REDUCE

User's Manual

Free Version

Anthony C. Hearn and Rainer Schöpf

https://reduce-algebra.sourceforge.io/

Revision 7146

June 23, 2025

Contents

Abstract

This document provides the user with a description of the algebraic programming system REDUCE. The capabilities of this system include:

1. expansion and ordering of polynomials and rational functions,
2. substitutions and pattern matching in a wide variety of forms,
3. automatic and user controlled simplification of expressions,
4. calculations with symbolic matrices,
5. arbitrary precision integer and real arithmetic,
6. facilities for defining new functions and extending program syntax,
7. analytic differentiation and integration,
8. factorization of polynomials,
9. facilities for the solution of a variety of algebraic equations,
10. facilities for the output of expressions in a variety of formats,
11. facilities for generating numerical programs from symbolic input,
12. expansion of expressions into power series,
13. graphical display of functions and data,
14. Dirac matrix calculations of interest to high energy physicists.

Acknowledgment

The production of this version of the manual has been the result of the contributions of a large number of individuals who have taken the time and effort to suggest improvements to previous versions, and to draft new sections. Particular thanks are due to Gerry Rayna, who provided a draft rewrite of most of the first half of the manual. Other people who have made significant contributions have included John Fitch, Martin Griss, Stan Kameny, Jed Marti, Herbert Melenk, Don Morrison, Arthur Norman, Eberhard Schrüfer, Larry Seward and Walter Tietze. Finally, Richard Hitt produced a TEX version of the REDUCE 3.3 manual, which has been a useful guide for the production of the LATEX version of this manual.

Chapter 1

Introductory Information

REDUCE is a system for carrying out algebraic operations accurately, no matter how complicated the expressions become. It can manipulate polynomials in a variety of forms, both expanding and factoring them, and extract various parts of them as required. REDUCE can also do differentiation and integration, but we shall only show trivial examples of this in this introduction. Other topics not considered include the use of arrays, the definition of procedures and operators, the specific routines for high energy physics calculations, the use of files to eliminate repetitious typing and for saving results, and the editing of the input text.

Also not considered in any detail in this introduction are the many options that are available for varying computational procedures, output forms, number systems used, and so on.

REDUCE is designed to be an interactive system, so that the user can input an algebraic expression and see its value before moving on to the next calculation. For those systems that do not support interactive use, or for those calculations, especially long ones, for which a standard script can be defined, REDUCE can also be used in batch mode. In this case, a sequence of commands can be given to REDUCE and results obtained without any user interaction during the computation.

In this introduction, we shall limit ourselves to the interactive use of REDUCE, since this illustrates most completely the capabilities of the system. When REDUCE is called, it begins by printing a banner message like:

```
Reduce (Free CSL version), 25-Oct-14 ...
```

where the version number and the system release date will change from time to time. It proceeds to execute the commands in user's startup (`reducerc`) file, if such a file is present, then prompts the user for input by:

```
1:
```

You can now type a REDUCE statement, terminated by a semicolon to indicate the end of the expression, for example:

```
(x+y+z)^2;
```

This expression would normally be followed by another character (a Return on an ASCII keyboard) to "wake up" the system, which would then input the expression, evaluate it, and return the result:

```
 2                      2            2
x  + 2*x*y + 2*x*z + y  + 2*y*z + z
```

Let us review this simple example to learn a little more about the way that REDUCE works. First, we note that REDUCE deals with variables, and constants like other computer languages, but that in evaluating the former, a variable can stand for itself. Expression evaluation normally follows the rules of high school algebra, so the only surprise in the above example might be that the expression was expanded. REDUCE normally expands expressions where possible, collecting like terms and ordering the variables in a specific manner. However, expansion, ordering of variables, format of output and so on is under control of the user, and various declarations are available to manipulate these.

Another characteristic of the above example is the use of lower case on input and upper case on output. In fact, input may be in either mode, but output is usually in lower case. To make the difference between input and output more distinct in this manual, all expressions intended for input will be shown in lower case and output in upper case. However, for stylistic reasons, we represent all single identifiers in the text in upper case.

Finally, the numerical prompt can be used to reference the result in a later computation.

As a further illustration of the system features, the user should try:

```
for i:= 1:40 product i;
```

The result in this case is the value of 40!,

```
815915283247897734345611269596115894272000000000
```

You can also get the same result by saying

```
factorial 40;
```

Since we want exact results in algebraic calculations, it is essential that integer arithmetic be performed to arbitrary precision, as in the above example. Further-

more, the FOR statement in the above is illustrative of a whole range of combining forms that REDUCE supports for the convenience of the user.

Among the many options in REDUCE is the use of other number systems, such as multiple precision floating point with any specified number of digits — of use if roundoff in, say, the 100^{th} digit is all that can be tolerated.

In many cases, it is necessary to use the results of one calculation in succeeding calculations. One way to do this is via an assignment for a variable, such as

```
u := (x+y+z)^2;
```

If we now use u in later calculations, the value of the right-hand side of the above will be used.

The results of a given calculation are also saved in the variable ws (for WorkSpace), so this can be used in the next calculation for further processing.

For example, the expression

```
df(ws,x);
```

following the previous evaluation will calculate the derivative of (x+y+z)^2 with respect to x. Alternatively,

```
int(ws,y);
```

would calculate the integral of the same expression with respect to y.

REDUCE is also capable of handling symbolic matrices. For example,

```
matrix m(2,2);
```

declares m to be a two by two matrix, and

```
m := mat((a,b),(c,d));
```

gives its elements values. Expressions that include m and make algebraic sense may now be evaluated, such as 1/m to give the inverse, 2*m - u*m^2 to give us another matrix and det(m) to give us the determinant of m.

REDUCE has a wide range of substitution capabilities. The system knows about elementary functions, but does not automatically invoke many of their well-known properties. For example, products of trigonometrical functions are not converted automatically into multiple angle expressions, but if the user wants this, he can say, for example:

```
(sin(a+b)+cos(a+b))*(sin(a-b)-cos(a-b))
```

```
        where cos(~x)*cos(~y)  = (cos(x+y)+cos(x-y))/2,
              cos(~x)*sin(~y)  = (sin(x+y)-sin(x-y))/2,
              sin(~x)*sin(~y)  = (cos(x-y)-cos(x+y))/2;
```

where the tilde in front of the variables `x` and `y` indicates that the rules apply for all values of those variables. The result of this calculation is

```
-(cos(2*a) + sin(2*b))
```

See also the sections on ASSIST (chapter 20.2), CAMAL (chapter 20.8) and TRIGSIMP (section 8.7).

Another very commonly used capability of the system, and an illustration of one of the many output modes of REDUCE, is the ability to output results in a FORTRAN compatible form. Such results can then be used in a FORTRAN based numerical calculation. This is particularly useful as a way of generating algebraic formulas to be used as the basis of extensive numerical calculations.

For example, the statements

```
on fort;
df(log(x)*(sin(x)+cos(x))/sqrt(x),x,2);
```

will result in the output

```
 ANS=(-4.*LOG(X)*COS(X)*X**2-4.*LOG(X)*COS(X)*X+3.*
 . LOG(X)*COS(X)-4.*LOG(X)*SIN(X)*X**2+4.*LOG(X)*
 . SIN(X)*X+3.*LOG(X)*SIN(X)+8.*COS(X)*X-8.*COS(X)-8.
 . *SIN(X)*X-8.*SIN(X))/(4.*SQRT(X)*X**2)
```

These algebraic manipulations illustrate the algebraic mode of REDUCE. REDUCE is based on Standard Lisp. A symbolic mode is also available for executing Lisp statements. These statements follow the syntax of Lisp, e.g.

```
symbolic car '(a);
```

Communication between the two modes is possible.

With this simple introduction, you are now in a position to study the material in the full REDUCE manual in order to learn just how extensive the range of facilities really is. If further tutorial material is desired, the seven REDUCE Interactive Lessons by David R. Stoutemyer are recommended. These are normally distributed with the system.

Chapter 2

Structure of Programs

A REDUCE program consists of a set of functional commands which are evaluated sequentially by the computer. These commands are built up from declarations, statements and expressions. Such entities are composed of sequences of numbers, variables, operators, strings, reserved words and delimiters (such as commas and parentheses), which in turn are sequences of basic characters.

2.1 The REDUCE Standard Character Set

The basic characters which are used to build REDUCE symbols are the following:

1. The 26 letters `a` through `z`
2. The 10 decimal digits `0` through `9`
3. The special characters `_ ! " $ % ' ( ) * + , - . / : ; < > = { }` ⟨*blank*⟩

With the exception of strings and characters preceded by an exclamation mark, the case of characters is ignored: depending of the underlying LISP they will all be converted internally into lower case or upper case: `ALPHA`, `Alpha` and `alpha` represent the same symbol. Most implementations allow you to switch this conversion off. The operating instructions for a particular implementation should be consulted on this point. For portability, we shall limit ourselves to the standard character set in this exposition.

2.2 Numbers

There are several different types of numbers available in REDUCE. Integers consist of a signed or unsigned sequence of decimal digits written without a decimal point, for example:

```
-2, 5396, +32
```

In principle, there is no practical limit on the number of digits permitted as exact arithmetic is used in most implementations. (You should however check the specific instructions for your particular system implementation to make sure that this is true.) For example, if you ask for the value of 2^{2000} you get it displayed as a number of 603 decimal digits, taking up several lines of output on an interactive display. It should be borne in mind of course that computations with such long numbers can be quite slow.

Numbers that aren't integers are usually represented as the quotient of two integers, in lowest terms: that is, as rational numbers.

In essentially all versions of REDUCE it is also possible (but not always desirable!) to ask REDUCE to work with floating point approximations to numbers again, to any precision. Such numbers are called *real*. They can be input in two ways:

1. as a signed or unsigned sequence of any number of decimal digits with an embedded or trailing decimal point;

2. as in 1. followed by a decimal exponent which is written as the letter `e` followed by a signed or unsigned integer.

E.g. `32.`, `+32.0`, `0.32e2` and `320.e-1` are all representations of 32.

The declaration `scientific_notation` controls the output format of floating point numbers. At the default settings, any number with five or less digits before the decimal point is printed in a fixed-point notation, e.g., `12345.6`. Numbers with more than five digits are printed in scientific notation, e.g., `1.234567e+5`. Similarly, by default, any number with eleven or more zeros after the decimal point is printed in scientific notation. To change these defaults, `scientific_notation` can be used in one of two ways.

`SCIENTIFIC_NOTATION` *m*,

with *m* a positive integer, sets the printing format so that a number with more than *m* digits before the decimal point, or *m* or more zeros after the decimal point, is printed in scientific notation.

`SCIENTIFIC_NOTATION{`*m*`,`*n*`}`,

with *m* and *n* both positive integers, sets the format so that a number with more

than *m* digits before the decimal point, or *n* or more zeros after the decimal point is printed in scientific notation.

CAUTION: The unsigned part of any number may *not* begin with a decimal point, as this causes confusion with the `CONS (.)` operator, i.e., *NOT ALLOWED ARE*: `.5`, `-.23`, `+.12`; use `0.5`, `-0.23`, `+0.12` instead.

2.3 Identifiers

Identifiers in REDUCE consist of one or more alphanumeric characters (i.e. alphabetic letters or decimal digits) the first of which must be alphabetic. The maximum number of characters allowed is implementation dependent, although twenty-four is permitted in most implementations. In addition, the underscore character (_) is considered a letter if it is *within* an identifier. For example,

```
a az p1 q23p  a_very_long_variable
```

are all identifiers, whereas

```
_a
```

is not.

A sequence of alphanumeric characters in which the first is a digit is interpreted as a product. For example, `2ab3c` is interpreted as $2 \times ab3c$. There is one exception to this: If the first letter after a digit is `e`, the system will try to interpret that part of the sequence as a real number, which may fail in some cases. For example, `2e12` is the real number 2.0×10^{12} and `2e3c` is $2000.0 \times c$. If the `e` is not followed by a number, 0 is assumed as the decimal exponent, thus `2e` is interpreted as 2 and `2ebc` as $2 \times bc$.

Special characters, such as $-$, $\star$, and blank, may be used in identifiers too, even as the first character, but each must be preceded by an exclamation mark in input. For example:

```
light!-years    d!*!*n          good! morning
!$sign          !5goldrings
```

CAUTION: Many system identifiers have such special characters in their names (especially $\star$ and =). If the user accidentally picks the name of one of them for his own purposes it may have catastrophic consequences for his REDUCE run. Users are therefore advised to avoid such names.

Identifiers are used as variables, labels and to name arrays, operators and procedures.

In graphical environments with typeset mathematics enabled, the (shared) variable `fancy_lower_digits` can be set to one of the values `t`, `nil` or `all` to control the display of digits within identifiers. The default value is `t`. Digits in an identifier are typeset as subscripts if `fancy_lower_digits = all` or if `fancy_lower_digits = t` and the digits are all at the end of the identifier. For example, with the following values assigned to `fancy_lower_digits`, the identifiers `ab12cd34` and `abcd34` are displayed as follows:

`fancy_lower_digits`	`ab12cd34`	`abcd34`
`t`	$ab12cd34$	$abcd_{34}$
`all`	$ab_{12}cd_{34}$	$abcd_{34}$
`nil`	$ab12cd34$	$abcd34$

Restrictions

The reserved words listed in Appendix A may not be used as identifiers. No spaces may appear within an identifier, and an identifier may not extend over a line of text.

2.4 Variables

Every variable is named by an identifier, and is given a specific type. The type is of no concern to the ordinary user. Most variables are allowed to have the default type, called *scalar*. These can receive, as values, the representation of any ordinary algebraic expression. In the absence of such a value, they stand for themselves.

Reserved Variables

Several variables in REDUCE have particular properties which should not be changed by the user. These variables include:

`Catalan` Catalan's constant, defined as

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2}.$$

`e` Intended to represent the base of the natural logarithms. `log(e)`, if it occurs in an expression, is automatically replaced by 1. If `rounded` is on, `e` is replaced by the value of e to the current degree of floating point precision.

`Euler_Gamma` Euler's constant, also available as $-\psi(1)$.

`Golden_Ratio` The number $\frac{1+\sqrt{5}}{2}$.

`i` Intended to represent the square root of -1. `i^2` is replaced by -1, and appropriately for higher powers of `i`. This applies only to the symbol `i` used on the top level, not as a formal parameter in a procedure, a local variable, nor in the context `for i:= ...`.

`infinity` Intended to represent ∞ in limit and power series calculations for example, as well as in definite integration. Note however that the current system does *not* do proper arithmetic on ∞. For example, `infinity + infinity` is `2*infinity`.

`Khinchin` Khinchin's constant, defined as

$$\prod_{n=1}^{\infty}\left(1+\frac{1}{n(n+2)}\right)^{\log_2 n}.$$

`negative` Used in the ROOTS package.

`nil` In REDUCE (algebraic mode only) taken as a synonym for zero. Therefore `nil` cannot be used as a variable.

`pi` Intended to represent the circular constant. With `rounded` on, it is replaced by the value of π to the current degree of floating point precision.

`positive` Used in the ROOTS package.

`t` Must not be used as a formal parameter or local variable in procedures, since conflict arises with the symbolic mode meaning of `t` as *true*.

Other reserved variables, such as `low_pow`, described in other sections, are listed in Appendix A. Using these reserved variables inappropriately will lead to errors.

There are also internal variables used by REDUCE that have similar restrictions. These usually have an asterisk in their names, so it is unlikely a casual user would use one. An example of such a variable is `k!*` used in the asymptotic command package.

Certain words are reserved in REDUCE. They may only be used in the manner intended. A list of these is given in Appendix A. There are, of course, an impossibly large number of such names to keep in mind. The reader may therefore want to make himself a copy of the list, deleting the names he doesn't think he is likely to use by mistake.

2.5 Strings

Strings are used in `write` statements, in other output statements (such as error messages), and to name files. A string consists of any number of characters enclosed in double quotes. For example:

```
"A String"
```

Lower case characters within a string are not converted to upper case.

The string `""` represents the empty string. A double quote may be included in a string by preceding it by another double quote. Thus `"a""b"` is the string `a"b`, and `""""` is the string consisting of the single character `"`.

2.6 Comments

Text can be included in program listings for the convenience of human readers, in such a way that REDUCE pays no attention to it. There are three ways to do this:

1. Everything from the word `comment` to the next statement terminator, normally `;` or `$`, is ignored. Such comments can be placed anywhere a blank could properly appear. (Note that `end` and >> are *not* treated as `comment` delimiters!)

2. Everything from the symbol `%` to the end of the line on which it appears is ignored. Such comments can be placed as the last part of any line. Statement terminators have no special meaning in such comments. Remember to put a semicolon before the `%` if the earlier part of the line is intended to be so terminated. Remember also to begin each line of a multi-line `%` comment with a `%` sign.

3. C-style inline comments: everything from `/*` to `*/` is ignored.

2.7 Operators

Operators in REDUCE are specified by name and type. There are two types, infix and prefix. Operators can be purely abstract, just symbols with no properties; they can have values assigned (using `:=` or simple `let` declarations) for specific arguments; they can have properties declared for some collection of arguments (using more general `let` declarations); or they can be fully defined (usually by a procedure declaration).

Infix operators have a definite precedence with respect to one another, and normally occur between their arguments. For example:

`a + b - c`	(spaces optional)
`x<y and y=z`	(spaces required where shown)

Spaces can be freely inserted between operators and variables or operators and operators. They are required only where operator names are spelled out with letters (such as the `and` in the example) and must be unambiguously separated from another such or from a variable (like `y`). Wherever one space can be used, so can any larger number.

Prefix operators occur to the left of their arguments, which are written as a list enclosed in parentheses and separated by commas, as with normal mathematical functions, e.g.,

```
cos(u)
df(x^2,x)
q(v+w)
```

Unmatched parentheses, incorrect groupings of infix operators and the like, naturally lead to syntax errors. The parentheses can be omitted (replaced by a space following the operator name) if the operator is unary and the argument is a single symbol or begins with a prefix operator name:

`cos y`	means cos(y)
`cos (-y)`	– parentheses necessary
`log cos y`	means log(cos(y))
`log cos (a+b)`	means log(cos(a+b))

but

`cos a*b`	means (cos a)*b
`cos -y`	is erroneous (treated as a variable "`cos`" minus the variable `y`)

A unary prefix operator has a precedence higher than any infix operator, including unary infix operators. In other words, REDUCE will always interpret `cos y + 3` as `(cos y) + 3` rather than as `cos(y + 3)`.

Infix operators may also be used in a prefix format on input, e.g., `+(a,b,c)`. On output, however, such expressions will always be printed in infix form (i.e., `a + b + c` for this example).

A number of prefix operators are built into the system with predefined properties. Users may also add new operators and define their rules for simplification. The built in operators are described in another section.

Built-In Infix Operators

The following infix operators are built into the system. They are all defined internally as procedures.

⟨*infix operator*⟩ ⟶ `where` | `:=` | `or` | `and` | `member` | `memq` | `=` | `neq` | `eq` | `>=` | `>` | `<=` | `<` | `+` | `-` | `*` | `/` | `^` | `**` | `.`

These operators may be further divided into the following subclasses:

⟨*assignment operator*⟩ ⟶ `:=`
⟨*logical operator*⟩ ⟶ `or` | `and` | `member` | `memq`
⟨*relational operator*⟩ ⟶ `=` | `neq` | `eq` | `>=` | `>` | `<=` | `<`
⟨*substitution operator*⟩ ⟶ `where`
⟨*arithmetic operator*⟩ ⟶ `+` | `-` | `*` | `/` | `^` | `**`
⟨*construction operator*⟩ ⟶ `.`

`memq` and `eq` are not used in the algebraic mode of REDUCE. They are explained in the section on symbolic mode. `where` is described in the section on substitutions.

In previous versions of REDUCE, `not` was also defined as an infix operator. In the present version it is a regular prefix operator, and interchangeable with `null`.

For compatibility with the intermediate language used by REDUCE, each special character infix operator has an alternative alphanumeric identifier associated with it. These identifiers may be used interchangeably with the corresponding special character names on input. This correspondence is as follows:

`:=`	`setq`	(the assignment operator)
`=`	`equal`	
`>=`	`geq`	
`>`	`greaterp`	
`<=`	`leq`	
`<`	`lessp`	
`+`	`plus`	
`-`	`difference`	(if unary, `minus`)
`*`	`times`	
`/`	`quotient`	(if unary, `recip`)
`^ or **`	`expt`	(raising to a power)
`.`	`cons`	

Note: `neq` is used to mean *not equal*. There is no special symbol provided for it.

The above operators are binary, except `not` which is unary and + and ⋆ which are nary (i.e., taking an arbitrary number of arguments). In addition, − and / may be used as unary operators, e.g., `/2` means the same as `1/2`. Any other operator is parsed as a binary operator using a left association rule. Thus `a/b/c` is interpreted as `(a/b)/c`. There are two exceptions to this rule: := and . are right associative. Example: `a:=b:=c` is interpreted as `a:=(b:=c)`. Unlike ALGOL and PASCAL, ^ is left associative. In other words, `a^b^c` is interpreted as `(a^b)^c`.

The operators <, <=, >, >= can only be used for making comparisons between numbers. No meaning is currently assigned to this kind of comparison between general expressions.

Parentheses may be used to specify the order of combination. If parentheses are omitted then this order is by the ordering of the precedence list defined by the right-hand side of the ⟨*infix operator*⟩ table at the beginning of this section, from lowest to highest. In other words, `where` has the lowest precedence, and . (the dot operator) the highest.

Chapter 3

Expressions

REDUCE expressions may be of several types and consist of sequences of numbers, variables, operators, left and right parentheses and commas. The most common types are as follows:

3.1 Scalar Expressions

Using the arithmetic operations `+ - * / ^` (power) and parentheses, scalar expressions are composed from numbers, ordinary "scalar" variables (identifiers), array names with subscripts, operator or procedure names with arguments and statement expressions.

Examples:

```
x
x^3 - 2*y/(2*z^2 - df(x,z))
(p^2 + m^2)^(1/2)*log (y/m)
a(5) + b(i,q)
```

The symbol `**` may be used as an alternative to the caret symbol (`^`) for forming powers, particularly in those systems that do not support a caret symbol. For details of operator precedence and associativity, see section 2.7.

Statement expressions, usually in parentheses, can also form part of a scalar expression, as in the example

```
w + (c:=x+y) + z
```

When the algebraic value of an expression is needed, REDUCE determines it, starting with the algebraic values of the parts, roughly as follows:

Variables and operator symbols with an argument list have the algebraic values they were last assigned, or if never assigned stand for themselves. However, array elements have the algebraic values they were last assigned, or, if never assigned, are taken to be 0.

Procedures are evaluated with the values of their actual parameters.

In evaluating expressions, the standard rules of algebra are applied. Unfortunately, this algebraic evaluation of an expression is not as unambiguous as is numerical evaluation. This process is generally referred to as "simplification" in the sense that the evaluation usually but not always produces a simplified form for the expression.

There are many options available to the user for carrying out such simplification. If the user doesn't specify any method, the default method is used. The default evaluation of an expression involves expansion of the expression and collection of like terms, ordering of the terms, evaluation of derivatives and other functions and substitution for any expressions which have values assigned or declared (see assignments and `let` statements). In many cases, this is all that the user needs.

The declarations by which the user can exercise some control over the way in which the evaluation is performed are explained in other sections. For example, if a real (floating point) number is encountered during evaluation, the system will normally convert it into a ratio of two integers. If the user wants to use real arithmetic, he can effect this by the command `on rounded;`. Other modes for coefficient arithmetic are described elsewhere.

If an illegal action occurs during evaluation (such as division by zero) or functions are called with the wrong number of arguments, and so on, an appropriate error message is generated.

3.2 Integer Expressions

These are expressions which, because of the values of the constants and variables in them, evaluate to whole numbers.

Examples:

```
2,      37 * 999,      (x + 3)^2 - x^2 - 6*x
```

are obviously integer expressions.

```
j + k - 2 * j^2
```

is an integer expression when `j` and `k` have values that are integers, or if not integers are such that "the variables and fractions cancel out", as in

```
k - 7/3 - j + 2/3 + 2*j^2.
```

3.3 Boolean Expressions

A boolean expression returns a truth value. In the algebraic mode of REDUCE, boolean expressions have the syntactical form:

⟨*expression*⟩ ⟨*relational operator*⟩ ⟨*expression*⟩

or

⟨*boolean operator*⟩ (⟨*arguments*⟩)

or

⟨*boolean expression*⟩ ⟨*logical operator*⟩ ⟨*boolean expression*⟩.

Parentheses can also be used to control the precedence of expressions.

In addition to the logical and relational operators defined earlier as infix operators, the following boolean operators are also defined:

`evenp(u)`	determines if the number `u` is even or not;
`fixp(u)`	determines if the expression `u` is integer or not;
`freeof(u,v)`	determines if the expression `u` does not contain the kernel `v` anywhere in its structure;
`numberp(u)`	determines if `u` is a number or not;
`ordp(u,v)`	determines if `u` is ordered ahead of `v` by some canonical ordering (based on the expression structure and an internal ordering of identifiers);
`primep(u)`	true if `u` is a prime object, i.e., any object other than 0 and plus or minus 1 which is only exactly divisible by itself or a unit.

Examples:

```
j<1
x>0  or  x=-2
```

```
numberp x
fixp x and evenp x
numberp x and x neq 0
```

Boolean expressions can only appear directly within `if`, `for`, `while`, and `until` statements, as described in other sections. Such expressions cannot be used in place of ordinary algebraic expressions, or assigned to a variable.

NB: For those familiar with symbolic mode, the meaning of some of these operators is different in that mode. For example, `numberp` is true only for integers and reals in symbolic mode.

When two or more boolean expressions are combined with `and`, they are evaluated one by one until a *false* expression is found. The rest are not evaluated. Thus

```
numberp x and numberp y and x>y
```

does not attempt to make the `x>y` comparison unless `x` and `y` are both verified to be numbers.

Similarly, evaluation of a sequence of boolean expressions connected by `or` stops as soon as a *true* expression is found.

NB: In a boolean expression, and in a place where a boolean expression is expected, the algebraic value 0 is interpreted as *false*, while all other algebraic values are converted to *true*. So in algebraic mode a procedure can be written for direct usage in boolean expressions, returning say 1 or 0 as its value as in

```
procedure polynomialp(u,x);
   if den(u)=1 and deg(u,x)>=1 then 1 else 0;
```

One can then use this in a boolean construct, such as

```
if polynomialp(q,z) and not polynomialp(q,y) then ...
```

In addition, any procedure that does not have a defined return value (for example, a block without a `return` statement in it) has the boolean value *false*.

3.4 Equations

Equations are a particular type of expression with the syntax

⟨*expression*⟩ = ⟨*expression*⟩.

In addition to their role as boolean expressions, they can also be used as arguments

to several operators (e.g., `solve`), and can be returned as values.

Under normal circumstances, the right-hand-side of the equation is evaluated but not the left-hand-side. This also applies to any substitutions made by the `sub` operator. If both sides are to be evaluated, the switch `evallhseqp` should be turned on.

To facilitate the handling of equations, two selectors, `lhs` and `rhs`, which return the left- and right-hand sides of an equation respectively, are provided. For example,

```
        lhs(a+b=c) -> a+b
and
        rhs(a+b=c) -> c.
```

3.5 Proper Statements as Expressions

Several kinds of proper statements deliver an algebraic or numerical result of some kind, which can in turn be used as an expression or part of an expression. For example, an assignment statement itself has a value, namely the value assigned. So

```
        2 * (x := a+b)
```

is equal to `2*(a+b)`, as well as having the "side-effect" of assigning the value `a+b` to `x`. In context,

```
        y := 2 * (x := a+b);
```

sets `x` to `a+b` and `y` to `2*(a+b)`.

The sections on the various proper statement types indicate which of these statements are also useful as expressions.

Chapter 4

Lists

A list is an object consisting of a sequence of other objects (including lists themselves), separated by commas and surrounded by braces. Examples of lists are:

```
{a,b,c}

{1,a-b,c=d}

{{a},{{b,c},d},e}.
```

The empty list is represented as

```
{}.
```

4.1 Operations on Lists

Several operators in the system return their results as lists, and a user can create new lists using braces and commas. Alternatively, one can use the operator `list` to construct a list. An important class of operations on lists are `map` and `select` operations. For details, please refer to the chapters on `map`, `select` and the `for` command. See also the documentation on the ASSIST (chapter 20.2) package.

To facilitate the use of lists, a number of operators are also available for manipulating them. `part(`⟨*list*⟩`,n)` for example will return the n^{th} element of a list. `length` will return the length of a list. Several operators are also defined uniquely for lists. For those familiar with them, these operators in fact mirror the operations defined for Lisp lists. These operators are as follows:

4.1.1 list

The operator `list` is an alternative to the usage of curly brackets. `list` accepts an arbitrary number of arguments and returns a list of its arguments. This operator is useful in cases where operators have to be passed as arguments. E.g.,

```
list(a,list(list(b,c),d),e);        ->  {{a},{{b,c},d},e}
```

4.1.2 FIRST

This operator returns the first member of a list. An error occurs if the argument is not a list, or the list is empty.

4.1.3 SECOND

`second` returns the second member of a list. An error occurs if the argument is not a list or has no second element.

4.1.4 THIRD

This operator returns the third member of a list. An error occurs if the argument is not a list or has no third element.

4.1.5 REST

`rest` returns its argument with the first element removed. An error occurs if the argument is not a list, or is empty.

4.1.6 . (Cons) Operator

This operator adds ("conses") an expression to the front of a list. For example:

```
a . {b,c}      ->   {a,b,c}.
```

4.1.7 APPEND

This operator appends its first argument to its second to form a new list. *Examples:*

```
append({a,b},{c,d})      ->      {a,b,c,d}
append({{a,b}},{c,d})    ->      {{a,b},c,d}.
```

4.1.8 REVERSE

The operator `reverse` returns its argument with the elements in the reverse order. It only applies to the top level list, not any lower level lists that may occur. Examples are:

```
reverse({a,b,c})        ->      {c,b,a}
reverse({{a,b,c},d})    ->      {d,{a,b,c}}.
```

4.1.9 List Arguments of Other Operators

If an operator other than those specifically defined for lists is given a single argument that is a list, then the result of this operation will be a list in which that operator is applied to each element of the list. For example, the result of evaluating `log{a,b,c}` is the expression `{log(a),log(b),log(c)}`.

There are two ways to inhibit this operator distribution. Firstly, the switch `listargs`, if on, will globally inhibit such distribution. Secondly, one can inhibit this distribution for a specific operator by the declaration `listargp`. For example, with the declaration `listargp log`, `log{a,b,c}` would evaluate to `log({a,b,c})`.

If an operator has more than one argument, no such distribution occurs.

4.1.10 Caveats and Examples

Some of the natural list operations such as *member* or *delete* are available only after loading the package *ASSIST* (chapter 20.2).

Please note that a non-list as second argument to CONS (a "dotted pair" in LISP terms) is not allowed and causes an "invalid as list" error.

```
a := 17 . 4;

***** 17 4 invalid as list
```

Also, the initialization of a scalar variable is not the empty list – one has to set list type variables explicitly, as in the following example:

```
load_package assist;

procedure lotto (n,m);
 begin scalar list_1_n, luckies, hit;
    list_1_n := {};
```

```
    luckies := {};
    for k:=1:n do list_1_n := k . list_1_n;
    for k:=1:m do
      << hit := part(list_1_n,random(n-k+1) + 1);
         list_1_n := delete(hit,list_1_n);
         luckies := hit . luckies >>;
    return luckies;
end;
                % In Germany, try lotto (49,6);
```

Another example: Find all coefficients of a multivariate polynomial with respect to a list of variables:

```
procedure allcoeffs(q,lis);
   % q : polynomial, lis: list of vars
   allcoeffs1 (list q,lis);

procedure allcoeffs1(q,lis);
  if lis={} then q else
    allcoeffs1(foreach qq in q join coeff(qq,first lis),
               rest lis);
```

Chapter 5

Statements

A statement is any combination of reserved words and expressions, and has the syntax

⟨*statement*⟩ ⟶ ⟨*expression*⟩ | ⟨*proper statement*⟩

A REDUCE program consists of a series of commands which are statements followed by a terminator:

⟨*terminator*⟩ ⟶ `;` | `$`

The division of the program into lines is arbitrary. Several statements can be on one line, or one statement can be freely broken onto several lines. If the program is run interactively, statements ending with ; or $ are not processed until an end-of-line character is encountered. This character can vary from system to system, but is normally the Return key on an ASCII terminal. Specific systems may also use additional keys as statement terminators.

If a statement is a proper statement, the appropriate action takes place.

Depending on the nature of the proper statement some result or response may or may not be printed out, and the response may or may not depend on the terminator used.

If a statement is an expression, it is evaluated. If the terminator is a semicolon, the result is printed. If the terminator is a dollar sign, the result is not printed. Because it is not usually possible to know in advance how large an expression will be, no explicit format statements are offered to the user. However, a variety of output declarations are available so that the output can be produced in different forms. These output declarations are explained in Section 8.3.3.

The following sub-sections describe the types of proper statements in REDUCE.

5.1 Assignment Statements

These statements have the syntax

⟨*assignment statement*⟩ ⟶ ⟨*expression*⟩ `:=` ⟨*expression*⟩

The ⟨*expression*⟩ on the left side is normally the name of a variable, an operator symbol with its list of arguments filled in, or an array name with the proper number of integer subscript values within the array bounds. For example:

```
a1 := b + c
h(l,m) := x-2*y          (where h is an operator)
k(3,5) := x-2*y          (where k is a 2-dim. array)
```

More general assignments such as `a+b := c` are also allowed. The effect of these is explained in Section 11.2.5.

An assignment statement causes the expression on the right-hand-side to be evaluated. If the left-hand-side is a variable, the value of the right-hand-side is assigned to that unevaluated variable. If the left-hand-side is an operator or array expression, the arguments of that operator or array are evaluated, but no other simplification done. The evaluated right-hand-side is then assigned to the resulting expression. For example, if `a` is a single-dimensional array, `a(1+1) := b` assigns the value `b` to the array element `a(2)`.

If a semicolon is used as the terminator when an assignment is issued as a command (i.e. not as a part of a group statement or procedure or other similar construct), the left-hand side symbol of the assignment statement is printed out, followed by a "`:=`", followed by the value of the expression on the right.

It is also possible to write a multiple assignment statement:

⟨*expression*⟩ `:=` ... `:=` ⟨*expression*⟩ `:=` ⟨*expression*⟩

In this form, each ⟨*expression*⟩ but the last is set to the value of the last ⟨*expression*⟩. If a semicolon is used as a terminator, each expression except the last is printed followed by a "`:=`" ending with the value of the last expression.

5.1.1 Set and Unset Statements

In some cases, it is desirable to perform an assignment in which *both* the left- and right-hand sides of an assignment are evaluated. In this case, the `set` statement can be used with the syntax:

`set(`⟨*expression*⟩`,` ⟨*expression*⟩`);`

For example, the statements

```
j := 23;
set(mkid(a,j),x);
```

assigns the value `x` to `a23`. (See also `mkid`.)

To remove a value from such a variable, the `unset` statement can be used with the syntax:

`unset(`⟨*expression*⟩`);`

For example, the statement

```
j := 23;
unset(mkid(a,j));
```

clears the value of `a23`.

The command `unset` also acts like an indirect version of `clear` for operator values; for each operator value it clears the value assigned to the operator value, rather than the operator value itself. For example

```
operator a, b;
a(1) := x;  b(1) := 4;
a(2) := y;  b(2) := 3;
set(a(1), b(1));          % performs x := 4
set(a(2), b(2));          % performs y := 3
x*y;                      % returns  12
unset a(1), a(2);         % performs clear x, y
x*y;                      % returns  x*y
{a(1),b(1),a(2),b(2)};    % returns  {x,4,y,3}
```

Note that the `unset` command clears the values of `x` and `y`, but leaves `a(1)`, `b(1)`, `a(2)`, `b(2)` assigned.

5.2 Group Statements

The group statement is a construct used where REDUCE expects a single statement, but a series of actions needs to be performed. It is formed by enclosing one or more statements (of any kind) between the symbols << and >>, separated by semicolons or dollar signs – it doesn't matter which. The statements are executed one after another.

Examples will be given in the sections on `if` and other types of statements in which the << ... >> construct is useful.

If the last statement in the enclosed group has a value, then that is also the value of the group statement. Care must be taken not to have a semicolon or dollar sign after the last grouped statement, if the value of the group is relevant: such an extra terminator causes the group to have the value NIL or zero.

5.3 Conditional Statements

The conditional statement has the following syntax:

⟨*conditional statement*⟩ ⟶ `if` ⟨*boolean expression*⟩
`then` ⟨*statement*⟩
[`else` ⟨*statement*⟩]

The boolean expression is evaluated. If this is *true*, the first ⟨*statement*⟩ is executed. If it is *false*, the second is.

Examples:

```
if x=5 then a:=b+c else d:=e+f

if x=5 and numberp y
   then <<ff:=q1; a:=b+c>>
   else <<ff:=q2; d:=e+f>>
```

Note the use of the group statement.

Conditional statements associate to the right; i.e.,

```
IF <a> THEN <b> ELSE IF <c> THEN <d> ELSE <e>
```

is equivalent to:

```
IF <a> THEN <b> ELSE (IF <c> THEN <d> ELSE <e>)
```

In addition, the construction

```
IF <a> THEN IF <b> THEN <c> ELSE <d>
```

parses as

```
IF <a> THEN (IF <b> THEN <c> ELSE <d>).
```

If the value of the conditional statement is of primary interest, it is often called a conditional expression instead. Its value is the value of whichever statement was

executed. (If the executed statement has no value, the conditional expression has no value or the value 0, depending on how it is used.)

Examples:

```
a:=if x<5 then 123 else 456;
b:=u + v^(if numberp z then 10*z  else 1) + w;
```

If the value is of no concern, the `else` clause may be omitted if no action is required in the *false* case.

```
if x=5 then a:=b+c;
```

Note: As explained in Section 3.3, if a scalar or numerical expression is used in place of the boolean expression – for example, a variable is written there – the *true* alternative is followed unless the expression has the value 0.

5.4 FOR Statements

The `for` statement is used to define a variety of program loops. Its general syntax is as follows:

$$\texttt{for}\ \left\{\begin{array}{l}\langle var\rangle\ \texttt{:=}\ \langle number\rangle\ \left\{\begin{array}{c}\texttt{step}\ \langle number\rangle\ \texttt{until}\\ \texttt{:}\end{array}\right\}\ \langle number\rangle\\ \texttt{each}\ \langle var\rangle\ \left\{\begin{array}{l}\texttt{in}\\ \texttt{on}\end{array}\right\}\ \langle list\rangle\end{array}\right\}\ \langle action\rangle\ \langle exprn\rangle$$

where

$$\langle action\rangle \longrightarrow \texttt{do} \mid \texttt{product} \mid \texttt{sum} \mid \texttt{collect} \mid \texttt{join}.$$

The assignment form of the `for` statement defines an iteration over the indicated numerical range. If expressions that do not evaluate to numbers are used in the designated places, an error will result.

The `for each` form of the `for` statement is designed to iterate down a list. Again, an error will occur if a list is not used.

The action `do` means that ⟨*exprn*⟩ is simply evaluated and no value kept; the statement returning 0 in this case (or no value at the top level). `collect` means that the results of evaluating ⟨*exprn*⟩ each time are linked together to make a list, and `join` means that the values of ⟨*exprn*⟩ are themselves lists that are joined to make one list (similar to `conc` in Lisp). Finally, `product` and `sum` form the respective combined value out of the values of ⟨*exprn*⟩.

In all cases, ⟨*exprn*⟩ is evaluated algebraically within the scope of the current value of ⟨*var*⟩. If ⟨*action*⟩ is `do`, then nothing else happens. In other cases, ⟨*action*⟩ is

a binary operator that causes a result to be built up and returned by `for`. In those cases, the loop is initialized to a default value (`0` for `sum` `1` for `product`, and an empty list for the other actions). The test for the end condition is made before any action is taken. As in Pascal, if the variable is out of range in the assignment case, or the ⟨*list*⟩ is empty in the `for each` case, ⟨*exprn*⟩ is not evaluated at all.

Examples:

1. If `a`, `b` have been declared to be arrays, the following stores 5^2 through 10^2 in `a(5)` through `a(10)`, and at the same time stores the cubes in the `b` array:

```
for i := 5 step 1 until 10 do
    <<a(i):=i^2; b(i):=i^3>>
```

2. As a convenience, the common construction

```
step 1 until
```

   may be abbreviated to a colon. Thus, instead of the above we could write:

```
for i := 5:10 do <<a(i):=i^2; b(i):=i^3>>
```

3. The following sets `c` to the sum of the squares of 1,3,5,7,9; and `d` to the expression `x*(x+1)*(x+2)*(x+3)*(x+4)`:

```
c := for j:=1 step 2 until 9 sum j^2;
d := for k:=0 step 1 until 4 product (x+k);
```

4. The following forms a list of the squares of the elements of the list `{a,b,c}`:

```
for each x in {a,b,c} collect x^2;
```

5. The following forms a list of the listed squares of the elements of the list `{a,b,c}` (i.e., `{{a^2},{b^2},{c^2}}`):

```
for each x in {a,b,c} collect {x^2};
```

6. The following also forms a list of the squares of the elements of the list `{a,b,c}`, since the `join` operation joins the individual lists into one list:

```
for each x in {a,b,c} join {x^2};
```

The control variable used in the `for` statement is actually a new variable, not related to the variable of the same name outside the `for` statement. In other words, executing a statement `for i:=` ... doesn't change the system's assumption that $i^2 = -1$. Furthermore, in algebraic mode, the value of the control variable is substituted in ⟨*exprn*⟩ only if it occurs explicitly in that expression. It will not replace a variable of the same name in the value of that expression. For example:

```
b := a; for a := 1:2 do write b;
```

prints `A` twice, not 1 followed by 2.

5.5 WHILE ... DO

The `for ... do` feature allows easy coding of a repeated operation in which the number of repetitions is known in advance. If the criterion for repetition is more complicated, `while ... do` can often be used. Its syntax is:

`while` ⟨*boolean expression*⟩ `do` ⟨*statement*⟩

The `while ... do` controls the single statement following `do`. If several statements are to be repeated, as is almost always the case, they must be grouped using the `<< ... >>` or `begin ... end` as in the example below.

The `while` condition is tested each time *before* the action following the `do` is attempted. If the condition is false to begin with, the action is not performed at all. Make sure that what is to be tested has an appropriate value initially.

Example:

Suppose we want to add up a series of terms, generated one by one, until we reach a term which is less than 1/1000 in value. For our simple example, let us suppose the first term equals 1 and each term is obtained from the one before by taking one third of it and adding one third its square. We would write:

```
ex:=0; term:=1;
while num(term - 1/1000) >= 0  do
        <<ex := ex+term; term:=(term + term^2)/3>>;
ex;
```

As long as `term` is greater than or equal to (>=) 1/1000 it will be added to `ex` and the next `term` calculated. As soon as `term` becomes less than 1/1000 the `while` test fails and the `term` will not be added.

5.6 REPEAT ... UNTIL

`repeat ... until` is very similar in purpose to `while ... do`. Its syntax is:

`repeat` ⟨*statement*⟩ `until` ⟨*boolean expression*⟩

(PASCAL users note: Only a single statement – usually a group statement – is allowed between the `repeat` and the `until`.)

There are two essential differences:

1. The test is performed *after* the controlled statement (or group of statements) is executed, so the controlled statement is always executed at least once.
2. The test is a test for when to stop rather than when to continue, so its "polarity" is the opposite of that in `while ... do`.

As an example, we rewrite the example from the `while ...do` section:

```
ex:=0; term:=1;
repeat <<ex := ex+term; term := (term + term^2)/3>>
   until num(term - 1/1000) < 0;
ex;
```

In this case, the answer will be the same as before, because in neither case is a term added to `ex` which is less than 1/1000.

5.7 Compound Statements

Often the desired process can best (or only) be described as a series of steps to be carried out one after the other. In many cases, this can be achieved by use of the group statement. However, each step often provides some intermediate result, until at the end we have the final result wanted. Alternatively, iterations on the steps are needed that are not possible with constructs such as `while` or `repeat` statements. In such cases the steps of the process must be enclosed between the words `begin` and `end` forming what is technically called a *block* or *compound* statement. Such a compound statement can in fact be used wherever a group statement appears. The converse is not true: `begin ...end` can be used in ways that `<< ... >>` cannot.

If intermediate results must be formed, local variables must be provided in which to store them. *Local* means that their values are deleted as soon as the block's operations are complete, and there is no conflict with variables outside the block that happen to have the same name. Local variables are created by a `scalar` declaration immediately after the `begin`:

```
scalar a,b,c,z;
```

If more convenient, several `scalar` declarations can be given one after another:

```
scalar a,b,c;
scalar z;
```

In place of `scalar` one can also use the declarations `integer` or `real`. In the present version of REDUCE variables declared `integer` are expected to have only integer values, and are initialized by default to 0. `real` variables on the other hand are currently treated as algebraic mode `scalar`s.

CAUTION: `integer`, `real` and `scalar` declarations can only be given immediately after a `begin`. An error will result if they are used after other statements in a block (including `array` and `operator` declarations, which are global in scope), or outside the top-most block (e.g., at the top level). All variables declared `scalar` are automatically initialized by default to zero in algebraic mode (`nil` in symbolic mode).

Optionally, each variable appearing in a `scalar`, `integer` or `real` declaration can be followed by an assignment operator and an initial value, which overrides the default initial value. For example,

```
scalar x := 5;
```

has the same effect as

```
scalar x; x := 5;
```

Any symbols not declared as local variables in a block refer to the variables of the same name in the current calling environment. In particular, if they are not so declared at a higher level (e.g., in a surrounding block or as parameters in a calling procedure), their values can be permanently changed.

Following the `scalar` declaration(s), if any, write the statements to be executed, one after the other, separated by delimiters (e.g., `;` or `$`) (it doesn't matter which). However, from a stylistic point of view, `;` is preferred.

The last statement in the body, just before `end`, need not have a terminator (since the `begin ... end` are in a sense brackets confining the block statements). The last statement must also be the command `return` followed by the variable or expression whose value is to be the value returned by the procedure. If the `return` is omitted (or nothing is written after the word `return`) the procedure will have no value or the value zero, depending on how it is used (and `nil` in symbolic mode). Remember to put a terminator after the `end`.

Examples:

Given a previously assigned integer value for n, the following block will compute the Legendre polynomial of degree n in the variable x:

```
begin scalar seed,deriv,top,fact;
   seed:=1/(y^2 - 2*x*y +1)^(1/2);
   deriv:=df(seed,y,n);
   top:=sub(y=0,deriv);
   fact:=for i:=1:n product i;
   return top/fact
end;
```

This block uses explicit initialization and computes the 10th Fibonacci number:

```
begin integer f1 := 1, f2 := 1, f3;
   for i := 3 : 10 do
      << f3 := f1 + f2; f1 := f2; f2 := f3 >>;
   return f3
end;
```

5.7.1 Compound Statements with GO TO

It is possible to have more complicated structures inside the `begin... end` brackets than indicated in the previous example. That the individual lines of the program need not be assignment statements, but could be almost any other kind of statement or command, needs no explanation. For example, conditional statements, and `while` and `repeat` constructions, have an obvious role in defining more intricate blocks.

If these structured constructs don't suffice, it is possible to use labels and `go tos` within a compound statement, and also to use `return` in places within the block other than just before the `end`. The following subsections discuss these matters in detail. For many readers the following example, presenting one possible definition of a process to calculate the factorial of n for preassigned n will suffice:

Example:

```
begin scalar m;
    m:=1;
 l: if n=0 then return m;
    m:=m*n;
    n:=n-1;
    go to l
end;
```

5.7.2 Labels and GO TO Statements

Within a `begin ... end` compound statement it is possible to label statements, and transfer to them out of sequence using `go to` statements. Only statements on the top level inside compound statements can be labeled, not ones inside subsidiary constructions like `<< ... >>`, `if ... then ...`, `while ... do ...`, etc.

Labels and `go to` statements have the syntax:

⟨*go to statement*⟩ ⟶ `go to` ⟨*label*⟩ | `goto` ⟨*label*⟩
⟨*label*⟩ ⟶ ⟨*identifier*⟩
⟨*labeled statement*⟩ ⟶ ⟨*label*⟩ **:** ⟨*statement*⟩

Note that statement names cannot be used as labels.

While `go to` is an unconditional transfer, it is frequently used in conditional statements such as

```
if x>5 then go to abcd;
```

giving the effect of a conditional transfer.

Transfers using `go to`s can only occur within the block in which the `go to` is used. In other words, you cannot transfer from an inner block to an outer block using a `go to`. However, if a group statement occurs within a compound statement, it is possible to jump out of that group statement to a point within the compound statement using a `go to`.

5.7.3 RETURN Statements

The value corresponding to a `begin ... end` compound statement, such as a procedure body, is normally 0 (`nil` in symbolic mode). By executing a `return` statement in the compound statement a different value can be returned. After a `return` statement is executed, no further statements within the compound statement are executed.

`Examples:`

```
return x+y;
return m;
return;
```

Note that parentheses are not required around the `x+y`, although they are permitted. The last example is equivalent to `return 0` or `return nil`, depending on whether the block is used as part of an expression or not.

Since `return` actually moves up only one block level, in a sense the casual user is not expected to understand, we tabulate some cautions concerning its use.

1. `return` can be used on the top level inside the compound statement, i.e. as one of the statements bracketed together by the `begin ... end`
2. `return` can be used within a top level `<< ... >>` construction within the compound statement. In this case, the `return` transfers control out of both the group statement and the compound statement.
3. `return` can be used within an `if ... then ... else ...` on the top level within the compound statement.

NOTE: At present, there is no construct provided to permit early termination of a `for`, `while`, or `repeat` statement. In particular, the use of `return` in such cases results in a syntax error. For example,

```
begin scalar y;
   y := for i:=0:99 do if a(i)=x then return b(i);
   ...
```

will lead to an error.

Chapter 6

Commands and Declarations

A command is an order to the system to do something. Some commands cause visible results (such as calling for input or output); others, usually called declarations, set options, define properties of variables, or define procedures. Commands are formally defined as a statement followed by a terminator

⟨*command*⟩ ⟶ ⟨*statement*⟩⟨*terminator*⟩
⟨*terminator*⟩ ⟶ `;` | `$`

Some REDUCE commands and declarations are described in the following subsections.

6.1 Array Declarations

Array declarations in REDUCE are similar to FORTRAN dimension statements. For example:

```
array a(10),b(2,3,4);
```

Array indices each range from 0 to the value declared. An element of an array is referred to in standard FORTRAN notation, e.g. `a(2)`.

We can also use an expression for defining an array bound, provided the value of the expression is a positive integer. For example, if `x` has the value 10 and `y` the value 7 then `array c(5*x+y)` is the same as `array c(57)`.

If an array is referenced by an index outside its range, an error occurs. If the array is to be one-dimensional, and the bound a number or a variable (not a more general expression) the parentheses may be omitted:

```
array a 10, c 57;
```

The operator `length` applied to an array name returns a list of its dimensions.

All array elements are initialized to 0 at declaration time. In other words, an array element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used instead.

Array declarations can appear anywhere in a program. Once a symbol is declared to name an array, it can not also be used as a variable, or to name an operator or a procedure. It can however be re-declared to be an array, and its size may be changed at that time. An array name can also continue to be used as a parameter in a procedure, or a local variable in a compound statement, although this use is not recommended, since it can lead to user confusion over the type of the variable.

Arrays once declared are global in scope, and so can then be referenced anywhere in the program. In other words, unlike arrays in most other languages, a declaration within a block (or a procedure) does not limit the scope of the array to that block, nor does the array go away on exiting the block (use `clear` instead for this purpose).

6.2 Mode Handling Declarations

The `on` and `off` declarations are available to the user for controlling various system options. Each option is represented by a *switch* name. `on` and `off` take a list of switch names as argument and turn them on and off respectively, e.g.,

```
on time;
```

causes the system to print a message after each command giving the elapsed CPU time since the last command, or since `time` was last turned off, or the session began. Another useful switch with interactive use is `demo`, which causes the system to pause after each command in a file (with the exception of comments) until a Return is typed on the terminal. This enables a user to set up a demonstration file and step through it command by command.

As with most declarations, arguments to `on` and `off` may be strung together separated by commas. For example,

```
off time,demo;
```

will turn off both the time messages and the demonstration switch.

We note here that while most `on` and `off` commands are obeyed almost instantaneously, some trigger slower actions such as reading in necessary modules from secondary storage.

A diagnostic message is printed if `on` or `off` are used with a switch that is not

known to the system. For example, if you misspell `demo` and type

```
on demq;
```

you will get the message

```
***** demq not defined as switch.
```

6.3 END

The identifier `end` has two separate uses.

1) Its use in a `begin ... end` bracket has been discussed in connection with compound statements.

2) Files to be read using `in` should end with an extra `end`; command. The reason for this is explained in the section on the `in` command. This use of `end` does not allow an immediately preceding `end` (such as the `end` of a procedure definition), so we advise using `;end;` there.

6.4 BYE Command

The command `bye;` (or alternatively `quit;`) stops the execution of REDUCE, closes all open output files, and returns you to the calling program (usually the operating system). Your REDUCE session is normally destroyed.

6.5 Timing Facilities

The command `showtime` causes REDUCE to print the elapsed time since the last call of this command or, on its first call, since the current REDUCE session began.

Turning on the switch `time` causes REDUCE to print a message after each command giving the elapsed CPU time since the last command, or since `time` was last turned off, or the session began.

The time is normally given in milliseconds and gives the time as measured by a system clock. The operations covered by this measure are system dependent.

The Lisp function `time()` returns CPU time since the session began in milliseconds as an integer. It could be used in algebraic mode to time a computation like this:

```
% Make time() available in algebraic mode:
```

```
symbolic operator time;

start_time := time();
% perform a computation...
time_taken := time() - start_time;

write "Time taken was ", time_taken, " ms.";
```

Beware that modern computers can perform many computations in less than a millisecond, so the time taken may be reported inaccurately as zero.

6.6 DEFINE Command

The command `define` allows a user to supply a new name for any identifier or replace it by any well-formed expression. Its argument is a list of expressions of the form

⟨*identifier*⟩ = ⟨*number*⟩ | ⟨*identifier*⟩ | ⟨*operator*⟩ |
⟨*reserved word*⟩ | ⟨*expression*⟩

Example:

```
define be==,x=y+z;
```

means that `be` will be interpreted as an equal sign, and `x` as the expression `y+z` from then on. This renaming is done at parse time, and therefore takes precedence over any other replacement declared for the same identifier. It stays in effect until the end of the REDUCE run.

The identifiers `algebraic` and `symbolic` have properties which prevent `define` from being used on them. To define `alg` to be a synonym for `algebraic`, use the more complicated construction

```
put('alg,'newnam,'algebraic);
```

6.7 RESETREDUCE Command

The command `resetreduce` works through the history of previous commands, and clears any values which have been assigned, plus any rules, arrays and the like. It also sets the various switches to their initial values. It is not complete, but does work for most things that cause a gradual loss of space.

Chapter 7

Built-in Prefix Operators

In the following subsections are descriptions of the most useful prefix operators built into REDUCE that are not defined in other sections (such as substitution operators). Some are fully defined internally as procedures; others are more nearly abstract operators, with only some of their properties known to the system.

In many cases, an operator is described by a prototypical header line as follows. Each formal parameter is given a name and followed by its allowed type. The names of classes referred to in the definition are printed in lower case, and parameter names in upper case. If a parameter type is not commonly used, it may be a specific set enclosed in brackets `{ ... }`. Operators that accept formal parameter lists of arbitrary length have the parameter and type class enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. Optional parameters and their type classes are enclosed in angle brackets.

7.1 Numerical Operators

REDUCE includes a number of functions that are analogs of those found in most numerical systems. With numerical arguments, such functions return the expected result. However, they may also be called with non-numerical arguments. In such cases, except where noted, the system attempts to simplify the expression as far as it can. In such cases, a residual expression involving the original operator usually remains. These operators are as follows:

7.1.1 ABS

`abs` returns the absolute value of its single argument, if that argument has a numerical value. A non-numerical argument is returned as an absolute value, with an

overall numerical coefficient taken outside the absolute value operator. For example:

```
abs(-3/4)      ->  3/4
abs(2a)        ->  2*abs(a)
abs(i)         ->  1
abs(-x)        ->  abs(x)
```

7.1.2 CEILING

This operator returns the ceiling (i.e., the least integer greater than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
ceiling(-5/4) ->  -1
ceiling(-a)   ->  ceiling( - a)
```

7.1.3 CONJ

This returns the complex conjugate of an expression, if that argument has a numerical value. By default the complex conjugate of a non-numerical argument is returned as an expression in the operators `repart` and `impart`. For example:

```
conj(1+i)      -> - i + 1
conj(a+i*b)    -> - impart(a)*i - impart(b)
                  + repart(a) - repart(b)*i
```

If rules have been previously defined for the complex conjugate(s) of one or more non-numerical terms appearing in the argument, these rules are applied and the expansion in terms of the operators `repart` and `impart` is suppressed.

For example:

```
realvalued a,b;
conj(a+i*b)    ->   a - b*i
let conj z => z!*, conj c => c!*;
conj(a+b*z*z!*+z*c!*)   ->   a + b*z*z* + c*z*
conj atan z    ->   atan(z*)
```

Note that in defining the rule `conj z => z!*`, the rule `conj z!* => z` is (in effect) automatically defined and *should not* be entered by the user. A more convenient method of associating two identifiers as mutual complex-conjugates is to use the `complex_conjugates` declaration as described in the section Declaring

Complex Conjugates.

The main use of rules for `conj` is to associate two identifiers as complex conjugates as in the examples above. In addition rules of the form `let conj(z)=>z, conj(w)=>-w` may be used. They imply that `z` is real-valued and `w` is purely imaginary, although the effect of the first rule can also be obtained by declaring `z` to be `realvalued`.

Rules of the form `let conj z => «some-expression»` may be used, but are not recommended. More useful results will usually be obtained by defining the equivalent rule `let z => conj(«some-expression»)`. Rules of the form `let conj z => «some-expression»` are particularly problematic if `«some-expression»` involves `z` itself as they may be inconsistent, for example `let conj z => z+1`. Even where they are consistent, better results may usually achieved by defining alternative rules. For example, given:

```
realvalued a,b;
let conj z => 2*a-z, conj w => w-2*b*i;
```

so that the real part of `z` is `a` and the imaginary part of `w` is `b`, more useful results will be obtained by defining the mathematically equivalent rules:

```
realvalued a,b,x,y;
let z => a +i*y, w => x + b*i;
```

Note also that the standard elementary functions and their inverses (where appropriate) are automatically defined to be `selfconjugate` so that `conj(f(z))` is simplified to `f(conj(z))`. User-defined operators may be declared to be self-conjugate with the declaration `selfconjugate`.

7.1.4 Conversion between degree and radians

These operators convert an angle in degrees to radians and vice-versa. `rad2deg` converts the radian value to an angle in degrees expressed as a single floating point value (according to the currently specified system precision). The value to be converted may be an integer, a rational or a floating value or indeed any expression that simplifies to a rounded value. In particular numerical constants such as π may be used in the input expression. `deg2rad` performs the inverse conversion.

`rad2dms` converts the radian value to an angle expressed in degrees, minutes and seconds returned as a three element list. The degree and minute values are integers the latter in the range $0 \ldots 59$ inclusive and the seconds value is a floating point value in the half-open interval $[0, 60.0)$. Similarly, `deg2dms` converts an angle given in degrees into a such a three element list.

The purpose of the operators `dms2rad` and `dms2deg` should also be obvious. The degree, minute and second value to be converted is passed to the conversion function as a three element list. There is considerable flexibility allowed in format of the list supplied as parameter – all three values may be integers, rational numbers or rounded values or any combination of these; the minute and second values need not lie between zero and sixty. The list supplied is simplified with the appropriate *carrys and borrows* performed (in effect at least) between the three values. For example

```
{60.5, 9.2, 11.234}       =>   {60, 39, 23.234}
{45, 0, -1}               =>   {44, 59, 59}
```

7.1.5 FACTORIAL

If the single argument of `factorial` evaluates to a non-negative integer, its factorial is returned. Otherwise an expression involving `factorial` is returned. For example:

```
factorial(5)   ->  120
factorial(a)   ->  factorial(a)
```

7.1.6 FIX

This operator returns the fixed value (i.e., the integer part of the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
fix(-5/4)    ->  -1
fix(a)       ->  fix(a)
```

7.1.7 FLOOR

This operator returns the floor (i.e., the greatest integer less than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
floor(-5/4)    ->  -2
floor(a)       ->  floor(a)
```

7.1.8 IMPART

This operator returns the imaginary part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `repart` and `impart`. For example:

```
impart(1+i)          -> 1
impart(sin(3+4*i)) -> cos(3)*sinh(4)
impart(log(2+i))    -> atan(1/2)
impart(asin(1+i))   -> acosh(sqrt(5)+2)/2
impart(a+i*b)        -> impart(a) + repart(b)
```

For the inverse trigometric and hyperbolic functions with non-numeric arguments the output is usually more compact when the `factor` is on.

7.1.9 LEGENDRE_SYMBOL

The operator `legendre_symbol`(a,p) denotes the Legendre symbol

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

which, by its very definition can only have one of the values $\{-1, 0, 1\}$.

7.1.10 MAX/MIN

`max` and `min` can take an arbitrary number of expressions as their arguments. If all arguments evaluate to numerical values, the maximum or minimum of the argument list is returned. If any argument is non-numeric, an appropriately reduced expression is returned. For example:

```
max(2,-3,4,5) ->  5
min(2,-2)     ->  -2.
max(a,2,3)    ->  max(A,3)
min(x)        ->  X
```

`max` or `min` of an empty list returns 0.

7.1.11 NEXTPRIME

`nextprime` returns the next prime greater than its integer argument, using a probabilistic algorithm. A type error occurs if the value of the argument is not an integer. For example:

```
nextprime(5)       ->  7
nextprime(-2)      ->  2
nextprime(-7)      -> -5
nextprime 1000000 -> 1000003
```

whereas `nextprime(a)` gives a type error.

7.1.12 RANDOM

`random(`*n*`)` returns a random number r in the range $0 \leq r < n$. A type error occurs if the value of the argument is not a positive integer in algebraic mode, or positive number in symbolic mode. For example:

```
random(5)          ->    3
random(1000)       ->  191
```

whereas `random(a)` gives a type error.

7.1.13 RANDOM_NEW_SEED

`random_new_seed(`*n*`)` reseeds the random number generator to a sequence determined by the integer argument n. It can be used to ensure that a repeatable pseudo-random sequence will be delivered regardless of any previous use of `random`, or can be called early in a run with an argument derived from something variable (such as the time of day) to arrange that different runs of a REDUCE program will use different random sequences. When a fresh copy of REDUCE is first created it is as if `random_new_seed(1)` has been obeyed.

A type error occurs if the value of the argument is not a positive integer.

7.1.14 REIMPART

This returns a two-element list of the real and imaginary parts of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `repart` and `impart`. This is more efficient than calling `repart` and `impart` separately particularly if its argument is complicated. For example:

```
reimpart(1+i)   -> {1,1}
reimpart(sin(3+4*i)) ->
      {cosh(4)*sin(3),cos(3)*sinh(4)}
reimpart(log(2+i))   ->
      {log(5)/2,atan(1/2)}
```

```
reimpart(asin(1+i))  ->
      {acos(sqrt(5)2)/2,acosh(sqrt(5)+2)/2}
reimpart(a+i*b)      ->
      { - impart(b) + repart(a),
        impart(a) + repart(b)}
```

For the inverse trigometric and hyperbolic functions with non-numeric arguments the output is usually more compact when the `FACTOR` is on.

7.1.15 REPART

This returns the real part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `repart` and `impart`. For example:

```
repart(1+i)   -> 1
repart(sin(3+4*i)) -> cosh(4)*sin(3)
repart(log(2+i))   -> log(5)/2
repart(asin(1+i))  -> acos(sqrt(5)-2)/2
repart(a+i*b)      ->  - impart(b) + repart(a)
```

For the inverse trigometric and hyperbolic functions with non-numeric arguments the output is usually more compact when the `FACTOR` is on.

7.1.16 ROUND

This operator returns the rounded value (i.e, the nearest integer) of its single argument if that argument has a numerical value. A non-numeric argument is returned as an expression in the original operator. For example:

```
round(-5/4)   ->  -1
round(a)      ->  round(a)
```

7.1.17 SIGN

`sign` tries to evaluate the sign of its argument. If this is possible `sign` returns one of 1, 0 or -1. Otherwise, the result is the original form or a simplified variant. For example:

```
sign(-5)      ->  -1
sign(-a^2*b)  ->  - sign(b)
```

Note that even powers of formal expressions are assumed to be positive only as long as the switch `complex` is off.

7.2 Mathematical Functions

REDUCE knows that the following represent mathematical functions that can take arbitrary scalar expressions as their argument(s).

7.2.1 Elementary Functions

Trigonometric, hyperbolic and exponential functions:

```
sin cos tan cot csc sec sinh
cosh tanh coth csch sech exp
```

The trigonometric functions listed above take arguments given in radians; degree-based versions are described below.

Inverse trigonometric, hyperbolic and exponential functions:

```
asin acos atan acot acsc asec
asinh acosh atanh acoth acsch asech
log log10 logb
```

where `log` is the natural logarithm, `log10` is the logarithm to base 10, and `logb` has two arguments of which the second is the logarithmic base. Note on the CSL GUI and other graphical interfaces the inverse trigonometric and hyperbolic functions are output as arcsin etc.

REDUCE defines the identifier `ln` to be an abstract operator with no properties; see section 2.7. You can use it any way you want. One way to use the identifier `ln` to represent the natural logarithm (for input only) is to execute the command

```
define ln = log;
```

The degree variants of the trigonometric functions and their inverses:

```
sind cosd tand cotd cscd secd
asind acosd atand acotd acscd asecd
```

The names of the degree-based functions are those of the normal trig functions with the letter d appended, for example `sind`, `cosd` and `tand` denote the sine, cosine and tangent repectively and their corresponding inverse functions are `asind`,

`acosd` and `atand`. The secant, cosecant and cotangent functions and their inverses are also supported and, indeed, are treated more as *first class* objects than their corresponding radian-based functions which are often converted to expressions involving sine and cosine by some of the standard REDUCE simplifications rules.

These degree-based functions are probably best regarded as functions defined for *real* values only, but complex arguments are supported for completeness. The numerical evaluation routines are fairly comprehensive for both real and complex arguments. However, few simplifications occur for trigd functions with complex arguments.

The range of the principal values returned by the inverse functions is consistent with those of the corresponding radian-valued functions. More precisely, for `asind`, `atand` and `acscd` the (closure of the) range is $[-90, 90]$ whilst for `acosd`, `acotd` and `asecd` the (closure of the) range is $[0, 180]$. In addition the operator `atan2d` is the degree valued version of the two argument inverse tangent function which returns an angle in the half-open interval $(-180, 180]$ in the correct quadrant depending on the signs of its two arguments. For $x > 0$, `atan2d(y, x)` returns the same numerical value as `atand(y/x)`. If $x = 0$ then ± 90 is returned depending on the sign of y.

Miscellaneous functions:

```
sqrt  hypot  atan2
norm  arg    argd
```

The function `hypot` takes two arguments x and y and returns the value $\sqrt{x^2 + y^2}$ but, when the switch `rounded` is ON, problems with rounding and possible overflow for large numerical arguments are reduced.

The function `atan2` also takes two arguments y and x respectively and returns a value of $\arctan(y/x)$ in the range $(-\pi, \pi]$ taking account of the signs of its two arguments and avoiding an error if $x = 0$.

The operator `norm` returns the modulus (or absolute value or norm) of a complex number when the switches `rounded` and `complex` are on. When the switches `rounded` and `complex` are both `on`, `arg` will return the argument in radians of the complex number supplied as its parameter — supplying zero as the parameter causes an error to be raised. With a real numerical value as parameter, it returns 0 or π when the value is positive or negative respectively. `argd` is similar to `arg`, but returns the argument expressed in degrees. Currently these are *purely numeric* operators; when `rounded` is `off` they basically return the input expression (perhaps with their parameter simplified). Example

```
1: on rounded;
```

```
2: {argd(-5), argd(1+i)};

{180.0,argd(i + 1)}

3: on complex;

*** Domain mode rounded changed to complex-rounded

4: {argd(1+i), argd(-1-i)};

{45.0, - 135.0}

5: {arg(3+4i), norm(3+4i)};

{0.927295218002,5.0}
```

REDUCE knows various elementary identities and properties of these functions. For example:

```
cos(-x) = cos(x)          sin(-x) = -sin(x)
cos(n*pi) = (-1)^n        sin(n*pi) = 0
log(e) = 1                e^(i*pi/2) = i
log(1) = 0                e^(i*pi) = -1
log(e^x) = x              e^(3*i*pi/2) = -i
sin(asin(x) = x           atan(0) = 0
atan2(0, -1) = pi         atan2(1, 0) = pi/2
```

The derivatives of all the elementary functions except `hypot` are also known to the system. Beside these identities, there are a lot of simplifications for elementary functions defined in REDUCE as rulelists. In order to view these, the SHOWRULES operator can be used, e.g.

```
showrules tan;

{tan(~n*arbint(~i)*pi + ~~x)

  => tan(x) when fixp(n),

 tan(~x) => trigquot(sin(x),cos(x))

  when knowledge_about(sin,x,tan),
```

```
      ~x + ~~k*pi
tan(-------------) =>
         ~~d

         x                  k
 - cot(--- + i*pi*impart(---))
         d                  d

                   k     1
 when abs(repart(---))=---,
                   d     2

      ~~w + ~~k*pi
tan(--------------) =>
         ~~d

      w      k                 k
tan(--- + (--- - fix(repart(---)))*pi) when ((
      d      d                 d

ratnump(rp) and abs(rp)>=1) where rp

            k
 => repart(---)),
            d

tan(atan(~x)) => x,

                              2
df(tan(~x),~x) => 1 + tan(x) }
```

For further simplification, especially of expressions involving trigonometric functions, see section 8.7.

Functions not listed above may be defined in the special functions package SPECFN.

The user can add further rules for the reduction of expressions involving these operators by using the `let` command.

In many cases it is desirable to expand product arguments of logarithms, or collect a sum of logarithms into a single logarithm. Since these are inverse operations, it is not possible to provide rules for doing both at the same time and preserve the REDUCE concept of idempotent evaluation. As an alternative, REDUCE provides two switches `expandlogs` and `combinelogs` to carry out these operations.

Both are off by default, and are subject to the value of the switch `precise`. This switch is on by default and prevents modifications that may be false in a complex domain. Thus to expand `log(3*y)` into a sum of logs, one can say

```
on expandlogs; log(3*y);
```

whereas to expand `log(x*y)` into a sum of logs, one needs to say

```
off precise; on expandlogs; log(x*y);
```

To combine this sum into a single log:

```
off precise; on combinelogs; log(x) + log(y);
```

These switches affect the logarithmic functions `log10` (base 10) and `logb` (arbitrary base) as well.

At the present time, it is possible to have both switches on at once, which could lead to infinite recursion. However, an expression is switched from one form to the other in this case. Users should not rely on this behavior, since it may change in the future.

The current version of REDUCE does a poor job of simplifying surds. In particular, expressions involving the product of variables raised to non-integer powers do not usually have their powers combined internally, even though they are printed as if those powers were combined. For example, the expression

```
x^(1/3)*x^(1/6);
```

will print as

```
sqrt(x)
```

but will have an internal form containing the two exponentiated terms. If you now subtract `sqrt(x)` from this expression, you will *not* get zero. Instead, the confusing form

```
sqrt(x) - sqrt(x)
```

will result. To combine such exponentiated terms, the switch `combineexpt` should be turned on.

The square root function can be input using the name `sqrt`, or the power operation `^(1/2)`. On output, unsimplified square roots are normally represented by the operator `sqrt` rather than a fractional power. With the default system switch settings, the argument of a square root is first simplified, and any divisors of the

expression that are perfect squares taken outside the square root argument. The remaining expression is left under the square root. Thus the expression

```
sqrt(-8a^2*b)
```

becomes

```
2*a*sqrt(-2*b).
```

Note that such simplifications can cause trouble if `A` is eventually given a value that is a negative number. If it is important that the positive property of the square root and higher even roots always be preserved, the switch `precise` should be set on (the default value). This causes any non-numerical factors taken out of surds to be represented by their absolute value form. With `precise` on then, the above example would become

```
2*abs(a)*sqrt(-2*b).
```

However, this is incorrect in the complex domain, where $\sqrt{x^2}$ is not identical to $|x|$. To avoid the above simplification, the switch `precise_complex` should be set on (default is off). For example:

```
on precise_complex; sqrt(-8a^2*b);
```

yields the output

```
              2
2*sqrt( - 2*a *b)
```

If the switch `rounded` is on, any of the elementary functions

```
acos acosd acosh acot acotd acoth acsc acscd
acsch asec asecd asech asin asind asinh atan
atan2 atanh atan2 atan2d cos cosd cosh cot
cotd coth csc cscd csch exp hypot log logb
log10 sec secd sech sin sind sinh sqrt tan
tand tanh
```

when given a numerical argument has its value calculated to the current degree of floating point precision. In addition, real (non-integer valued) powers of numbers will also be evaluated.

If the `complex` switch is turned on in addition to `rounded`, these functions will also calculate a real or complex result, again to the current degree of floating point precision, if given complex arguments. For example, with `on`

```
rounded,complex;

    2.3^(5.6i)   ->   - 0.0480793490914 - 0.998843519372*i
    cos(2+3i)    ->   - 4.18962569097 - 9.10922789376*i
```

For `log` and the inverse trigonometric and hyperbolic functions which are multi-valued, the principal value is returned. The branch cuts chosen (except for `acot` and `acotd`) are now those recommended by W. Kahan ([Kah87])

The exception for `acot` and `acotd` is necessary as elsewhere in REDUCE $\mathrm{acot}(-z)$ is taken to be $\pi - \mathrm{acot}(z)$ rather than $-\mathrm{acot}(z)$, and $\mathrm{acotd}(-z) = 180 - \mathrm{acotd}(z)$. The branch cuts are:

log, sqrt:	$\{r \mid r \in \mathbb{R} \wedge r < 0\}$
asin, asind, acos, acosd:	$\{r \mid r \in \mathbb{R} \wedge (r > 1 \vee r < -1)\}$
acsc, acscd, asec, asecd:	$\{r \mid r \in \mathbb{R} \wedge r \neq 0 \wedge r > -1 \wedge r < 1\}$
atan, atand, acot, acotd:	$\{r * i \mid r \in \mathbb{R} \wedge (r > 1 \vee r < -1)\}$
asinh:	$\{r * i \mid r \in \mathbb{R} \wedge (r \geq 1 \vee r \leq -1)\}$
acsch:	$\{r * i \mid r \in \mathbb{R} \wedge r \neq 0 \wedge r \geq -1 \wedge r \leq 1\}$
acosh:	$\{r \mid r \in \mathbb{R} \wedge r < 1\}$
asech:	$\{r \mid r \in \mathbb{R} \wedge (r > 1 \vee r < 0)\}$
atanh:	$\{r \mid r \in \mathbb{R} \wedge (r > 1 \vee r < -1)\}$
acoth:	$\{r \mid r \in \mathbb{R} \wedge r > -1 \wedge r < 1\}$

7.2.2 Special Functions

The functions in this section are either built-in or are autoloading functions from the package `SPECFN`. On the CSL GUI and other graphical interfaces many of the functions will be output in standard form; for example `BesselJ(nu,x)` will be output as $J_\nu(x)$ and `Fresnel_S(u)` as $\mathrm{S}(u)$. For most of the non-unary special functions in this section (`Lerch_Phi` is an exception), the last parameter is the 'main' variable and the earlier parameters are the order (or orders) usually rendered in the literature as subscipts and/or superscripts.

The information provided below is fairly rudimentary; more complete information may be found in the SPECFN package. Quick Reference Tables are also available.

Integral Functions:

```
Ei Li Si Ci Shi Chi Erf Fresnel_S Fresnel_C
```

All these functions are unary; the first six are the exponential, logarithmic, sine and cosine integrals and their hyperbolic counterparts. `Erf`, `Fresnel_S` and `Fresnel_C` are the error function and the Fresnel sine and cosine integrals respectively.

Beta, Gamma and Related Functions:

```
Beta ibeta Gamma iGamma psi Polygamma
```

The `Gamma` operator represents the Gamma function $\Gamma(x)$ when used with one argument; with two arguments it is the upper incomplete Gamma function $\Gamma(a, x)$. `Beta` is binary. The binary function `iGamma` and ternary function `iBeta` are the (normalised) lower incomplete Gamma ($\gamma(a, x)$) and Beta functions respectively. The unary function `psi` is sometimes known as the Digamma function and the binary function `Polygamma` with integer first parameter n is the nth derivative of the function `psi`.

Bessel and Related Functions:

```
BesselJ BesselY BesselI BesselK Hankel1 Hankel2
```

All of these functions are binary, their first argument being the order of the function.

For the special functions below, a second Quick Reference Table is available.

Airy Functions:

```
Airy_Ai Airy_Aiprime Airy_Bi Airy_Biprime
```

These are all unary functions.

Kummer, Lommel, Struve and Whittaker Functions:

```
KummerM KummerU Lommel1 Lommel2
StruveH StruveL WhittakerM WhittakerW
```

The Struve functions are both binary whilst the remaining ones are all ternary.

Riemann Zeta and Lambert's W Function:

```
zeta  Lambert_W
```

These are both unary functions.

Polylogarithms and Related Functions

```
dilog Polylog Lerch_Phi
```

These take one, two and three arguments respectively.

Associated Legendre functions:

```
SphericalHarmonicY SolidHarmonicY
```

These functions take four and six arguments respectively.

7.2.3 Polynomial Functions

The polynomial functions below are from the non-core package SPECFN and for the most part are not autoloading. This package needs to be loaded before they may be used with the command:

```
load_package specfn;
```

The names of the REDUCE operators for the polynomial functions below are mostly built by adding a P to the name of the polynomial, e.g. `EulerP` implements the Euler polynomials.

The information in this subsection is fairly rudimentary; more complete information may be found in the SPECFN package.
A Quick Reference Table is available for all the polynomial functions below.

Orthogonal Polynomials

Some well-known orthogonal polynomials are available:

- Hermite polynomials: (`HermiteP`);
- Chebyshev polynomials: (`ChebyshevT`, `ChebyshevU`);
- Legendre polynomials: (`LegendreP`);
- Laguerre polynomials: (`LaguerreP`);
- Associated Legendre functions: (`LegendreP`);
- Generalised Laguerre (or Sonin) polynomials: (`LaguerreP`);
- Gegenbauer polynomials: (`GegenbauerP`);
- Jacobi polynomials: (`JacobiP`).

The first three of these functions are binary and the first argument should be an integer specifying the order of the required polynomial. The functions `LegendreP` and `LaguerreP` may be used either as binary operators or ternary ones and represent the corresponding 'basic' and associated functions respectively. Finally the Gegenbauer polynomials are ternary whilst the Jacobi polynomials are quaternary.

Most definitions are equivalent to those in [AS72], except for the ternary associated Legendre functions:

$$P_n^{(m)}(x) = (-1)^m (1-x^2)^{m/2} \frac{\mathrm{d}^m P_n(x)}{\mathrm{d}x^m}\,.$$

These are sometimes mistakenly referred to as associated Legendre polynomials, but they are only polynomial when m is even.

Other Polynomial Functions

Fibonacci Polynomials are computed by the binary operator `FibonacciP`, where `FibonacciP(n,x)` returns the nth Fibonacci polynomial in the variable x. If n is an integer, it will be

evaluated using the recursive definition:

$$F_0(x) = 0; \qquad F_1(x) = 1; \qquad F_n(x) = xF_{n-1}(x) + F_{n-2}(x)\,.$$

Euler Polynomials are computed by the binary operator `EulerP`, where `EulerP(n,x)` returns the nth Euler polynomial in the variable x.

Bernoulli Polynomials are computed by the binary operator `BernoulliP`, where `BernoulliP(n,x)` returns the nth Bernoulli polynomial in the variable x.

7.2.4 Elliptic Functions and Integrals

All the functions documented in this subsection are autoloading functions from the package `ELLIPFN`. On the CSL GUI and other graphical interfaces these functions will be output in standard mathematical form; for example `jacobisn(x,k)` will be output as $\mathrm{sn}(x,k)$ and `weierstrass(x,omega1,omega3)` as $\wp(x,\omega_1,\omega_3)$.

Jacobi Elliptic Functions:

```
jacobisn jacobicn jacobidn
```

and their three reciprocals

```
jacobins jacobinc jacobind
```

and six quotients

```
jacobisc jacobisd jacobicd
jacobics jacobids jacobidc
```

All are binary functions with the second argument being the modulus. The binary function `jacobiam` is the amplitude.

Complete and Incomplete Elliptic Integrals of the First & Second Kinds:

```
ellipticK ellipticE ellipticF
jacobiE jacobiZeta
```

The function `ellipticE` may take one or two arguments to denote the complete and Legendre's form of the incomplete elliptic integrals of the second kind respectively. The complete integral of the first kind `ellipticK` is unary whilst `ellipticF`, `jacobiE` and `jacobiZeta` are binary and represent the incomplete integral of the first kind, Jacobi's form of the incomplete elliptic integral of the second kind and Jacobi's Zeta function respectively.

Jacobi's Theta Functions:

```
elliptictheta1 elliptictheta2
elliptictheta3 elliptictheta4
```

are all binary functions with the second argument being the 'parameter' τ, the nome q being given by $q = \exp(i\pi\tau)$

Weierstrassian Elliptic Functions:

```
weierstrass     weierstrassZeta
weierstrass_sigma  weierstrass_sigma1
weierstrass_sigma2   weierstrass_sigma3
weierstrass1   weierstrassZeta1
```

are all ternary functions with the second and third arguments of the first six functions being the the lattice period parameters ω_1 and ω_3. The remaining two functions are alternative versions of the Weierstrass functions with the second and third arguments being the lattice invariants g_2 and g_3.

Inverse Elliptic Functions:

```
arcsn  arccn  arcdn   arcns  arcnc  arcnd
arcsc  arccs  arcsd   arcds  arccd  arcdc
```

These are all binary functions with the second argument being the modulus k. They are the inverses of the corresponding Jacobi elliptic functions `jacobisn`, `jacobicn` etc.(wrt their first argument).

For the elliptic functions above a Quick Reference Table is available.

7.3 Combinatorial Numbers

Binomial coefficients are provided by the binary operator `Binomial`. The value of `Binomial(n, m)`, where n and m are non-negative integers with $n \geq m$, is the number of ways of choosing m items from a set of n distinct items as well, of course, as being the coefficient of x^m in the expansion of $(1 + x)^n$.

The function call `Binomial(n,m)`, where n and m are non-negative integers, will return the expected integer value (from Pascal's triangle). For other real numerical values the result will usually involve the Γ function, but if the switch `rounded` is ON the Γ functions will be evaluated numerically. This will also be the case for complex numerical arguments if the switch `complex` is ON. For non-numeric arguments the result returned will involve the original operator `binomial`, or its pretty-printed equivalent on graphical interfaces.

Stirling numbers of the first and second kind are computed by the binary operators `Stirling1` and `Stirling2` respectively using explicit formulae.
`Stirling1(n, k)` is $(-1)^{n-k}$ $\times$ *(the number of permutations of the set* $\{1, 2, \ldots, n\}$ *into exactly* k *cycles).*
`Stirling2(n, k)` is *the number of partitions of the set* $\{1, 2, \ldots, n\}$ *into exactly* k *non-empty subsets.*
Here n and k should be non-negative integers with $n \geq k$.

For integer arguments an integer result will be returned, otherwise a result involving the original operator will be returned. Note on graphical user interfaces `Stirling1(n,m)` and `Stirling2(n,m)` are rendered as s_n^m and S_n^m respectively.

Stirling numbers are implemented in the non-core package SPECFN and are not currently autoloading. Before use this package should be loaded with the command:

```
load_package specfn;
```

For more information see here.

A Motzkin number M_n (named after Theodore Motzkin) is the number of different ways of drawing non-intersecting chords on a circle between n points. For a non-negative integer n, the operator `Motzkin(n)` returns the nth Motzkin number, according to the recursion formula

$$M_0 = 1; \qquad M_1 = 1; \qquad M_{n+1} = \frac{2n+3}{n+3} M_n + \frac{3n}{n+3} M_{n-1}\,.$$

The recursion is, of course, optimised as a simple loop to avoid repeated computation of lower-order numbers.

For the functions in this and the section below a Quick Reference Table is available. It also includes a list of reserved constants known to REDUCE.

7.4 Bernoulli, Euler and Fibonacci Numbers

Bernoulli numbers are provided by the unary operator `Bernoulli`. If n is a non-negative integer, the call `Bernoulli(n)` evaluates to the nth Bernoulli number;

all of the odd Bernoulli numbers, except `Bernoulli(1)`, are zero. Otherwise the result involves the original operator `Bernoulli`; on graphical interfaces this is rendered as B_n.

Euler numbers are computed by the unary operator `Euler`. If n is a non-negative integer, the call `Euler(n)` returns the nth Euler number; all of the odd Euler numbers are zero. Otherwise the result returned involves the original operator `Euler`; on graphical interfaces this is rendered as E_n.

Fibonacci numbers are provided by the unary operator `Fibonacci`, where `Fibonacci(n)` evaluates to the nth Fibonacci number; if n is an integer, this will be evaluated following the recursive definition:

$$F_0 = 0; \qquad F_1 = 1; \qquad F_n = F_{n-1} + F_{n-2}\,.$$

The recursion is, of course, optimised as a simple loop to avoid repeated computation of lower-order numbers. Otherwise the result returned involves the original operator `fibonacci`; on graphical interfaces this is rendered as F_n.

7.5 CHANGEVAR Operator

Author: G. Üçoluk.

The operator `changevar` does a variable transformation in a set of differential equations. Syntax:

`changevar(`⟨*depvars*⟩, ⟨*newvars*⟩, ⟨*eqlist*⟩, ⟨*diffeq*⟩`)`

⟨*diffeq*⟩ is either a single differential equation or a list of differential equations, ⟨*depvars*⟩ are the dependent variables to be substituted, ⟨*newvars*⟩ are the new depend variables, and ⟨*eqlist*⟩ is a list of equations of the form ⟨*depvar*⟩ = ⟨*expression*⟩ where ⟨*expression*⟩ is some function in the new dependent variables.

The three lists ⟨*depvars*⟩, ⟨*newvars*⟩, and ⟨*eqlist*⟩ must be of the same length. If there is only one variable to be substituted, then it can be given instead of the list. The same applies to the list of differential equations, i.e., the following two commands are equivalent

```
changevar(u,y,x=e^y,df(u(x),x) - log(x));

changevar({u},{y},{x=e^y},{df(u(x),x) - log(x)});
```

except for one difference: the first command returns the transformed differential equation, the second one a list with a single element.

The switch `dispjacobian` governs the display the entries of the inverse Jacobian, it is `off` per default.

The mathematics behind the change of independent variable(s) in differential equations is quite straightforward. It is basically the application of the chain rule. If the dependent variable of the differential equation is F, the independent variables are x_i and the new independent variables are u_i (where i=1...n) then the first derivatives are:

$$\frac{\partial F}{\partial x_i} = \frac{\partial F}{\partial u_j}\frac{\partial u_j}{\partial x_i}$$

We assumed Einstein's summation convention. Here the problem is to calculate the $\partial u_j/\partial x_i$ terms if the change of variables is given by

$$x_i = f_i(u_1, \ldots, u_n)$$

The first thought might be solving the above given equations for u_j and then differentiating them with respect to x_i, then again making use of the equations above, substituting new variables for the old ones in the calculated derivatives. This is not always a preferable way to proceed. Mainly because the functions f_i may not always be easily invertible. Another approach that makes use of the Jacobian is better. Consider the above given equations which relate the old variables to the new ones. Let us differentiate them:

$$\frac{\partial x_j}{\partial x_i} = \frac{\partial f_j}{\partial x_i}$$
$$\delta_{ij} = \frac{\partial f_j}{\partial u_k}\frac{\partial u_k}{\partial x_i}$$

The first derivative is nothing but the (j, k) th entry of the Jacobian matrix.

So if we speak in matrix language

$$\mathbf{1} = \mathbf{J} \cdot \mathbf{D}$$

where we defined the Jacobian

$$\mathbf{J}_{ij} \stackrel{\triangle}{=} \frac{\partial f_i}{\partial u_j}$$

and the matrix of the derivatives we wanted to obtain as

$$\mathbf{D}_{ij} \stackrel{\triangle}{=} \frac{\partial u_i}{\partial x_j}.$$

If the Jacobian has a non-vanishing determinant then it is invertible and we are able to write from the matrix equation above:

$$\mathbf{D} = \mathbf{J}^{-\mathbf{1}}$$

so finally we have what we want

$$\frac{\partial u_i}{\partial x_j} = \left[\mathbf{J}^{-\mathbf{1}}\right]_{ij}$$

The higher derivatives are obtained by the successive application of the chain rule and using the definitions of the old variables in terms of the new ones. It can be easily verified that the only derivatives that are needed to be calculated are the first order ones which are obtained above.

7.5.1 CHANGEVAR example: The 2-dim. Laplace Equation

The 2-dimensional Laplace equation in cartesian coordinates is:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Now assume we want to obtain the polar coordinate form of Laplace equation. The change of variables is:

$$x = r\cos\theta, \qquad y = r\sin\theta$$

The solution using `changevar` is as follows

```
changevar({u},{r,theta},{x=r*cos theta,y=r*sin theta},
          {df(u(x,y),x,2)+df(u(x,y),y,2)} );
```

Here we could omit the curly braces in the first and last arguments (because those lists have only one member) and the curly braces in the third argument (because they are optional), but you cannot leave off the curly braces in the second argument. So one could equivalently write

```
changevar(u,{r,theta},x=r*cos theta,y=r*sin theta,
          df(u(x,y),x,2)+df(u(x,y),y,2) );
```

If you have tried out the above example, you will notice that the denominator contains a $\cos^2\theta + \sin^2\theta$ which is actually equal to 1. This has of course nothing to do with `changevar`. One has to be overcome these pattern matching problems by the conventional methods REDUCE provides (a rule, for example, will fix it).

Secondly you will notice that your `u(x,y)` operator has changed to `u(r,theta)` in the result. Nothing magical about this. That is just what we do with pencil and paper. `u(r,theta)` represents the the transformed dependent variable.

7.5.2 Another CHANGEVAR example: An Euler Equation

Consider a differential equation which is of Euler type, for instance:

$$x^3 y''' - 3x^2 y'' + 6xy' - 6y = 0$$

where prime denotes differentiation with respect to x. As is well known, Euler type of equations are solved by a change of variable:

$$x = e^u$$

So our call to `changevar` reads as follows:

```
changevar(y, u, x=e**u, x**3*df(y(x),x,3)-
          3*x**2*df(y(x),x,2)+6*x*df(y(x),x)-6*y(x));
```

and returns the result

```
df(y(u),u,3) - 6*df(y(u),u,2) + 11*df(y(u),u) - 6*y(u)
```

7.6 CONTINUED_FRACTION Operator

The operator `continued_fraction` generates the continued fraction expansion of a rational number argument. For irrational or rounded arguments, it approximates the real number as a rational number to the current system precision and generates the continued fraction expansion. Currently the operator `cf` is a complete synonym for `continued_fraction` although this may change in future updates of the package RATAPRX.

The operator `continued_fraction` accepts one, two or three arguments: the number to be expanded; an **optional** maximum size permitted for the denominator of the convergent and an **optional** number of continuents to be generated:

`continued_fraction(`⟨*num*⟩`)`
`continued_fraction(`⟨*num*⟩`,` ⟨*size*⟩`)`
`continued_fraction(`⟨*num*⟩`,` ⟨*size*⟩`,` ⟨*numterms*⟩`)`

The result is the special operator `contfrac` with three arguments: the original number to be expanded ⟨*num*⟩, secondly the rational number approximation (the final convergent) and thirdly a list of continuents of the continued fraction (i.e. a list of pairs of partial numerators and denominators)

```
{t0, {1, t1}, {1, t2}, ....  }
```

which represents the same value according to the definition

```
t0 + 1/(t1 + 1/(t2 + ...)).
```

Note that, although with the current algorithm all the partial numerators have the value 1, they are stored in the list of continuents. This is for compatibility with the output of other continued fractions functions `cfrac` and `cf_euler`. This

facilitates pretty-printing and the implementation of various equivalence transformations all of which are documented in the continued fraction subsection of the `rataprx` manual (Section 20.48.2).

Precision: the second optional parameter ⟨*size*⟩ is an upper bound for the absolute value of the denominator of the convergent.
Number of terms: the third optional parameter ⟨*numterms*⟩ is the maximum number of terms (continuents) to be generated.
If both optional parameters omitted, the expansion performed is exact for rational number arguments and for irrational or rounded arguments it is up to the current system precision. If both optional parameters are given the expansion is halted when the desired precision is reached or when the specified maximum number of terms have been generated whichever is the sooner. If the size parameter is zero, its value is ignored. Thus to obtain a continued fraction expansion to, for example, 10 terms one would specify the ⟨*size*⟩ parameter to be 0 and the ⟨*numterms*⟩ parameter to be 10.

Note that the operator `contfrac` is not normally seen as the output is pretty-printed, unless the number of continuents generated is larger than 12.

Examples:

```
continued_fraction(6/11);
```

```
  6                 1
{----,exact,---------------}
  11                  1
             1 + ---------
                       1
                  1 + ---
                       5
```

```
continued_fraction(pi,1000);
```

```
     355              1
{pi,-----,3 + -----------------}
     113                1
                7 + ----------
                           1
                     15 + ---
                           1
```

```
continued_fraction(pi,0,6);
```

```
     104348                    1
```

```
{pi,--------,3 + ------------------------------}
     33215                          1
                  7 + ------------------------
                                        1
                       15 + -----------------
                                          1
                             1 + -----------
                                           1
                                  292 + ---
                                          1
```

```
continued_fraction(pi,1000,3);

     333          1
{pi,-----,3 + ----------}
     106            1
                7 + ----
                    15
```

```
continued_fraction(pi,1000,6);

     355             1
{pi,-----,3 + ----------------}
     113               1
                7 + ----------
                           1
                    15 + ---
                           1
```

```
continued_fraction e;

{e,

  13580623
 ----------,
  4996032

 {2, {1,1}, {1,2}, {1,1}, {1,1}, {1,4}, {1,1}, {1,1},
  {1,6}, {1,1}, {1,1}, {1,8}, {1,1}, {1,1}, {1,10},
  {1,1}, {1,1}, {1,12}}}
```

7.7 DF Operator

The operator `df` is used to represent partial differentiation with respect to one or more variables. It is used with the syntax:

`df(`⟨*exprn:algebraic*⟩[, ⟨*var:kernel*⟩<, ⟨*num:integer*⟩>]) : *algebraic*.

The first argument is the expression to be differentiated. The remaining arguments specify the differentiation variables and the number of times they are applied.

The number `num` may be omitted if it is 1. For example,

`df(y,x)`	$= \partial y/\partial x$
`df(y,x,2)`	$= \partial^2 y/\partial x^2$
`df(y,x1,2,x2,x3,2)`	$= \partial^5 y/\partial x_1^2 \partial x_2 \partial x_3^2$.

The evaluation of `df(y,x)` proceeds as follows: first, the values of `y` and `x` are found. Let us assume that `x` has no assigned value, so its value is `x`. Each term or other part of the value of `y` that contains the variable `x` is differentiated by the standard rules. If `z` is another variable, not `x` itself, then its derivative with respect to `x` is taken to be 0, unless `z` has previously been declared to `depend` on `x`, in which case the derivative is reported as the symbol `df(z,x)`.

7.7.1 Switches influencing differentiation

Consider `df(u,x,y,z)`, assuming `u` depends on each of `x,y,z` in some way. If none of `x,y,z` is equal to `u` then the order of differentiation is commuted into a canonical form, unless the switch `nocommutedf` is turned on (default is off). If at least one of `x,y,z` is equal to `u` then the order of differentiation is *not* fully commuted and the derivative is *not* simplified to zero, unless the switch `commutedf` is turned on. It is off by default.

If `commutedf` is off and the switch `simpnoncomdf` is on then simplify as follows:

```
df(u,x,u)          ->  df(u,x,2) / df(u,x)
df(u,x,n,u)        ->  df(u,x,n+1) / df(u,x)
```

provided `u` depends only on the one variable `x`. This simplification removes the non-commutative aspect of the derivative.

If the switch `expanddf` is turned on then REDUCE uses the chain rule to expand symbolic derivatives of indirectly dependent variables provided the result is unambiguous, i.e. provided there is no direct dependence. It is off by default. Thus, for example, given

```
depend f,u,v; depend {u,v},x;
on expanddf;
df(f,x)     -> df(f,u)*df(u,x) + df(f,v)*df(v,x)
```

whereas after

```
depend f,x;
```

`df(f,x)` does not expand at all (since the result would be ambiguous and the algorithm would loop).

Turning on the switch `allowdfint` allows "differentiation under the integral sign", i.e.

```
df(int(y, x), v) -> int(df(y, v), x)
```

if this results in a simplification. If the switch `dfint` is also turned on then this happens regardless of whether the result simplifies. Both switches are off by default.

7.7.2 Adding Differentiation Rules

The `let` statement can be used to introduce rules for differentiation of user-defined operators. Its general form is

`for all` ⟨*var1*⟩`, ...,` ⟨*varn*⟩
`let df(`⟨*operator*⟩⟨*varlist*⟩`,` ⟨*vari*⟩`) =` ⟨*expression*⟩

where

⟨*varlist*⟩ $\longrightarrow$ (⟨*var1*⟩, ..., ⟨*varn*⟩),

and ⟨*var1*⟩, ..., ⟨*varn*⟩ are the dummy variable arguments of ⟨*operator*⟩.

An analogous form applies to infix operators.

Examples:

```
for all x let df(tan x,x) = 1 + tan(x)^2;
```

(This is how the tan differentiation rule appears in the REDUCE source.)

```
for all x,y let df(f(x,y),x)=2*f(x,y),
               df(f(x,y),y)=x*f(x,y);
```

Notice that all dummy arguments of the relevant operator must be declared arbitrary by the `for all` command, and that rules may be supplied for operators with any number of arguments. If no differentiation rule appears for an argument in an operator, the differentiation routines will return as result an expression in terms of `df`. For example, if the rule for the differentiation with respect to the second argument of `f` is not supplied, the evaluation of `df(f(x,z),z)` would leave this expression unchanged. (No `depend` declaration is needed here, since `f(x,z)` obviously "depends on" `z`.)

Once such a rule has been defined for a given operator, any future differentiation rules for that operator must be defined with the same number of arguments for that operator, otherwise we get the error message

```
Incompatible DF rule argument length for <operator>
```

7.7.3 Options controlling display of derivatives

If the switch `dfprint` is turned on (it is off by default) then derivatives are displayed using subscripts, as illustrated below. In graphical environments with typeset mathematics enabled, the (shared) variable `fancy_print_df` can be set to one of the values `partial`, `total` or `indexed` to control the display of derivatives. The default value is `partial`. However, if the switch `dfprint` is on then `fancy_print_df` is ignored. For example, with the following settings, derivatives are displayed as follows (assuming `depend f,x,y` and `operator g`):

Setting	`df(f,x,2,y)`	`df(g(x,y),x,2,y)`
`fancy_print_df:=partial`	$\dfrac{\partial^3 f}{\partial x^2 \partial y}$	$\dfrac{\partial^3 g(x,y)}{\partial x^2 \partial y}$
`fancy_print_df:=total`	$\dfrac{d^3 f}{dx^2 dy}$	$\dfrac{d^3 g(x,y)}{dx^2 dy}$
`fancy_print_df:=indexed`	$f_{x,x,y}$	$g(x,y)_{x,x,y}$
`on dfprint`	$f_{x,x,y}$	$g_{x,x,y}$

7.8 INT Operator

`int` is an operator in REDUCE for indefinite or definite integration.

7.8.1 Indefinite integration

Indefinite integration is performed using a combination of the Risch-Norman algorithm and pattern matching [NM77, Har79, ND79]. It is used with the syntax:

`int(`⟨*exprn:algebraic*⟩`,` ⟨*var:kernel*⟩`)` : *algebraic*.

This will return correctly the indefinite integral for expressions comprising polynomials, log functions, exponential functions and tan and atan. The arbitrary constant is not represented. If the integral cannot be done in closed terms, it returns a formal integral for the answer in one of two ways:

1. It returns the input, `int(...,...)` unchanged.
2. It returns an expression involving `int`s of some other functions (sometimes more complicated than the original one, unfortunately).

Rational functions can be integrated when the denominator is factorizable by the program. In addition it will attempt to integrate expressions involving error functions, dilogarithms and other trigonometric expressions. In these cases it might not always succeed in finding the solution, even if one exists.

Examples:

```
int(log(x),x) ->  x*(log(x) - 1),
int(e^x,x)    ->  e**x.
```

The program checks that the second argument is a variable and gives an error if it is not.

7.8.2 Definite Integration

If `int` is used with the syntax

```
INT(EXPRN:algebraic,VAR:kernel,
     LOWER:algebraic,UPPER:algebraic):algebraic.
```

The definite integral of `exprn` with respect to `var` is calculated between the limits `lower` and `upper`. This is calculated by several methods that are tried one after the other: pattern matching, by first finding the indefinite integral and then substituting the limits into this, by contour integration for some types integrands with polynomial denominator, or by transforming the integrand into one or two Meijer G-functions. For details, see the documentation on the DEFINT package described in section D.1[1].

[1]This code was written by Kerry Gaskell, Stanley M. Kameny, Winfried Neun.

7.8.3 Options

The switch `trint` when on will trace the operation of the algorithm. It produces a great deal of output in a somewhat illegible form, and is not of much interest to the general user. It is normally off.

The switch `trintsubst` when on will trace the heuristic attempts to solve the integral by substitution. It is normally off.

The switch `trdefint` when on will trace the operation of the definite integration algorithm.

If the switch `failhard` is on the algorithm will terminate with an error if the integral cannot be done in closed terms, rather than return a formal integration form. `failhard` is normally off.

The switch `nolnr` suppresses the use of the linear properties of integration in cases when the integral cannot be found in closed terms. It is normally off.

The switch `nointsubst` disables the heuristic attempts to solve the integral by substitution. It is normally off.

7.8.4 Advanced Use

If a function appears in the integrand that is not one of the functions `exp, Erf, tan, atan, log, dilog` then the algorithm will make an attempt to integrate the argument if it can, differentiate it and reach a known function. However the answer cannot be guaranteed in this case. If a function is known to be algebraically independent of this set it can be flagged transcendental by

```
flag('(trilog),'transcendental);
```

in which case this function will be added to the permitted field descriptors for a genuine decision procedure. If this is done the user is responsible for the mathematical correctness of his actions.

The standard version does not deal with algebraic extensions. Thus integration of expressions involving square roots and other like things can lead to trouble. The extension package `ALGINT` will analytically integrate a wide range of expressions involving square roots where the answer exists in that class of functions. It is an implementation of the work described by J.H. Davenport [Dav81].

The extension package is loaded automatically when the switch `algint` is turned on. One enters an expression for integration, as with the regular integrator, for example:

```
int(sqrt(x+sqrt(x**2+1))/x,x);
```

If one later wishes to integrate expressions without using the facilities of this package, the switch `algint` should be turned off.

The switches supported by the standard integrator (e.g., `trint`) are also supported by this package. In addition, the switch `tra`, if on, will give further tracing information about the specific functioning of the algebraic integrator.

7.9 LENGTH Operator

`length` is a generic operator for finding the length of various objects in the system. The meaning depends on the type of the object. In particular, the length of an algebraic expression is the number of additive top-level terms its expanded representation.

Examples:

```
length(a+b)     ->  2
length(2)       ->  1.
```

Other objects that support a length operator include arrays, lists and matrices. The explicit meaning in these cases is included in the description of these objects.

7.10 LIMIT Operator

LIMITS is a fast limit package for REDUCE for functions which are continuous except for computable poles and singularities, written by Stanley L. Kameny, based on some earlier work by Ian Cohen and John P. Fitch. The Truncated Power Series package is used for non-critical points, at which the value of the function is the constant term in the expansion around that point. l'Hôpital's rule is used in critical cases, with preprocessing of $\infty - \infty$ forms and reformatting of product forms in order to apply l'Hôpital's rule. A limited amount of bounded arithmetic is also employed where applicable.

The standard way of calling limit, applying all of the methods, is

`limit(`⟨*exprn:algebraic*⟩, ⟨*var:kernel*⟩, ⟨*limpoint:algebraic*⟩`)` : *algebraic*

The result is the limit of exprn as var approaches limpoint. To compute the of $\sin(x)/x$ at the point 0, enter

```
limit(sin(x)/x,x,0);
```

```
1
```

If the limit depends upon the direction of approach to the `limpoint`, the onesided limit functions `limit!+` and `limit!-` may be used:

`limit!+(`⟨*exprn:algebraic*⟩, ⟨*var:kernel*⟩, ⟨*limpoint:algebraic*⟩`)` : *algebraic*
`limit!-(`⟨*exprn:algebraic*⟩, ⟨*var:kernel*⟩, ⟨*limpoint:algebraic*⟩`)` : *algebraic*

they are defined by:

`limit!+ (limit!-)` (exp,var,limpoint) →`limit`(exp*,ϵ,0),
exp*=sub(var=var+(-)ϵ^2,exp)

for example,

```
limit!+(sqrt x/sin x,x,0);

infinity;
```

7.11 MAP Operator

The `map` operator applies a uniform evaluation pattern to all members of a composite structure: a matrix, a list, or the arguments of an operator expression. The evaluation pattern can be a unary procedure, an operator, or an algebraic expression with one free variable.

It is used with the syntax:

```
map(fnc:function,obj:object)
```

Here `obj` is a list, a matrix or an operator expression. `fnc` can be one of the following:

1. the name of an operator with a single argument: the operator is evaluated once with each element of `obj` as its single argument;
2. an algebraic expression with exactly one free variable, i.e. a variable preceded by the tilde symbol. The expression is evaluated for each element of `obj`, with the element substituted for the free variable;
3. a replacement rule of the form `var => rep` where `var` is a variable (a kernel without a subscript) and `rep` is an expression that contains `var`. The replacement expression `rep` is evaluated for each element of `obj` with the element substituted for `var`. The variable `var` may be optionally preceded by a tilde.

The rule form for `fnc` is needed when more than one free variable occurs.

Examples:

```
map(abs,{1,-2,a,-a})  ->  {1,2,ABS(A),ABS(A)}
map(int(~w,x), mat((x^2,x^5),(x^4,x^5))) ->

        [  3     6 ]
        [ x     x  ]
        [----  ----]
        [ 3     6  ]
        [          ]
        [  5     6 ]
        [ x     x  ]
        [----  ----]
        [ 5     6  ]

map(~w*6, x^2/3 = y^3/2 -1) -> 2*X^2=3*(Y^3-2)
```

You can use `map` in nested expressions. However, you cannot apply `map` to a non-composite object, e.g. an identifier or a number.

7.12 MKID Operator

In many applications, it is useful to create a set of identifiers for naming objects in a consistent manner. In most cases, it is sufficient to create such names from two components. The operator `mkid` is provided for this purpose. Its syntax is:

```
mkid(u:id,v:id|non-negative integer):id
```

for example

```
mkid(a,3)       -> a3
mkid(apple,s)   -> apples
```

while `mkid(a+b,2)` gives an error.

The `set` statement can be used to give a value to the identifiers created by `mkid`, for example

```
set(mkid(a,3),3);
```

will give `a3` the value 2. Similarly, the `unset` statement can be used to remove the value from these identifiers, for example

```
unset(mkid(a,3));
```

7.13 The Pochhammer Notation

The Pochhammer notation $(a)_k$ (also called Pochhammer's symbol) is supported by the binary operator `Pochhammer(a,k)`. For a non-negative integer k, it is defined as (http://dlmf.nist.gov/5.2.iii)

$$(a)_0 = 1,$$
$$(a)_k = a(a+1)(a+2)\cdots(a+k-1).$$

For $a \neq 0, -1, -2, -3, \ldots$, this is equivalent to

$$(a)_k = \frac{\Gamma(a+k)}{\Gamma(a)}.$$

When n is integral, the defining product is expanded (assuming the switch `exp` is on). With `rounded` off, this expression is evaluated numerically if a is numerical and k is integral, and otherwise may be simplified where appropriate. The simplification rules are based upon algorithms supplied by Wolfram Koepf [Koe92].

The Pochhammer symbol is used quite extensively in the simplification and numerical evaluation of special functions.

7.14 PF Operator

`pf(`⟨*exp*⟩`,`⟨*var*⟩`)` transforms the expression ⟨*exp*⟩ into a list of partial fractions with respect to the main variable, ⟨*var*⟩. `pf` does a complete partial fraction decomposition, and as the algorithms used are fairly unsophisticated (factorization and the extended Euclidean algorithm), the code may be unacceptably slow in complicated cases.

Example: Given `2/((x+1)^2*(x+2))` in the workspace, `pf(ws,x);` gives the result

```
     2        - 2           2
{-------,-------,--------------}  .
  x + 2    x + 1     2
                    x  + 2*x + 1
```

If you want the denominators in factored form, set the switch `exp` to off. Thus, with `2/((x+1)^2*(x+2))` in the workspace, the input `off exp; pf(ws,x);` gives the result

```
      2        - 2         2
{-------,-------,----------} .
  x + 2    x + 1           2
                    (x + 1)
```

To recombine the terms, `for each ... sum` can be used. So with the above list in the workspace, `for each j in ws sum j;` returns the result

```
          2
------------------
                   2
 (x + 2)*(x + 1)
```

Alternatively, one can use the operations on lists to extract any desired term.

7.15 RESIDUE and POLEORDER Operators

The residue $\operatorname*{Res}_{z=a} f(z)$ of a function $f(z)$ at the point $a \in \mathbb{C}$ is defined as

$$\operatorname*{Res}_{z=a} f(z) = \frac{1}{2\pi i} \oint f(z)\, dz \ ,$$

with integration along a closed curve around $z = a$ with winding number 1.

If $f(z)$ is given by a Laurent series expansion at $z = a$

$$f(z) = \sum_{k=-\infty}^{\infty} a_k \, (z-a)^k \ ,$$

then

$$\operatorname*{Res}_{z=a} f(z) = a_{-1} \ . \tag{7.1}$$

If $a = \infty$, one defines on the other hand

$$\operatorname*{Res}_{z=\infty} f(z) = -a_{-1} \tag{7.2}$$

for given Laurent representation

$$f(z) = \sum_{k=-\infty}^{\infty} a_k \, \frac{1}{z^k} \ .$$

The operator `residue(f,z,a)` determines the residue of f at the point $z = a$ if f is meromorphic at $z = a$. The calculation of residues at essential singularities of f is not supported, as are the residues of factorial terms.[2]

[2] This code was written by Wolfram Koepf.

`poleorder(f,z,a)` determines the pole order of f at the point $z = a$ if f is meromorphic at $z = a$.

Note that both functions use the operator `taylor` in connection with representations (7.1)–(7.2).

Here are some examples:

```
2: residue(x/(x^2-2),x,sqrt(2));

 1
---
 2

3: poleorder(x/(x^2-2),x,sqrt(2));

1

4: residue(sin(x)/(x^2-2),x,sqrt(2));

 sqrt(2)*sin(sqrt(2))
----------------------
          4

5: poleorder(sin(x)/(x^2-2),x,sqrt(2));

1

6: residue(1/(x-1)^m/(x-2)^2,x,2);

 - m

7: poleorder(1/(x-1)/(x-2)^2,x,2);

2

8: residue(sin(x)/x^2,x,0);

1

9: poleorder(sin(x)/x^2,x,0);

1

10: residue((1+x^2)/(1-x^2),x,1);

-1

11: poleorder((1+x^2)/(1-x^2),x,1);
```

```
1

12: residue((1+x^2)/(1-x^2),x,-1);

1

13: poleorder((1+x^2)/(1-x^2),x,-1);

1

14: residue(tan(x),x,pi/2);

-1

15: poleorder(tan(x),x,pi/2);

1

16: residue((x^n-y^n)/(x-y),x,y);

0

17: poleorder((x^n-y^n)/(x-y),x,y);

0

18: residue((x^n-y^n)/(x-y)^2,x,y);

  n
 y *n
------
  y

19: poleorder((x^n-y^n)/(x-y)^2,x,y);

1

20: residue(tan(x)/sec(x-pi/2)+1/cos(x),x,pi/2);

-2

21: poleorder(tan(x)/sec(x-pi/2)+1/cos(x),x,pi/2);

1

22: for k:=1:2 sum residue((a+b*x+c*x^2)/(d+e*x+f*x^2),x,
    part(part(solve(d+e*x+f*x^2,x),k),2));

 b*f - c*e
```

```
-----------
     2
    f

23: residue(x^3/sin(1/x)^2,x,infinity);

  - 1
------
  15

24: residue(x^3*sin(1/x)^2,x,infinity);

-1

25: residue(gamma(x),x,-1);

-1

26: residue(psi(x),x,-1);

-1

27: on fullroots;

28: for k:=1:3 sum
28: residue((a+b*x+c*x^2+d*x^3)/(e+f*x+g*x^2+h*x^3),x,
28: part(part(solve(e+f*x+g*x^2+h*x^3,x),k),2));

0
```

7.16 SELECT Operator

The `select` operator extracts from a list, or from the arguments of an n–ary operator, elements corresponding to a boolean predicate. It is used with the syntax:

`select` (⟨*fnc:function*⟩, ⟨*lst:list*⟩)

`fnc` can be one of the following forms:

1. the name of an operator with a single argument: the operator is evaluated once on each element of `lst`;
2. an algebraic expression with exactly one free variable, i.e. a variable preceded by the tilde symbol. The expression is evaluated for each element of ⟨*lst*⟩, with the element substituted for the free variable;

3. a replacement rule of the form ⟨*var*⟩ => ⟨*rep*⟩ where ⟨*var*⟩ is a variable (a kernel without subscript) and ⟨*rep*⟩ is an expression that contains ⟨*var*⟩. ⟨*rep*⟩ is evaluated for each element of `LST` with the element substituted for ⟨*var*⟩. ⟨*var*⟩ may be optionally preceded by a tilde.

The rule form for `fnc` is needed when more than one free variable occurs.

The result of evaluating `fnc` is interpreted as a boolean value corresponding to the conventions of REDUCE. These values are composed with the leading operator of the input expression.

Examples:

```
select( ~w>0 , {1,-1,2,-3,3}) -> {1,2,3}
select(evenp deg(~w,y),part((x+y)^5,0):=list)
       -> {X^5 ,10*X^3*Y^2 ,5*X*Y^4}
select(evenp deg(~w,x),2x^2+3x^3+4x^4)
       -> 4X^4 + 2X^2
```

7.17 SOLVE Operator

`solve` is an operator for solving one or more simultaneous algebraic equations. It is used with the syntax:

`solve`(⟨*exprn:algebraic*⟩[, ⟨*var:kernel*⟩ | , ⟨*varlist:list of kernels*⟩]) : *list*.

`exprn` is of the form ⟨*expression*⟩ or `{` ⟨*expression1*⟩,⟨*expression2*⟩, ... `}`. Each expression is an algebraic equation, or is the difference of the two sides of the equation. The second argument is either a kernel or a list of kernels representing the unknowns in the system. This argument may be omitted if the number of distinct, non-constant, top-level kernels equals the number of unknowns, in which case these kernels are presumed to be the unknowns and a message is printed:

```
solve(x^2 - 1);

->

Unknown: x

{x=1,x=-1}
```

For one equation, `solve` recursively uses factorization and decomposition, together with the known inverses of `log`, `sin`, `cos`, `^`, `acos`, `asin`, and linear, quadratic, cubic, quartic, or binomial factors. Solutions of equations built with exponentials or logarithms are often expressed in terms of Lambert's W function. This function is (partially) implemented in the special functions package.

Linear equations are solved by the multi-step elimination method due to Bareiss, unless the switch `cramer` is on, in which case Cramer's method is used. The Bareiss method is usually more efficient unless the system is large and dense.

Non-linear equations are solved using the Groebner basis package (chapter 20.26). Users should note that this can be quite a time consuming process.

Examples:

```
solve(log(sin(x+3))^5 = 8,x);
solve(a*log(sin(x+3))^5 - b, sin(x+3));
solve({a*x+y=3,y=-2},{x,y});
```

`solve` returns a list of solutions. If there is one unknown, each solution is an equation for the unknown. If a complete solution was found, the unknown will appear by itself on the left-hand side of the equation. On the other hand, if the solve package could not find a solution, the "solution" will be an equation for the

unknown in terms of the operator `root_of`. If there are several unknowns, each solution will be a list of equations for the unknowns. For example,

```
solve(x^2=1,x);              -> {x=-1,x=1}

solve(x^7-x^6+x^2=1,x)
                           6
       -> {x=root_of(x_  + x_ + 1,x_,tag_1),x=1}

solve({x+3y=7,y-x=1},{x,y}) -> {{x=1,y=2}}.
```

The TAG argument is used to uniquely identify those particular solutions. Solution multiplicities are stored in the global variable `root_multiplicities` rather than the solution list. The value of this variable is a list of the multiplicities of the solutions for the last call of `solve`. For example,

```
solve(x^2=2x-1,x); root_multiplicities;
```

gives the results

```
{x=1}

{2}
```

If you want the multiplicities explicitly displayed, the switch `multiplicities` can be turned on. For example

```
on multiplicities; solve(x^2=2x-1,x);
```

yields the result

```
{x=1,x=1}
```

7.17.1 Handling of Undetermined Solutions

When `solve` cannot find a solution to an equation, it normally returns an equation for the relevant indeterminates in terms of the operator `ROOT_OF`. For example, the expression

```
solve(cos(x) + log(x),x);
```

returns the result

```
{x=root_of(cos(x_) + log(x_),x_,tag_1)} .
```

An expression with a top-level `root_of` operator is implicitly a list with an unknown number of elements (since we don't always know how many solutions an equation has). If a substitution is made into such an expression, closed form solutions can emerge. If this occurs, the `root_of` construct is replaced by an operator `one_of`. At this point it is of course possible to transform the result of the original `solve` operator expression into a standard `solve` solution. To effect this, the operator `expand_cases` can be used.

The following example shows the use of these facilities:

```
solve(-a*x^3+a*x^2+x^4-x^3-4*x^2+4,x);
                2      3
{x=root_of(a*x_  - x_  + 4*x_ + 4,x_,tag_2),x=1}

sub(a=-1,ws);

{x=one_of({2,-1,-2},tag_2),x=1}

expand_cases ws;

{x=2,x=-1,x=-2,x=1}
```

7.17.2 Solutions of Equations Involving Cubics and Quartics

Since roots of cubics and quartics can often be very messy, a switch `fullroots` is available, that, when off (the default), will prevent the production of a result in closed form. The `root_of` construct will be used in this case instead.

In constructing the solutions of cubics and quartics, trigonometrical forms are used where appropriate. This option is under the control of a switch `trigform`, which is normally on.

The following example illustrates the use of these facilities:

```
let xx = solve(x^3+x+1,x);

xx;
              3
{x=root_of(x_  + x_ + 1,x_)}

on fullroots;

xx;

                                - sqrt(31)*i
```

```
                    atan(---------------)
                                3*sqrt(3)
{x=(i*(sqrt(3)*sin(----------------------)
                                  3

                      - sqrt(31)*i
                 atan(---------------)
                         3*sqrt(3)
         - cos(----------------------)))/sqrt(3),
                        3

                               - sqrt(31)*i
                          atan(---------------)
                                  3*sqrt(3)
 x=( - i*(sqrt(3)*sin(----------------------)
                                    3

                            - sqrt(31)*i
                       atan(---------------)
                               3*sqrt(3)
            + cos(----------------------)))/sqrt(
                              3

      3),

                    - sqrt(31)*i
             atan(---------------)
                     3*sqrt(3)
    2*cos(----------------------)*i
                       3
 x=-----------------------------------}
                  sqrt(3)

off trigform;

xx;
                                2/3
{x=( - (sqrt(31) - 3*sqrt(3))    *sqrt(3)*i

                                2/3    2/3
     - (sqrt(31) - 3*sqrt(3))     - 2    *sqrt(3)*i

        2/3                               1/3  1/3
     + 2    )/(2*(sqrt(31) - 3*sqrt(3))    *6
```

```
                1/6
              *3   ),

                              2/3
x=((sqrt(31) - 3*sqrt(3))    *sqrt(3)*i

                                 2/3    2/3
    - (sqrt(31) - 3*sqrt(3))    + 2   *sqrt(3)*i

       2/3                                1/3  1/3
    + 2   )/(2*(sqrt(31) - 3*sqrt(3))   *6

                1/6
              *3   ),

                               2/3    2/3
     (sqrt(31) - 3*sqrt(3))    - 2
x=-------------------------------------}
                              1/3  1/3  1/6
    (sqrt(31) - 3*sqrt(3))   *6   *3
```

7.17.3 Other Options

If `solvesingular` is on (the default setting), degenerate systems such as `x+y=0`, `2x+2y=0` will be solved by introducing appropriate arbitrary constants. The consistent singular equation 0=0 or equations involving functions with multiple inverses may introduce unique new indeterminant kernels `arbcomplex(j)`, or `arbint(j)`, (*j*=1,2,...), representing arbitrary complex or integer numbers respectively. To automatically select the principal branches, do `off allbranch`. To avoid the introduction of new indeterminant kernels do `off arbvars` – then no equations are generated for the free variables and their original names are used to express the solution forms. To suppress solutions of consistent singular equations do `off solvesingular`.

To incorporate additional inverse functions do, for example:

```
put('sinh,'inverse,'asinh);
put('asinh,'inverse,'sinh);
```

together with any desired simplification rules such as

```
for all x let sinh(asinh(x))=x, asinh(sinh(x))=x;
```

For completeness, functions with non-unique inverses should be treated as `^`, `sin`, and `cos` are in the `solve` module source.

Arguments of `asin` and `acos` are not checked to ensure that the absolute value of the real part does not exceed 1; and arguments of `log` are not checked to ensure that the absolute value of the imaginary part does not exceed π; but checks (perhaps involving user response for non-numerical arguments) could be introduced using `let` statements for these operators.

7.17.4 Parameters and Variable Dependency

The proper design of a variable sequence supplied as a second argument to `solve` is important for the structure of the solution of an equation system. Any unknown in the system not in this list is considered totally free. E.g. the call

```
solve({x=2*z,z=2*y},{z});
```

produces an empty list as a result because there is no function $z = z(x, y)$ which fulfills both equations for arbitrary x and y values. In such a case the share variable `requirements` displays a set of restrictions for the parameters of the system:

```
requirements;

{x - 4*y}
```

The non-existence of a formal solution is caused by a contradiction which disappears only if the parameters of the initial system are set such that all members of the requirements list take the value zero. For a linear system the set is complete: a solution of the requirements list makes the initial system solvable. E.g. in the above case a substitution $x = 4y$ makes the equation set consistent. For a non-linear system only one inconsistency is detected. If such a system has more than one inconsistency, you must reduce them one after the other. [3] The set shows you also the dependency among the parameters: here one of x and y is free and a formal solution of the system can be computed by adding it to the variable list of `solve`. The requirement set is not unique – there may be other such sets.

A system with parameters may have a formal solution, e.g.

```
solve({x=a*z+1,0=b*z-y},{z,x});

      y      a*y + b
```

[3] The difference between linear and non–linear inconsistent systems is based on the algorithms which produce this information as a side effect when attempting to find a formal solution; example: solve($\{x = a, x = b, y = c, y = d\}, \{x, y\}$) gives a set $\{a - b, c - d\}$ while $solve(\{x^2 = a, x^2 = b, y^2 = c, y^2 = d\}, \{x, y\}$ leads to $\{a - b\}$.

```
{{z=---,x=----------}}
     b         b
```

which is not valid for all possible values of the parameters. The variable `assumptions` contains then a list of restrictions: the solutions are valid only as long as none of these expressions vanishes. Any zero of one of them represents a special case that is not covered by the formal solution. In the above case the value is

```
assumptions;

{b}
```

which excludes formally the case $b = 0$; obviously this special parameter value makes the system singular. The set of assumptions is complete for both, linear and non–linear systems.

`solve` rearranges the variable sequence to reduce the (expected) computing time. This behavior is controlled by the switch `varopt`, which is on by default. If it is turned off, the supplied variable sequence is used or the system kernel ordering is taken if the variable list is omitted. The effect is demonstrated by an example:

```
s:= {y^3+3x=0,x^2+y^2=1};

solve(s,{y,x});

                 6       2
{{y=root_of(y_  + 9*y_  - 9,y_),

          3
       - y
  x=-------}}
        3

off varopt; solve(s,{y,x});

                 6       4         2
{{x=root_of(x_  - 3*x_  + 12*x_  - 1,x_),

              4       2
     x*( - x  + 2*x  - 10)
  y=----------------------}}
                 3
```

In the first case, `solve` forms the solution as a set of pairs $(y_i, x(y_i))$ because the degree of x is higher – such a rearrangement makes the internal computation of the Gröbner basis generally faster. For the second case the explicitly given variable sequence is used such that the solution has now the form $(x_i, y(x_i))$. Controlling the variable sequence is especially important if the system has one or more free variables. As an alternative to turning off `varopt`, a partial dependency among the variables can be declared using the `depend` statement: `solve` then rearranges the variable sequence but keeps any variable ahead of those on which it depends.

```
on varopt;
s:={a^3+b,b^2+c}$
solve(s,{a,b,c});

                          3       6
{{a=arbcomplex(1),b= - a ,c= - a }}

depend a,c; depend b,c; solve(s,{a,b,c});

{{c=arbcomplex(2),

              6
  a=root_of(a_  + c,a_),

        3
  b= - a }}
```

Here `solve` is forced to put c after a and after b, but there is no obstacle to interchanging a and b.

7.18 Support for Solving Inequalities

The operator `ineq_solve` tries to solves single inequalities and sets of coupled inequalities[4]. The following types of systems are supported:

- only numeric coefficients (no parametric system),
- a linear system of mixed equations and $\leq, \geq$ inequalities, applying the method of Fourier and Motzkin, as described by G. B. Dantzig in [Dan63],
- a univariate inequality with $\leq$, $\geq$, $<$ or $>$ operator and polynomial or rational left–hand and right–hand sides, or a system of such inequalities with only one variable.

[4]This code was written by Herbert Melenk.

For linear optimization problems please use the operator `simplex` of the LINALG package (cf. section 20.33).

Syntax:

```
ineq_solve(⟨expr⟩ [,⟨vl⟩])
```

where ⟨*expr*⟩ is an inequality or a list of coupled inequalities and equations, and the optional argument ⟨*vl*⟩ is a single variable (kernel) or a list of variables (kernels). If not specified, they are extracted automatically from ⟨*expr*⟩. For multivariate input an explicit variable list specifies the elimination sequence: the last member is the most specific one.

An error message occurs if the input cannot be processed by the currently implemented algorithms.

The result is a list. It is empty if the system has no feasible solution. Otherwise the result presents the admissible ranges as set of equations where each variable is equated to one expression or to an interval. The most specific variable is the first one in the result list and each form contains only preceding variables (resolved form). The interval limits can be formal **max** or **min** expressions. Algebraic numbers are encoded as rounded number approximations.

Examples:

```
ineq_solve({(2*x^2+x-1)/(x-1) >=  (x+1/2)^2, x>0});

{x=(0 .. 0.326583),x=(1 .. 2.56777)}

 reg:=
 {a + b - c>=0, a - b + c>=0, - a + b + c>=0, 0>=0,
  2>=0, 2*c - 2>=0, a - b + c>=0, a + b - c>=0,
   - a + b + c - 2>=0, 2>=0, 0>=0, 2*b - 2>=0,
  k + 1>=0, - a - b - c + k>=0,
   - a - b - c + k + 2>=0, - 2*b + k>=0,
   - 2*c + k>=0, a + b + c - k>=0,
  2*b + 2*c - k - 2>=0, a + b + c - k>=0}$

ineq_solve (reg,{k,a,b,c});

{c=(1 .. infinity),

 b=(1 .. infinity),

 a=(max( - b + c,b - c) .. b + c - 2),
```

```
k=a + b + c}
```

7.19 Finding Rational or Integer Zeros

The operators `r_solve` and `i_solve` compute the exact rational zeros of a single univariate polynomial using fast modular methods. The algorithm used is that described by R. Loos ([Loo83]). The operator `r_solve` computes all rational zeros whereas the operator `i_solve` computes only integer zeros in a way that is slightly more efficient than extracting them from the rational zeros. The `r_solve` and `i_solve` interfaces are almost identical, and are intended to be completely compatible with that of the general `r_solve` operator, although `r_solve` and `i_solve` give more convenient output when only rational or integer zeros respectively are required. The current implementation appears to be faster than `solve` by a factor that depends on the example, but is typically up to about 2.[5]

Extension to compute Gaussian integer and rational zeros and zeros of polynomial systems is planned.

7.19.1 The user interface

The first argument is required and must simplify to either a univariate polynomial expression or equation with integer, rational or rounded coefficients. Symbolic coefficients are not allowed (and currently complex coefficients are not allowed either.) The argument is simplified to a quotient of integer polynomials and the denominator is silently ignored.

`ARBRAT` Subsequent arguments are optional. If the polynomial variable is to be specified then it must be the first optional argument, and if the first optional argument is not a valid option (see below) then it is (mis-)interpreted as the polynomial variable. However, since the variable in a non-constant univariate polynomial can be deduced from the polynomial it is unnecessary to specify it separately, except in the degenerate case that the first argument simplifies to either 0 or $0 = 0$. In this case the result is returned by `i_solve` in terms of the operator `arbint` and by `r_solve` in terms of the (new) analogous operator `arbrat`. The operator `i_solve` will generally run slightly faster than `r_solve`.

The (rational or integer) zeros of the first argument are returned as a list and the default output format is the same as that used by `solve`. Each distinct zero is returned in the form of an equation with the variable on the left and the multiplicities of the zeros are assigned to the variable `root_multiplicities` as a list. However, if the switch `multiplicities` is turned on then each zero

[5] This code was written by Francis J. Wright.

is explicitly included in the solution list the appropriate number of times (and `root_multiplicities` has no value).

Optional keyword arguments acting as local switches allow other output formats. They have the following meanings:

`separate`: assign the multiplicity list to the global variable `root_multiplicities` (the default);

`expand` or **`multiplicities`:** expand the solution list to include multiple zeros multiple times (the default if the `multiplicities` switch is on);

`together`: return each solution as a list whose second element is the multiplicity;

`nomul`: do not compute multiplicities (thereby saving some time);

`noeqs`: do not return univariate zeros as equations but just as values.

7.19.2 Examples

```
r_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\left\{x = \frac{-4}{3}, x = 3, x = -3, x = \frac{4}{3}\right\}$$

```
i_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\{x = 3, x = -3\}$$

See the test/demonstration file `rsolve.tst` for more examples.

7.19.3 Tracing

The switch `trsolve` turns on tracing of the algorithm. It is off by default.

7.20 Modular Solve and Roots

The operators (m_solve) and (m_roots) are for modular polynomials and modular polynomial systems.[6] The moduli need not be primes. `m_solve` requires a modulus to be set. `m_roots` takes the modulus as a second argument. For example:

[6] This code was written by Herbert Melenk.

```
on modular; setmod 8;
m_solve(2x=4);             ->  {{X=2},{X=6}}
m_solve({x^2-y^3=3});
    ->  {{X=0,Y=5}, {X=2,Y=1}, {X=4,Y=5}, {X=6,Y=1}}
m_solve({x=2,x^2-y^3=3}); ->  {{X=2,Y=1}}
off modular;
m_roots(x^2-1,8);          ->  {1,3,5,7}
m_roots(x^3-x,7);          ->  {0,1,6}
```

7.21 Even and Odd Operators

An operator can be declared to be *even* or *odd* in its first argument by the declarations `even` and `odd` respectively. Expressions involving an operator declared in this manner are transformed if the first argument contains a minus sign. Any other arguments are not affected. In addition, if say `f` is declared odd, then `f(0)` is replaced by zero unless `f` is also declared *non zero* by the declaration `nonzero`. For example, the declarations

```
even f1; odd f2;
```

mean that

```
f1(-a)     ->    f1(a)
f2(-a)     ->   -f2(a)
f1(-a,-b)  ->    f1(a,-b)
f2(0)      ->    0.
```

To inhibit the last transformation, say `nonzero f2;`.

7.22 Linear Operators

An operator can be declared to be linear in its first argument over powers of its second argument. If an operator `f` is so declared, `f` of any sum is broken up into sums of `f`s, and any factors that are not powers of the variable are taken outside. This means that `f` must have (at least) two arguments. In addition, the second argument must be an identifier (or more generally a kernel), not an expression.

Example:

If `F` were declared linear, then

```
f(a*x^5+b*x+c,x) ->
```

```
      5
f(x ,x)*a + f(x,x)*b + f(1,x)*c
```

More precisely, not only will the variable and its powers remain within the scope of the `f` operator, but so will any variable and its powers that had been declared to `depend` on the prescribed variable; and so would any expression that contains that variable or a dependent variable on any level, e.g. `cos(sin(x))`.

To declare operators `f` and `g` to be linear operators, use:

```
linear f,g;
```

The analysis is done of the first argument with respect to the second; any other arguments are ignored. It uses the following rules of evaluation:

`f(0)`	$\longrightarrow$	`0`	
`f(-y,x)`	$\longrightarrow$	`-f(y,x)`	
`f(y+z,x)`	$\longrightarrow$	`f(y,x)+f(z,x)`	
`f(y*z,x)`	$\longrightarrow$	`z*f(y,x)`	if z does not depend on x
`f(y/z,x)`	$\longrightarrow$	`f(y,x)/z`	if z does not depend on x

To summarize, `y` "depends" on the indeterminate `x` in the above if either of the following hold:

1. `y` is an expression that contains `x` at any level as a variable, e.g., `cos(sin(x))`
2. Any variable in the expression `y` has been declared dependent on `x` by use of the declaration `depend`.

The use of such linear operators can be seen in the paper [FH74] which contains a complete listing of a program for definite integration of some expressions that arise in fourth order quantum electrodynamics.

7.23 Non-Commuting Operators

An operator can be declared to be non-commutative under multiplication by the declaration `noncom`.

Example:

After the declaration

```
noncom u,v;
```

the expressions `u(x)*u(y)-u(y)*u(x)` and `u(x)*v(y)-v(y)*u(x)` will remain unchanged on simplification, and in particular will not simplify to zero.

Note that it is the operators (`u` and `v` in the above example) and not the variable that have the non-commutative property.

The `let` statement may be used to introduce rules of evaluation for such operators. In particular, the boolean operator `ordp` is useful for introducing an ordering on such expressions.

Example:

The rule

```
for all x,y such that x neq y and ordp(x,y)
   let u(x)*u(y)= u(y)*u(x)+comm(x,y);
```

would introduce the commutator of `u(x)` and `u(y)` for all `x` and `y`. Note that since `ordp(x,x)` is *true*, the equality check is necessary in the degenerate case to avoid a circular loop in the rule.

7.24 Symmetric and Antisymmetric Operators

An operator can be declared to be symmetric with respect to its arguments by the declaration `symmetric`. For example

```
symmetric u,v;
```

means that any expression involving the top level operators `u` or `v` will have its arguments reordered to conform to the internal order used by REDUCE. The user can change this order for kernels by the command `korder`.

For example, `u(x,v(1,2))` would become `u(v(2,1),x)`, since numbers are ordered in decreasing order, and expressions are ordered in decreasing order of complexity.

Similarly the declaration `antisymmetric` declares an operator antisymmetric. For example,

```
antisymmetric l,m;
```

means that any expression involving the top level operators `l` or `m` will have its arguments reordered to conform to the internal order of the system, and the sign of the expression changed if there are an odd number of argument interchanges necessary to bring about the new order.

For example, `l(x,m(1,2))` would become `-l(-m(2,1),x)` since one inter-

change occurs with each operator. An expression like `l(x,x)` would also be replaced by 0.

7.25 Declaring New Prefix Operators

The user may add new prefix operators to the system by using the declaration `operator`. For example:

```
operator h,g1,arctan;
```

adds the prefix operators `h`, `g1` and `arctan` to the system.

This allows symbols like `h(w), h(x,y,z), g1(p+q), arctan(u/v)` to be used in expressions, but no meaning or properties of the operator are implied. The same operator symbol can be used equally well as a 0-, 1-, 2-, 3-, etc.-place operator.

To give a meaning to an operator symbol, or express some of its properties, `let` statements can be used, or the operator can be given a definition as a procedure.

If the user forgets to declare an identifier as an operator, the system will prompt the user to do so in interactive mode, or do it automatically in non-interactive mode. A diagnostic message will also be printed if an identifier is declared `operator` more than once.

Operators once declared are global in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the operator to that block, nor does the operator go away on exiting the block (use `clear` instead for this purpose).

An operator declared `print_indexed` has its arguments displayed as indices, e.g. after `print_indexed a;` the operator value `a(i,2)` is displayed as $a_{i,2}$. You can declare several operators together to be indexed, e.g.

```
print_indexed b, c;
```

and remove indexed declarations using `print_noindexed`.

7.26 Declaring New Infix Operators

Users can add new infix operators by using the `infix` and `precedence` declarations. For example,

```
infix mm;
```

```
precedence mm,-;
```

The declaration `infix mm;` would allow one to use the symbol `mm` as an infix operator:

`a mm b` instead of `mm(a,b).`

The declaration `precedence mm,-;` says that `mm` should be inserted into the infix operator precedence list just *after* the − operator. This gives it higher precedence than − and lower precedence than * . Thus

`a - b mm c - d` means `a - (b mm c) - d,`

while

`a * b mm c * d` means `(a * b) mm (c * d).`

Both infix and prefix operators have no transformation properties unless `let` statements or procedure declarations are used to assign a meaning.

We should note here that infix operators so defined are always binary:

`a mm b mm c` means `(a mm b) mm c.`

7.27 Creating / Removing Variable Dependency

There are several facilities in REDUCE, such as the differentiation operator and the linear operator facility, that can utilize knowledge of the dependency between various variables, or kernels. Such dependency may be expressed by the command `depend`. This takes an arbitrary number of arguments and sets up a dependency of the first argument on the remaining arguments. For example,

```
depend x,y,z;
```

says that `x` is dependent on both `y` and `z`.

```
depend z,cos(x),y;
```

says that `z` is dependent on `cos(x)` and `y`.

Dependencies introduced by `depend` can be removed by `nodepend`. The arguments of this are the same as for `depend`. For example, given the above dependencies,

```
nodepend z,cos(x);
```

says that `z` is no longer dependent on `cos(x)`, although it remains dependent on `y`.

As a convenience, one or more dependent variables can be specified together in a list for both the `depend` and `nodepend` commands, i.e.

`(no)depend {`y_1`, `y_2`, ...}, `x_1`, `x_2`, ...`

is equivalent to

`(no)depend `y_1`, `x_1`, `x_2`, ...; (no)depend `y_2`, `x_1`, `x_2`, ...; ...`

Both commands also accept a sequence of "dependence sequences", where the beginning of each new dependence sequence is indicated by a list of one or more dependent variables. For example,

```
depend {x,y,z},u,v,{theta},time;
```

is equivalent to

```
depend x,u,v;
depend y,u,v;
depend z,u,v;
depend theta,time;
```

Chapter 8

Display and Structuring of Expressions

In this section, we consider a variety of commands and operators that permit the user to obtain various parts of algebraic expressions and also display their structure in a variety of forms. Also presented are some additional concepts in the REDUCE design that help the user gain a better understanding of the structure of the system.

8.1 Kernels

REDUCE is designed so that each operator in the system has an evaluation (or simplification) function associated with it that transforms the expression into an internal canonical form. This form, which bears little resemblance to the original expression, is described in detail in [Hea71].

The evaluation function may transform its arguments in one of two alternative ways. First, it may convert the expression into other operators in the system, leaving no functions of the original operator for further manipulation. This is in a sense true of the evaluation functions associated with the operators `+`, `*` and `/` , for example, because the canonical form does not include these operators explicitly. It is also true of an operator such as the determinant operator `det` because the relevant evaluation function calculates the appropriate determinant, and the operator `det` no longer appears. On the other hand, the evaluation process may leave some residual functions of the relevant operator. For example, with the operator `cos`, a residual expression like `cos(x)` may remain after evaluation unless a rule for the reduction of cosines into exponentials, for example, were introduced. These residual functions of an operator are termed *kernels* and are stored uniquely like variables. Subsequently, the kernel is carried through the calculation as a variable unless transformations are introduced for the operator at a later stage.

In those cases where the evaluation process leaves an operator expression with non-trivial arguments, the form of the argument can vary depending on the state of the system at the point of evaluation. Such arguments are normally produced in expanded form with no terms factored or grouped in any way. For example, the expression `cos(2*x+2*y)` will normally be returned in the same form. If the argument `2*x+2*y` were evaluated at the top level, however, it would be printed as `2*(X+Y)`. If it is desirable to have the arguments themselves in a similar form, the switch `intstr` (for "internal structure"), if on, will cause this to happen.

In cases where the arguments of the kernel operators may be reordered, the system puts them in a canonical order, based on an internal intrinsic ordering of the variables. However, some commands allow arguments in the form of kernels, and the user has no way of telling what internal order the system will assign to these arguments. To resolve this difficulty, we introduce the notion of a *kernel form* as an expression that transforms to a kernel on evaluation.

Examples of kernel forms are:

```
a
cos(x*y)
log(sin(x))
```

whereas

```
a*b
(a+b)^4
```

are not.

We see that kernel forms can usually be used as generalized variables, and most algebraic properties associated with variables may also be associated with kernels.

8.2 The Expression Workspace

Several mechanisms are available for saving and retrieving previously evaluated expressions. The simplest of these refers to the last algebraic expression simplified. When an assignment of an algebraic expression is made, or an expression is evaluated at the top level, (i.e., not inside a compound statement or procedure) the results of the evaluation are automatically saved in a variable `ws` that we shall refer to as the workspace. (More precisely, the expression is assigned to the variable `ws` that is then available for further manipulation.)

Example:

If we evaluate the expression `(x+y)^2` at the top level and next wish to differen-

tiate it with respect to `y`, we can simply say

```
df(ws,y);
```

to get the desired answer.

If the user wishes to assign the workspace to a variable or expression for later use, the `saveas` statement can be used. It has the syntax

`saveas` ⟨*expression*⟩

For example, after the differentiation in the last example, the workspace holds the expression `2*x+2*y`. If we wish to assign this to the variable `z` we can now say

```
saveas z;
```

If the user wishes to save the expression in a form that allows him to use some of its variables as arbitrary parameters, the `for all` command can be used.

Example:

```
for all x saveas h(x);
```

with the above expression would mean that `h(z)` evaluates to `2*y+2*z`.

A further method for referencing more than the last expression is described in chapter 13 on interactive use of REDUCE.

8.3 Output of Expressions

A considerable degree of flexibility is available in REDUCE in the printing of expressions generated during calculations. No explicit format statements are supplied, as these are in most cases of little use in algebraic calculations, where the size of output or its composition is not generally known in advance. Instead, REDUCE provides a series of mode options to the user that should enable him to produce his output in a comprehensible and possibly pleasing form.

The most extreme option offered is to suppress the output entirely from any top level evaluation. This is accomplished by turning off the switch `output` which is normally on. It is useful for limiting output when loading large files or producing "clean" output from the prettyprint programs.

In most circumstances, however, we wish to view the output, so we need to know how to format it appropriately. As we mentioned earlier, an algebraic expression is normally printed in an expanded form, filling the whole output line with terms.

Certain output declarations, however, can be used to affect this format. To begin with, we look at an operator for changing the length of the output line.

8.3.1 LINELENGTH Operator

This operator is used with the syntax

```
linelength(num:integer):integer
```

and sets the output line length to the integer `num`. It returns the previous output line length (so that it can be stored for later resetting of the output line if needed).

8.3.2 Output Declarations

We now describe a number of switches and declarations that are available for controlling output formats. It should be noted, however, that the transformation of large expressions to produce these varied output formats can take a lot of computing time and space. If a user wishes to speed up the printing of the output in such cases, he can turn off the switch `pri`. If this is done, then output is produced in one fixed format, which basically reflects the internal form of the expression, and none of the options below apply. `pri` is normally on.

With `pri` on, the output declarations and switches available are as follows:

ORDER Declaration

The declaration `order` may be used to order variables on output. The syntax is:

```
order v1,...vn;
```

where the `vi` are kernels. Thus,

```
order x,y,z;
```

orders `x` ahead of `y`, `y` ahead of `z` and all three ahead of other variables not given an order. `order nil;` resets the output order to the system default. The order of variables may be changed by further calls of `order`, but then the reordered variables would have an order lower than those in earlier `order` calls. Thus,

```
order x,y,z;
order y,x;
```

would order `z` ahead of `y` and `x`. The default ordering is usually alphabetic.

FACTOR Declaration

This declaration takes a list of identifiers or kernels as argument. `factor` is not a factoring command (use `factorize` or the `factor` switch for this purpose); rather it is a separation command. All terms involving fixed powers of the declared expressions are printed as a product of the fixed powers and a sum of the rest of the terms.

For example, after the declaration

```
factor x;
```

the polynomial $(x + y + 1)^2$ will be printed as

```
 2                    2
x  + 2*x*(y + 1) + y  + 2*y + 1
```

All expressions involving a given prefix operator may also be factored by putting the operator name in the list of factored identifiers. For example:

```
factor x,cos,sin(x);
```

causes all powers of `x` and `sin(x)` and all functions of `cos` to be factored.

Note that `factor` does not affect the order of its arguments. You should also use `order` if this is important.

The declaration `remfac v1,...,vn;` removes the factoring flag from the expressions `v1` through `vn`.

8.3.3 Output Control Switches

In addition to these declarations, the form of the output can be modified by switching various output control switches using the declarations `on` and `off`. We shall illustrate the use of these switches by an example, namely the printing of the expression

```
x^2*(y^2+2*y)+x*(y^2+z)/(2*a) .
```

The relevant switches are as follows:

ALLFAC Switch

This switch will cause the system to search the whole expression, or any subexpression enclosed in parentheses, for simple multiplicative factors and print them

outside the parentheses. Thus our expression with `allfac` off will print as

```
     2  2         2           2
(2*x *y *a + 4*x *y*a + x*y  + x*z)/(2*a)
```

and with `allfac` on as

```
          2                  2
x*(2*x*y *a + 4*x*y*a + y  + z)/(2*a) .
```

`allfac` is normally on, and is on in the following examples, except where otherwise stated.

DIV Switch

This switch makes the system search the denominator of an expression for simple factors that it divides into the numerator, so that rational fractions and negative powers appear in the output. With `div` on, our expression would print as

```
       2                  2  (-1)          (-1)
x*(x*y  + 2*x*y + 1/2*y *a      + 1/2*a     *z) .
```

`div` is normally off.

HORNER Switch

This switch causes the system to print polynomials according to Horner's rule. With `horner` on, our expression prints as

```
     2
x*(y  + z + 2*(y + 2)*a*x*y)/(2*a) .
```

`horner` is normally off.

LIST Switch

This switch causes the system to print each term in any sum on a separate line. With `list` on, our expression prints as

```
          2
x*(2*x*y *a
```

```
        + 4*x*y*a

           2
        + y

        + z)/(2*a) .
```

`list` is normally off.

NOSPLIT Switch

Under normal circumstances, the printing routines try to break an expression across lines at a natural point. This is a fairly expensive process. If you are not overly concerned about where the end-of-line breaks come, you can speed up the printing of expressions by turning off the switch `nosplit`. This switch is normally on.

RAT Switch

This switch is only useful with expressions in which variables are factored with `factor`. With this mode, the overall denominator of the expression is printed with each factored sub-expression. We assume a prior declaration `factor x;` in the following output. We first print the expression with `rat` set to off:

```
    2                       2
(2*x *y*a*(y + 2) + x*(y  + z))/(2*a) .
```

With `rat` on the output becomes:

```
 2                    2
x *y*(y + 2) + x*(y  + z)/(2*a) .
```

`rat` is normally off.

Next, if we leave `x` factored, and turn on both `div` and `rat`, the result becomes

```
 2                         (-1)    2
x *y*(y + 2) + 1/2*x*a     *(y  + z) .
```

Finally, with `x` factored, `rat` on and `allfac` off we retrieve the original structure

```
 2   2                 2
X *(Y  + 2*Y) + X*(Y  + Z)/(2*A) .
```

RATPRI Switch

If the numerator and denominator of an expression can each be printed in one line, the output routines will print them in a two dimensional notation, with numerator and denominator on separate lines and a line of dashes in between. For example, `(a+b)/2` will print as

```
a + b
-----
  2
```

Turning this switch off causes such expressions to be output in a linear form.

REVPRI Switch

The normal ordering of terms in output is from highest to lowest power. In some situations (e.g., when a power series is output), the opposite ordering is more convenient. The switch `revpri` if on causes such a reverse ordering of terms. For example, the expression `y*(x+1)^2+(y+3)^2` will normally print as

```
 2               2
x *y + 2*x*y + y  + 7*y + 9
```

whereas with `REVPRI` on, it will print as

```
              2             2
9 + 7*y + y  + 2*x*y + x *y.
```

8.3.4 WRITE Command

In simple cases no explicit output command is necessary in REDUCE, since the value of any expression is automatically printed if a semicolon is used as a delimiter. There are, however, several situations in which such a command is useful.

In a `for`, `while`, or `repeat` statement it may be desired to output something each time the statement within the loop construct is repeated.

It may be desired for a procedure to output intermediate results or other information while it is running. It may be desired to have results labeled in special ways, especially if the output is directed to a file or device other than the terminal.

The `write` command consists of the word `write` followed by one or more items separated by commas, and followed by a terminator. There are three kinds of items that can be used:

1. Expressions (including variables and constants). The expression is evaluated, and the result is printed out.

2. Assignments. The expression on the right side of the `:=` operator is evaluated, and is assigned to the variable on the left; then the symbol on the left is printed, followed by a "`:=`", followed by the value of the expression on the right – almost exactly the way an assignment followed by a semicolon prints out normally. (The difference is that if the `write` is in a `for` statement and the left-hand side of the assignment is an array position or something similar containing the variable of the `for` iteration, then the value of that variable is inserted in the printout.)

3. Arbitrary strings of characters, preceded and followed by double-quote marks (e.g., `"string"`).

The items specified by a single `write` statement print side by side on one line. (The line is broken automatically if it is too long.) Strings print exactly as quoted. The `write` command itself however does not return a value.

The print line is closed at the end of a `write` command evaluation. Therefore the command `write "";` (specifying nothing to be printed except the empty string) causes a line to be skipped.

Examples:

1. if `a` is `x+5`, `b` is itself, `c` is 123, `m` is an array, and `q`=3, then

```
write m(q):=a," ",b/c," THANK YOU";
```

   will set `m(3)` to `x+5` and print

```
                b
m(3) := x + 5 ----- THANK YOU
              123
```

   The blanks between the `5` and the fraction, and the fraction and `t`, come from the blanks in the quoted strings.

2. To print a table of the squares of the integers from 1 to 20:

```
for i:=1:20 do write i," ",i^2;
```

3. To print a table of the squares of the integers from 1 to 20, and at the same time store them in positions 1 to 20 of an array `a`:

```
for i:=1:20 do <<a(i):=i^2; write i," ",a(i)>>;
```

This will give us two columns of numbers. If we had used

```
for i:=1:20 do write i," ",a(i):=i^2;
```

we would also get `a(i) :=` repeated on each line.

4. The following more complete example calculates the famous f and g series, first reported in [SLT65].

```
x1:= -sig*(mu+2*eps)$
x2:= eps - 2*sig^2$
x3:= -3*mu*sig$
f:= 1$
g:= 0$
for i:= 1 step 1 until 10 do begin
  f1:= -mu*g+x1*df(f,eps)+x2*df(f,sig)+x3*df(f,mu);
  write "f(",i,") := ",f1;
  g1:= f+x1*df(g,eps)+x2*df(g,sig)+x3*df(g,mu);
  write "g(",i,") := ",g1;
  f:=f1$
  g:=g1$
 end;
```

A portion of the output, to illustrate the printout from the `write` command, is as follows:

```
                ... <prior output> ...

                                  2
f(4)  := mu*(3*eps - 15*sig  + mu)

g(4)  := 6*sig*mu

                                        2
f(5)  := 15*sig*mu*( - 3*eps + 7*sig  - mu)

                                  2
g(5)  := mu*(9*eps - 45*sig  + mu)

                ... <more output> ...
```

When the switch `nat` is turned off, `write` adds a `$` character to the end of the output line, as illustrated below.

8.3.5 Suppression of Zeros

It is sometimes annoying to have zero assignments (i.e. assignments of the form ⟨*expression*⟩ `:= 0`) printed, especially in printing large arrays with many zero elements. The output from such assignments can be suppressed by turning on the switch `nero`.

8.3.6 FORTRAN Style Output Of Expressions

It is naturally possible to evaluate expressions numerically in REDUCE by giving all variables and sub-expressions numerical values. However, as we pointed out elsewhere the user must declare real arithmetical operation by turning on the switch `rounded`. However, it should be remembered that arithmetic in REDUCE is not particularly fast, since results are interpreted rather than evaluated in a compiled form. The user with a large amount of numerical computation after all necessary algebraic manipulations have been performed is therefore well advised to perform these calculations in a FORTRAN or similar system. For this purpose, REDUCE offers facilities for users to produce FORTRAN compatible files for numerical processing.

First, when the switch `fort` is on, the system will print expressions in a FORTRAN notation. Expressions begin in column seven. If an expression extends over one line, a continuation mark (.) followed by a blank appears on subsequent cards. After a certain number of lines have been produced (according to the value of the variable `card_no`), a new expression is started. If the expression printed arises from an assignment to a variable, the variable is printed as the name of the expression. Otherwise the expression is given the default name `ans`. An error occurs if identifiers or numbers are outside the bounds permitted by FORTRAN.

A second option is to use the `write` command to produce other programs.

Example:

The following REDUCE statements

```
on fort;
out "forfil";
write "C     this is a fortran program";
write " 1    format(e13.5)";
write "      u=1.23";
write "      v=2.17";
write "      w=5.2";
x:=(u+v+w)^11;
write "C     it was foolish to expand this expression";
write "      print 1,x";
```

```
write "      end";
shut "forfil";
off fort;
```

will generate a file `forfil` that contains:

```
c     this is a fortran program
 1    format(e13.5)
      u=1.23
      v=2.17
      w=5.2
      ans1=1980.0*u**2*v**7*w**2+4620.0*u**2*v**6*w**3+6930.0*u**2*v
     . **5*w**4+6930.0*u**2*v**4*w**5+4620.0*u**2*v**3*w**6+1980.0*u
     . **2*v**2*w**7+495.0*u**2*v*w**8+55.0*u**2*w**9+11.0*u*v**10+
     . 110.0*u*v**9*w+495.0*u*v**8*w**2+1320.0*u*v**7*w**3+2310.0*u*v
     . **6*w**4+2772.0*u*v**5*w**5+2310.0*u*v**4*w**6+1320.0*u*v**3*w
     . **7+495.0*u*v**2*w**8+110.0*u*v*w**9+11.0*u*w**10+v**11+11.0*v
     . **10*w+55.0*v**9*w**2+165.0*v**8*w**3+330.0*v**7*w**4+462.0*v
     . **6*w**5+462.0*v**5*w**6+330.0*v**4*w**7+165.0*v**3*w**8+55.0*
     . v**2*w**9+11.0*v*w**10+w**11
      x=u**11+11.0*u**10*v+11.0*u**10*w+55.0*u**9*v**2+110.0*u**9*v*w
     . +55.0*u**9*w**2+165.0*u**8*v**3+495.0*u**8*v**2*w+495.0*u**8*v
     . *w**2+165.0*u**8*w**3+330.0*u**7*v**4+1320.0*u**7*v**3*w+
     . 1980.0*u**7*v**2*w**2+1320.0*u**7*v*w**3+330.0*u**7*w**4+462.0
     . *u**6*v**5+2310.0*u**6*v**4*w+4620.0*u**6*v**3*w**2+4620.0*u**
     . 6*v**2*w**3+2310.0*u**6*v*w**4+462.0*u**6*w**5+462.0*u**5*v**6
     . +2772.0*u**5*v**5*w+6930.0*u**5*v**4*w**2+9240.0*u**5*v**3*w**
     . 3+6930.0*u**5*v**2*w**4+2772.0*u**5*v*w**5+462.0*u**5*w**6+
     . 330.0*u**4*v**7+2310.0*u**4*v**6*w+6930.0*u**4*v**5*w**2+
     . 11550.0*u**4*v**4*w**3+11550.0*u**4*v**3*w**4+6930.0*u**4*v**2
     . *w**5+2310.0*u**4*v*w**6+330.0*u**4*w**7+165.0*u**3*v**8+
     . 1320.0*u**3*v**7*w+4620.0*u**3*v**6*w**2+9240.0*u**3*v**5*w**3
     . +11550.0*u**3*v**4*w**4+9240.0*u**3*v**3*w**5+4620.0*u**3*v**2
     . *w**6+1320.0*u**3*v*w**7+165.0*u**3*w**8+55.0*u**2*v**9+495.0*
     . u**2*v**8*w+ans1
c     it was foolish to expand this expression
      print 1,x
      end
```

If the arguments of a `write` statement include an expression that requires continuation records, the output will need editing, since the output routine prints the arguments of `write` sequentially, and the continuation mechanism therefore generates its auxiliary variables after the preceding expression has been printed.

Finally, since there is no direct analog of *list* in FORTRAN, a comment line of the form

```
c ***** invalid fortran construct (list) not printed
```

will be printed if you try to print a list with `fort` on.

FORTRAN Output Options

There are a number of methods available to change the default format of the FORTRAN output.

The breakup of the expression into subparts is such that the number of continuation lines produced is less than a given number. This number can be modified by the assignment

```
card_no := ⟨number⟩;
```

where ⟨*number*⟩ is the *total* number of cards allowed in a statement. The default value of `card_no` is 20.

The width of the output expression is also adjustable by the assignment

```
fort_width := ⟨integer⟩;
```

`fort_width` which sets the total width of a given line to ⟨*integer*⟩. The initial FORTRAN output width is 70.

REDUCE automatically inserts a decimal point after each isolated integer coefficient in a FORTRAN expression (so that, for example, 4 becomes `4.`). To prevent this, set the `period` mode switch to `off`.

FORTRAN output is normally produced in lower case. If upper case is desired, the switch `fortupper` should be turned on.

Finally, the default name `ans` assigned to an unnamed expression and its subparts can be changed by the operator `varname`. This takes a single identifier as argument, which then replaces `ans` as the expression name. The value of `varname` is its argument.

Further facilities for the production of FORTRAN and other language output are provided by the GENTRAN and SCOPE packages described in sections 20.24 and 20.53.

8.3.7 Saving Expressions for Later Use as Input

It is often useful to save an expression on an external file for use later as input in further calculations. The commands for opening and closing output files are explained elsewhere. However, we see in the examples on output of expressions that the standard "natural" method of printing expressions is not compatible with the input syntax. So to print the expression in an input compatible form we must inhibit this natural style by turning off the switch `nat`. If this is done, a dollar sign will also be printed at the end of the expression.

Example:

The following sequence of commands

```
off nat; out "out"; x := (y+z)^2; write "end";
shut "out"; on nat;
```

will generate a file `out` that contains

```
x := y**2 + 2*y*z + z**2$
end$
```

8.3.8 Displaying Expression Structure

In those cases where the final result has a complicated form, it is often convenient to display the skeletal structure of the answer. The operator `structr`, that takes a single expression as argument, will do this for you. Its syntax is:

```
structr(exprn:algebraic
        [,id1:identifier[,id2:identifier]]);
```

The structure is printed effectively as a tree, in which the subparts are laid out with auxiliary names. If the optional `id1` is absent, the auxiliary names are prefixed by the root `ans`. This root may be changed by the operator `varname`. If the optional `id1` is present, and is an array name, the subparts are named as elements of that array, otherwise `id1` is used as the root prefix. (The second optional argument `id2` is explained later.)

The `exprn` can be either a scalar or a matrix expression. Use of any other will result in an error.

Example:

Let us suppose that the workspace contains `((a+b)^2+c)^3+d`. Then the input `STRUCTR ws;` will (with `exp` off) result in the output:

```
ans3

   where

                    3
      ans3 := ans2  + d

                    2
      ans2 := ans1  + c
```

```
ans1 := a + b
```

The workspace remains unchanged after this operation, since `structr` in the default situation returns no value (if `structr` is used as a sub-expression, its value is taken to be 0). In addition, the sub-expressions are normally only displayed and not retained. If you wish to access the sub-expressions with their displayed names, the switch `savestructr` should be turned on. In this case, `structr` returns a list whose first element is a representation for the expression, and subsequent elements are the sub-expression relations. Thus, with `savestructr` on, `structr ws` in the above example would return

```
                    3                2
{ans3,ans3=ans2  + d,ans2=ans1  + c,ans1=a + b}
```

The `part` operator can be used to retrieve the required parts of the expression. For example, to get the value of `ans2` in the above, one could say:

```
part(ws,3,2);
```

If `fort` is on, then the results are printed in the reverse order; the algorithm in fact guaranteeing that no sub-expression will be referenced before it is defined. The second optional argument `id2` may also be used in this case to name the actual expression (or expressions in the case of a matrix argument).

Example:

Let us suppose that `m`, a 2 by 1 matrix, contains the elements `((a+b)^2 + c)^3 + d` and `(a + b)*(c + d)` respectively, and that `v` has been declared to be an array. With `exp` off and `fort` on, the statement `structr(2*m,v,k);` will result in the output

```
v(1)=a+b
v(2)=v(1)**2+c
v(3)=v(2)**3+d
v(4)=c+d
k(1,1)=2.*v(3)
k(2,1)=2.*v(1)*v(4)
```

8.4 Changing the Internal Order of Variables

The internal ordering of variables (more specifically kernels) can have a significant effect on the space and time associated with a calculation. In its default state, REDUCE uses a specific order for this which may vary between sessions. However, it is possible for the user to change this internal order by means of the declaration

`korder`. The syntax for this is:

```
korder v1,...,vn;
```

where the `vi` are kernels. With this declaration, the `vi` are ordered internally ahead of any other kernels in the system. `v1` has the highest order, `v2` the next highest, and so on. A further call of `korder` replaces a previous one. `korder nil;` resets the internal order to the system default.

Unlike the `order` declaration, that has a purely cosmetic effect on the way results are printed, the use of `korder` can have a significant effect on computation time. In critical cases then, the user can experiment with the ordering of the variables used to determine the optimum set for a given problem.

8.5 Obtaining Parts of Algebraic Expressions

There are many occasions where it is desirable to obtain a specific part of an expression, or even change such a part to another expression. A number of operators are available in REDUCE for this purpose, and will be described in this section. In addition, operators for obtaining specific parts of polynomials and rational functions (such as a denominator) are described in another section.

8.5.1 COEFF Operator

Syntax:

```
coeff(exprn:polynomial,var:kernel)
```

`coeff` is an operator that partitions `exprn` into its various coefficients with respect to `var` and returns them as a list, with the coefficient independent of `var` first.

Under normal circumstances, an error results if `exprn` is not a polynomial in `var`, although the coefficients themselves can be rational as long as they do not depend on `var`. However, if the switch `ratarg` is on, denominators are not checked for dependence on `var`, and are taken to be part of the coefficients.

Example:

```
coeff((y^2+z)^3/z,y);
```

returns the result

```
   2
```

```
{z ,0,3*z,0,3,0,1/z}.
```

whereas

```
coeff((y^2+z)^3/y,y);
```

gives an error if `ratarg` is off, and the result

```
  3         2
{z /y,0,3*z /y,0,3*z/y,0,1/y}
```

if `ratarg` is on.

The length of the result of `coeff` is the highest power of `var` encountered plus 1. In the above examples it is 7. In addition, the variable `high_pow` is set to the highest non-zero power found in `exprn` during the evaluation, and `low_pow` to the lowest non-zero power, or zero if there is a constant term. If `exprn` is a constant, then `high_pow` and `low_pow` are both set to zero.

8.5.2 COEFFN Operator

The `coeffn` operator is designed to give the user a particular coefficient of a variable in a polynomial, as opposed to `coeff` that returns all coefficients. `coeffn` is used with the syntax

```
coeffn(exprn:polynomial,var:kernel,n:integer)
```

It returns the n^{th} coefficient of `var` in the polynomial `exprn`.

8.5.3 PART Operator

Syntax:

```
part(exprn:algebraic[,intexp:integer])
```

This operator works on the form of the expression as printed *or as it would have been printed at that point in the calculation* bearing in mind all the relevant switch settings at that point. The reader therefore needs some familiarity with the way that expressions are represented in prefix form in REDUCE to use these operators effectively. Furthermore, it is assumed that `pri` is on at that point in the calculation. The reason for this is that with `pri` off, an expression is printed by walking the tree representing the expression internally. To save space, it is never actually transformed into the equivalent prefix expression as occurs when `pri` is on. How-

ever, the operations on polynomials described elsewhere can be equally well used in this case to obtain the relevant parts.

The evaluation proceeds recursively down the integer expression list. In other words,

`part(`⟨*expression*⟩`,` ⟨*integer1*⟩`,` ⟨*integer2*⟩`)`
$\longrightarrow$ `part(part(`⟨*expression*⟩`,` ⟨*integer1*⟩`),` ⟨*integer2*⟩`)`

and so on, and

`part(`⟨*expression*⟩`)` $\longrightarrow$ ⟨*expression*⟩.

`intexp` can be any expression that evaluates to an integer. If the integer is positive, then that term of the expression is found. If the integer is 0, the operator is returned. Finally, if the integer is negative, the counting is from the tail of the expression rather than the head.

For example, if the expression `a+b` is printed as `a+b` (i.e., the ordering of the variables is alphabetical), then

```
        part(a+b,2)   ->   b
        part(a+b,-1)  ->   b
and
        part(a+b,0)   ->  plus
```

If *M* is either a matrix or a matrix-valued expression then `part(M,i)` evaluates to row *i represented as a list*, and (hence) `part(M,i,j)` evaluates to the matrix element in row *i* and column *j*.

An operator `arglength` is available to determine the number of arguments of the top level operator in an expression. If the expression does not contain a top level operator, then -1 is returned. For example,

```
        arglength(a+b+c) ->  3
        arglength(f())   ->  0
        arglength(a)     ->  -1
```

8.5.4 Substituting for Parts of Expressions

`part` may also be used to substitute for a given part of an expression. In this case, the `part` construct appears on the left-hand side of an assignment statement, and the expression to replace the given part on the right-hand side.

For example, with the normal settings of the REDUCE switches:

```
xx := a+b;
part(xx,2) := c;    ->  a+c
part(c+d,0) := -;    -> c-d
```

Note that `xx` in the above is not changed by this substitution. In addition, unlike expressions such as array and matrix elements that have an *instant evaluation* property, the values of `part(xx,2)` and `part(c+d,0)` are also not changed.

The `part` operator may be used to replace part of a matrix-valued expression, including part of a matrix element. If it is used to replace a row then the replacement must be a list with the same number of elements as the row.

8.6 COMPACT Operator

`compact` is an operator for the reduction of a polynomial in the presence of side relations. It applies the side relations to the polynomial so that an equivalent expression results with as few terms as possible. For example, the evaluation of

```
compact(s*(1-sin x^2)+c*(1-cos x^2)+sin x^2+cos x^2,
        {cos x^2+sin x^2=1});
```

yields the result

```
      2            2
cos(x) *s + sin(x) *c + 1
```

The switch `trcompact` can be used to trace the operation.

8.7 TRIGSIMP package: Simplification and factorization of trigonometric and hyperbolic functions

TRIGSIMP is a useful tool for all kinds of problems related to trigonometric and hyperbolic simplification and factorization.[1] There are three operators included: `trigsimp`, `trigfactorize` and `triggcd`. The first is for simplifying trigonometric or hyperbolic expressions and has many options, the second is for factorizing them and the third is for finding the greatest common divisor of two trigonometric or hyperbolic polynomials. This package is automatically loaded when one of these operators is used.

[1] This code was written by Wolfram Koepf.

8.7.1 Simplifying trigonometric expressions

As there is no normal form for trigonometric and hyperbolic expressions, the same function can convert in many different directions, e.g. $\sin(2x) \leftrightarrow 2\sin(x)\cos(x)$. The user has the possibility to give several parameters to the operator `trigsimp` in order to influence the transformations. It is possible to decide whether or not a rational expression involving trigonometric and hyperbolic functions vanishes.

To simplify an expression `f`, one uses `trigsimp(f[,options])`. For example:

```
trigsimp(sin(x)^2+cos(x)^2);
```

```
1
```

The possible options (where * denotes the default) are:

1. `sin`* or `cos`;
2. `sinh`* or `cosh`;
3. `expand`*, `combine` or `compact`;
4. `hyp`, `trig` or `expon`;
5. `keepalltrig`;
6. `tan` and/or `tanh`;
7. target arguments of the form *variable* / *positive integer*.

From each of the first four groups one can use at most one option, otherwise an error message will occur. Options can be given in any order.

The first group fixes the preference used while transforming a trigonometric expression:

```
trigsimp(sin(x)^2);
```

```
      2
sin(x)
```

```
trigsimp(sin(x)^2, cos);
```

```
         2
 - cos(x)  + 1
```

The second group is the equivalent for the hyperbolic functions.

The third group determines the type of transformation. With the default, `expand`, an expression is transformed to use only simple variables as arguments:

```
trigsimp(sin(2x+y));

                                      2
2*cos(x)*cos(y)*sin(x) - 2*sin(x) *sin(y) + sin(y)
```

With `combine`, products of trigonometric functions are transformed to trigonometric functions involving sums of variables:

```
trigsimp(sin(x)*cos(y), combine);

 sin(x - y) + sin(x + y)
-------------------------
            2
```

With `compact`, the REDUCE operator `compact` (cf. section 8.6) is applied to `f`. This often leads to a simple form, but in contrast to `expand` one does not get a normal form. For example:

```
trigsimp((1-sin(x)^2)^20*(1-cos(x)^2)^20, compact);

      40       40
cos(x)  *sin(x)
```

With an option from the fourth group, the input expression is transformed to trigonometric, hyperbolic or exponential form respectively:

```
trigsimp(sin(x), hyp);

 - sinh(i*x)*i

trigsimp(sinh(x), expon);

  2*x
 e    - 1
----------
      x
   2*e

trigsimp(e^x, trig);
```

```
cos(i*x) - sin(i*x)*i
```

Usually, `tan`, `cot`, `sec`, `csc` are expressed in terms of `sin` and `cos`. It can sometimes be useful to avoid this, which is handled by the option `keepalltrig`:

```
trigsimp(tan(x+y), keepalltrig);
```

```
  - (tan(x) + tan(y))
----------------------
  tan(x)*tan(y) - 1
```

Alternatively, the options `tan` and/or `tanh` can be given to convert the output to the specified form as far as possible:

```
trigsimp(tan(x+y), tan);
```

```
  - (tan(x) + tan(y))
----------------------
  tan(x)*tan(y) - 1
```

By default, the other functions used will be `cos` and/or `cosh`, unless the other desired functions are also specified in which case this choice will be respected.

The final possibility is to specify additional target arguments for the trigonometric or hyperbolic functions, each of which should have the form of a variable divided by a positive integer. These additional arguments are treated as if they had occurred within the expression to be simplified, and their denominators are used in determining the overall denominator to use for each variable in the simplified form:

```
trigsimp(csc x - cot x + csc y - cot y, x/2, y/2, tan);
```

```
     x          y
tan(---) + tan(---)
     2          2
```

It is possible to use the options of different groups simultaneously:

```
trigsimp(sin(x)^4, cos, combine);
```

```
 cos(4*x) - 4*cos(2*x) + 3
--------------------------
            8
```

Sometimes, it is necessary to handle an expression in separate steps:

```
trigsimp((sinh(x)+cosh(x))^n+(cosh(x)-sinh(x))^n,
         expon);

  1   n    n*x
(----)  + e
   x
  e

trigsimp(ws, hyp);

2*cosh(n*x)

trigsimp((cosh(a*n)*sinh(a)*sinh(p)+
          cosh(a)*sinh(a*n)*sinh(p)+
          sinh(a - p)*sinh(a*n))/sinh(a));

cosh(a*n)*sinh(p) + cosh(p)*sinh(a*n)

trigsimp(ws, combine);

sinh(a*n + p)
```

The `trigsimp` operator can be applied to equations, lists and matrices (and compositions thereof) as well as scalar expressions, and automatically maps itself recursively over such non-scalar data structures:

```
trigsimp( { sin(2x) = cos(2x) } );

                              2
{2*cos(x)*sin(x)= - 2*sin(x)  + 1}
```

8.7.2 Factorizing trigonometric expressions

With `trigfactorize(p,x)` one can factorize the trigonometric or hyperbolic polynomial `p` in terms of trigonometric functions of the argument `x`. The output has the same format as that from the standard REDUCE operator `factorize`. For example:

```
trigfactorize(sin(x), x/2);

             x            x
{{2,1},{sin(---),1},{cos(---),1}}
             2            2
```

If the polynomial is not coordinated or balanced [Roa], the output will equal the input. In this case, changing the value for `x` can help to find a factorization, e.g.

```
trigfactorize(1+cos(x), x);

{{cos(x) + 1,1}}

trigfactorize(1+cos(x), x/2);

              x
{{2,1},{cos(---),2}}
              2
```

The polynomial can consist of both trigonometric and hyperbolic functions:

```
trigfactorize(sin(2x)*sinh(2x), x);

{{4,1},

 {sinh(x),1},

 {cosh(x),1},

 {sin(x),1},

 {cos(x),1}}
```

The `trigfactorize` operator respects the standard REDUCE `factorize` switch `nopowers` – see the REDUCE manual for details. Turning it on gives the behaviour that was standard before REDUCE 3.7:

```
on nopowers;

trigfactorize(1+cos(x), x/2);

        x         x
{2,cos(---),cos(---)}
        2         2
```

8.7.3 GCDs of trigonometric expressions

The operator `triggcd` is essentially an application of the algorithm behind `trigfactorize`. With its help the user can find the greatest common divisor

of two trigonometric or hyperbolic polynomials. It uses the method described in [Roa]. The syntax is `triggcd(p,q,x)`, where `p` and `q` are the trigonometric polynomials and `x` is the argument to use. For example:

```
triggcd(sin(x), 1+cos(x), x/2);

     x
cos(---)
     2

triggcd(sin(x), 1+cos(x), x);

1
```

The polynomials p and q can consist of both trigonometric and hyperbolic functions:

```
triggcd(sin(2x)*sinh(2x), (1-cos(2x))*(1+cosh(2x)), x);

cosh(x)*sin(x)
```

8.7.4 Further Examples

With the help of this package the user can create identities:

```
trigsimp(tan(x)*tan(y));

 sin(x)*sin(y)
---------------
 cos(x)*cos(y)

trigsimp(ws, combine);

 cos(x - y) - cos(x + y)
-------------------------
 cos(x - y) + cos(x + y)

trigsimp((sin(x-a)+sin(x+a))/(cos(x-a)+cos(x+a)));

 sin(x)
--------
 cos(x)
```

```
trigsimp(cosh(n*acosh(x))-cos(n*acos(x)), trig);

0

trigsimp(sec(a-b), keepalltrig);

  csc(a)*csc(b)*sec(a)*sec(b)
------------------------------
 csc(a)*csc(b) + sec(a)*sec(b)

trigsimp(tan(a+b), keepalltrig);

  - (tan(a) + tan(b))
---------------------
  tan(a)*tan(b) - 1

trigsimp(ws, keepalltrig, combine);

tan(a + b)
```

Some difficult expressions can be simplified:

```
df(sqrt(1+cos(x)), x, 4);

            5           4            3       2
( - 4*cos(x)  - 4*cos(x)  - 20*cos(x) *sin(x)

             3            2       2            2
  + 12*cos(x)  - 24*cos(x) *sin(x)  + 20*cos(x)

                    4                   2
  - 15*cos(x)*sin(x)  + 12*cos(x)*sin(x)

                         4            2
  + 8*cos(x) - 15*sin(x)  + 16*sin(x) )/(16

                              4           3
   *sqrt(cos(x) + 1)*(cos(x)  + 4*cos(x)

                 2
       + 6*cos(x)  + 4*cos(x) + 1))

on rationalize;
```

```
trigsimp(ws);

 sqrt(cos(x) + 1)
------------------
        16

off rationalize;

taylor(sin(x+a)*cos(x+b), x, 0, 4);

cos(b)*sin(a) + (cos(a)*cos(b) - sin(a)*sin(b))*x

                                      2
 - (cos(a)*sin(b) + cos(b)*sin(a))*x

    2*( - cos(a)*cos(b) + sin(a)*sin(b))   3
 + ---------------------------------------*x
                     3

    cos(a)*sin(b) + cos(b)*sin(a)   4      5
 + -------------------------------*x  + O(x )
                 3

trigsimp(ws, combine);

 sin(a - b) + sin(a + b)
------------------------ + cos(a + b)*x
           2

               2    2*cos(a + b)   3
 - sin(a + b)*x  - --------------*x
                         3

    sin(a + b)   4      5
 + ------------*x  + O(x )
        3
```

Certain integrals whose evaluation was not possible in REDUCE (without preprocessing) are now computable:

```
int(trigsimp(sin(x+y)*cos(x-y)*tan(x)), x);

       2
(cos(x) *x - cos(x)*sin(x) - 2*cos(y)*log(cos(x))*sin(y)
```

```
          2
  + sin(x) *x)/2

int(trigsimp(sin(x+y)*cos(x-y)/tan(x)), x);

                                        x  2
(cos(x)*sin(x) - 2*cos(y)*log(tan(---)  + 1)*sin(y)
                                        2

                       x
  + 2*cos(y)*log(tan(---))*sin(y) + x)/2
                       2
```

Without the package, the integration fails, and in the second case one does not receive an answer for many hours.

```
trigfactorize(sin(2x)*cos(y)^2, y/2);

{{2*cos(x)*sin(x),1},

       y          y
 {cos(---) - sin(---),2},
       2          2

       y          y
 {cos(---) + sin(---),2}}
       2          2

trigfactorize(sin(y)^4-x^2, y);

        2               2
{{sin(y)  + x,1},{sin(y)  - x,1}}

trigfactorize(sin(x)*sinh(x), x/2);

{{4,1},

        x
 {sinh(---),1},
        2

        x
```

```
 {cosh(---),1},
        2

       x
 {sin(---),1},
       2

       x
 {cos(---),1}}
       2

triggcd(-5+cos(2x)-6sin(x), -7+cos(2x)-8sin(x), x/2);

       x        x
2*cos(---)*sin(---) + 1
       2        2

triggcd(1-2cosh(x)+cosh(2x), 1+2cosh(x)+cosh(2x), x/2);

        x  2
2*sinh(---)  + 1
        2
```

Chapter 9

Polynomials and Rationals

Many operations in computer algebra are concerned with polynomials and rational functions. In this section, we review some of the switches and operators available for this purpose. These are in addition to those that work on general expressions (such as `df` and `int`) described elsewhere. In the case of operators, the arguments are first simplified before the operations are applied. In addition, they operate only on arguments of prescribed types, and produce a type mismatch error if given arguments which cannot be interpreted in the required mode with the current switch settings. For example, if an argument is required to be a kernel and `a/2` is used (with no other rules for `a`), an error

```
a/2 invalid as kernel
```

will result.

With the exception of those that select various parts of a polynomial or rational function, these operations have potentially significant effects on the space and time associated with a given calculation. The user should therefore experiment with their use in a given calculation in order to determine the optimum set for a given problem.

One such operation provided by the system is an operator `length` which returns the number of top level terms in the numerator of its argument. For example,

```
length ((a+b+c)^3/(c+d));
```

has the value 10. To get the number of terms in the denominator, one would first select the denominator by the operator `den` and then call `length`, as in

```
length den ((a+b+c)^3/(c+d));
```

Other operations currently supported, the relevant switches and operators, and the

required argument and value modes of the latter, follow.

9.1 Controlling the Expansion of Expressions

The switch `exp` controls the expansion of expressions. If it is off, no expansion of powers or products of expressions occurs. Users should note however that in this case results come out in a normal but not necessarily canonical form. This means that zero expressions simplify to zero, but that two equivalent expressions need not necessarily simplify to the same form.

Example: With `exp` on, the two expressions

```
(a+b)*(a+2*b)
```

and

```
a^2+3*a*b+2*b^2
```

will both simplify to the latter form. With `exp` off, they would remain unchanged. The switch `exp` is normally on. Note that if the complete factoring (`allfac`) option is in force it affects the output independently of the setting of `exp`.

Several operators that expect a polynomial as an argument behave differently when `exp` is off, since there is often only one term at the top level. For example, with `exp` off

```
length((a+b+c)^3/(c+d));
```

returns the value 1.

9.2 Factorization of Polynomials

REDUCE is capable of factorizing univariate and multivariate polynomials that have integer coefficients, finding all factors that also have integer coefficients. The package for doing this was written by Dr. Arthur C. Norman and Ms. P. Mary Ann Moore at The University of Cambridge. It is described in [NM81].

The easiest way to use this facility is to turn on the switch `factor`, which causes all expressions to be output in a factored form. For example, with `factor` on, the expression `a^2-b^2` is returned as `(a+b)*(a-b)`.

It is also possible to factorize a given expression explicitly. The operator `factorize` that invokes this facility is used with the syntax

`factorize` (⟨*polynomial*⟩ [,prime integer]) :⟨*list*⟩,

the optional argument of which will be described later. Thus to find and display all factors of the cyclotomic polynomial $x^{105} - 1$, one could write:

```
factorize(x^105-1);
```

The result is a list of factor,exponent pairs. In the above example, there is no overall numerical factor in the result, so the results will consist only of polynomials in x. The number of such polynomials can be found by using the operator `length`. If there is a numerical factor, as in factorizing $12x^2 - 12$, the first member of the result will be a list with two elements: the numerical factor and its multiplicity (1). It will however not be factored further. Prime factors of such numbers can be found, using a probabilistic algorithm, by turning on the switch `ifactor`. For example,

```
on ifactor; factorize(12x^2-12);
```

would result in the output

```
{{2,2},{3,1},{x + 1,1},{x - 1,1}}.
```

If the first argument of `factorize` is an integer, it will be decomposed into its prime components, whether or not `ifactor` is on.

Note that the `ifactor` switch only affects the result of `factorize`. It has no effect if the `factor` switch is also on.

The order in which the factors occur in the result (with the exception of a possible overall numerical coefficient which comes first) can be system dependent and should not be relied on. Similarly it should be noted that any pair of individual factors can be negated without altering their product, and that REDUCE may sometimes do that.

The factorizer works by first reducing multivariate problems to univariate ones and then solving the univariate ones modulo small primes. It normally selects both evaluation points and primes using a random number generator that should lead to different detailed behavior each time any particular problem is tackled. If, for some reason, it is known that a certain (probably univariate) factorization can be performed effectively with a known prime, `p` say, this value of `p` can be handed to `factorize` as a second argument. An error will occur if a non-prime is provided to `factorize` in this manner. It is also an error to specify a prime that divides the discriminant of the polynomial being factored, but users should note that this

condition is not checked by the program, so this capability should be used with care.

Factorization can be performed over a number of polynomial coefficient domains in addition to integers. The particular description of the relevant domain should be consulted to see if factorization is supported. For example, the following statements will factorize $x^4 + 1$ modulo 7:

```
setmod 7;
on modular;
factorize(x^4+1);
```

The factorization module is provided with a trace facility that may be useful as a way of monitoring progress on large problems, and of satisfying curiosity about the internal workings of the package. The most simple use of this is enabled by issuing the REDUCE command `on trfac;` . Following this, all calls to the factorizer will generate informative messages reporting on such things as the reduction of multivariate to univariate cases, the choice of a prime and the reconstruction of full factors from their images. Further levels of detail in the trace are intended mainly for system tuners and for the investigation of suspected bugs. For example, `trallfac` gives tracing information at all levels of detail. `on overview;` reduces the amount of detail presented in other forms of trace. Other forms of trace output are enabled by directives of the form

```
symbolic set!-trace!-factor(<number>,<filename>);
```

where useful numbers are 1, 2, 3 and 100, 101, ... This facility is intended to make it possible to discover in fairly great detail what just some small part of the code has been doing — the numbers refer mainly to depths of recursion when the factorizer calls itself, and to the split between its work forming and factorizing images and reconstructing full factors from these. If `nil` is used in place of a filename the trace output requested is directed to the standard output stream. After use of this trace facility the generated trace files should be closed by calling

```
symbolic close!-trace!-files();
```

NOTE: Using the factorizer with `mcd` off will result in an error.

9.3 Cancellation of Common Factors

Facilities are available in REDUCE for cancelling common factors in the numerators and denominators of expressions, at the option of the user. The system will perform this greatest common divisor computation if the switch `gcd` is on. (`gcd`

is normally off.)

A check is automatically made, however, for common variable and numerical products in the numerators and denominators of expressions, and the appropriate cancellations made.

When `gcd` is on, and `exp` is off, a check is made for square free factors in an expression. This includes separating out and independently checking the content of a given polynomial where appropriate. (For an explanation of these terms, see [Hea79].)

Example: With `exp` off and `gcd` on, the polynomial `a*c+a*d+b*c+b*d` would be returned as `(a+b)*(c+d)`.

Under normal circumstances, GCDs are computed using an algorithm described in the above paper. It is also possible in REDUCE to compute GCDs using an alternative algorithm, called the EZGCD Algorithm, which uses modular arithmetic. The switch `ezgcd`, if on in addition to `gcd`, makes this happen.

In non-trivial cases, the EZGCD algorithm is almost always better than the basic algorithm, often by orders of magnitude. We therefore *strongly* advise users to use the `ezgcd` switch where they have the resources available for supporting the package.

For a description of the EZGCD algorithm, [MY73].

NOTE: This package shares code with the factorizer, so a certain amount of trace information can be produced using the factorizer trace switches.

An implementation of the heuristic GCD algorithm, first introduced by B.W. Char, K.O. Geddes and G.H. Gonnet, as described in [DP85], is also available on an experimental basis. To use this algorithm, the switch `heugcd` should be on in addition to `gcd`. Note that if both `ezgcd` and `heugcd` are on, the former takes precedence.

9.3.1 Determining the GCD of Two Polynomials

This operator, used with the syntax

```
gcd(exprn1:polynomial,exprn2:polynomial):polynomial,
```

returns the greatest common divisor of the two polynomials `exprn1` and `exprn2`.

Examples:

```
gcd(x^2+2*x+1,x^2+3*x+2)  ->  x+1
gcd(2*x^2-2*y^2,4*x+4*y)  ->  2*x+2*y
gcd(x^2+y^2,x-y)          ->  1.
```

9.4 Working with Least Common Multiples

Greatest common divisor calculations can often become expensive if extensive work with large rational expressions is required. However, in many cases, the only significant cancellations arise from the fact that there are often common factors in the various denominators which are combined when two rationals are added. Since these denominators tend to be smaller and more regular in structure than the numerators, considerable savings in both time and space can occur if a full GCD check is made when the denominators are combined and only a partial check when numerators are constructed. In other words, the true least common multiple of the denominators is computed at each step. The switch `lcm` is available for this purpose, and is normally on.

In addition, the operator `lcm`, used with the syntax

```
lcm(exprn1:polynomial,exprn2:polynomial):polynomial,
```

returns the least common multiple of the two polynomials `exprn1` and `exprn2`.

Examples:

```
lcm(x^2+2*x+1,x^2+3*x+2) ->  x**3 + 4*x**2 + 5*x + 2
lcm(2*x^2-2*y^2,4*x+4*y) ->  4*(x**2 - y**2)
```

9.5 Controlling Use of Common Denominators

When two rational functions are added, REDUCE normally produces an expression over a common denominator. However, if the user does not want denominators combined, he or she can turn off the switch `mcd` which controls this process. The latter switch is particularly useful if no greatest common divisor calculations are desired, or excessive differentiation of rational functions is required.

CAUTION: With `mcd` off, results are not guaranteed to come out in either normal or canonical form. In other words, an expression equivalent to zero may in fact not be simplified to zero. This option is therefore most useful for avoiding expression swell during intermediate parts of a calculation.

`mcd` is normally on.

9.6 Euclidean Division

The operators `divide`, `poly_quotient` and `mod` / `remainder` implement Euclidean division of polynomials (over the current number domain). The

`remainder` operator is used with the syntax

```
remainder(exprn1:polynomial,exprn2:polynomial):
     polynomial.
```

It returns the remainder when `exprn1` is divided by `exprn2`. This is the true remainder based on the internal ordering of the variables, and not the pseudo-remainder.

Examples:

```
remainder((x + y)*(x + 2*y), x + 3*y)  ->  2*y^2
remainder(2*x + y, 2)                  ->  y
```

CAUTION: In the default case, remainders are calculated over the integers. If you need the remainder with respect to another domain, it must be declared explicitly.

Example:

```
remainder(x^2 - 2, x + sqrt(2));  ->  x^2 - 2
load_package arnum;
defpoly sqrt2^2 - 2;
remainder(x^2 - 2, x + sqrt2);    ->  0
```

(Note the use of `sqrt2` in place of `sqrt(2)` in the second call of `remainder`.)

The infix operator `mod` is an alias for `remainder` when at least one operand is explicitly polynomial, e.g.

```
(x^2 + y^2) mod (x - y);

   2
2*y
```

However, when both operands are integers, `mod` implements the integer modulus operation, regardless of the current number domain, e.g.

```
7 mod 4  ->  3
```

The Euclidean division operator `divide` is used with the syntax

```
divide(exprn1:polynomial,exprn2:polynomial):
     list(polynomial,polynomial)
```

and returns both the quotient and the remainder together as the first and second elements of a list, e.g.

```
divide(x^2 + y^2, x - y);

           2
{x + y,2*y }
```

It can also be used as an infix operator:

```
(x^2 + y^2) divide (x - y);

           2
{x + y,2*y }
```

The infix operator `poly_quotient` returns only the quotient, i.e. the first element of the list returned by `divide`.

All Euclidean division operators (when used in prefix form) accept an optional third argument, which specifies the main variable to be used during the division. The default is the leading kernel in the current global ordering. Specifying the main variable does not change the ordering of any other variables involved, nor does it change the global environment. For example

```
divide(x^2 + y^2, x - y, y);

                2
{ - (x + y),2*x }
```

Specifying x as main variable gives the same behaviour as the default shown earlier, i.e.

```
divide(x^2 + y^2, x - y, x);

           2
{x + y,2*y }
```

All Euclidean division operators accept a (possibly nested) list as first argument/operand and map over that list, e.g.

```
{x, x + 1, x^2 - 1} mod x - 1;

{1,2,0}
```

9.7 Polynomial Pseudo-Division

The polynomial division discussed above is normally most useful for a univariate polynomial over a field, otherwise the division is likely to fail giving trivially a zero quotient and a remainder equal to the dividend. (A ring of univariate polynomials is a Euclidean domain only if the coefficient ring is a field.) For example, over the integers:

```
divide(x^2 + y^2, 2(x - y));

    2    2
{0,x  + y }
```

The division of a polynomial $u(x)$ of degree m by a polynomial $v(x)$ of degree $n \leq m$ can be performed over any commutative ring with identity (such as the integers, or any polynomial ring) if the polynomial $u(x)$ is first multiplied by $\mathrm{lc}(v,x)^{m-n+1}$ (where lc denotes the leading coefficient). This is called *pseudo-division*. The polynomial pseudo-division operators `pseudo_divide`, `pseudo_quotient` and `pseudo_remainder` are implemented as prefix operators (only). When multivariate polynomials are pseudo-divided it is important which variable is taken as the main variable, because the leading coefficient of the divisor is computed with respect to this variable. Therefore, if this is allowed to default and there is any ambiguity, i.e. the polynomials are multivariate or contain more than one kernel, the pseudo-division operators output a warning message to indicate which kernel has been selected as the main variable – it is the first kernel found in the internal forms of the dividend and divisor. (As usual, the warning can be turned off by setting the switch `msg` to off.) For example

```
pseudo_divide(x^2 + y^2, x - y);

*** Main division variable selected is x

          2
{x + y,2*y }

pseudo_divide(x^2 + y^2, x - y, x);

          2
{x + y,2*y }

pseudo_divide(x^2 + y^2, x - y, y);

               2
```

```
{ - (x + y),2*x }
```

If the leading coefficient of the divisor is a unit (invertible element) of the coefficient ring then division and pseudo-division should be identical, otherwise they are not, e.g.

```
divide(x^2 + y^2, 2(x - y));

    2    2
{0,x  + y }

pseudo_divide(x^2 + y^2, 2(x - y));

*** Main division variable selected is x

               2
{2*(x + y),8*y }
```

The pseudo-division gives essentially the same result as would division over the field of fractions of the coefficient ring (apart from the overall factors [contents] of the quotient and remainder), e.g.

```
on rational;

divide(x^2 + y^2, 2(x - y));

  1             2
{---*(x + y),2*y }
  2

pseudo_divide(x^2 + y^2, 2(x - y));

*** Main division variable selected is x

               2
{2*(x + y),8*y }
```

Polynomial division and pseudo-division can only be applied to what REDUCE regards as polynomials, i.e. rational expressions with denominator 1, e.g.

```
off rational;

pseudo_divide((x^2 + y^2)/2, x - y);
```

```
          2    2
         x  + y
***** --------- invalid as polynomial
           2
```

All pseudo-division operators accept a (possibly nested) list as first argument/operand and map over that list.

Pseudo-division is implemented using an algorithm ([Knu81], Algorithm R, page 407) that does not perform any actual division at all (which proves that it applies over a ring). It is more efficient than the naive algorithm, and it also has the advantage that it works over coefficient domains in which REDUCE may not be able to perform in practice divisions that are possible mathematically. An example of this is coefficient domains involving algebraic numbers, such as the integers extended by $\sqrt{2}$, as illustrated in the file `polydiv.tst`.

The implementation attempts to be reasonably efficient, except that it always computes the quotient internally even when only the remainder is required (as does the standard remainder operator).

9.8 RESULTANT Operator

This is used with the syntax

`resultant(`⟨*polynomial*⟩,⟨*polynomial*⟩,⟨*kernel*⟩`)`:⟨*polynomial*⟩.

It computes the resultant of the two given polynomials with respect to the given variable, the coefficients of the polynomials can be taken from any domain. The result can be identified as the determinant of a Sylvester matrix, but can often also be thought of informally as the result obtained when the given variable is eliminated between the two input polynomials. If the two input polynomials have a non-trivial GCD their resultant vanishes.

The switch `bezout` controls the computation of the resultants. It is off by default. In this case a subresultant algorithm is used. If the switch Bezout is turned on, the resultant is computed via the Bezout Matrix. However, in the latter case, only polynomial coefficients are permitted.

The sign conventions used by the resultant function follow those in [Loo82]. namely, with a and b not dependent on x:

$$\begin{aligned}\text{resultant}(p(x), q(x), x) &= (-1)^{\deg(p)\cdot\deg(q)} \cdot \text{resultant}(q(x), p(x), x)\\ \text{resultant}(a, q(x), x) &= a^{\deg(p)}\\ \text{resultant}(a, b, x) &= 1\end{aligned}$$

Examples:

```
                                    2
resultant(x/r*u+y,u*y,u)    ->  - y
```

calculation in an algebraic extension:

```
load arnum;
defpoly sqrt2**2 - 2;

resultant(x + sqrt2,sqrt2 * x +1,x) -> -1
```

or in a modular domain:

```
setmod 17;
on modular;

resultant(2x+1,3x+4,x)     -> 5
```

9.9 DECOMPOSE Operator

The `decompose` operator takes a multivariate polynomial as argument, and returns an expression and a list of equations from which the original polynomial can be found by composition. Its syntax is:

```
decompose(exprn:polynomial):list.
```

For example:

```
decompose(x^8-88*x^7+2924*x^6-43912*x^5+263431*x^4-
              218900*x^3+65690*x^2-7700*x+234)
             2                   2            2
     -> {u  + 35*u + 234, u=v  + 10*v, v=x  - 22*x}
                                 2
decompose(u^2+v^2+2u*v+1)  -> {w  + 1, w=u + v}
```

Users should note however that, unlike factorization, this decomposition is not unique.

9.10 INTERPOL Operator

Syntax:

`interpol(`⟨*values*⟩`,` ⟨*variable*⟩`,` ⟨*points*⟩`);`

where ⟨*values*⟩ and ⟨*points*⟩ are lists of equal length and `<variable>` is an algebraic expression (preferably a kernel).

`interpol` generates an interpolation polynomial f in the given variable of degree `length(`⟨*values*⟩`)`-1. The unique polynomial f is defined by the property that for corresponding elements v of ⟨*values*⟩ and p of ⟨*points*⟩ the relation $f(p) = v$ holds.

The Aitken-Neville interpolation algorithm is used which guarantees a stable result even with rounded numbers and an ill-conditioned problem.

9.11 Obtaining Parts of Polynomials and Rationals

These operators select various parts of a polynomial or rational function structure. Except for the cost of rearrangement of the structure, these operations take very little time to perform.

For those operators in this section that take a kernel `var` as their second argument, an error results if the first expression is not a polynomial in `var`, although the coefficients themselves can be rational as long as they do not depend on `var`. However, if the switch `ratarg` is on, denominators are not checked for dependence on `var`, and are taken to be part of the coefficients.

9.11.1 DEG Operator

This operator is used with the syntax

```
deg(exprn:polynomial,var:kernel):integer.
```

It returns the leading degree of the polynomial `exprn` in the variable `var`. If `var` does not occur as a variable in `exprn`, 0 is returned.

Examples:

```
deg((a+b)*(c+2*d)^2,a) ->  1
deg((a+b)*(c+2*d)^2,d) ->  2
deg((a+b)*(c+2*d)^2,e) ->  0.
```

Note also that if `ratarg` is on,

```
deg((a+b)^3/a,a)        ->  3
```

since in this case, the denominator `a` is considered part of the coefficients of the numerator in `a`. With `ratarg` off, however, an error would result in this case.

9.11.2 DEN Operator

This is used with the syntax:

```
den(exprn:rational):polynomial.
```

It returns the denominator of the rational expression `exprn`. If `exprn` is a polynomial, 1 is returned.

Examples:

```
den(x/y^2)    ->  Y**2
den(100/6)    ->  3
     % since 100/6 is first simplified to 50/3
den(a/4+b/6) ->  12
den(a+b)      ->  1
```

9.11.3 LCOF Operator

`lcof` is used with the syntax

```
lcof(exprn:polynomial,var:kernel):polynomial.
```

It returns the leading coefficient of the polynomial `exprn` in the variable `var`. If `var` does not occur as a variable in `exprn`, `exprn` is returned. *Examples:*

```
lcof((a+b)*(c+2*d)^2,a) ->  c**2+4*c*d+4*d**2
lcof((a+b)*(c+2*d)^2,d) ->  4*(a+b)
lcof((a+b)*(c+2*d),e)   ->  a*c+2*a*d+b*c+2*b*d
```

9.11.4 LPOWER Operator

Syntax:

```
lpower(exprn:polynomial,var:kernel):polynomial.
```

`lpower` returns the leading power of `exprn` with respect to `var`. If `exprn` does not depend on `var`, 1 is returned.

Examples:

```
lpower((a+b)*(c+2*d)^2,a) ->  a
lpower((a+b)*(c+2*d)^2,d) ->  d**2
lpower((a+b)*(c+2*d),e)   ->  1
```

9.11.5 LTERM Operator

Syntax:

```
lterm(exprn:polynomial,var:kernel):polynomial.
```

`lterm` returns the leading term of `exprn` with respect to `var`. If `exprn` does not depend on `var`, `exprn` is returned.

Examples:

```
lterm((a+b)*(c+2*d)^2,a) ->  a*(c**2+4*c*d+4*d**2)
lterm((a+b)*(c+2*d)^2,d) ->  4*d**2*(a+b)
lterm((a+b)*(c+2*d),e)   ->  a*c+2*a*d+b*c+2*b*d
```

Compatibility Note: In some earlier versions of REDUCE, `lterm` returned `0` if the `exprn` did not depend on `var`. In the present version, `exprn` is always equal to `lterm(exprn,var) + reduct(exprn,var)`.

9.11.6 MAINVAR Operator

Syntax:

```
mainvar(exprn:polynomial):expression.
```

Returns the main variable (based on the internal polynomial representation) of `exprn`. If `exprn` is a domain element, 0 is returned.

Examples:

Assuming `a` has higher kernel order than `b`, `c`, or `d`:

```
mainvar((a+b)*(c+2*d)^2) ->  a
mainvar(2)               ->  0
```

9.11.7 NUM Operator

Syntax:

```
num(exprn:rational):polynomial.
```

Returns the numerator of the rational expression `exprn`. If `exprn` is a polynomial, that polynomial is returned.

Examples:

```
num(x/y^2)   ->  x
num(100/6)    ->  50
num(a/4+b/6) ->  3*a+2*b
num(a+b)      ->  a+b
```

9.11.8 REDUCT Operator

Syntax:

```
reduct(exprn:polynomial,var:kernel):polynomial.
```

Returns the reductum of `exprn` with respect to `var` (i.e., the part of `exprn` left after the leading term is removed). If `exprn` does not depend on the variable `var`, 0 is returned.

Examples:

```
reduct((a+b)*(c+2*d),a) ->  b*(c + 2*d)
reduct((a+b)*(c+2*d),d) ->  c*(a + b)
reduct((a+b)*(c+2*d),e) ->  0
```

Compatibility Note: In some earlier versions of REDUCE, `reduct` returned `exprn` if it did not depend on `var`. In the present version, `exprn` is always equal to `lterm(exprn,var) + reduct(exprn,var)`.

9.11.9 TOTALDEG Operator

Syntax:

```
totaldeg(a*x^2+b*x+c, x)  => 2
totaldeg(a*x^2+b*x+c, {a,b,c})  => 1
totaldeg(a*x^2+b*x+c, {x, a})  => 3
totaldeg(a*x^2+b*x+c, {x,b})  => 2
totaldeg(a*x^2+b*x+c, {p,q,r})  => 0
```

`totaldeg(u, kernlist)` finds the total degree of the polynomial `u` in the variables in `kernlist`. If `kernlist` is not a list it is treated as a simple single

variable. The denominator of `u` is ignored, and "degree" here does not pay attention to fractional powers. Mentions of a kernel within the argument to any operator or function (eg sin, cos, log, sqrt) are ignored. Really `u` is expected to be just a polynomial.

9.12 Polynomial Coefficient Arithmetic

REDUCE allows for a variety of numerical domains for the numerical coefficients of polynomials used in calculations. The default mode is integer arithmetic, although the possibility of using real coefficients has been discussed elsewhere. Rational coefficients have also been available by using integer coefficients in both the numerator and denominator of an expression, using the `on div` option to print the coefficients as rationals. However, REDUCE includes several other coefficient options in its basic version which we shall describe in this section. All such coefficient modes are supported in a table-driven manner so that it is straightforward to extend the range of possibilities. A description of how to do this is given in [BHPS86].

9.12.1 Rational Coefficients in Polynomials

Instead of treating rational numbers as the numerator and denominator of a rational expression, it is also possible to use them as polynomial coefficients directly. This is accomplished by turning on the switch `rational`.

Example: With `rational` off, the input expression `a/2` would be converted into a rational expression, whose numerator was `a` and denominator 2. With `rational` on, the same input would become a rational expression with numerator `1/2*a` and denominator `1`. Thus the latter can be used in operations that require polynomial input whereas the former could not.

9.12.2 Real Coefficients in Polynomials

The switch `rounded` permits the use of arbitrary sized real coefficients in polynomial expressions. The actual precision of these coefficients can be set by the operator `precision`. For example, `precision 50;` sets the precision to fifty decimal digits. The default precision is system dependent and can be found by `precision 0;`. In this mode, denominators are automatically made monic, and an appropriate adjustment is made to the numerator.

Example: With `ROUNDED` on, the input expression `a/2` would be converted into a rational expression whose numerator is `0.5*a` and denominator `1`.

Internally, REDUCE uses floating point numbers up to the precision supported by

the underlying machine hardware, and so-called *bigfloats* for higher precision or whenever necessary to represent numbers whose value cannot be represented in floating point. The internal precision is two decimal digits greater than the external precision to guard against roundoff inaccuracies. Bigfloats represent the fraction and exponent parts of a floating-point number by means of (arbitrary precision) integers, which is a more precise representation in many cases than the machine floating point arithmetic, but not as efficient. If a case arises where use of the machine arithmetic leads to problems, a user can force REDUCE to use the bigfloat representation at all precisions by turning on the switch `roundbf`. In rare cases, this switch is turned on by the system, and the user informed by the message

```
ROUNDBF turned on to increase accuracy
```

Rounded numbers are normally printed to the specified precision. However, if the user wishes to print such numbers with less precision, the printing precision can be set by the command `print_precision`. For example, `print_precision 5;` will cause such numbers to be printed with five digits maximum.

Under normal circumstances when `rounded` is on, REDUCE converts the number 1.0 to the integer 1. If this is not desired, the switch `noconvert` can be turned on.

Numbers that are stored internally as bigfloats are normally printed with a space between every five digits to improve readability. If this feature is not required, it can be suppressed by turning off the switch `bfspace`.

Further information on the bigfloat arithmetic may be found in T. Sasaki, "Manual for Arbitrary Precision Real Arithmetic System in REDUCE", Department of Computer Science, University of Utah, Technical Note No. TR-8 (1979).

When a real number is input, it is normally truncated to the precision in effect at the time the number is read. If it is desired to keep the full precision of all numbers input, the switch `adjprec` (for *adjust precision*) can be turned on. While on, `adjprec` will automatically increase the precision, when necessary, to match that of any integer or real input, and a message printed to inform the user of the precision increase.

When `rounded` is on, rational numbers are normally converted to rounded representation. However, if a user wishes to keep such numbers in a rational form until used in an operation that returns a real number, the switch `roundall` can be turned off. This switch is normally on.

Results from rounded calculations are returned in rounded form with two exceptions: if the result is recognized as `0` or `1` to the current precision, the integer result is returned.

9.12.3 Modular Number Coefficients in Polynomials

REDUCE includes facilities for manipulating polynomials whose coefficients are computed modulo a given base. To use this option, two commands must be used; `setmod` ⟨*integer*⟩, to set the prime modulus, and `on modular` to cause the actual modular calculations to occur. For example, with `setmod 3;` and `on modular;`, the polynomial `(a+2*b)^3` would become `a^3+2*b^3`.

The argument of `setmod` is evaluated algebraically, except that non-modular (integer) arithmetic is used. Thus the sequence

```
setmod 3; on modular; setmod 7;
```

will correctly set the modulus to 7.

Modular numbers are by default represented by integers in the interval [0,p-1] where p is the current modulus. Sometimes it is more convenient to use an equivalent symmetric representation in the interval [-p/2+1,p/2], or more precisely [-floor((p-1)/2), ceiling((p-1)/2)], especially if the modular numbers map objects that include negative quantities. The switch `balanced_mod` allows you to select the symmetric representation for output.

Users should note that the modular calculations are on the polynomial coefficients only. It is not currently possible to reduce the exponents since no check for a prime modulus is made (which would allow x^{p-1} to be reduced to 1 mod p). Note also that any division by a number not co-prime with the modulus will result in the error "Invalid modular division".

9.12.4 Complex Number Coefficients in Polynomials

Although REDUCE routinely treats the square of the variable *i* as equivalent to -1, this is not sufficient to reduce expressions involving *i* to lowest terms, or to factor such expressions over the complex numbers. For example, in the default case,

```
factorize(a^2+1);
```

gives the result

```
{{a**2+1,1}}
```

and

```
(a^2+b^2)/(a+i*b)
```

is not reduced further. However, if the switch `complex` is turned on, full complex

arithmetic is then carried out. In other words, the above factorization will give the result

```
{{a + i,1},{a - i,1}}
```

and the quotient will be reduced to `a-i*b`.

The switch `complex` may be combined with `rounded` to give complex real numbers; the appropriate arithmetic is performed in this case. Similarly, combining `complex` with `rational` performs polynomial arithmetic with complex rational numbers.

Complex conjugation is used to remove complex numbers from denominators of expressions. To do this if `complex` is off, you must turn the switch `rationalize` on.

9.12.5 Algebraic Numbers as Coefficients in Polynomial

The ARNUM package[1] provides facilities for handling algebraic numbers as polynomial coefficients in REDUCE calculations. It includes facilities for introducing indeterminates to represent algebraic numbers, for calculating splitting fields, and for factoring and finding greatest common divisors in such domains.

Algebraic numbers are the solutions of an irreducible polynomial over some ground domain. The algebraic number i (imaginary unit), for example, would be defined by the polynomial $i^2 + 1$. The arithmetic of algebraic number s can be viewed as a polynomial arithmetic modulo the defining polynomial.

Given a defining polynomial for an algebraic number a

$$a^n \ + \ p_{n-1}a^{n-1} \ + \ ... \ + \ p_0$$

All algebraic numbers which can be built up from a are then of the form:

$$r_{n-1}a^{n-1} \ + \ r_{n-2}a^{n-2} \ + \ ... \ + \ r_0$$

where the r_j's are rational numbers.

The operation of addition is defined by

$$(r_{n-1}a^{n-1} \ + \ r_{n-2}a^{n-2} \ + \ ...) \ + \ (s_{n-1}a^{n-1} \ + \ s_{n-2}a^{n-2} \ + \ ...) \ = \\ (r_{n-1} + s_{n-1})a^{n-1} \ + \ (r_{n-2} + s_{n-2})a^{n-2} \ + \ ... \quad (9.1)$$

Multiplication of two algebraic numbers can be performed by normal polynomial multiplication followed by a reduction of the result with the help of the defining polynomial.

[1] This package was written by Eberhard Schrüfer.

$$(r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + ...) \times (s_{n-1}a^{n-1} + s_{n-2}a^{n-2} + ...) =$$
$$r_{n-1}s^{n-1}a^{2n-2} + ... \textbf{ modulo } (a^n + p_{n-1}a^{n-1} + ... + p_0)$$
$$= q_{n-1}a^{n-1} + q_{n-2}a^{n-2} + ... \quad (9.2)$$

Division of two algebraic numbers r and s yields another algebraic number q.

$\frac{r}{s} = q$ or $r = qs$.

The last equation written out explicitly reads

$$\begin{aligned}&(r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + \ldots)\\ &\quad = (q_{n-1}a^{n-1} + q_{n-2}a^{n-2} + \ldots) \times (s_{n-1}a^{n-1} + s_{n-2}a^{n-2} + \ldots)\\ &\qquad \textbf{modulo } (a^n + p_{n-1}a^{n-1} + \ldots)\\ &\quad = (t_{n-1}a^{n-1} + t_{n-2}a^{n-2} + \ldots)\end{aligned}$$

The t_i are linear in the q_j. Equating equal powers of a yields a linear system for the quotient coefficients q_j.

With this, all field operations for the algebraic numbers are available. The translation into algorithms is straightforward. For an implementation we have to decide on a data structure for an algebraic number. We have chosen the representation REDUCE normally uses for polynomials, the so-called standard form. Since our polynomials have in general rational coefficients, we must allow for a rational number domain inside the algebraic number.

⟨ *algebraic number* ⟩ ::=
`:ar:` . ⟨ *univariate polynomial over the rationals* ⟩

⟨ *univariate polynomial over the rationals* ⟩ ::=
⟨ *variable* ⟩ .** ⟨ *ldeg* ⟩ .* ⟨ *rational* ⟩ .+ ⟨ *reductum* ⟩

⟨ *ldeg* ⟩ ::= integer

⟨ *rational* ⟩ ::=
`:rn:` . ⟨ *integer numerator* ⟩ . ⟨ *integer denominator* ⟩ : integer

⟨ *reductum* ⟩ ::= ⟨ *univariate polynomial* ⟩ : ⟨ *rational* ⟩ : nil

This representation allows us to use the REDUCE functions for adding and multiplying polynomials on the tail of the tagged algebraic number. Also, the routines for solving linear equations can easily be used for the calculation of quotients. We are still left with the problem of introducing a particular algebraic number. In the current version this is done by giving the defining polynomial to the statement **defpoly**. The algebraic number sqrt(2), for example, can be introduced by

```
defpoly sqrt2**2 - 2;
```

This statement associates a simplification function for the translation of the variable in the defining polynomial into its tagged internal form and also generates a power reduction rule used by the operations **times** and **quotient** for the reduction of their result modulo the defining polynomial. A basis for the representation of an algebraic number is also set up by the statement. In the working version, the basis is a list of powers of the indeterminate of the defining polynomial up to one less then its degree. Experiments with integral bases, however, have been very encouraging, and these bases might be available in a later version. If the defining polynomial is not monic, it will be made so by an appropriate substitution.

Example 1

```
defpoly sqrt2**2-2;

1/(sqrt2+1);

sqrt2 - 1

(x**2+2*sqrt2*x+2)/(x+sqrt2);

x + sqrt2

on gcd;

(x**3+(sqrt2-2)*x**2-(2*sqrt2+3)*x-3*sqrt2)/(x**2-2);

  2
(x  - 2*x - 3)/(x - sqrt2)

off gcd;

sqrt(x**2-2*sqrt2*x*y+2*y**2);

abs(x - sqrt2*y)
```

Until now we have dealt with only a single algebraic number. In practice this is not sufficient as very often several algebraic numbers appear in an expression. There are two possibilities for handling this: one can use multivariate extensions [Dav81] or one can construct a defining polynomial that contains all specified extensions. This package implements the latter case (the so called primitive representation). The algorithm we use for the construction of the primitive element is the same as given by Trager [Tra76]. In the implementation, multiple extensions can be given

as a list of equations to the statement **defpoly**, which, among other things, adds the new extension to the previously defined one. All algebraic numbers are then expressed in terms of the primitive element.

Example 2

```
defpoly sqrt2**2-2,cbrt5**3-5;

*** defining polynomial for primitive element:

  6       4        3        2
a1  - 6*a1  - 10*a1  + 12*a1  - 60*a1 + 17

sqrt2;

  48     5     45     4    320     3    780     2
------*a1  + ------*a1  - ------*a1  - ------*a1
 1187         1187         1187         1187

    735         1820
 + ------*a1 - ------
    1187        1187

sqrt2**2;

   2
```

We can provide factorization of polynomials over the algebraic number domain by using Trager's algorithm. The polynomial to be factored is first mapped to a polynomial over the integers by computing the norm of the polynomial, which is the resultant with respect to the primitive element of the polynomial and the defining polynomial. After factoring over the integers, the factors over the algebraic number field are recovered by GCD calculations.

Example 3

```
   defpoly a**2-5;

   on factor;

   x**2 + x - 1;

   (x + (1/2*a + 1/2))*(x - (1/2*a - 1/2))
```

We have also incorporated a function `split_field` for the calculation of a primitive element of minimal degree for which a given polynomial splits into linear factors. The algorithm as described in Trager's article is essentially a repeated primitive element calculation.

Example 4

```
split_field(x**3-3*x+7);

*** Splitting field is generated by:

  6        4        2
a2  - 18*a2  + 81*a2  + 1215



         4          2
{1/126*a2  - 5/42*a2  - 1/2*a2 + 2/7,


           4          2
 - (1/63*a2  - 5/21*a2  + 4/7),


        4          2
1/126*a2  - 5/42*a2  + 1/2*a2 + 2/7}


for each j in ws product (x-j);

 3
x  - 3*x + 7
```

A more complete description can be found in [BHPS86].

9.13 Finding Roots

The simplest way to find roots of a univariate polynomial with real or complex coefficients is to call `solve` with the switch `rounded` set to on. For example, the evaluation of

```
on rounded,complex;
solve(x**3+x+5,x);
```

yields the result

```
{x=0.757990113846 + 1.65034755069*i,

 x=0.757990113846 - 1.65034755069*i,

 x= - 1.51598022769}
```

In the following, the independent use of the roots finder is described. It can be used to find some or all of the roots of univariate polynomials with real or complex coefficients, to the accuracy specified by the user.[2]

9.13.1 Root Finding Strategies

For all polynomials handled by the root finding package, strategies of factoring are employed where possible to reduce the amount of required work. These include square-free factoring and separation of complex polynomials into a product of a polynomial with real coefficients and one with complex coefficients. Whenever these succeed, the resulting smaller polynomials are solved separately, except that the root accuracy takes into account the possibility of close roots on different branches. One other strategy used where applicable is the powergcd method of reducing the powers of the initial polynomial by a common factor, and deriving the roots in two stages, as roots of the reduced power polynomial. Again here, the possibility of close roots on different branches is taken into account.

9.13.2 Top Level Functions

The top level functions can be called either as symbolic operators from algebraic mode, or they can be called directly from symbolic mode with symbolic mode arguments. Outputs are expressed in forms that print out correctly in algebraic mode.

Functions that refer to real roots only

Three top level functions refer only to real roots. Each of these functions can receive 1, 2 or 3 arguments.

The first argument is the polynomial p, that can be complex and can have multiple or zero roots. If arg2 and arg3 are not present, all real roots are found. If the additional arguments are present, they restrict the region of consideration.

[2]This code was written by Stanley L. Kameny.

- If arguments are (p,arg2) then arg2 must be `positive` or `negative`. If arg2=`negative` then only negative roots of p are included; similarly, if arg2=`positive` then only positive roots of p are included. Zero roots are excluded.

- If arguments are (p,arg2,arg3) then Arg2 and Arg3 must be r (a real number) or `exclude` r, or a member of the list `positive`, `negative`, `infinity`, `-infinity`. `exclude` r causes the value r to be excluded from the region. The order of the sequence arg2, arg3 is unimportant. Assuming that arg2 $\leq$ arg3 when both are numeric, then

{`-infinity`,`infinity`}	is equivalent to	{} represents all roots;
{arg2,`negative`}	represents	$-\infty < r < arg2$;
{arg2,`positive`}	represents	$arg2 < r < \infty$;

  In each of the following, replacing an *arg* with `exclude` *arg* converts the corresponding inclusive $\leq$ to the exclusive $<$

{arg2,`-infinity`}	represents	$-\infty < r \leq arg2$;
{arg2,`infinity`}	represents	$arg2 \leq r < \infty$;
{arg2,arg3}	represents	$arg2 \leq r \leq arg3$;

- If zero is in the interval the zero root is included.

`realroots` This function finds the real roots of the polynomial p, using the `REALROOT` package to isolate real roots by the method of Sturm sequences, then polishing the root to the desired accuracy. Precision of computation is guaranteed to be sufficient to separate all real roots in the specified region. (cf. `multiroot` for treatment of multiple roots.)

`isolater` This function produces a list of rational intervals, each containing a single real root of the polynomial p, within the specified region, but does not find the roots.

`rlrootno` This function computes the number of real roots of p in the specified region, but does not find the roots.

Functions that return both real and complex roots

`roots p;` This is the main top level function of the roots package. It will find all roots, real and complex, of the polynomial p to an accuracy that is sufficient to separate them and which is a minimum of 6 decimal places. The value returned by `roots` is a list of equations for all roots. In addition, `roots` stores separate lists of real roots and complex roots in the global variables `rootsreal` and `rootscomplex`.

The order of root discovery by `roots` is highly variable from system to system, depending upon very subtle arithmetic differences during the computation. In order to make it easier to compare results obtained on different computers, the output of `roots` is sorted into a standard order: a root with smaller real part precedes a root with larger real part; roots with identical real parts are sorted so that larger imaginary part precedes smaller imaginary part. (This is done so that for complex pairs, the positive imaginary part is seen first.)

However, when a polynomial has been factored (by square-free factoring or by separation into real and complex factors) then the root sorting is applied to each factor separately. This makes the final resulting order less obvious. However it is consistent from system to system.

`roots_at_prec` p; Same as `roots` except that roots values are returned to a minimum of the number of decimal places equal to the current system precision.

`root_val` p; Same as `roots_at_prec`, except that instead of returning a list of equations for the roots, a list of the root value is returned. This is the function that SOLVE calls.

`nearestroot`(p,s); This top level function uses an iterative method to find the root to which the method converges given the initial starting origin s, which can be complex. If there are several roots in the vicinity of s and s is not significantly closer to one root than it is to all others, the convergence could arrive at a root that is not truly the nearest root. This function should therefore be used only when the user is certain that there is only one root in the immediate vicinity of the starting point s.

`firstroot` p; `roots` is called, but only the first root determined by `roots` is computed. Note that this is not in general the first root that would be listed in `roots` output, since the `roots` outputs are sorted into a canonical order. Also, in some difficult root finding cases, the first root computed might be incorrect.

Other top level functions

`getroot`(n,rr); If rr has the form of the output of ROOTS, REALROOTS, or NEARESTROOTS; `GETROOT` returns the rational, real, or complex value of the root equation. An error occurs if $n < 1$ or $n >$ the number of roots in rr.

`mkpoly` rr; This operator can be used to reconstruct a polynomial whose root equation list is rr and whose denominator is 1. Thus one can verify that if $rr := roots\ p$, and $rr1 := roots\ mkpoly\ rr$, then $rr1 = rr$. (This will be

true if `multiroot` and `ratroot` are ON, and `rounded` is off.) However, $mkpoly\ rr - num\ p = 0$ will be true if and only if all roots of p have been computed exactly.

Functions available for diagnostic or instructional use only

`gfnewt`(p,r,cpx); This function will do a single pass through the function `gfnewton` for polynomial p and root r. If cpx=T, then any complex part of the root will be kept, no matter how small.

`gfroot`(p,r,cpx); This function will do a single pass through the function `GFROOTFIND` for polynomial p and root r. If cpx=T, then any complex part of the root will be kept, no matter how small.

9.13.3 Switches Used in Input

The input of polynomials in algebraic mode is sensitive to the switches `complex`, `rounded`, and `adjprec`. The correct choice of input method is important since incorrect choices will result in undesirable truncation or rounding of the input coefficients.

Truncation or rounding may occur if `rounded` is on and one of the following is true:

1. a coefficient is entered in floating point form or rational form.
2. `complex` is on and a coefficient is imaginary or complex.

Therefore, to avoid undesirable truncation or rounding, then:

1. `rounded` should be off and input should be in integer or rational form; or
2. `rounded` can be on if it is acceptable to truncate or round input to the current value of system precision; or both `rounded` and `adjprec` can be on, in which case system precision will be adjusted to accommodate the largest coefficient which is input; or
3. if the input contains complex coefficients with very different magnitude for the real and imaginary parts, then all three switches `rounded`, `adjprec` and `complex` must be on.

integer and complex modes (**off** `rounded`) any real polynomial can be input using integer coefficients of any size; integer or rational coefficients can be

used to input any real or complex polynomial, independent of the setting of the switch `complex`. These are the most versatile input modes, since any real or complex polynomial can be input exactly.

modes rounded and complex-rounded (on `rounded`) polynomials can be input using integer coefficients of any size. Floating point coefficients will be truncated or rounded, to a size dependent upon the system. If complex is on, real coefficients can be input to any precision using integer form, but coefficients of imaginary parts of complex coefficients will be rounded or truncated.

9.13.4 Internal and Output Use of Switches

The REDUCE arithmetic mode switches `rounded` and `complex` control the behavior of the root finding package. These switches are returned in the same state in which they were set initially, (barring catastrophic error).

`complex` The root finding package controls the switch `complex` internally, turning the switch on if it is processing a complex polynomial. For a polynomial with real coefficients, the starting point argument for `nearestroot` can be given in algebraic mode in complex form as rl + im * I and will be handled correctly, independent of the setting of the switch `complex`. Complex roots will be computed and printed correctly regardless of the setting of the switch `complex`. However, if `complex` is off, the imaginary part will print out ahead of the real part, while the reverse order will be obtained if `complex` is on.

`rounded` The root finding package performs computations using the arithmetic mode that is required at the time, which may be integer, Gaussian integer, rounded, or complex rounded. The switch `bftag` is used internally to govern the mode of computation and precision is adjusted whenever necessary. The initial position of switches `rounded` and `complex` are ignored. At output, these switches will emerge in their initial positions.

9.13.5 Root Package Switches

`ratroot` (Default off) If `RATROOT` is on all root equations are output in rational form. Assuming that the mode is `complex` (i.e. `rounded` is off,) the root equations are guaranteed to be able to be input into REDUCE without truncation or rounding errors. (Cf. the function `mkpoly` described above.)

`multiroot` (Default on) Whenever the polynomial has complex coefficients or has real coefficients and has multiple roots, as determined by the Sturm

function, the function `sqfrf` is called automatically to factor the polynomial into square-free factors. If `multiroot` is on, the multiplicity of the roots will be indicated in the output of `roots` or `realroots` by printing the root output repeatedly, according to its multiplicity. If `multiroot` is off, each root will be printed once, and all roots should be normally be distinct. (Two identical roots should not appear. If the initial precision of the computation or the accuracy of the output was insufficient to separate two closely-spaced roots, the program attempts to increase accuracy and/or precision if it detects equal roots. If, however, the initial accuracy specified was too low, and it was not possible to separate the roots, the program will abort.)

`trroot` (Default off) If switch `trroot` is on, trace messages are printed out during the course of root determination, to show the progress of solution.

`rootmsg` (Default off) If switch `rootmsg` is on in addition to switch `trroot`, additional messages are printed out to aid in following the progress of Laguerre and Newton complex iteration. These messages are intended for debugging use primarily.

9.13.6 Operational Parameters and Parameter Setting.

`ROOTACC!#` (Default 6) This parameter can be set using the function `rootacc` n; which causes `rootacc!#` to be set to max(n,6). (If roots are closely spaced, a higher number of significant places is computed where needed.)

system precision The roots package, during its operation, will change the value of system precision but will restore the original value of system precision at termination except that the value of system precision is increased if necessary to allow the full roots output to be printed.

`PRECISION` n; If the user sets system precision, using the command `precision n;` then the effect is to increase the system precision to n, and to have the same effect on `roots` as `rootacc` n; ie. roots will now be printed with minimum accuracy n. The original conditions can then be restored by using the command `PRECISION RESET;` or `PRECISION NIL;`.

`ROOTPREC` n; The roots package normally sets the computation mode and precision automatically. However, if `rootprec` n; is called and n is greater than the initial system precision then all root computation will be done initially using a minimum system precision n. Automatic operation can be restored by input of `rootprec` 0;.

9.13.7 Avoiding truncation of polynomials on input

The roots package will not internally truncate polynomials. However, it is possible that a polynomial can be truncated by input reading functions of the embedding lisp system, particularly when input is given in floating point (rounded) format.

To avoid any difficulties, input can be done in integer or Gaussian integer format, or mixed, with integers or rationals used to represent quantities of high precision. There are many examples of this in the test package. It is usually best to let the roots package determine the precision needed to compute roots.

The number of digits that can be safely represented in floating point in the lisp system are contained in the global variable `!!nfpd`. Similarly, the maximum number of significant figures in floating point output are contained in the global variable `!!flim`. The roots package computes these values, which are needed to control the logic of the program.

The values of intermediate root iterations (that are printed when `TRROOT` is on) are given in bigfloat format even when the actual values are computed in floating point. This avoids intrusive rounding of root printout.

Chapter 10

Assigning and Testing Algebraic Properties

Sometimes algebraic expressions can be further simplified if there is additional information about the value ranges of its components. The following section describes how to inform REDUCE of such assumptions.

10.1 REALVALUED Declaration and Check

The declaration `realvalued` may be used to restrict variables to the real numbers. The syntax is:

```
realvalued v1,...vn;
```

For such variables the operator `impart` gives the result zero. Thus, with

```
realvalued x,y;
```

the expression `impart(x+sin(y))` is evaluated as zero. You may also declare an operator as real valued with the meaning, that this operator maps real arguments always to real values. Example:

```
operator h; realvalued h,x;
impart h(x);

   0

impart h(w);
```

```
impart(h(w))
```

Such declarations are not needed for the standard elementary functions.

To remove the propery from a variable or an operator use the declaration `notrealvalued` with the syntax:

```
notrealvalued v1,...vn;
```

The boolean operator `realvaluedp` allows you to check if a variable, an operator, or an operator expression is known as real valued. Thus,

```
realvalued x;
write if realvaluedp(sin x) then "yes" else "no";
write if realvaluedp(sin z) then "yes" else "no";
```

would print first `yes` and then `no`. For general expressions test the impart for checking the value range:

```
realvalued x,y; w:=(x+i*y); w1:=conj w;
impart(w*w1);

   0

impart(w*w);

   2*x*y
```

10.2 SELFCONJUGATE Declaration

The declaration `selfconjugate` may be used to declare an operator to be self-conjuate in the sense that `conj(f(z)) = f(conj(z))`. The syntax is:

```
 selfconjugate f1,...fn;
```

Such declarations are not needed for the standard elementary functions nor for the inverses `atan, acot, asinh, acsch`. The remaining inverse functions `log, asin, acos, atanh, acosh` etc. and `sqrt` fail to be self-conjugate on their branch cuts (which are all subsets of the real axis).

10.3 Declaring Complex Conjugates

The argument u of a declaration complex_conjugates should consist of one or more (comma-separated) lists of two identifiers. This declaration associates the two identifiers as mutual complex-conjugates. If the first is an operator, the second is also declared as an operator, if it is not one already. A fancy print symbol is automatically constructed and installed for the second identifier from that of the first by adding over-lining. For example:

```
operator f;
complex_conjugates {f, fbar}, {z, zb};
conj zb     -> z
conj(f(z))  -> fbar(zb)
```

This will associate f & fbar and z & zb as mutual complex conjugates and declare fbar as an operator. On graphical interfaces zb and fbar will be rendered as $\overline{z}$ and $\overline{f}$ respectively. If the first identifier already has a *fancy special symbol* defined, this will be over-lined to produce the fancy print symbol for the second identifier. Should the user not wish to have a fancy print symbol automatically generated, they may instead use explicit let statements as described in the subsection on the operator conj.

10.4 Declaring Expressions Positive or Negative

Detailed knowlege about the sign of expressions allows REDUCE to simplify expressions involving exponentials or abs. You can express assumptions about the *positivity* or *negativity* of expressions by rules for the operator sign. Examples:

```
abs(a*b*c);

   abs(a*b*c);

let sign(a)=>1,sign(b)=>1; abs(a*b*c);

   abs(c)*a*b

on precise; sqrt(x^2-2x+1);

   abs(x - 1)

ws where sign(x-1)=>1;
```

```
     x - 1
```

Here factors with known sign are factored out of an `abs` expression.

```
on precise; on factor;

(q*x-2q)^w;

                 w
   ((x - 2)*q)

ws where sign(x-2)=>1;

    w          w
   q *(x - 2)
```

In this case the factor $(x-2)^w$ may be extracted from the base of the exponential because it is known to be positive.

Note that REDUCE knows a lot about sign propagation. For example, with x and y also $x+y$, $x+y+\pi$ and $(x+e)/y^2$ are known as positive. Nevertheless, it is often necessary to declare additionally the sign of a combined expression. E.g. at present a positivity declaration of $x-2$ does not automatically lead to sign evaluation for $x-1$ or for x.

Chapter 11

Substitution Commands

An important class of commands in REDUCE define substitutions for variables and expressions to be made during the evaluation of expressions. Such substitutions use the prefix operator `sub`, various forms of the command `let`, and rule sets.

11.1 SUB Operator

Syntax:

`sub(`⟨*substitution_list*⟩, ⟨*exprn1:algebraic*⟩`)` : *algebraic*

where ⟨*substitution_list*⟩ is a list of one or more equations of the form

⟨*var:kernel*⟩ = ⟨*exprn:algebraic*⟩

or a kernel that evaluates to such a list.

The `sub` operator gives the algebraic result of replacing every occurrence of the variable `var` in the expression `exprn1` by the expression `exprn`. Specifically, `exprn1` is first evaluated using all available rules. Next the substitutions are made, and finally the substituted expression is reevaluated. When more than one variable occurs in the substitution list, the substitution is performed by recursively walking down the tree representing `exprn1`, and replacing every `var` found by the appropriate `exprn`. The `exprn` are not themselves searched for any occurrences of the various `var`s. The trivial case `sub(exprn1)` returns the algebraic value of `exprn1`.

Examples:

```
sub({x=a+y,y=y+1},x^2+y^2);
```

```
 2               2
a  + 2*a*y + 2*y  + 2*y + 1
```

and

```
s := {x=a+y,y=y+1}$

sub(s,x^2+y^2);

 2               2
a  + 2*a*y + 2*y  + 2*y + 1
```

Note that the global assignments `x:=a+y`, etc., do not take place.

`exprn1` can be any valid algebraic expression whose type is such that a substitution process is defined for it (e.g., scalar expressions, lists and matrices). An error will occur if an expression of an invalid type for substitution occurs either in `exprn` or `exprn1`.

The braces around the substitution list may also be omitted, as in:

```
sub(x=a+y,y=y+1,x^2+y^2);

 2               2
a  + 2*a*y + 2*y  + 2*y + 1
```

11.2 LET Rules

Unlike substitutions introduced via `sub`, `let` rules are global in scope and stay in effect until replaced or `clear`ed.

The simplest use of the `let` statement is in the form

`let` ⟨*substitution list*⟩

where ⟨*substitution list*⟩ is a list of rules separated by commas, each of the form:

⟨*variable*⟩ `=>` ⟨*expression*⟩

or

⟨*prefix operator*⟩`(`⟨*argument*⟩, . . . , ⟨*argument*⟩`)` `=>` ⟨*expression*⟩

or

⟨*argument*⟩⟨*infix operator*⟩, . . . , ⟨*argument*⟩ `=>` ⟨*expression*⟩

For example,

```
let {x => y^2,
     h(u,v) => u - v,
     cos(pi/3) => 1/2,
     a*b => c,
     l+m => n,
     w^3 => 2*z - 3,
     z^10 => 0}
```

An equal sign (=) can be used instead of the "replaceby" sign (=>) and the list brackets can be left out if preferred. The above rules could also have been entered as seven separate `let` statements.

After such `let` rules have been input, `x` will always be evaluated as the square of `y`, and so on. This is so even if at the time the `let` rule was input, the variable `y` had a value other than `y`. (In contrast, the assignment `x:=y^2` will set `x` equal to the square of the current value of `y`, which could be quite different.)

The rule `let a*b=>c` means that whenever `a` and `b` are both factors in an expression their product will be replaced by `c`. For example, `a^5*b^7*w` would be replaced by `c^5*b^2*w`.

The rule for `l+m` will not only replace all occurrences of `l+m` by `n`, but will also normally replace `l` by `n-m`, but not `m` by `n-l`. A more complete description of this case is given in Section 11.2.5.

The rule pertaining to `w^3` will apply to any power of `w` greater than or equal to the third.

Note especially the last example, `let z^10=>0`. This declaration means, in effect: ignore the tenth or any higher power of `z`. Such declarations, when appropriate, often speed up a computation to a considerable degree. (See Section 11.4 for more details.)

Any new operators occurring in such `let` rules will be automatically declared `operator` by the system, if the rules are being read from a file. If they are being entered interactively, the system will ask `Declare ... operator? (Y or N)`. Answer `Y` or `N` and hit Return.

In each of these examples, substitutions are only made for the explicit expressions given; i.e., none of the variables may be considered arbitrary in any sense. For example, the command

```
let h(u,v) => u - v;
```

will cause `h(u,v)` to evaluate to `u - v`, but will not affect `h(u,z)` or `h` with any arguments other than precisely the symbols `u,v`.

These simple `let` rules are on the same logical level as assignments made with the := operator. An assignment `x := p+q` cancels a rule `let x => y^2` made earlier, and vice versa.

CAUTION: A recursive rule such as

```
let x => x + 1;
```

is erroneous, since any subsequent evaluation of `x` would lead to a non-terminating chain of substitutions:

```
x -> x + 1
  -> (x + 1) + 1
  -> ((x + 1) + 1) + 1
  -> ...
```

Similarly, coupled substitutions such as

```
let l => m + n, n => l + r;
```

would lead to the same error. As a result, if you try to evaluate an `x`, `l` or `n` defined as above, you will get an error such as

```
x improperly defined in terms of itself
```

Array and matrix elements can appear on the left-hand side of a `let` statement. However, because of their *instant evaluation* property, it is the value of the element

that is substituted for, rather than the element itself. E.g.,

```
array a(5);
a(2) := b;
let a(2) => c;
```

results in `b` being substituted by `c`; the assignment for `a(2)` does not change.

Finally, if an error occurs in any equation in a `let` statement (including generalized statements involving `for all` and `such that`), the remaining rules are not evaluated.

11.2.1 FOR ALL ... LET

If a substitution for all possible values of a given argument of an operator is required, the declaration `for all` may be used. The syntax of such a command is

`for all` ⟨*variable*⟩,...,⟨*variable*⟩ ⟨*let statement*⟩⟨*terminator*⟩

e.g.,

```
for all x,y let h(x,y) => x-y;
for all x let k(x,y) => x^y;
```

The first of these declarations would cause `h(a,b)` to be evaluated as `a-b`, `h(u+v,u+w)` as `v-w`, etc. If the operator symbol `h` is used with more or fewer arguments, not two, the `let` would have no effect, and no error would result.

The second declaration would cause `k(a,y)` to be evaluated as `a^y`, but would have no effect on `k(a,z)` since the rule didn't say `for all y`....

As with simple `let` rules for backward compatibility with earlier versions of REDUCE, an equals sign may be used instead of => in `for all` commands and rule lists (see below).

Where we used `x` and `y` in the examples, any variables could have been used. This use of a variable doesn't affect the value it may have outside the `let` statement. However, you should remember what variables you actually used. If you want to delete the rule subsequently, you must use the same variables in the `clear` command.

It is possible to use more complicated expressions as a template for a `let` statement, as explained in the section on substitutions for general expressions. In nearly all cases, the rule will be accepted, and a consistent application made by the system. However, if there is a sole constant or a sole free variable on the left-hand side

of a rule (e.g., `let 2 => 3` or `for all x let x => 2`), then the system is unable to handle the rule, and the error message

```
Substitution for ... not allowed
```

will be issued. Any variable listed in the `for all` part will have its symbol preceded by an equal sign: `x` in the above example will appear as `=x`. An error will also occur if a variable in the `for all` part is not properly matched on both sides of the `let` equation.

11.2.2 FOR ALL ... SUCH THAT ... LET

If a substitution is desired for more than a single value of a variable in an operator or other expression, but not all values, a conditional form of the `for all ... let` declaration can be used.

Example:

```
for all x such that numberp x and x<0 let h(x)=>0;
```

will cause `h(-5)` to be evaluated as 0, but `h` of a positive integer, or of an argument that is not an integer at all, would not be affected. Any boolean expression can follow the `such that` keywords.

11.2.3 Removing Assignments and Substitution Rules

The user may remove all assignments and substitution rules from any expression by the command `clear`, in the form

`clear` ⟨*expression*⟩, ..., ⟨*expression*⟩⟨*terminator*⟩

e.g.

```
clear x, h(x,y);
```

Because of their *instant evaluation* property, array and matrix elements cannot be cleared with `clear`. For example, if `a` is an array, you must say

```
a(3) := 0;
```

rather than

```
clear a(3);
```

to "clear" element `a(3)`.

On the other hand, a whole array (or matrix) `a` can be cleared by the command `clear a`; This means much more than resetting to 0 all the elements of `a`. The fact that `a` is an array, and what its dimensions are, are forgotten, so `a` can be redefined as another type of object, for example an operator.

If you need to clear a variable whose name must be computed, see the `unset` statement.

The more general types of `let` declarations can also be deleted by using `clear`. Simply repeat the `let` rule to be deleted, using `clear` in place of `let`, and omitting the equal sign and right-hand part. The same dummy variables must be used in the `for all` part, and the boolean expression in the `such that` part must be written the same way. (The placing of blanks doesn't have to be identical.)

Example: The `let` rule

```
for all x such that numberp x and x<0 let h(x)=>0;
```

can be erased by the command

```
for all x such that numberp x and x<0 clear h(x);
```

11.2.4 Overlapping LET Rules

`clear` is not the only way to delete a `let` rule. A new `let` rule identical to the first, but with a different expression after the equal sign, replaces the first. Replacements are also made in other cases where the existing rule would be in conflict with the new rule. For example, a rule for `x^4` would replace a rule for `x^5`. The user should however be cautioned against having several `let` rules in effect that relate to the same expression. No guarantee can be given as to which rules will be applied by REDUCE or in what order. It is best to `clear` an old rule before entering a new related `let` rule.

11.2.5 Substitutions for General Expressions

The examples of substitutions discussed in other sections have involved very simple rules. However, the substitution mechanism used in REDUCE is very general, and can handle arbitrarily complicated rules without difficulty.

The general substitution mechanism used in REDUCE is discussed in [Hea68], and [Hea69]. For the reasons given in these references, REDUCE does not attempt to implement a general pattern matching algorithm. However, the present system uses far more sophisticated techniques than those discussed in the above papers. It is now possible for the rules appearing in arguments of `let` to have the form

⟨*substitution expression*⟩ => ⟨*expression*⟩

where any rule to which a sensible meaning can be assigned is permitted. However, this meaning can vary according to the form of ⟨*substitution expression*⟩. The semantic rules associated with the application of the substitution are completely consistent, but somewhat complicated by the pragmatic need to perform such substitutions as efficiently as possible. The following rules explain how the majority of the cases are handled.

To begin with, the ⟨*substitution expression*⟩ is first partly simplified by collecting like terms and putting identifiers (and kernels) in the system order. However, no substitutions are performed on any part of the expression with the exception of expressions with the *instant evaluation* property, such as array and matrix elements, whose actual values are used. It should also be noted that the system order used is not changeable by the user, even with the `korder` command. Specific cases are then handled as follows:

1. If the resulting simplified rule has a left-hand side that is an identifier, an expression with a top-level algebraic operator or a power, then the rule is added without further change to the appropriate table.

2. If the operator ⋆ appears at the top level of the simplified left-hand side, then any constant arguments in that expression are moved to the right-hand side of the rule. The remaining left-hand side is then added to the appropriate table. For example,

   ```
   let 2*x*y => 3
   ```

   becomes

   ```
   let x*y => 3/2
   ```

   so that `x*y` is added to the product substitution table, and when this rule is applied, the expression `x*y` becomes 3/2, but neither `x` nor `y` by themselves are replaced.

3. If the operators +, − or / appear at the top level of the simplified left-hand side, all but the first term is moved to the right-hand side of the rule. Thus the rules

```
let l+m=>n, x/2=>y, a-b=>c
```

become

```
let l=>n-m, x=>2*y, a=>c+b.
```

One problem that can occur in this case is that if a quantified expression is moved to the right-hand side, a given free variable might no longer appear on the left-hand side, resulting in an error because of the unmatched free variable. E.g.,

```
for all x,y let f(x)+f(y)=>x*y
```

would become

```
for all x,y let f(x)=>x*y-f(y)
```

which no longer has `y` on both sides.

The fact that array and matrix elements are evaluated in the left-hand side of rules can lead to confusion at times. Consider for example the statements

```
array a(5); let x+a(2)=>3; let a(3)=>4;
```

The left-hand side of the first rule will become `x`, and the second 0. Thus the first rule will be instantiated as a substitution for `x`, and the second will result in an error.

The order in which a list of rules is applied is not easily understandable without a detailed knowledge of the system simplification protocol. It is also possible for this order to change from release to release, as improved substitution techniques are implemented. Users should therefore assume that the order of application of rules is arbitrary, and program accordingly.

After a substitution has been made, the expression being evaluated is reexamined in case a new allowed substitution has been generated. This process is continued until no more substitutions can be made.

As mentioned elsewhere, when a substitution expression appears in a product, the substitution is made if that expression divides the product. For example, the rule

```
let a^2*c => 3*z;
```

would cause `a^2*c*x` to be replaced by `3*z*x` and `a^2*c^2` by `3*z*c`. If the substitution is desired only when the substitution expression appears in a product with the explicit powers supplied in the rule, the command `match` should be used instead..

For example,

```
match a^2*c => 3*z;
```

would cause `a^2*c*x` to be replaced by `3*z*x`, but `a^2*c^2` would not be replaced. `match` can also be used with the `for all` constructions described above.

To remove substitution rules of the type discussed in this section, the `clear` command can be used, combined, if necessary, with the same `for all` clause with which the rule was defined, for example:

```
for all x clear log(e^x),e^log(x),cos(w*t+theta(x));
```

Note, however, that the arbitrary variable names in this case *must* be the same as those used in defining the substitution.

11.3 Rule Lists

Rule lists offer an alternative approach to defining substitutions that is different from either `sub` or `let`. In fact, they provide the best features of both, since they have all the capabilities of `let`, but the rules can also be applied locally as is possible with `sub`. In time, they will be used more and more in REDUCE. However, since they are relatively new, much of the REDUCE code you see uses the older constructs.

A rule list is a list of *rules* that have the syntax

⟨*expression*⟩ `=>` ⟨*expression*⟩ (`when` ⟨*boolean expression*⟩)

For example,

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(~n*pi)      => (-1)^n when remainder(n,2)=0}
```

The tilde preceding a variable marks that variable as *free* for that rule, much as a variable in a `for all` clause in a `let` statement. The first occurrence of that variable in each relevant rule must be so marked on input, otherwise inconsistent results can occur. For example, the rule list

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(x)^2        => (1+cos(2x))/2}
```

designed to replace products of cosines, would not be correct, since the second rule would only apply to the explicit argument `x`. Later occurrences in the same rule may also be marked, but this is optional (internally, all such rules are stored

with each relevant variable explicitly marked). The optional `when` clause allows constraints to be placed on the application of the rule, much as the `such that` clause in a `let` statement.

A rule list may be named, for example

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) => (cos(x-y)-cos(x+y))/2,
          cos(~x)^2       => (1+cos(2*x))/2,
          sin(~x)^2       => (1-cos(2*x))/2};
```

Such named rule lists may be inspected as needed. E.g., the command `trig1;` would cause the above list to be printed.

Rule lists may be used in two ways. They can be globally instantiated by means of the command `let`. For example,

```
let trig1;
```

would cause the above list of rules to be globally active from then on until cancelled by the command `clearrules`, as in

```
clearrules trig1;
```

`clearrules` has the syntax

`clearrules` ⟨*rule list*⟩ | ⟨*name of rule list*⟩(,...)

The second way to use rule lists is to invoke them locally by means of a `where` clause. For example

```
cos(a)*cos(b+c)
   where {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2};
```

or

```
cos(a)*sin(b) where trigrules;
```

The syntax of an expression with a `where` clause is:

⟨*expression*⟩ `where` ⟨*rule list*⟩ | ⟨*rule list*⟩(,⟨*rule list*⟩ | ⟨*rule list*⟩ ...)

so the first example above could also be written

```
cos(a)*cos(b+c)
```

```
where cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2;
```

The effect of this construct is that the rule list(s) in the `where` clause only apply to the expression on the left of `where`. They have no effect outside the expression. In particular, they do not affect previously defined `where` clauses or `let` statements. For example, the sequence

```
let a=>2;
a where a=>4;
a;
```

would result in the output

```
4

2
```

Although `where` has a precedence less than any other infix operator, it still binds higher than keywords such as `else`, `then`, `do`, `repeat` and so on. Thus the expression

```
if a=2 then 3 else a+2 where a=>3
```

will parse as

```
if a=2 then 3 else (a+2 where a=>3)
```

`where` may be used to introduce auxiliary variables in symbolic mode expressions, as described in Section 21.4. However, the symbolic mode use has different semantics, so expressions do not carry from one mode to the other.

Compatibility Note: In order to provide compatibility with older versions of rule lists released through the Network Library, it is currently possible to use an equal sign interchangeably with the replacement sign `=>` in rules and `let` statements. However, since this will change in future versions, the replacement sign is preferable in rules and the equal sign in non-rule-based `let` statements. When an equal sign is used in rules a warning

```
** Please use => instead of = in rules
```

will be printed.

Advanced Use of Rule Lists

Some advanced features of the rule list mechanism make it possible to write more complicated rules than those discussed so far, and in many cases to write more compact rule lists. These features are:

- Free operators
- Double slash operator
- Double tilde variables.

A *free operator* in the left hand side of a pattern will match any operator with the same number of arguments. The free operator is written in the same style as a variable. For example, the implementation of the product rule of differentiation can be written as:

```
operator diff, !~f, !~g;

prule := {diff(~f(~x) * ~g(~x),x) =>
             diff(f(x),x) * g(x) + diff(g(x),x) * f(x)};

let prule;

diff(sin(z)*cos(z),z);

         cos(z)*diff(sin(z),z) + diff(cos(z),z)*sin(z)
```

The *double slash operator* may be used as an alternative to a single slash (quotient) in order to match quotients properly. E.g., in the example of the Gamma function above, one can use:

```
gammarule :=
  {gamma(~z)//(~c*gamma(~zz)) =>
        gamma(z)/(c*gamma(zz-1)*zz)
                  when fixp(zz -z) and (zz -z) >0,
   gamma(~z)//gamma(~zz) =>
        gamma(z)/(gamma(zz-1)*zz)
                  when fixp(zz -z) and (zz -z) >0};

let gammarule;

gamma(z)/gamma(z+3);
```

```
          1
----------------------
  3       2
 z  + 6*z  + 11*z + 6
```

The above example suffers from the fact that two rules had to be written in order to perform the required operation. This can be simplified by the use of *double tilde variables*. E.g. the rule list

```
GGrule :=  {
   gamma(~z)//(~~c*gamma(~zz)) =>
        gamma(z)/(c*gamma(zz-1)*zz)
            when fixp(zz -z) and (zz -z) >0};
```

will implement the same operation in a much more compact way. In general, double tilde variables are bound to the neutral element with respect to the operation in which they are used.

Pattern given	Argument used	Binding
~z + ~~y	x	z=x; y=0
~z + ~~y	x+3	z=x; y=3 or z=3; y=x
~z * ~~y	x	z=x; y=1
~z * ~~y	x*3	z=x; y=3 or z=3; y=x
~z / ~~y	x	z=x; y=1
~z / ~~y	x/3	z=x; y=3

Remarks: A double tilde variable as the numerator of a pattern is not allowed. Also, using double tilde variables may lead to recursion errors when the zero case is not handled properly.

```
let f(~~a * ~x,x)  => a * f(x,x) when freeof (a,x);

f(z,z);

***** f(z,z) improperly defined in terms of itself

% BUT:

let ff(~~a * ~x,x)
       => a * ff(x,x) when freeof (a,x) and a neq 1;
```

```
ff(z,z);
                    ff(z,z)

ff(3*z,z);
                    3*ff(z,z)
```

Displaying Rules Associated with an Operator

The operator `showrules` takes a single identifier as argument, and returns in rule-list form the operator rules associated with that argument. For example:

```
showrules log;

                       1
{df(log(~x),~x) => ---,
                       x

          ~x
 df(log(----),~z) => df(log(x),z) - df(log(y),z)}
          ~y
```

Such rules can then be manipulated further as with any list. For example `rhs first ws;` has the value `1`. Note that an operator may have other properties that cannot be displayed in such a form, such as the fact it is an odd function, or has a definition defined as a procedure.

Order of Application of Rules

If rules have overlapping domains, their order of application is important. In general, it is very difficult to specify this order precisely, so that it is best to assume that the order is arbitrary. However, if only one operator is involved, the order of application of the rules for this operator can be determined from the following:

1. Rules containing at least one free variable apply before all rules without free variables.
2. Rules activated in the most recent `let` command are applied first.
3. `let` with several entries generate the same order of application as a corresponding sequence of commands with one rule or rule set each.
4. Within a rule set, the rules containing at least one free variable are applied in their given order. In other words, the first member of the list is applied first.

5. Consistent with the first item, any rule in a rule list that contains no free variables is applied after all rules containing free variables.

Example: The following rule set enables the computation of exact values of the Gamma function:

```
operator gamma,gamma_error;
gamma_rules :=
{gamma(~x)=>sqrt(pi)/2 when x=1/2,
 gamma(~n)=>factorial(n-1) when fixp n and n>0,
 gamma(~n)=>gamma_error(n) when fixp n,
 gamma(~x)=>(x-1)*gamma(x-1) when fixp(2*x) and x>1,
 gamma(~x)=>gamma(x+1)/x when fixp(2*x)};
```

Here, rule by rule, cases of known or definitely uncomputable values are sorted out; e.g. the rule leading to the error expression will be applied for negative integers only, since the positive integers are caught by the preceding rule, and the last rule will apply for negative odd multiples of $1/2$ only. Alternatively the first rule could have been written as

```
gamma(1/2) => sqrt(pi)/2,
```

but then the case $x = 1/2$ should be excluded in the `when` part of the last rule explicitly because a rule without free variables cannot take precedence over the other rules.

11.4 Asymptotic Commands

In expansions of polynomials involving variables that are known to be small, it is often desirable to throw away all powers of these variables beyond a certain point to avoid unnecessary computation. The command `let` may be used to do this. For example, if only powers of `x` up to `x^7` are needed, the command

```
let x^8 => 0;
```

will cause the system to delete all powers of `x` higher than 7.

CAUTION: This particular simplification works differently from most substitution mechanisms in REDUCE in that it is applied during polynomial manipulation rather than to the whole evaluated expression. Thus, with the above rule in effect, `x^10/x^5` would give the result zero, since the numerator would simplify to zero. Similarly `x^20/x^10` would give a `Zero divisor` error message, since both numerator and denominator would first simplify to zero.

The method just described is not adequate when expressions involve several variables having different degrees of smallness. In this case, it is necessary to supply an asymptotic weight to each variable and count up the total weight of each product in an expanded expression before deciding whether to keep the term or not. There are two associated commands in the system to permit this type of asymptotic constraint. The command `weight` takes a list of equations of the form

⟨*kernel form*⟩ = ⟨*number*⟩

where ⟨*number*⟩ must be a positive integer (not just evaluate to a positive integer). This command assigns the weight ⟨*number*⟩ to the relevant kernel form. A check is then made in all algebraic evaluations to see if the total weight of the term is greater than the weight level assigned to the calculation. If it is, the term is deleted. To compute the total weight of a product, the individual weights of each kernel form are multiplied by their corresponding powers and then added.

The weight level of the system is initially set to 1. The user may change this setting by the command

`wtlevel` ⟨*number*⟩`;`

which sets ⟨*number*⟩ as the new weight level of the system. ⟨*number*⟩ must evaluate to a positive integer. `wtlevel` will also allow `nil` as an argument, in which case the current weight level is returned.

Chapter 12

File Handling Commands

In many applications, it is desirable to load previously prepared REDUCE files into the system, or to write output on other files. REDUCE offers five main commands for this purpose, namely, `in`, `out`, `shut`, `load`, and `load_package`. The first three are described here; `load` and `load_package` are discussed in Section 23.2.

12.1 IN Command

This command takes a list of file names as argument and directs the system to input each file (that should contain REDUCE statements and commands) into the system. File names can either be an identifier or a string. The explicit format of these will be system dependent and, in many cases, site dependent. The explicit instructions for the implementation being used should therefore be consulted for further details. For example:

```
in f1,"ggg.rr.s";
```

will first load file `f1`, then `ggg.rr.s`. When a semicolon is used as the terminator of the `in` statement, the statements in the file are echoed on the terminal or written on the current output file. If `$` is used as the terminator, the input is not shown. Echoing of all or part of the input file can be prevented, even if a semicolon was used, by placing an `off echo;` command in the input file.

Files to be read using `in` should end with `;end;`. Note the two semicolons! First of all, this is protection against obscure difficulties the user will have if there are, by mistake, more `begin`s than `end`s on the file. Secondly, it triggers some file control book-keeping which may improve system efficiency. If `end` is omitted, an error message `"End-of-file read"` will occur.

While a file is being loaded, the special identifier `!__line__` is replaced by the number of the current line in the file currently being read. Similarly, `!__file__` is replaced by the name of the file currently being read.

12.2 IN_TEX Command

This is a variant of the `in` command. Its purpose is to document a REDUCE session by interspersing a LaTeX document with REDUCE commands to be executed.

When a file is input into REDUCE with this command, every line is simply echoed to the output except those enclosed by `\begin{reduce}...\end{reduce}`, which are processed as usual.

The effect is to produce a LaTeX document with REDUCE output.

12.3 OUT Command

This command takes a single file name as argument, and directs output to that file from then on, until another `out` changes the output file, or `shut` closes it. Output can go to only one file at a time, although many can be open. If the file has previously been used for output during the current job, and not `shut`, the new output is appended to the end of the file. Any existing file is erased before its first use for output in a job, or if it had been `shut` before the new `out`.

To output on the terminal without closing the output file, the reserved file name `t` (for terminal) may be used. For example, `out ofile;` will direct output to the file `ofile` and `out t;` will direct output to the user's terminal.

The output sent to the file will be in the same form that it would have on the terminal. In particular `x^2` would appear on two lines, an `x` on the lower line and a 2 on the line above. If the purpose of the output file is to save results to be read in later, this is not an appropriate form. We first must turn off the `nat` switch that specifies that output should be in standard mathematical notation.

Example: To create a file `abcd` from which it will be possible to read – using `in` – the value of the expression `xyz`:

```
off echo$       % needed if your input is from a file.
off nat$        % output in IN-readable form. Each
                % expression printed will end with a $ .
out abcd$       % output to new file
linelength 72$ % for systems with fixed input
                % line length.
xyz:=xyz;       % will output "xyz := " followed by
```

```
                % the value of xyz
write ";end"$   % standard for ending files for in
shut abcd$      % save abcd, return to terminal output
on nat$                 % restore usual output form
```

12.4 SHUT Command

This command takes a list of names of files that have been previously opened via an `out` statement and closes them. Most systems require this action by the user before he ends the REDUCE job (if not sooner), otherwise the output may be lost. If a file is shut and a further `out` command issued for the same file, the file is erased before the new output is written.

If it is the current output file that is shut, output will switch to the terminal. Attempts to shut files that have not been opened by `out`, or an input file, will lead to errors.

12.5 Using Variables as Filenames

The commands `in`, `out` and `shut` treat their arguments as constants, but you can use variables if you make them into group expressions by enclosing them between << and >> symbols, e.g.

```
filename := "something/long/and/complicated.red";
out <<filename>>;
% perform some computation...
shut <<filename>>;
```

12.6 Accessing the Operating System

REDUCE provides limited access to the operating system via the operator `system`, which takes one argument that must evaluate to a string. It passes the content of this string to the default shell and returns a number, which will be 0 if it succeeds and non-zero if it fails. The output from the shell is displayed on the default output device, which is normally the terminal. There is no straightforward way to process this output within REDUCE. The operator `system` can be useful for operations such as copying, moving, renaming and deleting files, or for listing a directory interactively. It is used within a few of the REDUCE packages, such as GNUPLOT.

The correct shell syntax to use with `system` depends on your operating system.

On Microsoft Windows, `system` invokes the `cmd.exe` shell (not `PowerShell`); on other operating systems it probably invokes `bash`. Note that backslash (`\`) is not an escape character in REDUCE and so can be included in the argument to `system` without any special precautions. However, to include a double-quote character (`"`) within a REDUCE string it must be doubled.

For example, on Windows, you could use the following to delete a file called "`C:\long dir name\file.red`"

```
system "del ""C:\long dir name\file.red""";
```

whereas, on most other platforms, you could use the following to delete a file called "`/long dir name/file.red`"

```
system "rm '/long dir name/file.red'";
```

More sophisticated use of `system` is possible, but requires symbolic-mode programming. See the REDUCE source code for examples.

12.7 REDUCE Startup File

At the start of a REDUCE session, the system checks for the existence of a user's startup file, and executes the REDUCE statements in it. This is equivalent to inputting the file with the `in` command.

To find the directory/folder where the file resides, the system checks the existence of the following environment variables:

1. `HOME`,
2. `HOMEDRIVE` and `HOMEPATH` together (Windows).

If none of these are set, the current directory is used. The file itself must be named either `.reducerc` or `reduce.rc`[1].

[1] If none of these exist, the system checks for a file called `reduce.INI` in the current directory. This is historical and may be removed in future.

Chapter 13

Commands for Interactive Use

REDUCE is designed as an interactive system, but naturally it can also operate in a batch processing or background mode by taking its input command by command from the relevant input stream. There is a basic difference, however, between interactive and batch use of the system. In the former case, whenever the system discovers an ambiguity at some point in a calculation, such as a forgotten type assignment for instance, it asks you for the correct interpretation. In batch operation, it is not practical to terminate the calculation at such points and require resubmission of the job, so the system makes the most obvious guess of your intentions and continues the calculation.

13.1 Error Handling: `errcont`, `retry`

There is also a difference in the handling of errors. In the former case, the computation can continue since you have the opportunity to correct the mistake. In batch mode, the error may lead to consequent erroneous (and possibly time consuming) computations. So in the default case, no further evaluation occurs, although the remainder of the input is checked for syntax errors. A message `"Continuing with parsing only"` informs you that this is happening. On the other hand, the switch `errcont`, if on, will cause the system to continue evaluating expressions after such errors occur.

When a syntactical error occurs, the place where the system detected the error is marked with three dollar signs (`$$$`). In interactive mode, you can then use `ed` to correct the error, or retype the command. When a non-syntactical error occurs in interactive mode, the command being evaluated at the time the last error occurred is saved, and may later be reevaluated by the command `retry`.

13.2 Referencing Previous Results: `input`, `ws`, `display`

It is often useful to be able to reference results of previous computations during a REDUCE session; see also 8.2. For this purpose, REDUCE maintains a history of all interactive inputs and the results of all interactive computations during a given session. These results are referenced by the command number that REDUCE prints automatically in interactive mode. To use a previous input expression in a new computation, write `input(`n`)`, where n is the command number. To use a previous output expression, write `ws(`n`)` (where WS stands for WorkSpace). `ws` used as a variable (rather than a function) references the previous output expression. For example:

```
1: int(x-1, x);

 x*(x - 2)
-----------
     2

:

7: (x^2-1)/(x+1);

x - 1

:

15: 2*input(1)-ws(7)^2;

-1

16: 2*ws(1)-ws(7)^2;

-1

17: x := 101;

x := 101

18: ws(7);

100
```

Inputs 15 and 16 above yield the same result, but input 16 does so *without* re-

computing the integral. However, an output expression referenced using `ws` is re-evaluated in the current context, as shown by the last two statements above.

Note that input that causes an error, and some commands such as `let` statements, file handling and mode changing, do not produce an output expression, so the output from such input cannot be accessed. `ws` used as a variable returns the last output expression, which does not necessarily correspond to the last input, and `ws` used as a function reports an error if you try to access non-existent output. For example:

```
1: 6*7;

42

2: 0/0;

***** 0/0 formed

3: ws;

42

4: ws 2;

***** Entry 2 not found

5: let x => 0;

6: ws;

42

7: ws 5;

***** Entry 5 not found
```

The operator `display` is available to display previous inputs. If its argument is a positive integer, n say, then the previous n inputs are displayed. If its argument is `all` (or in fact any non-numerical expression), then all previous inputs are displayed.

13.3 Interactive Editing: `ed`, `editdef`

It is possible when working interactively to edit any REDUCE input that comes from your terminal, and also some user-defined procedure definitions. At the top level, you can access any previous command string by the command `ed(`n`)`, where n is the desired command number as prompted by the system in interactive mode. The command `ed` (with no argument) accesses the previous command.

After `ed` has been called, you can now edit the displayed string using a string editor with the following commands:

`b`	move pointer to beginning
`c`⟨*character*⟩	replace next character by ⟨*character*⟩
`d`	delete next character
`e`	end editing and reread text
`f`⟨*character*⟩	move pointer to next occurrence of ⟨*character*⟩
`i`⟨*string*⟩⟨*escape*⟩	insert ⟨*string*⟩ in front of pointer
`k`⟨*character*⟩	delete all characters until ⟨*character*⟩
`p`	print string from current pointer
`q`	give up with error exit
`s`⟨*string*⟩⟨*escape*⟩	search for first occurrence of ⟨*string*⟩, positioning pointer just before it
`space` or `x`	move pointer right one character.

The above table can be displayed online by typing a question mark followed by a carriage return to the editor. The editor prompts with an angle bracket. Commands can be combined on a single line, and all command sequences must be followed by a carriage return to become effective.

Thus, to change the command `x := a+1;` to `x := a+2;` and cause it to be executed, the following edit command sequence could be used:

```
f1c2e<return>
```

You can also use the interactive editor to edit a user-defined procedure that has not been compiled. To do this, use:

`editdef` ⟨*id*⟩`;`

where ⟨*id*⟩ is the name of the procedure. The procedure definition will then be displayed in editing mode, and may then be edited and redefined on exiting from the editor.

Some versions of REDUCE include input editing that uses the capabilities of modern window systems. Please consult your system dependent documentation to see

if this is possible. Such editing techniques are usually much easier to use then `ed` or `editdef`.

13.4 Interactive File Control: `int`, `pause`, `cont`

If input is coming from an external file, the system treats it as a batch processed calculation. If you desire interactive response in this case, you can include the command `on int`; in the file. Likewise, you can issue the command `off int`; in the main program if you do not desire continual questioning from the system. Regardless of the setting of the switch `int`, input commands from a file are not kept in the system, and so cannot be referenced using `input` or `ws`, or edited using `ed`. However, an implementation of REDUCE may provide a link to an external system editor that can be used for such editing. The specific instructions for the particular implementation should be consulted for information on this.

Two commands are available in REDUCE for interactive use of files. `pause`; may be inserted at any point in an input file. When this command is encountered on input, the system prints the message `Cont?  (Y or N)` on your terminal and halts. If you respond `y` (for yes), the calculation continues from that point in the file. If you respond `n` (for no), control is returned to the terminal, and you can input further statements and commands. Later on you can use the command `cont;` to transfer control back to the point in the file following the last `pause;` encountered. A top-level `pause;` from the terminal has no effect.

Chapter 14

Matrix Calculations

A very powerful feature of REDUCE is the ease with which matrix calculations can be performed. To extend our syntax to this class of calculations we need to add another prefix operator, `mat`, and a further variable and expression type as follows:

14.1 MAT Operator

This prefix operator is used to represent $n \times m$ matrices. `mat` has n arguments interpreted as rows of the matrix, each of which is a list of m expressions representing elements in that row. For example, the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

would be written as `mat((a,b,c),(d,e,f))`.

Note that the single column matrix

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

becomes `mat((x),(y))`. The inside parentheses are required to distinguish it from the single row matrix

$$\begin{pmatrix} x & y \end{pmatrix}$$

that would be written as `mat((x,y))`.

14.2 Matrix Variables

An identifier may be declared a matrix variable by the declaration `matrix`. The size of the matrix may be declared explicitly in the matrix declaration, or by default in assigning such a variable to a matrix expression. For example,

```
matrix x(2,1),y(3,4),z;
```

declares `x` to be a 2 x 1 (column) matrix, `y` to be a 3 x 4 matrix and `z` a matrix whose size is to be declared later.

Matrix declarations can appear anywhere in a program. Once a symbol is declared to name a matrix, it can not also be used to name an array, operator or a procedure, or used as an ordinary variable. It can however be redeclared to be a matrix, and its size may be changed at that time. Note however that matrices once declared are *global* in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the matrix to that block, nor does the matrix go away on exiting the block (use `clear` instead for this purpose). An element of a matrix is referred to in the expected manner; thus `x(1,1)` gives the first element of the matrix `x` defined above. References to elements of a matrix whose size has not yet been declared leads to an error. All elements of a matrix whose size is declared are initialized to 0. As a result, a matrix element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used to name the matrix elements as in:

```
matrix m; operator x;  m := mat((x(1,1),x(1,2));
```

14.3 Matrix Expressions

These follow the normal rules of matrix algebra as defined by the following syntax:

⟨*matrix expression*⟩ ⟶ `mat`⟨*matrix description*⟩ | ⟨*matrix variable*⟩ |
⟨*scalar expression*⟩ ∗ ⟨*matrix expression*⟩ |
⟨*matrix expression*⟩ ∗ ⟨*matrix expression*⟩ |
⟨*matrix expression*⟩ + ⟨*matrix expression*⟩ |
⟨*matrix expression*⟩ `^` ⟨*integer*⟩ |
⟨*matrix expression*⟩ / ⟨*matrix expression*⟩

Sums and products of matrix expressions must be of compatible size; otherwise an error will result during their evaluation. Similarly, only square matrices may be raised to a power. A negative power is computed as the inverse of the matrix raised to the corresponding positive power. `a/b` is interpreted as `a*b^(-1)`.

Examples:

Assuming `x` and `y` have been declared as matrices, the following are matrix expressions

```
y
```

```
y^2*x-3*y^(-2)*x
y + mat((1,a),(b,c))/2
```

The computation of the quotient of two matrices normally uses a two-step elimination method due to Bareiss. An alternative method using Cramer's method is also available. This is usually less efficient than the Bareiss method unless the matrices are large and dense, although we have no solid statistics on this as yet. To use Cramer's method instead, the switch `cramer` should be turned on.

14.4 Operators with Matrix Arguments

The operator `length` applied to a matrix returns a list of the number of rows and columns in the matrix. Other operators useful in matrix calculations are defined in the following subsections. Attention is also drawn to the LINALG (section 20.33) and NORMFORM (section 20.40) packages.

14.4.1 DET Operator

Syntax:

`det(`⟨*exprn:matrix_expression*⟩`)` : *algebraic* .

The operator `det` is used to represent the determinant of a square matrix expression. E.g.,

```
det(y^2)
```

is a scalar expression whose value is the determinant of the square of the matrix `Y`, and

```
det mat((a,b,c),(d,e,f),(g,h,j));
```

is a scalar expression whose value is the determinant of the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$$

Determinant expressions have the *instant evaluation* property. In other words, the statement

```
let det mat((a,b),(c,d)) = 2;
```

sets the *value* of the determinant to 2, and does not set up a rule for the determinant itself.

14.4.2 MATEIGEN Operator

Syntax:

`mateigen(`⟨*exprn:matrix_expression,id*⟩`)` : ⟨*list*⟩ .

`mateigen` calculates the eigenvalue equation and the corresponding eigenvectors of a matrix, using the variable `id` to denote the eigenvalue. A square free decomposition of the characteristic polynomial is carried out. The result is a list of lists of 3 elements, where the first element is a square free factor of the characteristic polynomial, the second its multiplicity and the third the corresponding eigenvector (as an n by 1 matrix). If the square free decomposition was successful, the product of the first elements in the lists is the minimal polynomial. In the case of degeneracy, several eigenvectors can exist for the same eigenvalue, which manifests itself in the appearance of more than one arbitrary variable in the eigenvector. To extract the various parts of the result use the operations defined on lists.

Example: The command

```
mateigen(mat((2,-1,1),(0,1,1),(-1,1,1)),eta);
```

gives the output

```
{{eta - 1,2,

  [arbcomplex(1)]
  [              ]
  [arbcomplex(1)]
  [              ]
  [      0       ]

  },

 {eta - 2,1,

  [      0       ]
  [              ]
  [arbcomplex(2)]
  [              ]
  [arbcomplex(2)]
```

```
}}
```

14.4.3 TP Operator

Syntax:

`tp(`⟨*exprn:matrix_expression*⟩`)` : ⟨*matrix*⟩ .

This operator takes a single matrix argument and returns its transpose.

14.4.4 Trace Operator

Syntax:

`trace(`⟨*exprn:matrix_expression*⟩`)` : ⟨*algebraic*⟩.

The operator `trace` is used to represent the trace of a square matrix.

14.4.5 Matrix Cofactors

Syntax:

`cofactor(`⟨*exprn:matrix_expression*⟩`,`⟨*row:integer*⟩`,`⟨*column:integer*⟩`)` : algebraic

The operator `cofactor` returns the cofactor of the element in row `row` and column `column` of the matrix `matrix`. Errors occur if `row` or `column` do not simplify to integer expressions or if `matrix` is not square.

14.4.6 NULLSPACE Operator

Syntax:

`nullspace(`⟨*exprn:matrix_expression*⟩`)` : ⟨*list*⟩

`nullspace` calculates for a matrix A a list of linear independent vectors (a basis) whose linear combinations satisfy the equation $Ax = 0$. The basis is provided in a form such that as many upper components as possible are isolated.

Note that with `b := nullspace a` the expression `length b` is the *nullity* of A, and that `second length a - length b` calculates the *rank* of A. The

rank of a matrix expression can also be found more directly by the `rank` operator described below.

Example: The command

```
nullspace mat((1,2,3,4),(5,6,7,8));
```

gives the output

```
{
 [ 1   ]
 [     ]
 [ 0   ]
 [     ]
 [ - 3]
 [     ]
 [ 2   ]
,
 [ 0   ]
 [     ]
 [ 1   ]
 [     ]
 [ - 2]
 [     ]
 [ 1   ]
 }
```

In addition to the REDUCE matrix form, `nullspace` accepts as input a matrix given as a list of lists, that is interpreted as a row matrix. If that form of input is chosen, the vectors in the result will be represented by lists as well. This additional input syntax facilitates the use of `nullspace` in applications different from classical linear algebra.

14.4.7 RANK Operator

Syntax:

`rank` (⟨*exprn:matrix_expression*⟩) : ⟨*integer*⟩ .

`rank` calculates the rank of its argument, that, like `nullspace` can either be a standard matrix expression, or a list of lists, that can be interpreted either as a row matrix or a set of equations.

`Example:`

```
rank mat((a,b,c),(d,e,f));
```

returns the value 2.

14.5 Matrix Assignments

Matrix expressions may appear in the right-hand side of assignment statements. If the left-hand side of the assignment, which must be a variable, has not already been declared a matrix, it is declared by default to the size of the right-hand side. The variable is then set to the value of the right-hand side.

Such an assignment may be used very conveniently to find the solution of a set of linear equations. For example, to find the solution of the following set of equations

```
a11*x(1) + a12*x(2) = y1
a21*x(1) + a22*x(2) = y2
```

we simply write

```
x := 1/mat((a11,a12),(a21,a22))*mat((y1),(y2));
```

14.6 Evaluating Matrix Elements

Once an element of a matrix has been assigned, it may be referred to in standard array element notation. Thus `y(2,1)` refers to the element in the second row and first column of the matrix `y`.

The easiest way to access an element of a matrix-valued expression is to assign the expression to a variable and then use the syntax described above. Another way is to use the `part` operator: if *M* is either a matrix or a matrix-valued expression then `part(`*M*`, `*i*`)` evaluates to row *i represented as a list*, and (hence) `part(`*M*`, `*i*`, `*j*`)` evaluates to the matrix element in row *i* and column *j*.

Chapter 15

Procedures

It is often useful to name a statement for repeated use in calculations with varying parameters, or to define a complete evaluation procedure for an operator. REDUCE offers a procedural declaration for this purpose. Its general syntax is:

[⟨*procedural type*⟩] `procedure` ⟨*name*⟩[⟨*varlist*⟩]`;` ⟨*statement*⟩`;`

where

⟨*varlist*⟩ ⟶ `(`⟨*variable*⟩`, ..., `⟨*variable*⟩`)`

This will be explained more fully in the following sections.

In the algebraic mode of REDUCE the ⟨*procedural type*⟩ can be omitted, since the default is `algebraic`. Procedures of type `integer` or `real` may also be used. In the former case, the system checks that the value of the procedure is an integer. At present, such checking is not done for a real procedure, although this will change in the future when a more complete type checking mechanism is installed. Users should therefore only use these types when appropriate. An empty variable list may also be omitted.

All user-defined procedures are automatically declared to be operators.

In order to allow users relatively easy access to the whole REDUCE source program, system procedures are not protected against user redefinition. If a procedure is redefined, a message

```
*** <procedure name> redefined
```

is printed. If this occurs, and the user is not redefining his own procedure, he is well advised to rename it, and possibly start over (because he has *already* redefined some internal procedure whose correct functioning may be required for his job!)

All required procedures should be defined at the top level, since they have global scope throughout a program. In particular, an attempt to define a procedure within a procedure will cause an error to occur.

15.1 Procedure Heading

Each procedure has a heading consisting of the word `procedure` (optionally preceded by the word `algebraic`), followed by the name of the procedure to be defined, and followed by its formal parameters – the symbols that will be used in the body of the definition to illustrate what is to be done. There are three cases:

1. No parameters. Simply follow the procedure name with a terminator (semicolon or dollar sign).

   ```
   procedure abc;
   ```

   When such a procedure is used in an expression or command, `abc()`, with empty parentheses, must be written.

2. One parameter. Enclose it in parentheses *or* just leave at least one space, then follow with a terminator.

   ```
   procedure abc(x);
   ```

   or

   ```
   procedure abc x;
   ```

3. More than one parameter. Enclose them in parentheses, separated by commas, then follow with a terminator.

   ```
   procedure abc(x,y,z);
   ```

Referring to the last example, if later in some expression being evaluated the symbols `abc(u,p*q,123)` appear, the operations of the procedure body will be carried out as if `x` had the same value as `u` does, `y` the same value as `p*q` does, and `z` the value 123. The values of `x`, `y`, `z`, after the procedure body operations are completed are unchanged. So, normally, are the values of `u`, `p`, `q`, and (of course) 123. (This is technically referred to as call by value.)

The reader will have noted the word *normally* a few lines earlier. The call by value protections can be bypassed if necessary, as described elsewhere.

15.2 Procedure Body

Following the delimiter that ends the procedure heading must be a *single* statement defining the action to be performed or the value to be delivered. A terminator must follow the statement. If it is a semicolon, the name of the procedure just defined is printed. It is not printed if a dollar sign is used.

If the result wanted is given by a formula of some kind, the body is just that formula, using the variables in the procedure heading.

Simple Example:

If `f(x)` is to mean `(x+5)*(x+6)/(x+7)`, the entire procedure definition could read

```
procedure f x; (x+5)*(x+6)/(x+7);
```

Then `f(10)` would evaluate to 240/17, `f(a-6)` to `a*(a-1)/(a+1)`, and so on.

More Complicated Example:

Suppose we need a function `p(n,x)` that, for any positive integer `n`, is the Legendre polynomial of order n. We can define this operator using the textbook formula defining these functions:

$$p_n(x) = \frac{1}{n!}\frac{d^n}{dy^n}\left.\frac{1}{(y^2-2xy+1)^{\frac{1}{2}}}\right|_{y=0}$$

Put into words, the Legendre polynomial $p_n(x)$ is the result of substituting $y = 0$ in the n^{th} partial derivative with respect to y of a certain fraction involving x and y, then dividing that by $n!$.

This verbal formula can easily be written in REDUCE:

```
procedure p(n,x);
   sub(y=0,df(1/(y^2-2*x*y+1)^(1/2),y,n))
       /(for i:=1:n product i);
```

Having input this definition, the expression evaluation

```
2p(2,w);
```

would result in the output

```
   2
3*w  - 1 .
```

If the desired process is best described as a series of steps, then a group or compound statement can be used.

Example:

The above Legendre polynomial example can be rewritten as a series of steps instead of a single formula as follows:

```
procedure p(n,x);
  begin scalar seed,deriv,top,fact;
       seed:=1/(y^2 - 2*x*y +1)^(1/2);
       deriv:=df(seed,y,n);
       top:=sub(y=0,deriv);
       fact:=for i:=1:n product i;
       return top/fact
  end;
```

Procedures may also be defined recursively. In other words, the procedure body can include references to the procedure name itself, or to other procedures that themselves reference the given procedure. As an example, we can define the Legendre polynomial through its standard recurrence relation:

```
procedure p(n,x);
   if n<0 then rederr "Invalid argument to P(N,X)"
    else if n=0 then 1
    else if n=1 then x
    else ((2*n-1)*x*p(n-1,x)-(n-1)*p(n-2,x))/n;
```

The operator `rederr` in the above example provides for a simple error exit from an algebraic procedure (and also a block). It can take a string as argument.

It should be noted however that all the above definitions of `p(n,x)` are quite inefficient if extensive use is to be made of such polynomials, since each call effectively recomputes all lower order polynomials. It would be better to store these expressions in an array, and then use say the recurrence relation to compute only those polynomials that have not already been derived. We leave it as an exercise for the reader to write such a definition.

15.3 Matrix- and List-valued Procedures

Normally, procedures can only return scalar values. In order for a procedure to return a matrix, it has to be declared of type **matrixproc**:

```
matrixproc SkewSym1 (w);
```

```
mat((0,-w(3,1),w(2,1)),
    (w(3,1),0,-w(1,1)),
    (-w(2,1), w(1,1), 0));
```

Following this declaration, the call to `SkewSym1` can be used as a matrix, e.g.

```
X := SkewSym1(mat((qx),(qy),(qz)));
```

```
      [  0     - qz    qy  ]
      [                    ]
x := [ qz       0     - qx]
      [                    ]
      [ - qy    qx      0  ]
```

```
X * mat((rx),(ry),(rz));
```

```
[ qy*rz - qz*ry  ]
[                ]
[ - qx*rz + qz*rx]
[                ]
[ qx*ry - qy*rx  ]
```

Similarly, by using the keyword **listproc**, an algebraic procedure can be declared to return a list. For example, the following procedure returns a normalized version of the vector provided as its argument, represented as a list (i.e. the returned vector has unit Euclidean norm):

```
listproc normalize v;
   begin scalar n := sqrt for each vi in v sum vi^2;
      return for each vi in v collect vi/n
   end;
```

(Note that the LISTVECOPS package provides elegant vector operations on lists, which allow the above procedure to be written much more succinctly; see the version at the end of the LISTVECOPS section.)

15.4 Using LET Inside Procedures

By using `let` instead of an assignment in the procedure body it is possible to bypass the call-by-value protection. If `x` is a formal parameter or local variable of the procedure (i.e. is in the heading or in a local declaration), and `let` is used

instead of := to make an assignment to x, e.g.

```
let x => 123;
```

then it is the variable that is the value of x that is changed. This effect also occurs with local variables defined in a block. If the value of x is not a variable, but a more general expression, then it is that expression that is used on the left-hand side of the let statement. For example, if x had the value p*q, it is as if let p*q => 123 had been executed.

15.5 LET Rules as Procedures

The let statement offers an alternative syntax and semantics for procedure definition.

In place of

```
procedure abc(x,y,z); <procedure body>;
```

one can write

```
for all x,y,z let abc(x,y,z) => <procedure body>;
```

There are several differences to note.

If the procedure body contains an assignment to one of the formal parameters, e.g.

```
x := 123;
```

in the procedure case it is a variable holding a copy of the first actual argument that is changed. The actual argument is not changed.

In the let case, the actual argument is changed. Thus, if abc is defined using let, and abc(u,v,w) is evaluated, the value of u changes to 123. That is, the let form of definition allows the user to bypass the protections that are enforced by the call by value conventions of standard procedure definitions.

Example: We take our earlier factorial procedure and write it as a let statement.

```
for all n let factorial n =>
            begin scalar m,s;
            m:=1; s:=n;
        l1: if s=0 then return m;
            m:=m*s;
```

```
            s:=s-1;
            go to l1
        end;
```

The reader will notice that we introduced a new local variable, `s`, and set it equal to `n`. The original form of the procedure contained the statement `n:=n-1;`. If the user asked for the value of `factorial(5)` then `n` would correspond to, not just have the value of, 5, and REDUCE would object to trying to execute the statement $5 := 5 - 1$.

If `pqr` is a procedure with no parameters,

```
procedure pqr;
   <procedure body>;
```

it can be written as a `let` statement quite simply:

```
let pqr => <procedure body>;
```

To call *procedure* `pqr`, if defined in the latter form, the empty parentheses would not be used: use `pqr` not `pqr()` where a call on the procedure is needed.

The two notations for a procedure with no arguments can be combined. `pqr` can be defined in the standard `procedure` form. Then a `let` statement

```
let pqr => pqr();
```

would allow a user to use `pqr` instead of `pqr()` in calling the procedure.

A feature available with `let`-defined procedures and not with procedures defined in the standard way is the possibility of defining partial functions.

```
for all x such that numberp x
   let uvw(x) => <procedure body>;
```

Now `uvw` of an integer would be calculated as prescribed by the procedure body, while `uvw` of a general argument, such as `z` or `p+q` (assuming these evaluate to themselves) would simply stay `uvw(z)` or `uvw(p+q)` as the case may be.

15.6 REMEMBER Statement

Setting the remember option for an algebraic procedure by

`remember (`⟨*procname:procedure*⟩`);`

saves all intermediate results of such procedure evaluations, including recursive calls. Subsequent calls to the procedure can then be determined from the saved results, and thus the number of evaluations (or the complexity) can be reduced. This mode of evalation costs extra memory, of course. In addition, the procedure must be free of side–effects.

The following examples show the effect of the remember statement on two well–known examples.

```
procedure H(n);       % Hofstadter's function
 if numberp n then
 << cnn := cnn +1;    % counts the calls
 if n < 3 then 1 else H(n-H(n-1))+H(n-H(n-2))>>;

remember h;

<< cnn := 0; H(100); cnn>>;

100

% H has been called 100 times only.

procedure A(m,n);     % Ackermann function

 if m=0 then n+1 else
  if n=0 then A(m-1,1) else
  A(m-1,A(m,n-1));

remember a;

A(3,3);
```

Chapter 16

Series Expansion

Expanding an algebraic expression into a series can be done by standard REDUCE operators, namely `df`, `sub`, and possibly `limit`. Nevertheless, there are many cases where this straightforward method fails. REDUCE offers two different operators for this purpose:

`taylor` computes a truncated power series.

`ps` computes extendible power series.

`fps` computes formal power series.

16.1 Taylor Expansion

This package carries out the Taylor expansion of an expression in one or more variables and efficient manipulation of the resulting Taylor series. Capabilities include basic operations (addition, subtraction, multiplication and division) and also application of certain algebraic and transcendental functions.[1]

The most important operator is `taylor`. It is used as follows:

`taylor(`⟨*exp:algebraic*⟩`,`
⟨*var:kernel*⟩`,`⟨*var0:algebraic*⟩`,`⟨*order:integer*⟩[,...]`)` : algebraic.

where `exp` is the expression to be expanded. It can be any REDUCE object, even an expression containing other Taylor kernels. `var` is the kernel with respect to which `exp` is to be expanded. `var0` denotes the expansion point about which and `order` the order up to which expansion is to take place. If more than one `(var,`

[1] This code was written by Rainer Schöpf.

`var0, order)` triple is specified `taylor` will expand its first argument independently with respect to each variable in turn. For example,

```
taylor(e^(x^2+y^2),x,0,2,y,0,2);
```

will calculate the Taylor expansion up to order $X^2 * Y^2$:

```
     2    2    2  2       3  3
1 + y  + x  + y *x  + O(x ,y )
```

Note that once the expansion has been done it is not possible to calculate higher orders. Instead of a kernel, `var` may also be a list of kernels. In this case expansion will take place in a way so that the *sum* of the degrees of the kernels does not exceed `order`. If `var0` evaluates to the special identifier `infinity`, expansion is done in a series in 1/var instead of `var`.

The expansion is performed variable per variable, i.e. in the example above by first expanding $\exp(x^2 + y^2)$ with respect to x and then expanding every coefficient with respect to y.

There are two extra operators to compute the Taylor expansions of implicit and inverse functions:

`implicit_taylor(`⟨*f:algebraic*⟩`,`
⟨*var:kernel*⟩`,`⟨*depvar:kernel*⟩`,`
⟨*var0:algebraic*⟩`,`⟨*depvar0:algebraic*⟩`,`
⟨*order:integer*⟩`)` : algebraic.

takes a function f depending on two variables var and depvar and computes the Taylor series of the implicit function depvar(var) given by the equation f(var,depvar) = 0, around the point var0. (Violation of the necessary condition f(var0,depvar0)=0 causes an error.) For example,

```
implicit_taylor(x^2 + y^2 - 1,x,y,0,1,5);
```

gives the output

```
     1   2    1   4       6
1 - ---*x  - ---*x  + O(x )
     2        8
```

The operator

```
inverse_taylor(⟨f:algebraic⟩,
               ⟨var:kernel⟩,⟨depvar:kernel⟩,
               ⟨var0:algebraic⟩,⟨order:integer⟩)
: algebraic.
```

takes a function f depending on var and computes the Taylor series of the inverse of f with respect to var0. For example,

```
inverse_taylor(exp(x)-1,x,y,0,8);
```

yields

```
      1   2    1   3    1   4    1   5
y - ---*y  + ---*y  - ---*y  + ---*y  + (3 terms)
      2        3        4        5

       9
 + O(y )
```

When a Taylor kernel is printed, only a certain number of (non-zero) coefficients are shown. If there are more, an expression of the form `(`n `terms)` is printed to indicate how many non-zero terms have been suppressed. The number of terms printed is given by the value of the shared algebraic variable `taylorprintterms`. Allowed values are integers and the special identifier `ALL`. The latter setting specifies that all terms are to be printed. The default setting is 5.

The `part` operator can be used to extract subexpressions of a Taylor expansion in the usual way. All terms can be accessed, irregardless of the value of the variable `taylorprintterms`.

If the switch `taylorkeeporiginal` is set to `on` the original expression exp is kept for later reference. It can be recovered by means of the operator

`taylororiginal`(exp:*exprn*):*exprn*

An error is signalled if exp is not a Taylor kernel or if the original expression was not kept, i.e. if `taylorkeeporiginal` was `off` during expansion. The template of a Taylor kernel, i.e. the list of all variables with respect to which expansion took place together with expansion point and order can be extracted using .

`taylortemplate`(exp:*exprn*):*list*

This returns a list of lists with the three elements (var,var0,order). As with `taylororiginal`, an error is signalled if exp is not a Taylor kernel.

`taylorcoefflist`(exp:*exprn*):*list*

This returns a list of two element lists (list of exponents,coefficient). Each exponent corresponds to a variable in the template. For homogenous expansion, each exponent is replaced by a list of exponents, as the template has a list of variables instead of a single one. Again, an error is signalled if exp is not a Taylor kernel.

See the test file for examples.

The operator

`taylortostandard`(exp:*exprn*):*exprn*

converts all Taylor kernels in exp into standard form and resimplifies the result.

The boolean operator

`taylorseriesp`(exp:*exprn*):*boolean*

may be used to determine if exp is a Taylor kernel. (Note that this operator is subject to the same restrictions as, e.g., `ordp` or `numberp`, i.e. it may only be used in boolean expressions in `if` or `let` statements.

Finally there is

`taylorcombine`(exp:*exprn*):*exprn*

which tries to combine all Taylor kernels found in exp into one. Operations currently possible are:

- Addition, subtraction, multiplication, and division.
- Roots, exponentials, and logarithms.
- Trigonometric and hyperbolic functions and their inverses.

Application of unary operators like `log` and `atan` will nearly always succeed. For binary operations their arguments have to be Taylor kernels with the same template. This means that the expansion variable and the expansion point must match. Expansion order is not so important, different order usually means that one of them is truncated before doing the operation.

If `taylorkeeporiginal` is set to `on` and if all Taylor kernels in `exp` have their original expressions kept `taylorcombine` will also combine these and store the result as the original expression of the resulting Taylor kernel. There is also the switch `taylorautoexpand` (see below).

There are a few restrictions to avoid mathematically undefined expressions: it is not possible to take the logarithm of a Taylor kernel which has no terms (i.e. is zero), or to divide by such a beast. There are some provisions made to detect singularities during expansion: poles that arise because the denominator has zeros at the expansion point are detected and properly treated, i.e. the Taylor kernel will start with a negative power. (This is accomplished by expanding numerator and

denominator separately and combining the results.) Essential singularities of the known functions (see above) are handled correctly.

Differentiation of a Taylor expression is possible. If you differentiate with respect to one of the Taylor variables the order will decrease by one.

Substitution is a bit restricted: Taylor variables can only be replaced by other kernels. There is one exception to this rule: you can always substitute a Taylor variable by an expression that evaluates to a constant. Note that REDUCE will not always be able to determine that an expression is constant.

Only simple taylor kernels can be integrated. More complicated expressions that contain Taylor kernels as parts of themselves are automatically converted into a standard representation by means of the `taylortostandard` operator. In this case a suitable warning is printed.

It is possible to revert a Taylor series of a function f, i.e., to compute the first terms of the expansion of the inverse of f from the expansion of f. This is done by the operator

`taylorrevert`(exp:*exprn*,oldvar:*kernel*, NEWVAR:*kernel*):*exprn*

EXP must evaluate to a Taylor kernel with OLDVAR being one of its expansion variables. Example:

```
taylor (u - u**2, u, 0, 5)$
taylorrevert (ws, u, x);
```

gives

```
      2      3      4       5      6
x + x  + 2*x  + 5*x  + 14*x  + O(x )
```

This package introduces a number of new switches:

`taylorautocombine` causes Taylor expressions to be automatically combined during the simplification process. This is equivalent to applying `taylorcombine` to every expression that contains Taylor kernels. Default is `on`.

`taylorautoexpand` makes Taylor expressions "contagious" in the sense that `taylorcombine` tries to Taylor expand all non-Taylor subexpressions and to combine the result with the rest. Default is `off`.

`taylorkeeporiginal` forces the package to keep the original expression, i.e. the expression that was Taylor expanded. All operations performed on the Taylor kernels are also applied to this expression which can be recovered using the operator `taylororiginal`. Default is `off`.

taylorprintorder causes the remainder to be printed in big-O notation. Otherwise, three dots are printed. Default is `on`.

verboseload will cause REDUCE to print some information when the Taylor package is loaded. This switch is already present in PSL systems. Default is `off`.

16.1.1 Caveats

`taylor` should always detect non-analytical expressions in its first argument. As an example, consider the function $xy/(x+y)$ that is not analytical in the neighborhood of $(x, y) = (0, 0)$: Trying to calculate

```
taylor(x*y/(x+y),x,0,2,y,0,2);
```

causes an error

```
***** Not a unit in argument to QUOTTAYLOR
```

Note that it is not generally possible to apply the standard REDUCE operators to a Taylor kernel. For example, `coeff` or `coeffn` cannot be used. Instead, the expression at hand has to be converted to standard form first using the `taylortostandard` operator.

16.1.2 Warning messages

***** Cannot expand further... truncation done**
You will get this warning if you try to expand a Taylor kernel to a higher order.

***** Converting Taylor kernels to standard representation**
This warning appears if you try to integrate an expression containing Taylor kernels.

16.1.3 Error messages

******* Branch point detected in ...**
This occurs if you take a rational power of a Taylor kernel and raising the lowest order term of the kernel to this power yields a non analytical term (i.e. a fractional power).

******* Cannot replace part ... in Taylor kernel**
The `part` operator can only be used to either replace the template of a Taylor kernel (part 2) or the original expression that is kept for reference (part 3).

******* Computation loops (recursive definition?): ...**
Most probably the expression to be expanded contains an operator whose derivative involves the operator itself.

******* Error during expansion (possible singularity)**
The expression you are trying to expand caused an error. As far as I know this can only happen if it contains a function with a pole or an essential singularity at the expansion point. (But one can never be sure.)

******* Essential singularity in ...**
An essential singularity was detected while applying a special function to a Taylor kernel.

******* Expansion point lies on branch cut in ...**
The only functions with branch cuts this package knows of are (natural) logarithm, inverse circular and hyperbolic tangent and cotangent. The branch cut of the logarithm is assumed to lie on the negative real axis. Those of the arc tangent and arc cotangent functions are chosen to be compatible with this: both have essential singularities at the points $\pm i$. The branch cut of arc tangent is the straight line along the imaginary axis connecting $+1$ to -1 going through ∞ whereas that of arc cotangent goes through the origin. Consequently, the branch cut of the inverse hyperbolic tangent resp. cotangent lies on the real axis and goes from -1 to $+1$, that of the latter across 0, the other across ∞.

The error message can currently only appear when you try to calculate the inverse tangent or cotangent of a Taylor kernel that starts with a negative degree. The case of a logarithm of a Taylor kernel whose constant term is a negative real number is not caught since it is difficult to detect this in general.

******* Input expression non-zero at given point**
Violation of the necessary condition f(var0,depvar0)=0 for the arguments of `implicit_taylor`.

******* Invalid substitution in Taylor kernel: ...**
You tried to substitute a variable that is already present in the Taylor kernel or on which one of the Taylor variables depend.

******* Not a unit in ...**
This will happen if you try to divide by or take the logarithm of a Taylor series whose constant term vanishes.

******* Not implemented yet (...)**
Sorry, but this feature is not implemented, although it is possible to do so.

******* Reversion of Taylor series not possible: ...**
You tried to call the `taylorrevert` operator with inappropriate arguments. The second half of this error message tells you why this operation is not possible.

******* Taylor kernel doesn't have an original part**
The Taylor kernel upon which you try to use `taylororiginal` was created with the switch `taylorkeeporiginal` set to `off` and does therefore not keep the original expression.

******* Wrong number of arguments to TAYLOR**
You try to use the operator `taylor` with a wrong number of arguments.

******* Zero divisor in TAYLOREXPAND**
A zero divisor was found while an expression was being expanded. This should not normally occur.

******* Zero divisor in Taylor substitution**
That's exactly what the message says. As an example consider the case of a Taylor kernel containing the term `1/x` and you try to substitute `x` by `0`.

******* ... invalid as kernel**
You tried to expand with respect to an expression that is not a kernel.

******* ... invalid as order of Taylor expansion**
The order parameter you gave to `taylor` is not an integer.

******* ... invalid as Taylor kernel**
You tried to apply `taylororiginal` or `taylortemplate` to an expression that is not a Taylor kernel.

******* ... invalid as Taylor Template element**
You tried to substitute the `taylortemplate` part of a Taylor kernel with a list of incorrect form. For the correct form see the description of the `taylortemplate` operator.

******* ... invalid as Taylor variable**
You tried to substitute a Taylor variable by an expression that is not a kernel.

******* ... invalid as value of TaylorPrintTerms**
You have assigned an invalid value to `taylorprintterms`. Allowed values are: an integer or the special identifier `all`.

TAYLOR PACKAGE (...): this can't happen ...
This message shows that an internal inconsistency was detected. This is not your fault, at least as long as you did not try to work with the internal data structures of REDUCE. Send input and output to the REDUCE developers mailing list, together with the version information that is printed out.

16.1.4 Comparison to other packages

At the moment there is only one REDUCE package that I know of: the extendible power series package by Alan Barnes and Julian Padget. In my opinion there are two major differences:

- The interface. They use the domain mechanism for their power series, I decided to invent a special kind of kernel. Both approaches have advantages and disadvantages: with domain modes, it is easier to do certain things automatically, e.g., conversions.
- The concept of an extendible series: their idea is to remember the original expression and to compute more coefficients when more of them are needed. My approach is to truncate at a certain order and forget how the unexpanded expression looked like. I think that their method is more widely usable, whereas mine is more efficient when you know in advance how many terms you need.

16.2 TPS: Extendible Power Series

16.2.1 Introduction

This package implements formal Laurent power series expansions in one variable using the domain mechanism of REDUCE. This means that power series objects can be added, multiplied, differentiated etc. like other first class objects in the system. A lazy evaluation scheme is used in the package and thus terms of the series are not evaluated until they are required for printing or for use in calculating terms in other power series. The series are extendible giving the user the impression that the full infinite series is being manipulated. The errors that can sometimes occur using series that are truncated at some fixed depth (for example when a term in the required series depends on terms of an intermediate series beyond the truncation depth) are thus avoided.

The package was originally based on an earlier *truncated power series* package developed by Julian Padget in the 1980's. The name of the original package was TPS and this was never changed. The alternative (more accurate) name EPS was perhaps rejected because of possible confusion with the acronym for *encapsulated PostScript*.

In the first subsection below a brief description of the main operators available for series expansion are given together with some examples of their use.

16.2.2 Basic Use

The most important operator is `ps` which is used as follows:

```
ps(EXP:algebraic, VAR:kernel,
   ABOUT:algebraic):algebraic.
```

The `ps` operator returns a Laurent power series object (a tagged domain element) representing the univariate formal Laurent power series expansion of `EXP` with respect to the dependent variable `VAR` about the expansion point `ABOUT`. `EXP` may itself contain power series objects. If the function has a pole at the expansion point then the correct Laurent series expansion will be produced.

The algebraic expression `ABOUT` should simplify to an expression which is independent of the dependent variable `VAR`, otherwise an error will result. If `ABOUT` is the identifier `infinity` then the power series expansion about ∞ is obtained in ascending powers of `1/VAR`.

Examples

```
a := ps(sin x, x, 0);
ps(sin a, x, 0);
ps(cos x/x^2, x, 0);
ps(x/(1+x),x,infinity);
```

Operations on Power Series

As power series objects are domain elements they may be added, subtracted, multiplied and divided in the normal way. For example if A and B are power series objects with the *same expansion variable and expansion point*:

```
a+b; a*b;
1/b; a/b;
```

will produce power series objects representing the sum, product, reciprocal, and quotient of the power series objects A and B respectively.

Differentiation

Similarly, if A is a power series object depending on X then the input `df(a, x);` will produce the power series expansion of the derivative of A with respect to X.

Integration

The power series expansion of an integral may also be obtained (even if REDUCE cannot evaluate the integral in closed form). An example of this is

```
ps(int(exp(exp x),x),x,0);
```

Note that if the integration variable is the same as the expansion variable, the integration package is not called. If on the other hand the two variables are different the integrator is called to integrate each of the coefficients in the power series expansion of the integrand. The constant of integration is zero by default.

Note that the Laurent series domain is not closed under integration with respect to the expansion variable; if the term of degree -1 is non-zero a logarithmic singularity error will occur on integration.

Exponentiation The Laurent series domain is closed under exponentiation by an *integer* power. Thus, with respect to integer exponentiation, power series are first class objects and for example the following results in automatic expansion of the final result:

```
a:= ps(cos x,x,0);
b:= ps(sin x,x,0);
a^2+b^(-2);
```

However, for more general exponents automatic expansion does not occur. For example given power series `a` and `b` defined as above, the following commands are necessary:

```
ps(a^(1/2),x,0);
ps(a^pi,x,0);
ps(a^b,x,0);
```

Note *any* power of a power series of *order zero* (that is with a non-zero term of degree zero) can be expanded as a power series (again of order zero) provided only that the power is non-singular at the expansion point. As the third example above shows the exponent may itself be a power series.

However in general the Laurent series domain is not closed under exponentiation. If the result is to be a Laurent series some restrictions on the allowed values of the exponent and order of the original series are necessary. Namely, if the order of the power series is non-zero (σ say) and the exponent is rational with denominator q say, then σq must be integral. If the exponent is rational, but σq is not an integer, a *branch point* error is generated. For other exponents a *logarithmic singularity* error is usually generated. For example,

```
a := ps(1-cos x,x,0);  % series has order 2
ps(a^(1/2),x,0);   % series has order 1
ps(a^(2/3),x,0);   % branch point error
ps(a^pi,x,0);      % logarithmic singularity error
```

Power series of user defined functions

New user-defined functions may be expanded provided the user provides a rule or rule list defining the derivative of the function and optionally its value at the expansion point. For example

```
operator u;
let df(u(~x),~x)= exp(e^x);
let u(0) = e;
ps(u(sin x),x,0);
```

Of course the rules defined must be such that the function actually has a Taylor series expansion about the specified point.

Restrictions and Known Bugs

Currently automatic expansion of quotients with an integer denominator does not normally occur. One must use:

```
    a:=ps(sin x,x,0);
    ps(a/5,x,0);
or
    on rational;    % or on rounded;
    a/5;
```

Currently the following does not produce a power series object (although the result is formally valid):

```
  a := ps(cos x, x, 0);
  ps(2^a,x,0);
% instead use:
  ps(2^cos x,x,0);
```

If A is a power series object and X is a variable which evaluates to itself then expressions such as `a*x` or `int(a, x);` do not automtically expand to a single power series object (although the result returned is formally valid). Instead expressions such as `ps(a*x,x,0)` and `ps(int(a,x),x,0` should be used.

Currently the handling of essential sigularities is rather erratic; sometimes an Essential Singularity or Logarithmic Singularity error message is output, but often the system fails rather ungracefully.

There is no simple way to write the results of power series calculation to a file and read them back into REDUCE at a later stage.

Taylor Series Expansion

The operator `pstaylor` may be used as follows:

```
pstaylor(EXP:algebraic, VAR:kernel,
         ABOUT:algebraic):algebraic.
```

which uses the classic Taylor series algorithm for expanding `EXP` and returning an extendible Taylor series object.

The `pstaylor` operator may be useful in contexts where the operator `ps` fails to build a suitable recurrence relation automatically and reports too deep a recursion in `ps!:unknown!-crule`. A typical example is the expansion of the Γ function about an expansion point which is not a non-positive integer [2].

Note, however, that `pstaylor` always returns a *Taylor* series whose order is non-negative. Attempting to use `pstaylor` to expand a function about a pole will fail with a zero divisor error message.

Also in many cases the use of an automatically generated recurrence relation built by `ps` is more efficient than using `pstaylor`, particularly if a large number of terms is required; expansion of `tan` is a typical example where the number of terms in the nth derivative grows exponentially.

16.2.3 Printing Power Series

If the command `ps` or `pstaylor` is terminated by a semi-colon, a power series object is compiled and then a number of terms of the power series expansion are evaluated and printed.

psexplim Operator

The expansion is carried out as far as the value specified by an internal variable (with a default value of 6). This variable can be accessed via the operator `psexplim`.

```
  psexplim(UPTO:integer):integer.
or
  psexplim():integer
```

If `psexplim` is called with an integer value, the internal variable is updated to the value of `UPTO` and its previous value is returned. If `psexplim` is called with no argument the current value is unaltered and that value is returned.

If `psexplim` is used to increase the expansion limit, sufficient information is stored in the power series object to enable the additional terms to be calculated without recalculating the terms already obtained.

If the command is terminated by a dollar symbol, a power series object is compiled

[2] Actually the TPS code now detects this case and automatically uses `pstaylor` where appropriate.

and the first term is calculated, but no output is printed.

psprintorder Switch

When the switch `psprintorder` is ON the trailing terms of power series beyond `psexplim` are represented in print by a big-O notation, otherwise, three dots are printed. This switch is ON by default. However, if expression being expanded is a polynomial in the expansion variable and all non-zero terms have been output then the big-O or trailing dots are omitted to indicate that the series is complete.

16.2.4 Accessor Functions

In this section a number of accessor functions which allow the user to extract information such as the dependent variable, expansion point, a particular term etc. of a power series object.

psdepvar Operator

```
psdepvar(TPS:power series object):identifier.
```

The operator `psdepvar` returns the expansion variable of the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results. If TPS is an integer, the identifier `undefined` is returned.

psexpansionpt operator

```
psexpansionpt(TPS:power-series-object):algebraic.
```

The operator `psexpansionpt` returns the expansion point of the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results. If TPS is an integer, the identifier `undefined` is returned. If the expansion is about infinity, the identifier `infinity` is returned.

psfunction Operator

```
psfunction(TPS:power-series-object):algebraic.
```

The operator `psfunction` returns the function whose expansion gave rise to the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results.

psterm Operator

```
psterm(TPS:power-series-object,
       NTH:integer):algebraic.
```

The operator `psterm` returns the NTH term of the existing power series object

TPS. If NTH does not evaluate to an integer or TPS to a power series object an error results. It should be noted that an integer is treated as a power series.

psorder Operator

```
psorder(TPS:power-series-object):integer.
```

The operator `psorder` returns the order, that is the degree of the first non-zero term, of the power series object TPS. TPS should evaluate to a power series object or an error results. If TPS is zero, the identifier `undefined` is returned.

pstruncate Operator

```
pstruncate(TPS:power-series-object,
           POWER:integer):algebraic.
```

This procedure truncates the power series `TPS` discarding terms of order higher than `POWER`. The series is extended automatically if the value of `POWER` is greater than the order of last term calculated to date. For example

```
a := ps(sin x, x, 0);
pstruncate(a, 11);
```

will output the eleventh order polynomial resulting in truncating the series for $sinx$ after the term involving x^{11}.

If `POWER` is less than the order of the series then 0 is returned. If `POWER` does not simplify to an integer or if `TPS` is not a power series object then a Reduce error result.

16.2.5 Power Series Reversion

In order to functionally invert a power series the operator `psreverse` is used.

```
psreverse(TPS:power-series-object)
          :power-series-object
```

Four cases arise:

1. If the order of the series is 1, then the expansion point of the inverted series is 0.
2. If the order is 0 *and* if the first order term in TPS is non-zero, then the expansion point of the inverted series is taken to be the coefficient of the zeroth order term in TPS.

3. If the order is -1 the expansion point of the inverted series is the point at infinity. In all other cases a REDUCE error is reported because the series cannot be inverted as a power series. Puiseux expansion would be required to handle these cases.

4. If the expansion point of TPS is finite it becomes the zeroth order term in the inverted series. For expansion about 0 or the point at infinity the order of the inverted series is one.

If TPS is not a power series object after evaluation an error results.

Some examples:

```
ps(sin x,x,0);
psreverse(ws); % produces series for asin x about x=0.
ps(exp x,x,0);
psreverse ws; % produces series for log x about x=1.
ps(sin(1/x),x,infinity);
psreverse(ws); % series for 1/asin(x) about x=0.
```

16.2.6 Power Series Composition

In order to functionally compose two power series the operator `pscompose` is used.

```
pscompose(TPS1:power-series-object,
          TPS2:power-series-object)
          :power-series-object
```

The power series TPS1 and TPS2 are functionally composed; that is to say that TPS2 is substituted for the expansion variable in TPS1 and the result expressed as a power series. The dependent variable and expansion point of the result coincide with those of TPS2. The following conditions apply to power series composition:

1. If the expansion point of TPS1 is 0 then the order of the TPS2 must be at least 1.

2. If the expansion point of TPS1 is finite, it should coincide with the coefficient of the zeroth order term in TPS2. The order of TPS2 should also be non-negative in this case.

3. If the expansion point of TPS1 is the point at infinity then the order of TPS2 must be less than or equal to -1.

If these conditions do not hold the series cannot be composed (with the current algorithm terms of the inverted series would involve infinite sums) and a REDUCE error occurs.

Some examples:

```
a:=ps(exp y,y,0);  b:=ps(sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(sin x)
% about x=0.

a:=ps(exp z,z,1); b:=ps(cos x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(cos x)
% about x=0.

a:=ps(cos(1/x),x,infinity);  b:=ps(1/sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of cos(sin x)
% about x=0.
```

16.2.7 pssum Operator

If an expression is known for the nth term of a power series, an extendible power series object may be constructed by the operator `pssum`

```
pssum(J:kernel = LOWLIM:integer,
      COEFF:algebraic, X:kernel,
      ABOUT:algebraic, POWER:algebraic)
      :power-series-object
```

The formal power series sum for J from LOWLIM to `infinity` of

```
COEFF*(X-ABOUT)**POWER
```

when `ABOUT` is finite or zero, whereas if ABOUT is `infinity`

```
COEFF*(1/X)**POWER
```

is constructed and returned. This enables power series whose general term is known to be constructed and manipulated using the other procedures of the power series package.

J and X should be distinct simple kernels. The algebraics ABOUT, COEFF and

POWER should not depend on the expansion variable X, similarly the algebraic ABOUT should not depend on the summation variable J. The algebraic POWER should be a strictly increasing integer-valued function of J for J in the range LOWLIM to `infinity`.

Some examples:

```
pssum(n=0,1,x,0,n*n);
% Produces the power series summation for n=0 to
% infinity of x**(n*n).

pssum(n=1,n,x,0,n);
% Produces the power series summation for n=1 to
% infinity of n*x**n.

pssum(m=1,(-1)**(m-1)/(2m-1),y,1,2m-1);
% Produces a power series which is actually the
% expansion of atan(y-1)  about y=1.

pssum(j=1,-1/j,x,infinity,j);
% Produces a power series which is actually the
% expansion of log(1-1/x) about the point at infinity.

pssum(n=0,1,x,0,2n**2+3n) + pssum(n=1,1,x,0,2n**2-3n);
% Produces the power series summation for n=-infinity
% to +infinity of x**(2n**2+3n).
```

It should be noted that a formal power series is produced which may not have a non-zero radius of convergence; the second example above illustrates this. Nevertheless these formal series may be added, multiplied, differentiated etc. by the TPS package. Of course, in general the result may also have a zero radius of convergence.

16.2.8 Miscellaneous Operators

pscopy Operator

```
pscopy(TPS:power-series-object):power-series-object
```

This procedure returns a copy of the power series `TPS`. The copy has no shared sub-structures in common with the original series. This enables substitutions to be performed on the series without side-effects on previously computed objects. For example:

```
clear a;
b := ps(sin(a*x)), x, 0);
b where a => 1;
```

will result in `a` being set to 1 in each of the terms of the power series and the resulting expressions being simplified. Owing to the way power series objects are implemented using Lisp vectors, this has the side-effect that the value of `b` is changed. This may be avoided by copying the series with `pscopy` before applying the substitution, thus:

```
b := ps(sin(a*x)), x, 0);
pscopy b where a => 1;
```

pschangevar Operator

```
pschangevar(TPS:power-series-object,
            X:kernel):power-series-object
```

The operator `pschangevar` changes the dependent variable of the power series object TPS to the variable X. TPS should evaluate to a power series object and X to a kernel, otherwise an error results. Also X should not appear as a parameter in TPS. The power series with the new dependent variable is returned.

psordlim Operator

```
   psordlim(UPTO:integer):integer
or
   psordlim():integer
```

An internal variable is set to the value of `UPTO` (which should evaluate to an integer). The value returned is the previous value of the variable. The default value is 100. If `psordlim` is called with no argument, the current value is returned.

The significance of this control is that the system attempts to find the order of the power series required, that is the order is the degree of the first non-zero term in the power series. If the order is greater than the value of this variable an error message is given and the computation aborts. This prevents infinite loops in certain cases, for example:

```
a:=ps(1-(cos x)^2,x,0);
b :=ps((sin x)^2,x,0);
b-a;
```

This will also occur in the rather unlikely situation where the expression being expanded is

1. identically zero, but is not recognized as such by REDUCE;
2. and its derivatives are not recognized as identically zero by Reduce;
3. but the values of all derivatives at the expansion point are simplified to zero by REDUCE.

16.3 FPS: Automatic Calculation of Formal Power Series

This package can expand a specific class of functions into their corresponding Laurent-Puiseux series.[3]

16.3.1 Introduction

This package can expand functions of certain type into their corresponding Laurent-Puiseux series as a sum of terms of the form

$$\sum_{k=0}^{\infty} a_k (x - x_0)^{mk/n+s}$$

where m is the 'symmetry number', s is the 'shift number', n is the 'Puiseux number', and x_0 is the 'point of development'. The following types are supported:

- textbffunctions of 'rational type', which are either rational or have a rational derivative of some order;
- **functions of 'hypergeometric type'** where $a(k+m)/a(k)$ is a rational function for some integer m;
- **functions of 'explike type'** which satisfy a linear homogeneous differential equation with constant coefficients.

The FPS package is an implementation of the method presented in [Koe92]. The implementations of this package for MAPLE (by D. Gruntz) and MATHEMATICA (by W. Koepf) served as guidelines for this one.

Numerous examples can be found in [Koe93b, Koe93a], most of which are contained in the test file `fps.tst`. Many more examples can be found in the extensive bibliography of Hansen [Han75].

16.3.2 REDUCE operator FPS

`fps(f,x,x0)` tries to find a formal power series expansion for `f` with respect to the variable `x` at the point of development `x0`. It also works for formal Laurent (negative exponents) and Puiseux series (fractional exponents). If the third argument is omitted, then `x0:=0` is assumed.

Examples: `fps(asin(x)^2,x)` results in

[3] This package was written by Wolfram Koepf and Winfried Neun.

```
          2*k  2*k              2  2
         x   *2   *factorial(k) *x
infsum(---------------------------,k,0,infinity)
         factorial(2*k + 1)*(k + 1)
```

`fps(sin x,x,pi)` gives

```
                    2*k       k
         ( - pi + x)   *( - 1) *( - pi + x)
infsum(-------------------------------------,k,0,
                 factorial(2*k + 1)

        infinity)
```

and `fps(sqrt(2-x^2),x)` yields

```
            2*k
          - x   *sqrt(2)*factorial(2*k)
infsum(-------------------------------,k,0,infinity)
           k             2
          8 *factorial(k) *(2*k - 1)
```

Note: The result contains one or more `infsum` terms such that it does not interfere with the REDUCE operator `sum`. In graphical oriented REDUCE interfaces this operator results in the usual $\sum$ notation.

If possible, the output is given using factorials. In some cases, the use of the Pochhammer symbol `pochhammer(a,k)` $:= a(a+1)\cdots(a+k-1)$ is necessary.

The operator `fps` uses the operator `SimpleDE` of the next section.

If an error message of type

```
Could not find the limit of:
```

occurs, you can set the corresponding limit yourself and try a recalculation. In the computation of `fps(atan(cot(x)),x,0)`, REDUCE is not able to find the value for the limit `limit(atan(cot(x)),x,0)` since the `atan` function is multi-valued. One can choose the branch of `atan` such that this limit equals $\pi/2$ so that we may set

```
let limit(atan(cot(~x)),x,0)=>pi/2;
```

and a recalculation of `fps(atan(cot(x)),x,0)` yields the output `pi - 2*x` which is the correct local series representation.

16.3.3 REDUCE operator SimpleDE

`SimpleDE(f,x)` tries to find a homogeneous linear differential equation with polynomial coefficients for f with respect to x. Make sure that y is not a used variable. The setting `factor df;` is recommended to receive a nicer output form.

Examples: `SimpleDE(asin(x)^2,x)` then results in

```
              2
df(y,x,3)*(x  - 1) + 3*df(y,x,2)*x + df(y,x)
```

`SimpleDE(exp(x^(1/3)),x)` gives

```
                2
27*df(y,x,3)*x  + 54*df(y,x,2)*x + 6*df(y,x) - y
```

and `SimpleDE(sqrt(2-x^2),x)` yields

```
           2
df(y,x)*(x  - 2) - x*y
```

The depth for the search of a differential equation for `f` is controlled by the variable `fps_search_depth`; A higher value for `fps_search_depth` will increase the chance to find the solution, but increases the complexity as well. The default value for `fps_search_depth` is 5. E. g., for `fps(sin(x^(1/3)),x)`, or `SimpleDE(sin(x^(1/3)),x)` a setting `fps_search_depth:=6` is necessary.

The output of the FPS package can be influenced by the switch `tracefps`. Setting `on tracefps` causes various prints of intermediate results.

16.3.4 Problems in the current version

The handling of logarithmic singularities is not yet implemented.

The rational type implementation is not yet complete.

The support of special functions [Koe94a] will be part of the next version.

Chapter 17

Solving Numerical Problems

The NUMERIC package implements some numerical (approximative) algorithms for REDUCE, based on the REDUCE rounded mode arithmetic.[1] These algorithms are implemented for standard cases:

This package implements basic algorithms of numerical analysis. These include:

- solution of algebraic equations by Newton's method

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2})
```

- solution of ordinary differential equations

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1),
             iterations=5)
```

- bounds of a function over an interval

```
bounds(sin x+x,x=(1 .. 2));
```

- minimizing a function (Fletcher Reeves steepest descent)

```
num_min(sin(x)+x/5, x);
```

- Chebyshev curve fitting

```
chebyshev_fit(sin x/x,x=(1 .. 3),5);
```

- numerical quadrature

```
num_int(sin x,x=(0 .. pi));
```

They should not be called for ill-conditioned problems; please use standard mathematical libraries for these.

[1] This code was written by Herbert Melenk.

17.1 Syntax

17.1.1 Intervals, Starting Points

Intervals are generally coded as lower bound and upper bound connected by the operator '`..`', usually associated to a variable in an equation. E.g.

```
x= (2.5 .. 3.5)
```

means that the variable x is taken in the range from 2.5 up to 3.5. Note, that the bounds can be algebraic expressions, which, however, must evaluate to numeric results. In cases where an interval is returned as the result, the lower and upper bounds can be extracted by the `part` operator as the first and second part respectively. A starting point is specified by an equation with a numeric righthand side, e.g.

```
x=3.0
```

If for multivariate applications several coordinates must be specified by intervals or as a starting point, these specifications can be collected in one parameter (which is then a list) or they can be given as separate parameters alternatively. The list form is more appropriate when the parameters are built from other REDUCE calculations in an automatic style, while the flat form is more convenient for direct interactive input.

17.1.2 Accuracy Control

The keyword parameters `accuracy=`$\langle a \rangle$ and `iterations=`$\langle i \rangle$, where $\langle a \rangle$ and $\langle i \rangle$ must be positive integer numbers, control the iterative algorithms: the iteration is continued until the local error is below 10^{-a}; if that is impossible within $\langle i \rangle$ steps, the iteration is terminated with an error message. The values reached so far are then returned as the result.

17.1.3 Tracing

Normally the algorithms produce only a minimum of printed output during their operation. In cases of an unsuccessful or unexpected long operation a trace of the iteration can be printed by setting

```
on trnumeric;
```

17.2 Minima

The Fletcher Reeves version of the *steepest descent* algorithms is used to find the minimum of a function of one or more variables. The function must have continuous partial derivatives with respect to all variables. The starting point of the search can be specified; if not, random values are taken instead. The steepest descent algorithms in general find only local minima.

Syntax:

num_min (⟨*exp*⟩, ⟨var_1⟩ [=val_1] [,var_2[=val_2] ...]
[,accuracy=⟨a⟩][,iterations=⟨i⟩])
or
num_min (⟨*exp*⟩, {⟨var_1⟩ [=val_1] [,var_2[=val_2] ...]}
[,accuracy=⟨a⟩][,iterations=⟨i⟩])

where ⟨*exp*⟩ is a function expression, ⟨var_1⟩, ⟨var_2⟩, ... are the variables in ⟨*exp*⟩ and ⟨val_1⟩, ⟨val_2⟩, ... are the (optional) start values.

num_min tries to find the next local minimum along the descending path starting at the given point. The result is a list with the minimum function value as first element followed by a list of equations, where the variables are equated to the coordinates of the result point.

Examples:

```
num_min(sin(x)+x/5, x);

{-0.0775896851944,{x=4.51103102502}}

num_min(sin(x)+x/5, x=0);

{-1.33422674662,{x=-1.77215826714}}

% Rosenbrock function (well known as hard to minimize).
fktn := 100*(x1**2-x2)**2 + (1-x1)**2;
num_min(fktn, x1=-1.2, x2=1, iterations=200);

{0.000000218702254529,{x1=0.999532844959,x2

    =0.99906807243}}
```

17.3 Roots of Functions / Solutions of Equations

An adaptively damped Newton iteration is used to find an approximative zero of a function, a function vector or the solution of an equation or an equation system. Equations are internally converted to a difference of lhs and rhs such that the Newton method (=zero detection) can be applied. The expressions must have continuous derivatives for all variables. A starting point for the iteration can be given. If not given, random values are taken instead. If the number of forms is not equal to the number of variables, the Newton method cannot be applied. Then the minimum of the sum of absolute squares is located instead.

With `on complex` solutions with imaginary parts can be found, if either the expression(s) or the starting point contain a nonzero imaginary part.

Syntax:

num_solve (⟨exp_1⟩,⟨var_1⟩[=val_1][,accuracy=a][,iterations=i])

or

num_solve $(\{exp_1,\ldots,exp_n\},var_1[=val_1],\ldots,var_n[=val_n]$

$[,accuracy=a][,iterations=i])$

or

num_solve $(\{exp_1,\ldots,exp_n\},\{var_1[=val_1],\ldots,var_n[=val_n]\}$

$[,accuracy=a][,iterations=i])$

where $exp_1,\ldots,exp_n$ are function expressions,

$var_1,\ldots,var_n$ are the variables,

$val_1,\ldots,val_n$ are optional start values.

`num_solve` tries to find a zero/solution of the expression(s). Result is a list of equations, where the variables are equated to the coordinates of the result point.

The Jacobian matrix is stored as a side effect in the shared variable `jacobian`.

Example:

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=1});

{x= - 1.85619449019,y=2.85619449019}

jacobian;
```

```
[cos(x)   sin(y)]
[                ]
[  1        1    ]
```

17.4 Integrals

For the numerical evaluation of univariate integrals over a finite interval the following strategy is used:

1. If the function has an antiderivative in close form which is bounded in the integration interval, this is used.
2. Otherwise a Chebyshev approximation is computed, starting with order 20, eventually up to order 80. If that is recognized as sufficiently convergent it is used for computing the integral by directly integrating the coefficient sequence.
3. If none of these methods is successful, an adaptive multilevel quadrature algorithm is used.

For multivariate integrals only the adaptive quadrature is used. This algorithm tolerates isolated singularities. The value $iterations$ here limits the number of local interval intersection levels. $Accuracy$ is a measure for the relative total discretization error (comparison of order 1 and order 2 approximations).

Syntax:

num_int $(exp, var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots]$

$[, accuracy = a][, iterations = i])$

where exp is the function to be integrated,

$var_1, var_2, \ldots$ are the integration variables,

$l_1, l_2, \ldots$ are the lower bounds,

$u_1, u_2, \ldots$ are the upper bounds.

Result is the value of the integral.

Example:

```
num_int(sin x,x=(0 .. pi));
```

```
2.0
```

17.5 Ordinary Differential Equations

A Runge-Kutta method of order 3 finds an approximate graph for the solution of a ordinary differential equation real initial value problem.

Syntax:

num_odesolve (exp,$depvar = dv$,$indepvar$=($from..to$)

$[, accuracy = a][, iterations = i]$)

where

exp is the differential expression/equation,

$depvar$ is an identifier representing the dependent variable (function to be found),

$indepvar$ is an identifier representing the independent variable,

exp is an equation (or an expression implicitly set to zero) which contains the first derivative of $depvar$ wrt $indepvar$,

$from$ is the starting point of integration,

to is the endpoint of integration (allowed to be below $from$),

dv is the initial value of $depvar$ in the point $indepvar = from$.

The ODE exp is converted into an explicit form, which then is used for a Runge Kutta iteration over the given range. The number of steps is controlled by the value of i (default: 20). If the steps are too coarse to reach the desired accuracy in the neighborhood of the starting point, the number is increased automatically.

Result is a list of pairs, each representing a point of the approximate solution of the ODE problem.

Remarks:

- Note that the dependent variable must be explicitly declared using a `depend` statement, e.g., `depend y,x`.
- The REDUCE package SOLVE is used to convert the form into an explicit ODE. If that process fails or has no unique result, the evaluation is stopped with an error message.

Example:

```
depend y,x;

num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5);
```

```
{{x,y},
 {0.0,1.0},
 {0.2,1.22140275816},
 {0.4,1.49182469764},
 {0.6,1.82211880039},
 {0.8,2.22554092849},
 {1.0,2.71828182846}}
```

17.6 Bounds of a Function

Upper and lower bounds of a real valued function over an interval or a rectangular multivariate domain are computed by the operator `bounds`. The algorithmic basis is the computation with inequalities: starting from the interval(s) of the variables, the bounds are propagated in the expression using the rules for inequality computation. Some knowledge about the behavior of special functions like ABS, SIN, COS, EXP, LOG, fractional exponentials etc. is integrated and can be evaluated if the operator `bounds` is called with rounded mode on (otherwise only algebraic evaluation rules are available).

If `bounds` finds a singularity within an interval, the evaluation is stopped with an error message indicating the problem part of the expression.

Syntax:

bounds $(exp, var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots])$

bounds $(exp, \{var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots]\})$

where exp is the function to be investigated,

$var_1, var_2, \ldots$ are the variables of exp,

$l_1, l_2, \ldots$ and $u_1, u_2, \ldots$ specify the area (intervals).

`bounds` computes upper and lower bounds for the expression in the given area. An interval is returned.

Example:

```
bounds(sin x,x=(1 .. 2));

- 1 .. 1

on rounded;
bounds(sin x,x=(1 .. 2));

0.841470984808 .. 1

bounds(x**2+x,x=(-0.5 .. 0.5));

 - 0.25 .. 0.75
```

17.7 Chebyshev Curve Fitting

The operator family $Chebyshev\_\ldots$ implements approximation and evaluation of functions by the Chebyshev method. Let $T_n^{(a,b)}(x)$ be the Chebyshev polynomial of order n transformed to the interval (a, b). Then a function $f(x)$ can be approximated in (a, b) by a series

$$f(x) \approx \sum_{i=0}^{N} c_i T_i^{(a,b)}(x)$$

The operator `chebyshev_fit` computes this approximation and returns a list, which has as first element the sum expressed as a polynomial and as second element the sequence of Chebyshev coefficients c_i. `chebyshev_df` and `chebyshev_int` transform a Chebyshev coefficient list into the coefficients of the corresponding derivative or integral respectively. For evaluating a Chebyshev approximation at a given point in the basic interval the operator `chebyshev_eval` can be used. Note that `Chebyshev_eval` is based on a recurrence relation which is in general more stable than a direct evaluation of the complete polynomial.

chebyshev_fit $(fcn, var = (lo..hi), n)$

chebyshev_eval $(coeffs, var = (lo..hi), var = pt)$

chebyshev_df $(coeffs, var = (lo..hi))$

chebyshev_int $(coeffs, var = (lo..hi))$

where ⟨*fcn*⟩ is an algebraic expression (the function to be fitted), ⟨*var*⟩ is the variable of ⟨*fcn*⟩, ⟨*lo*⟩ and ⟨*hi*⟩ are numerical real values which describe an interval ($lo < hi$), ⟨*n*⟩ is the approximation order, a positive integer, set to 20 if missing, ⟨*pt*⟩ is a numerical value in the interval and ⟨*coeffs*⟩ is a series of Chebyshev coefficients, computed by one of the operators `chebyshev_coeff`, `chebyshev_df`, or `chebyshev_int`.

Example:

```
on rounded;

w:=chebyshev_fit(sin x/x,x=(1 .. 3),5);

w := {0.0382345446975*x  - 0.239802588672*x

       + 0.0651206939005*x + 0.977836217464,

      {0.899091895826,-0.406599215895,

       -0.00519766024352,0.00946374143079,

       -0.0000948947435876}}

chebyshev_eval(second w, x=(1 .. 3), x=2.1);

0.411091086819
```

17.8 General Curve Fitting

The operator `num_fit` finds for a set of points the linear combination of a given set of functions (function basis) which approximates the points best under the objective of the least squares criterion (minimum of the sum of the squares of the deviation). The solution is found as zero of the gradient vector of the sum of squared errors.

Syntax:

num_fit $(vals, basis, var = pts)$

where $vals$ is a list of numeric values,

var is a variable used for the approximation,

pts is a list of coordinate values which correspond to var,

basis is a set of functions varying in *var* which is used for the approximation.

The result is a list containing as first element the function which approximates the given values, and as second element a list of coefficients which were used to build this function from the basis.

Example:

```
 % approximate a set of factorials by a polynomial
pts:=for i:=1 step 1 until 5 collect i$
vals:=for i:=1 step 1 until 5 collect
        for j:=1:i product j$

num_fit(vals,{1,x,x**2},x=pts);

                2
{14.5714285714*x  - 61.4285714286*x + 54.6,{54.6,

     - 61.4285714286,14.5714285714}}

num_fit(vals,{1,x,x**2,x**3,x**4},x=pts);

{2.20833333343*x  - 20.2500000011*x

  + 67.7916666713*x  - 93.7500000077*x

  + 45.0000000042,

 {45.0000000042, - 93.7500000077,67.7916666713,

  - 20.2500000011,2.20833333343}}
```

Chapter 18

Graphical Display

18.1 GNUPLOT: Display of Functions and Surfaces

Graphical display of functions and data is done via an interface, the REDUCE GNUPLOT[1] package, to the popular `gnuplot`[2] graphing utility. It allows you to display functions in 2D and surfaces in 3D on a variety of output devices including X terminals, PC monitors, and postscript and LaTeX printer files.

The binary distribution of REDUCE for Microsoft Windows includes `gnuplot` version 5.4, which has its own documentation. Web REDUCE also includes `gnuplot` version 5.4. On other platforms, REDUCE uses whatever version of `gnuplot` is available.

The GNUPLOT package provides easy to use graphics output for curves or surfaces which are defined by formulas and/or data sets, and supports a variety of output devices such as `VGA screen`, `postscript`, `picTEX`, `MS Windows`. It lets you generate `gnuplot` graphical output directly from inside REDUCE, either for the interactive display of curves/surfaces or for the production of pictures on paper.

18.1.1 Command `plot`

Under the REDUCE GNUPLOT package, `gnuplot` is used as a graphical output server, invoked by the command `plot(...)`. This command can have a variable number of parameters:

- A function to plot, which can be
  - an expression with one unknown, e.g., `u*sin(u)^2`;

[1] This interface, together with the code for plotting data, was written by Herbert Melenk.
[2] https://gnuplot.sourceforge.net/

  - a list of expressions with one (identical) unknown, e.g., `{sin(u), cos(u)}`;
  - an expression with two unknowns, e.g., `u*sin(u)^2+sqrt(v)`;
  - a list of expressions with two (identical) unknowns, e.g., `{x^2+y^2,x^2-y^2}`;
  - a parametic expression of the form `point(<u>,<v>)` (for 2D plots) or `point(<u>,<v>,<w>)` (for 3D plots) where `<u>,<v>,<w>` are expressions which depend of one or two parameters; if there is one parameter, the object describes a curve in the plane (only `<u>` and `<v>`) or in 3D space; if there are two parameters, the object describes a surface in 3D. The parameters are treated as independent variables. Example: `point(sin t,cos t,t/10);`
  - an equation with a symbol on the left-hand side and an expression with one or two unknowns on the right-hand side, e.g., `dome= 1/(x^2+y^2);`
  - an equation with an expression on the left-hand side and a zero on the right-hand side describing implicitly a one-dimensional variety in the plane (implicitly given curve), e.g., `x^3 + x*y^2 - 9x = 0`, or a two-dimensional surface in three-dimensional Euclidean space;
  - an equation with an expression in two variables on the left-hand side and a list of numbers on the right-hand side; the contour lines corresponding to the given values are drawn, e.g.,
    `x^3 - y^2 + x*y = {-2,-1,0,1,2};`
  - a list of points in 2 or 3 dimensions, e.g., `{{0,0},{0,1},{1,1}}` representing a curve;
  - a list of lists of points in 2 or 3 dimensions, e.g.,
    `{{{0,0},{0,1},{1,1}}, {{0,0},{0,1},{1,1}}}`
    representing a family of curves.

- A range for a variable: this has the form `<variable>=(<lower_bound> .. <upper_bound>)` where `<lower_bound>` and `<upper_bound>` must be expressions which evaluate to numbers. If no range is specified the default range for independent variables is `(-10 .. 10)` and for the dependent variable it is set to the maximum for the `gnuplot` executable (using double floats on most IEEE machines). However, each of the variables `plot_xrange`, `plot_yrange`, `plot_zrange` can be assigned an interval of the form `(<lower_bound> .. <upper_bound>)` that sets the default range for the x, y, z direction. The z direction corresponds to the function values or surface height and setting a z-range flattens the graph at the range limits, which is especially useful for functions with singularities.

  Additionally, the number of interval subdivisions can be assigned as a formal quotient of the form

`<variable>=(<lower_bound> .. <upper_bound>)/<it>`

where `<it>` is a positive integer; e.g. `(1 .. 5)/30` means the interval from 1 to 5 subdivided into 30 pieces of equal size. A subdivision parameter overrides the value of the option `points` for this variable.

- A plot option, either as a fixed keyword, e.g., `hidden3d` or as an equation e.g., `term=pictex`; free text such as titles and labels should be enclosed in string quotes. Further details are given below.

Please note that a space has to be inserted between a number and a dot when specifying an interval, otherwise the REDUCE translator will be misled.

If a function is given as an equation, the left-hand side is mainly used as a label for the axis of the dependent variable.

In two dimensions, `plot` can be called with more than one explicit function; all curves are drawn in one picture. However, all these must use the same independent variable name. One of the functions can be a point set or a point set list. Normally all functions and point sets are plotted by lines. A point set is drawn by points only if functions and the point set are drawn in one picture.

The same applies to three dimensions with explicit functions. However, an implicitly given curve must be the sole object for one picture.

In 2D implicit and contour plots, as well as 3D surface plots, the ordering of the two independent variables is by default the standard REDUCE ordering, e.g. x, y rather than y, x. However, if both independent variables are specified explicitly via options of the form `<variable> = <range>` then the option ordering is used as the variable ordering. In 2D, the first variable is plotted horizontally and the second vertically, and in 3D the first and second independent variables and the dependent variable form a right-handed coordinate system. (See the examples at the end of this section.)

The functional expressions are evaluated in `rounded` mode. This is done automatically, it is not necessary to turn on `rounded` mode explicitly.

Examples:

```
plot(cos x);
plot(s=sin phi, phi=(-3 .. 3));
plot(sin phi, cos phi, phi=(-3 .. 3));
plot (cos sqrt(x^2 + y^2), x=(-3 .. 3), y=(-3 .. 3),
      hidden3d);
plot {{0,0},{0,1},{1,1},{0,0},{1,0},{0,1},
      {0.5,1.5},{1,1},{1,0}};

% parametric: screw
```

```
on rounded;
w := for j := 1:200 collect
                {1/j*sin j, 1/j*cos j, j/200}$
plot w;

% parametric: globe
dd := pi/15$
w := for u := dd step dd until pi-dd collect
    for v := 0 step dd until 2pi collect
      {sin(u)*cos(v), sin(u)*sin(v), cos(u)}$
plot w;

% implicit: superposition of polynomials
plot((x^2+y^2-9)*x*y = 0);
```

Piecewise-defined functions

A composed graph can be defined by a rule-based operator. In that case each rule must contain a clause which restricts the rule application to numeric arguments, e.g.,

```
operator my_step1;
let {my_step1(~x) => -1 when numberp x and x<-pi/2,
     my_step1(~x) =>  1 when numberp x and x>pi/2,
     my_step1(~x) => sin x
         when numberp x and -pi/2<=x and x<=pi/2};
plot(my_step1(x));
```

Of course, such a rule may call a procedure:

```
procedure my_step3(x);
   if x<-1 then -1 else if x>1 then 1 else x;
operator my_step2;
let my_step2(~x) => my_step3(x) when numberp x;
plot(my_step2(x));
```

The direct use of a procedure with a numeric `if` clause is impossible.

Plot options

The following options are specific to the REDUCE `plot` command:

- `points=<integer>`: the number of unconditionally computed data points; for a grid, `points^2` grid points are used. The default value is 20. The value of `points` is used only for variables for which no individual interval subdivision has been included in the range specification.
- `refine=<integer>`: the maximum depth of adaptive interval intersections for 2D plots (only). The default is 8. A value of 0 switches any refinement off. Note that a high value may increase the computing time significantly.

Additional options

Additional `gnuplot` options are supported via the following syntax in the `plot` command.

The following options are implemented using the `gnuplot set` (or `unset` if the option begins with `no`) command, which offers far more control than is currently exposed via this interface. Note that the `gnuplot reset` command, which clears the effect of most `set` commands (not `set term` or `set output`) from previous calls of `plot`, is always issued before any `set` commands. See *Commands / Set-show* and *Commands / Reset* in the `gnuplot` documentation or the `gnuplot` Help for details.

- `title=<string>`: the specified title is put at the top of the picture.
- `xlabel=<string>`, `ylabel=<string>`, and `zlabel=<string>` for surfaces: the specified axis labels are displayed. If omitted the axes are labeled by the independent and dependent variable names from the expression. Note that `xlabel`, `ylabel`, and `zlabel` here are used in the usual sense, x for the horizontal and y for the vertical axis in 2D, and z for the perpendicular axis in 3D – these names do not refer to the variable names used in the expressions.

```
plot(1,x,(4*x^2-1)/2,(x*(12*x^2-5))/3, x=(-1 .. 1),
   ylabel="L(x,n)", title="Legendre Polynomials");
```

- `terminal=name`: prepare output for device type `name`. Every installation uses a default terminal as output device; some installations support additional devices such as printers; consult the `gnuplot` documentation or the `gnuplot` Help for details.
- `output="filename"`: redirect the output to a file.
- `size=<string>`: set the size of the plot via the `gnuplot` command `set size <string>`, for example...

  - `size="s_x,s_y"`: rescale the graph (not the window) where s_x and s_y are scaling factors for the x- and y-sizes, e.g.

    ```
    plot(1/(x^2+y^2), x=(0.1 .. 5), y=(0.1 .. 5),
       size="0.7,1");
    ```

    Defaults are $s_x = 1, s_y = 1$. Note that scaling factors greater than 1 will often cause the picture to be too big for the window.
  - `size="ratio -1"`: set the scales so that the unit has the same length on both the x and y axes, which is essential for geometrical plots to look correct, e.g.

    ```
    plot(x^2+y^2-1=0, x=(-2 .. 2), y=(-2 .. 2),
       size="ratio -1");
    ```

- `view="r_x,r_z"`: set the viewpoint in 3 dimensions by turning the object around the x or z axis; the values are degrees (integers). Defaults are $r_x = 60, r_z = 30$.

  ```
  plot(1/(x^2+y^2), x=(0.1 .. 5), y=(0.1 .. 5),
     view="30,130");
  ```

- `logscale`: set all axes to use logarithmic scales.
- `contour` resp. `nocontour`: in 3 dimensions an additional contour map is drawn (default: `nocontour`). Note that `contour` is an option which is executed by `gnuplot` by interpolating the precomputed function values. If you want to draw contour lines of a delicate formula, you had better use the contour form of the REDUCE `plot` command as described above.
- `surface` resp. `nosurface`: in 3 dimensions the surface is drawn, resp. suppressed (default: `surface`).
- `hidden3d`: apply hidden line removal in 3 dimensions.
- `pm3d`: draw a solid coloured (palette-mapped 3d) surface in 3 dimensions.

The following option is implemented using the `gnuplot` command `with <style>`, which offers far more control than is currently exposed via this interface. See *Plotting styles* in the `gnuplot` documentation or the `gnuplot` Help for details.

- `style=` one of `lines`, `points`, `linespoints`, `impulses`, `dots`, `errorbars`, `boxes`, `boxerrorbars`, `boxxyerrorbars`, `candlesticks`, `financebars`, `fsteps`, `histeps`, `steps`, `vector`, `xerrorbars`, `xyerrorbars`, `yerrorbars`: set the display style.

18.1.2 Paper output

The following example works for a PostScript printer. If your printer uses a different communication, please find the correct setting for the `terminal` variable in the `gnuplot` documentation.

For a PostScript output, you need to add the options `terminal=postscript` and `output="filename"` to your plot command, e.g.,

```
plot(sin x, x=(0 .. 10), terminal=postscript,
     output="sin.ps");
```

18.1.3 Mesh generation for implicit curves

The basic mesh for finding an implicitly-given curve, the x, y plane is subdivided into an initial set of triangles. Those triangles which have an explicit zero point or which have two points with different signs are refined by subdivision. A further refinement is performed for triangles which do not have exactly two zero neighbours because such places may represent crossings, bifurcations, turning points or other difficulties. The initial subdivision and the refinements are controlled by the option `points` which is initially set to 20: the initial grid is refined unconditionally until approximately `points * points` equally-distributed points in the x, y plane have been generated.

The final mesh can be visualized in the picture by setting

```
on show_grid;
```

18.1.4 Mesh generation for surfaces

By default the functions are computed at predefined mesh points: the ranges are divided by the number associated with the option `points` in both directions.

For two dimensions the given mesh is adaptively smoothed when the curves are too coarse, especially if singularities are present. On the other hand refinement can be rather time-consuming if used with complicated expressions. You can control it with the option `refine`. At singularities the graph is interrupted.

In three dimensions no refinement is possible as `gnuplot` supports surfaces only with a fixed regular grid. In the case of a singularity the near neighborhood is tested; if a point there allows a function evaluation, its clipped value is used instead, otherwise a zero is inserted.

When plotting surfaces in three dimensions you have the option of hidden line removal. Because of an error in Gnuplot 3.2 the axes cannot be labeled correctly

when hidden3d is used ; therefore they aren't labelled at all. Hidden line removal is not available with point lists.

18.1.5 gnuplot operation

The command `plotreset;` deletes the current `gnuplot` output window. The next call to `plot` will then open a new one.

If `gnuplot` is invoked directly by an output pipe (UNIX and Windows), an eventual error in the `gnuplot` data transmission might cause `gnuplot` to quit. As REDUCE is unable to detect the broken pipe, you have to reset the plot system by calling the command `plotreset;` explicitly. Afterwards new graphics output can be produced.

Under Windows 3.1 and Windows NT, `gnuplot` has a text and a graph window. If you don't want to see the text window, iconify it and activate the option `update wgnuplot.ini` from the graph window system menu – then the present screen layout (including the graph window size) will be saved and the text windows will come up iconified in future. You can also select some more features there and so tailor the graphic output. Before you terminate REDUCE you should terminate the graphic window by calling `plotreset;`. If you terminate REDUCE without deleting the `gnuplot` windows, use the command button from the `gnuplot` text window – it offers an exit function.

18.1.6 Saving gnuplot command sequences

If you want to use the internal `gnuplot` command sequence more than once (e.g., for producing a picture for a publication), you may set

```
on trplot, plotkeep;
```

`trplot` causes all `gnuplot` commands to be written additionally to the actual REDUCE output. Normally the data files are erased after calling `gnuplot`, however with `plotkeep` on the files are not erased.

18.1.7 Direct Call of gnuplot

`gnuplot` has a lot of facilities which are not accessed by the operators and parameters described above. Therefore genuine `gnuplot` commands can be sent by REDUCE. Please consult the `gnuplot` manual for the available commands and parameters. The general syntax for a `gnuplot` call inside REDUCE is

```
gnuplot(<cmd>,<p_1>,<p_2> ...)
```

where `cmd` is a command name and $p_1, p_2, \ldots$ are the parameters, inside REDUCE separated by commas. The parameters are evaluated by REDUCE and then transmitted to `gnuplot` in `gnuplot` syntax. Usually a drawing is built by a sequence of commands which are buffered by REDUCE or the operating system. For terminating and activating them use the REDUCE command `plotshow`. Example:

```
gnuplot(set,polar);
gnuplot(unset,parametric);
gnuplot(set,dummy,x);
gnuplot(plot, x*sin x);
plotshow;
```

In this example the function expression is transferred literally to `gnuplot`, while REDUCE is responsible for computing the function values when `plot` is called. Note that `gnuplot` restrictions with respect to variable and function names have to be taken into account when using this type of operation. **Important**: String quotes are not transferred to the `gnuplot` executable; if the `gnuplot` syntax needs string quotes, you must add doubled stringquotes *inside* the argument string, e.g.,

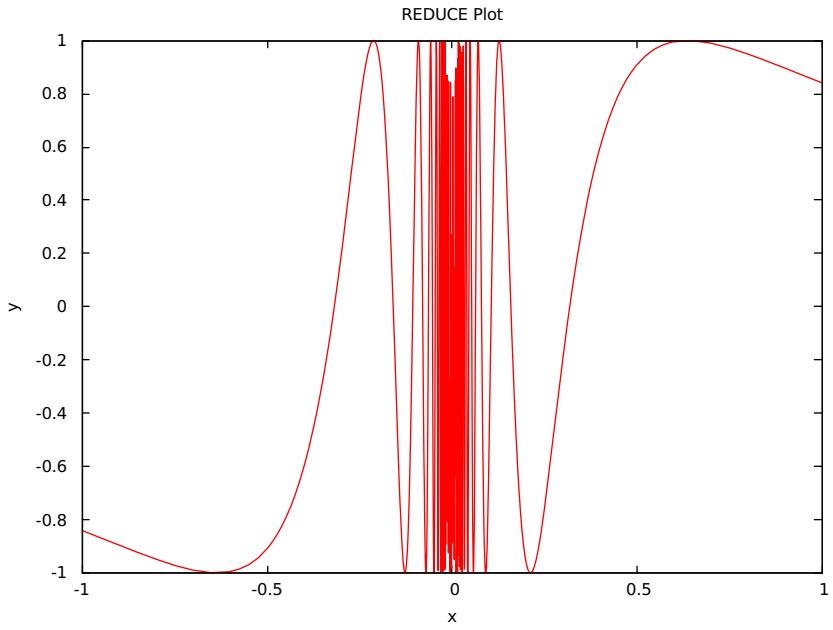
```
gnuplot(plot, """mydata""", "using 2:1");
```

18.1.8 Examples

The following are taken from a collection of sample plots (`gnuplot.tst`) and a set of tests for plotting special functions. The pictures are made using the `qt gnuplot` device and using the menu of the graphics window to export to PDF or PNG.

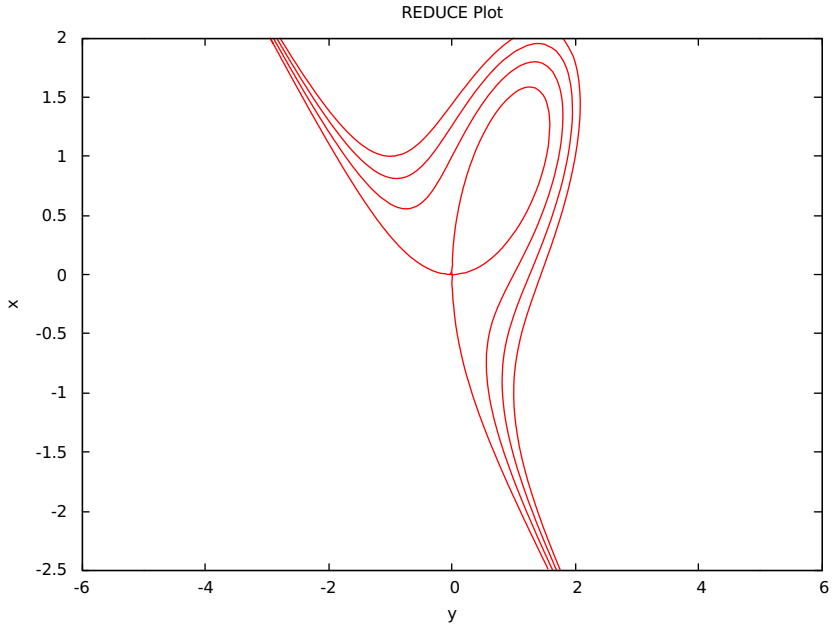
A simple plot for $\sin(1/x)$:

```
plot(sin(1/x), x=(-1 .. 1), y=(-3 .. 3));
```



Some implicitly-defined curves:

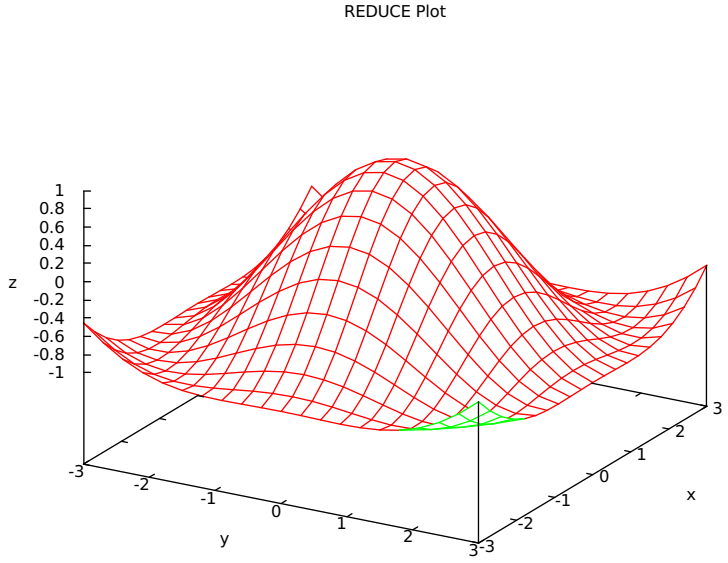
```
plot(x^3 + y^3 - 3*x*y = {0,1,2,3},
     y=(-5 .. 5), x=(-2.5 .. 2));
```



(Note that the y-axis is plotted horizontally since it is specified first among the options.)

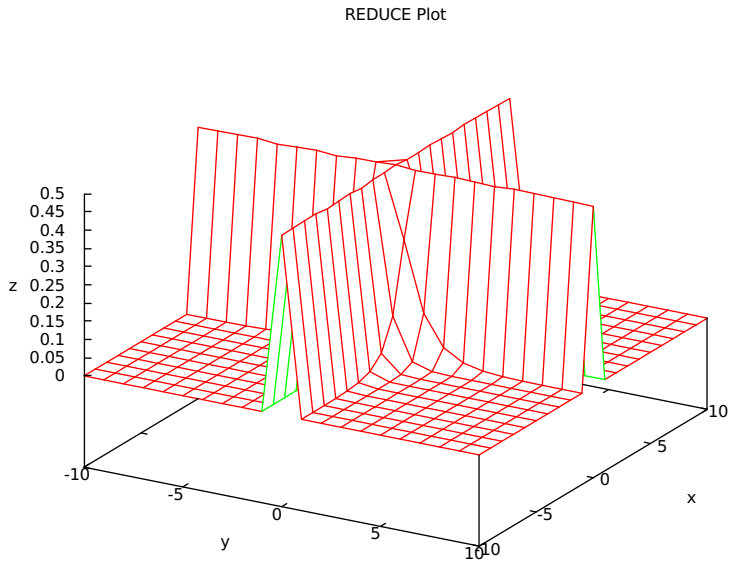
A test for hidden surfaces:

```
plot(cos sqrt(x^2 + y^2), y=(-3 .. 3), x=(-3 .. 3),
     hidden3d);
```



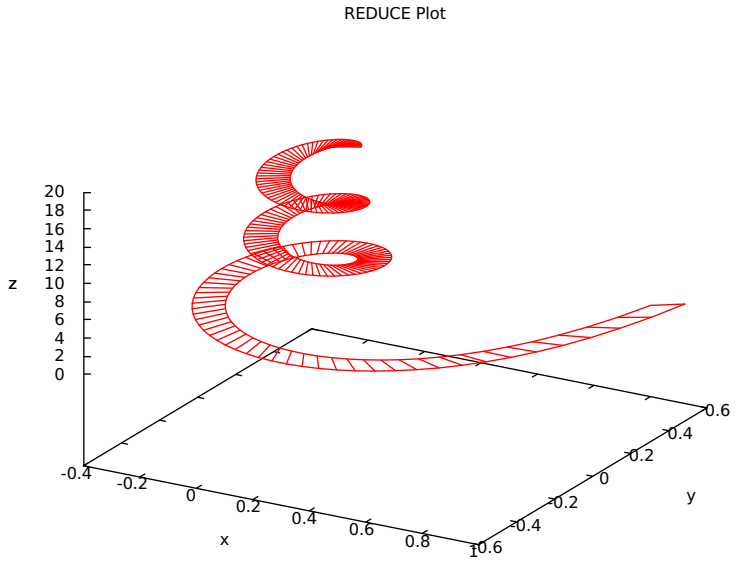
(Note the left-handed coordinate system since y is specified before x among the options; for a right-handed coordinate system specify x before y.)

```
plot(sinh(x*y)/sinh(2*x*y),
     y=(-10 .. 10), x=(-10 .. 10), hidden3d);
```



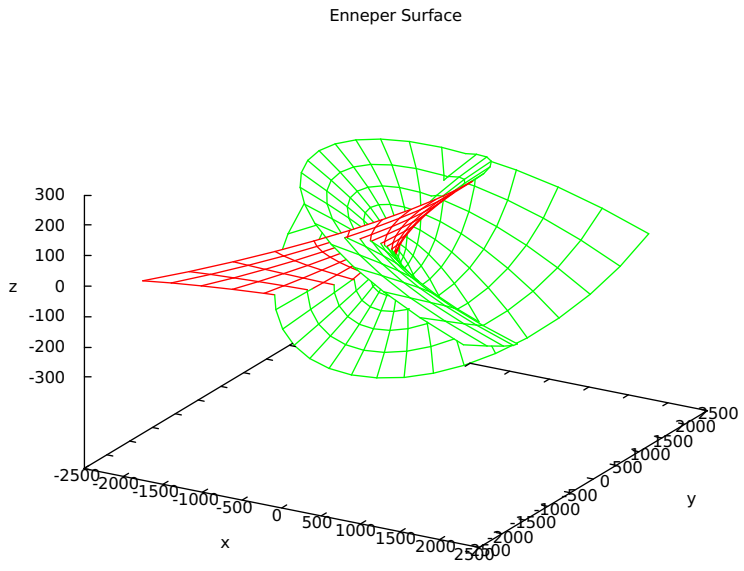
(Note the left-handed coordinate system since y is specified before x among the options; for a right-handed coordinate system specify x before y.)

```
on rounded;
w:= {for j:=1 step 0.1 until 20 collect
       {1/j*sin j, 1/j*cos j, j},
     for j:=1 step 0.1 until 20 collect
       {(0.1+1/j)*sin j, (0.1+1/j)*cos j, j} }$
plot w;
```



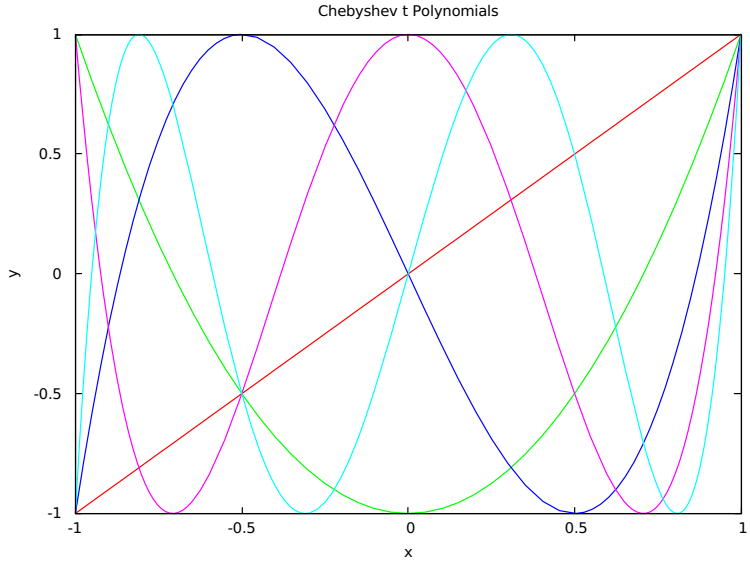
The following example is taken from: Cox, Little, O'Shea, *Ideals, Varieties and Algorithms*:

```
plot(point(3u+3u*v^2-u^3, 3v+3u^2*v-v^3, 3u^2-3v^2),
     hidden3d, title="Enneper Surface");
```

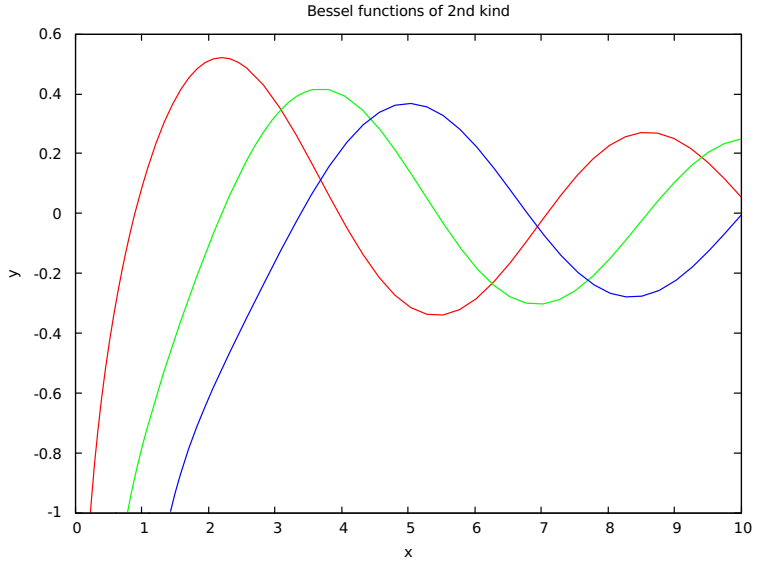


The following examples use the `specfn` package to draw a collection of Chebyshev T polynomials and Bessel Y functions. The special function package has to be loaded explicitely to make the operator ChebyshevT and BesselY available.

```
load_package specfn;
plot(chebyshevt(1,x), chebyshevt(2,x), chebyshevt(3,x),
     chebyshevt(4,x), chebyshevt(5,x),
     x=(-1 .. 1), title="Chebyshev t Polynomials");
```




```
plot(bessely(0,x), bessely(1,x), bessely(2,x),
     x=(0.1 .. 10), y=(-1 .. 1),
     title="Bessel functions of 2nd kind");
```



18.2 Turtle Graphics

18.2.1 Turtle Graphics

Turtle Graphics was originally developed in the 1960's as part of the LOGO system, and used in the classroom as an introduction to graphics and using computers to help with mathematics.

The LOGO language was created as part of an experiment to test the idea that programming may be used as an educational discipline to teach children. It was first intended to be used for problem solving, for illustrating mathematical concepts usually difficult to grasp, and for creation of experiments with abstract ideas.

At first LOGO had no graphics capabilities, but fast development enabled the incorporation of graphics, known as "Turtle Graphics" into the language. "Turtle Graphics" is regarded by many as the main use of LOGO.

For references, see [PZ97, LM94].

Main Idea: To use simple commands directing a turtle, such as forward, back, turnleft, in order to construct pictures as opposed to drawing lines connecting cartesian coordinate points.

The 'turtle' is at all times determined by its state $\{x,y,a,p\}$ – where x,y determine its position in the (x,y)-plane, a determines the angle (which describes the direction the turtle is facing) and p signals whether the pen is up or down (i.e. whether or not it is drawing on the paper).

Some alterations to the original "Turtle Graphics" commands have been made in this implementation due to the design of the graphics package *gnuplot* used in REDUCE.[3]

- It is not possible to draw lines individually and to see each separate line as it is added to the graph since gnuplot automatically replaces the last graph each time it calls on the plot function.

  Thus the whole sequence of commands must be input together if the complete picture is to be seen.

- This implementation does not make use of the standard turtle commands 'pen-up' or 'pen-down'. Instead, 'set' commands are included which allow the turtle to move without drawing a line.

- No facility is provided here to change the pen-colour, but gnuplot does have the capability to handle a few different colours (which could be included later).

[3]The code of the turtle package was written by Caroline Cotter.

- The user has no control over the range of output that can be seen on the screen since the gnuplot program automatically adjusts the picture to fit the window. Hence the size of each specified 'step' the turtle takes in any direction is not a fixed unit of length, rather it is relative to the scale chosen by gnuplot.

18.2.2 Turtle Functions

As previously mentioned, the turtle is determined at all times by its state $\{x,y,a\}$: its position on the (x,y)-plane and its angle(a) – its *heading* – which determines the direction the turtle is facing, in degrees, relative anticlockwise to the positive x-axis.

User Setting Functions

setheading Takes a number as its argument and resets the heading to this number. If the number entered is negative or greater than or equal to 360 then it is automatically checked to lie between 0 and 360.

Returns the turtle position $\{x,y\}$

SYNTAX: `setheading(`θ`)`

turnleft The turtle is turned anticlockwise through the stated number of degrees. Takes a number as its argument and resets the heading by adding this number to the previous heading setting.

Returns the turtle position $\{x,y\}$

SYNTAX: `turnleft(`α`)`

turnright Similar to `turnleft`, but the turtle is turned clockwise through the stated number of degrees. Takes a number as its argument and resets the heading by subtracting this number from the previous heading setting.

Returns the turtle position $\{x,y\}$

SYNTAX: `turnright(`β`)`

setx Relocates the turtle in the x direction. Takes a number as its argument and repositions the state of the turtle by changing its x-coordinate.

Returns {}

SYNTAX: `setx(`x`)`

sety Relocates the turtle in the y direction. Takes a number as its argument and repositions the state of the turtle by changing its y-coordinate.

Returns {}

SYNTAX: `sety(`y`)`

setposition Relocates the turtle from its current position to the new cartesian coordinate position described. Takes a pair of numbers as its arguments and repositions the state of the turtle by changing the x and y coordinates.

Returns {}

SYNTAX: `setposition(`x`,`y`)`

setheadingtowards Resets the heading so that the turtle is facing towards the given point, with respect to its current position on the coordinate axes. Takes a pair of numbers as its arguments and changes the heading, but the turtle stays in the same place.

Returns the turtle position {x,y}

SYNTAX: `setheadingtowards(`x`,`y`)`

setforward Relocates the turtle from its current position by moving forward (in the direction of its heading) the number of steps given. Takes a number as its argument and repositions the state of the turtle by changing the x and y coordinates.

Returns {}

SYNTAX: `setforward(`n`)`

setback As with `setforward`, but moves back (in the opposite direction of its heading) the number of steps given.

Returns {}

SYNTAX: `setback(`n`)`

Line-Drawing Functions

forward Moves the turtle forward (in the direction its heading) the number of steps given. Takes a number as its argument and draws a line from its current position to a new position on the coordinate plane. The x and y coordinates are reset to the new values.

Returns the list of points { {*old* x,*old* y}, {*new* x,*new* y} }

SYNTAX: `forward(`s`)`

back As with `forward` except the turtle moves back (in the opposite direction to its heading) the number of steps given.

Returns the list of points { {*old* x,*old* y}, {*new* x,*new* y} }

SYNTAX: `back(`s`)`

move Moves the turtle to a specified point on the coordinate plane. Takes a list of two numbers as its argument and draws a line from its current position to the position described. The x and y coordinates are set to these new values.

Returns the list of points { {*old* x,*old* y}, {*new* x,*new* y} }

SYNTAX: `move{`x`,`y`}`

Plotting Functions

draw This is the function the user calls within REDUCE to draw the list of turtle commands given into a picture. Takes a list as its argument, with each separate command being separated by a comma, and returns the graph drawn by following the commands.

SYNTAX: `draw{`*`command(command_args),...,`*
`command(command_args)``}`

Note: all commands may be entered in either long or shorthand form, and with a space before the arguments instead of parentheses only if just one argument is needed. Commands taking more than one argument must be written with parentheses and arguments separated by a comma.

Other Important Functions

info This function is called on its own in REDUCE to tell user the current state of the turtle. Takes no arguments but returns a list containing the current values of the x and y coordinates and the heading variable.

Returns the list {*x_coord*,*y_coord*,*heading*}

SYNTAX: `info()` or simply `info`

clearscreen This is also called on its own in REDUCE to get rid of the last gnuplot window, displaying the last turtle graphics picture, and to reset all the variables to 0. Takes no arguments and returns no printed output to the screen but the graphics window is simply cleared.

SYNTAX: `clearscreen()` or simply `clearscreen`

or `cls`

home This is a command which can be called within a plot function as well as outside of one. Takes no arguments, and simply resets the x and y coordinates and the heading variable to 0. When used in a series of turtle commands, it moves the turtle from its current position to the origin and sets the direction of the turtle along the x-axis, without drawing a line.

Returns {0,0}

SYNTAX: `home()` or simply `home`

Defining Functions

It is possible to use conditional statements (if ... then ... else ...) and 'for' statements (for i:=...collect{...}) in calls to draw. However, care must be taken – when using conditional statements the final else statement must return a point or at least {x_coord,y_coord} if the picture is to be continued at that point. Also, 'for' statements *must* include 'collect' followed by a list of turtle commands (in addition, the variable must begin counting from 0 if it is to be joined to the previous list of turtle commands at that point exactly, e.g. for i:=0:10 collect {...}).

SYNTAX: `(For user-defined Turtle functions)`

```
procedure func_name(func_args);
begin [scalar additional variables];
    ⋮
    (the procedure body containing some turtle commands)
    ⋮
    return (a list, or label to a list, of turtle commands
        as accepted by draw)
end;
```

For convenience, it is recommended that all user defined functions, such as those involving `if...then...else...` or `for i:=...collect{...}` are defined together in a separate file, then called into REDUCE using the `in "filename"` command.

18.2.3 Global variables

The following variables are global, so it is advised that these are not altered directly:

x_coord The current x coordinate.

y_coord The current y coordinate.

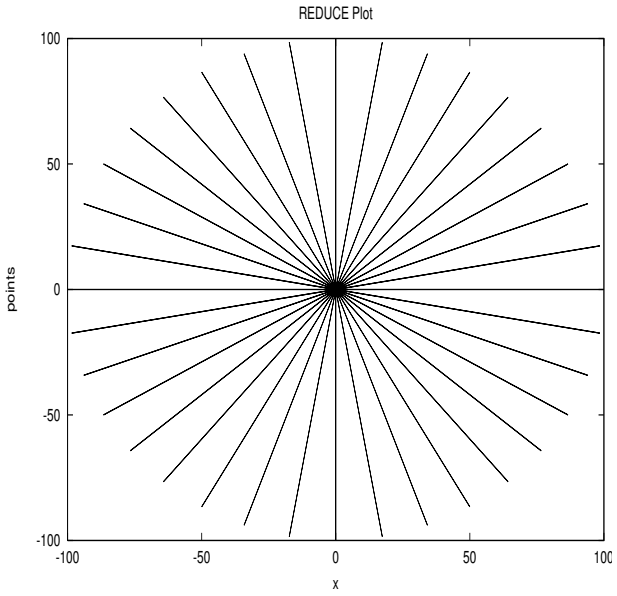
heading The current heading, as set by the `setheading` function.

18.2.4 Examples

The following examples are taken from the turtle.tst file. Examples 1, 2, 5 & 6 are simple calls to draw. Examples 3 & 4 show how more complicated commands can be built (which can take their own set of arguments) using procedures. Example 7 shows two graphs drawn together.

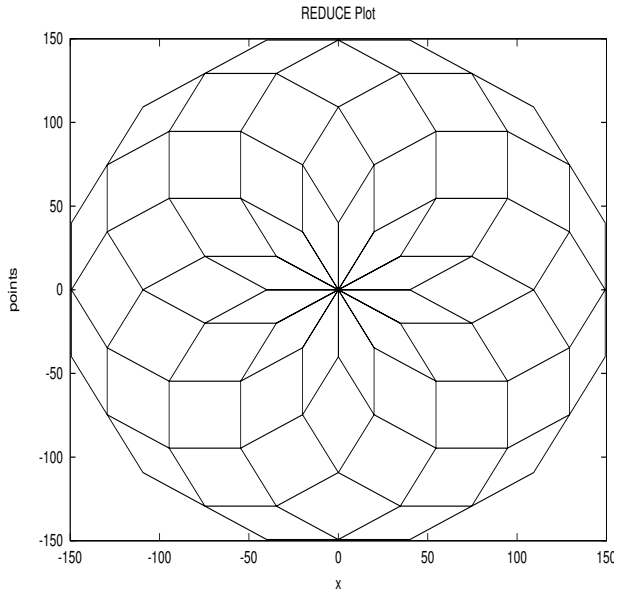
Example 1: Draw 36 rays of length 100

```
draw {for i:=1:36 collect
         {setheading(i*10), forward 100, back 100} };
```



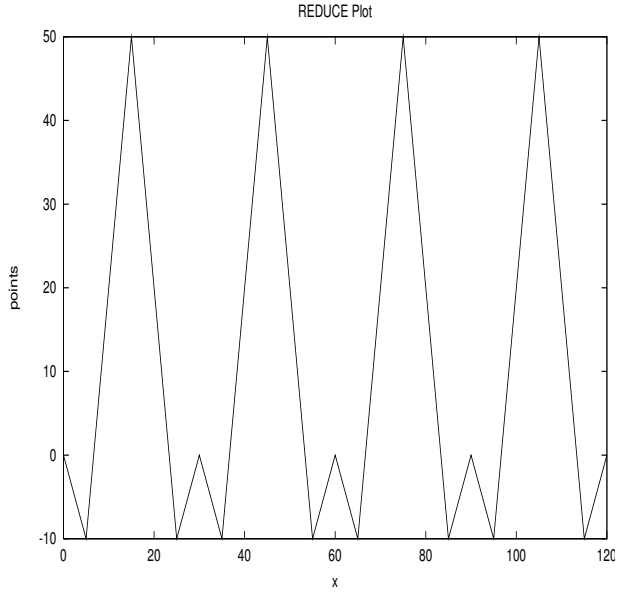
Example 2: Draw 12 regular polygons with 12 sides of length 40, each polygon forming an angle of 360/n degrees with the previous one.

```
draw {for i:=1:12 collect
          {turnleft(30),
           for j:=1:12 collect
               {forward 40, turnleft(30)}} };
```



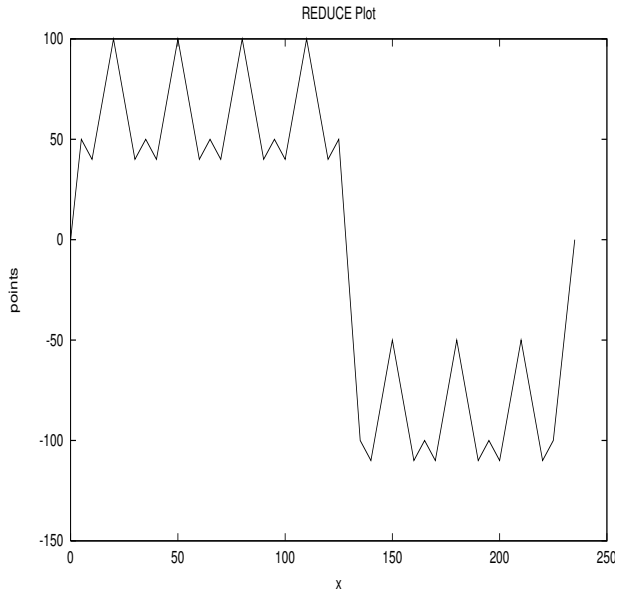
Example 3: A "peak" pattern - an example of a recursive procedure.

```
procedure peak(r);
begin;
  return for i:=0:r collect
              {move{x_coord+5,y_coord-10},
               move{x_coord+10,y_coord+60},
               move{x_coord+10,y_coord-60},
               move{x_coord+5,y_coord+10}};
end;

draw {home(), peak(3)};
```



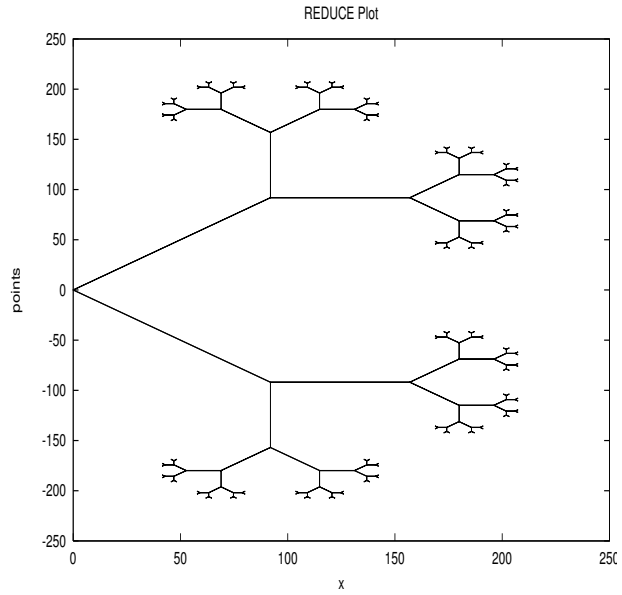
This procedure can then be part of a longer chain of commands:

```
draw {home(), move{5,50}, peak(3),
      move{x_coord+10,-100}, peak(2),
      move{x_coord+10,0}};
```



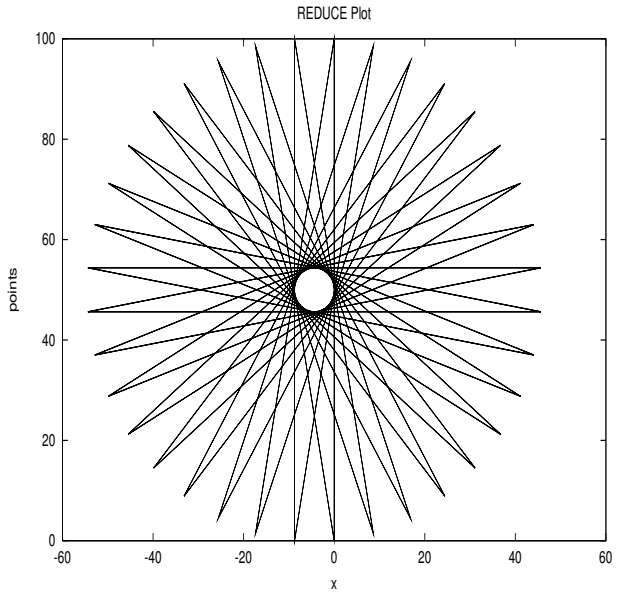
Example 4: Write a recursive procedure which draws "trees" such that every branch is half the length of the previous branch.

```
procedure tree(a,b);      %Here: a is the start length,
                          %b is the number of levels
begin;
  return if fixpb and b>0     %checking b is a positive
                              %integer

            then {turnleft(45), forward a,
                  tree(a/2,b-1), back a,
                  turnright(90), forward a,
                  tree(a/2,b-1), back a, turnleft(45)}
         else {x_coord,y_coord}; %default:
                                 %Turtle stays still
end;

draw {home(), tree(130,7)};
```



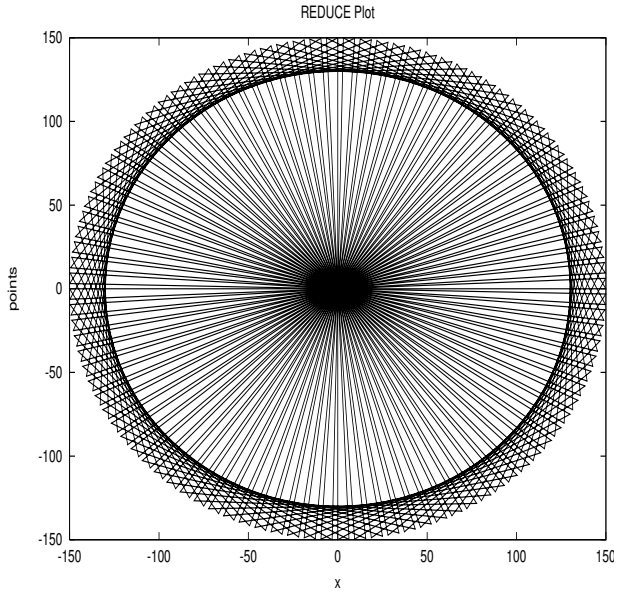
Example 5: A 36-point star.

```
draw {home(), for i:=1:36 collect
                  {turnleft(10), forward 100,
                   turnleft(10), back 100} };
```



Example 6: Draw 100 equilateral triangles with the leading points equally spaced on a circular path.

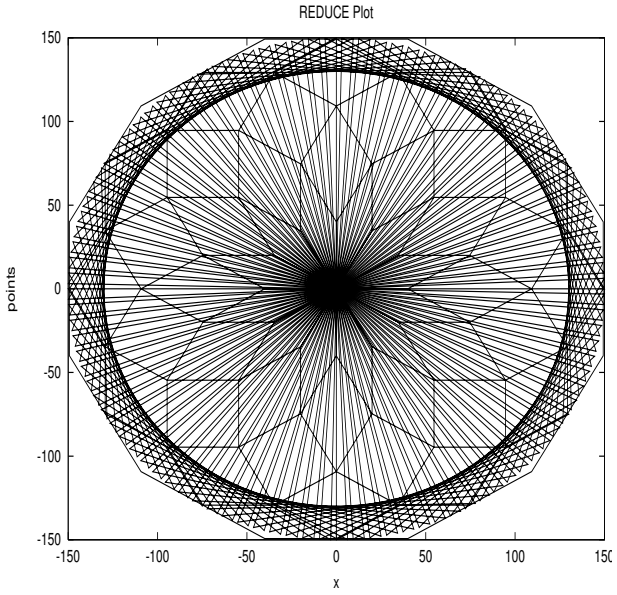
```
draw {home(), for i:=1:100 collect
                  {forward 150, turnright(60),
                   back(150), turnright(60),
                   forward 150, setheading(i*3.6)} };
```



Example 7: Two or more graphs can be drawn together (this is easier if the graphs

are named). Here we show graphs 2 and 6 on top of one another:

```
gr2:={home(), for i:=1:12 collect
                  {turnleft(30),
                   for j:=1:12 collect
                       {forward 40, turnleft(30)}} }$

gr6:={home(), for i:=1:100 collect
                  {forward 150, turnright(60),
                   back(150), turnright(60),
                   forward 150, setheading(i*3.6)} }$

draw {gr2, gr6};
```



18.3 Logo Turtle Graphics

18.3.1 Introduction

Logo Turtle Graphics[4] (henceforth referred to as "LogoTurtle") is a REDUCE emulation of traditional Logo turtle graphics with one turtle, modelled on Berkeley Logo by Brian Harvey and FMSLogo by David Costanzo (which is an updated version of George Mills' MSWLogo, a multimedia-enhanced version for Microsoft Windows, which is itself based on Berkeley Logo). This manual section is derived

[4]The Logo Turtle Graphics package was written by Francis Wright.

primarily from the Graphics chapter of the Berkeley Logo manual available from GitHub.

This package is inspired by, and related to, the REDUCE Turtle package by Caroline Cotter (ZIB, Berlin, 1998), and the word "Turtle" below (with a capital T) will refer specifically to that package. Both packages are built on the REDUCE Gnuplot package, which itself uses Gnuplot to display plots. This means that plotting is not fully interactive as it would be in traditional Logo; a plot is constructed invisibly and only displayed when requested. However, turning on the `LOGOTURTLE_AUTODRAW` switch makes LogoTurtle as interactive as possible. This package aims to be more efficient, more authentic, more interactive and more complete than Turtle.

Note that LogoTurtle and Turtle cannot both be run in the same REDUCE session because they define some procedures with the same names.

18.3.2 Design

LogoTurtle is entirely functional. It uses "getters" and "setters", and does not use any algebraic-mode variables (unlike Turtle). Most command names are as in Berkeley Logo and/or FMSLogo, and their function is the same or similar. (Identical behaviour is not always possible.) However, all commands are REDUCE procedure calls.

Getters (query procedures) return values that are accepted as input by their matching setters (command procedures). If more than one data value is involved then a list is used. For example, the `LABELFONT` query returns a list of the current label font face and size if both are set, which the corresponding `SETLABELFONT` command accepts as its argument.

LogoTurtle commands other than queries return nothing (`nil`) and plotting is achieved via side effects, not via returned values as for Turtle. A plot is displayed by calling the (non-traditional) `DRAW` command as for Turtle. The plot displayed need not be complete; `DRAW` displays the plot constructed so far, which allows an element of interactivity.

LogoTurtle makes essential use of commands to lower and raise the pen (unlike Turtle; see Pen and Background Control). Lowering the pen begins a "curve", namely a sequence of points connected by straight lines, and raising the pen ends that curve. Each time the pen is lowered, the turtle moved and the pen raised, a distinct curve is produced.

LogoTurtle uses Lisp floating-point numbers internally and does not require any particular REDUCE number domain settings. However, all command arguments and list elements relating to turtle position must be expressions that can be evaluated to real numbers. All returned values and list elements relating to turtle po-

sition will be floating-point numbers. Note that LogoTurtle lists are REDUCE algebraic-mode lists delimited by curly braces, `{ }`, not the square brackets used in traditional Logo, and that REDUCE list elements must be separated by commas.

18.3.3 User Interface

To use LogoTurtle, execute the REDUCE command

```
load_package logoturtle;
```

LogoTurtle sets the scaling of the two axes to be the same so that the aspect ratio is 1:1 and geometry is correct (although beware that Gnuplot may not always honour this). By default, LogoTurtle scales the graphics window so that turtle coordinates $(-100, -100)$ and $(100, 100)$ fit, and the center of the graphics window is turtle location $(0, 0)$, i.e. the origin or home position. But this fixed window can be turned off so that Gnuplot automatically sizes the display to include the whole plot (as for Turtle). The position of the origin (the turtle home location $(0, 0)$) is then not fixed and may be different for different plots. The window size can also be changed; see Turtle and Window Control for further details.

Positive x is to the right; positive y is up. Headings (angles) are measured in degrees clockwise from the positive y axis. (Note that this differs from the common mathematical convention, also used by Turtle, of measuring angles counterclockwise from the positive x axis!) Initially, the turtle is at the origin (Cartesian coordinates $(0, 0)$) facing up (heading 0 degrees) with the pen up.

LogoTurtle uses by default a white background, and pen, fill and label colours chosen automatically by Gnuplot, but you can set all these to any colour provided by Gnuplot.

Note that LogoTurtle command names are shown using upper case letters in the descriptions after the example below to distinguish them clearly from their arguments, but LogoTurtle is case insensitive, so commands can be entered in either case.

Commands that never take any arguments use special syntax and *need not be followed by empty parentheses*. Query commands that take no arguments can be used like (read-only) variables. For example, the following command increments the x-coordinate of the turtle by 10 steps:

```
setx(xcor + 10);
```

Multiple command arguments must be enclosed in parentheses; single or no arguments may be enclosed in parentheses, although parentheses can be, and usually are, omitted with a single argument or no arguments. However, if a single argument

is an expression involving infix operators then it must be enclosed in parentheses.

When the switch `TRLOGOTURTLE` is on, LogoTurtle outputs Cartesian coordinates corresponding to every move of the turtle and `DRAW` outputs the list of curves that it is about to draw. When the switch `TRPLOT` is on, the commands sent to Gnuplot (but not the actual plot data) are also output. Turning the switch `TRLOGOTURTLE` on or off also turns the switch `TRPLOT` on or off. Both switches are off by default.

18.3.4 A Simple Example

This example assumes LogoTurtle has just been loaded but not yet used. If this is not the case then first execute the commands

```
clearscreen; penup;
```

The following code draws an equilateral triangle with side length 100 centred on the origin, with one vertex on the positive Y axis. The sides are coloured red, green and blue. To make this example as interactive as possible, the plot is displayed after each side is drawn (but for this effect to be visible each line ending with `draw;` must be executed separately; if you input all the commands together then you will only see the complete triangle).

```
forward(100/sqrt 3); pendown;
setpencolor red; right 150; forward 100; draw;
setpencolor green; right 120; forward 100; draw;
setpencolor blue; right 120; forward 100; draw;
```

For more interesting examples, please see the files `logoturtle.tst` and `mondrian.tst`, which can be found online or in the `packages/plot` directory in a standard REDUCE distribution.

18.3.5 Turtle

The turtle is optionally displayed as an unfilled isosceles triangle; see Turtle and Window Control. The turtle is drawn using black default-thickness lines on top of the current plot; the colour and line thickness cannot currently be changed. The turtle is never wrapped, although it is clipped if any turtle mode is in effect, i.e. for drawing the turtle any turtle mode other than `false` is treated as `window`.

The turtle looks like an arrow head pointing in the direction of the turtle's heading. The height of the isosceles triangle (i.e. the length of the turtle) is $\ell(= 0.1)$ times the average of the maximum x and y coordinates set by the window size (even if windowing is not in effect) and the apex angle (at the head of the turtle) is twice

$\alpha(= 10°)$ (so that the two equal sides are at angles of α to the turtle's heading).[5]

The nominal turtle position is at the midpoint of the base (the short side). However, the turtle is drawn one step behind its nominal position, so that the display of the base of the turtle's triangle does not obscure a line drawn perpendicular to it (as would happen after drawing a square).

18.3.6 Colours

LogoTurtle offers both the traditional Berkeley Logo palette model and some of the Gnuplot model, which is much more flexible and usually more convenient. Colours may be input and output in several different formats, but colour numbers and RGB lists are used only for input and only the formats accepted by Gnuplot are used for output, since these are the only formats used internally.

A colour number (input only) is an integer between 0 and 15 inclusive. The initial colour assignments are

0	black	1	blue	2	green	3	cyan
4	red	5	magenta	6	yellow	7	white
8	brown	9	tan	10	forest	11	aqua
12	salmon	13	purple	14	orange	15	grey

but other colours can be assigned to numbers 8–15 by the `SETPALETTE` command. Colour numbers are useful for cycling programmatically through a range of colours.

An RGB list (input only) is a list of three nonnegative numbers not greater than 100 specifying the percent saturation of red, green, and blue in the desired colour. RGB lists are also easy to use programmatically.

An identifier or string (input and output) can be used to represent a colour in any way that is acceptable to Gnuplot, such as a colour name or hexadecimal number, e.g. `red`, `"red"` or `"#FF0000"`.

The identifier `FALSE` (input and output) means that no colour is set. In this case, Gnuplot uses its own automatic colour-choice algorithm.

[5] The turtle length relative to the window size ℓ and head half-angle α are respectively the Lisp float values of the symbolic-mode global variables `logoturtle!-rel!-len!*` and `logoturtle!-angle!*`. These values are used dynamically and in principle you could change them after LogoTurtle has loaded, but currently there is no algebraic-mode facility to do this.

18.3.7 Displaying Logo Turtle Graphics

Draw command

```
DRAW
```

can be used at any time to display the current plot, i.e. the plot and/or labels constructed so far (provided there is something to display). It initially opens a Gnuplot window and subsequently updates it.

Autodraw switch

When the switch `LOGOTURTLE_AUTODRAW` is on, making any visible change to a plot causes it to be redrawn (or drawn) automatically. In this situation it can be useful to turn on display of the turtle (using `SHOWTURTLE`), since this makes the turtle's heading and position visible even when the pen is up.

When using REDUCE programming constructs (e.g. group or loop statements) to run multiple LogoTurtle commands, it is best to have the `LOGOTURTLE_AUTODRAW` switch off, and instead to execute the `DRAW` command explicitly once the programming construct has completed, because intermediate changes to the plot will not be visible and repeatedly redrawing it is pointless.

18.3.8 Turtle Motion

Forward command

```
FORWARD dist
```

moves the turtle forward, in the direction that it's facing, by the specified distance (measured in turtle steps).

Back command

```
BACK dist
```

moves the turtle backward, i.e., exactly opposite to the direction that it's facing, by the specified distance. (The heading of the turtle does not change.)

Left command

```
LEFT degrees
```

turns the turtle counterclockwise by the specified angle, measured in degrees. (A degree is 1/360 of a full circle.)

Right command

```
RIGHT degrees
```

turns the turtle clockwise by the specified angle, measured in degrees.

Setpos command

```
SETPOS position
```

moves the turtle to an absolute position in the graphics window. The input is a list of two numbers, the x and y coordinates, e.g. "`SETPOS {50, 50}`".

Setxy command

```
SETXY(xcor, ycor)
```

moves the turtle to an absolute position in the graphics window. The two inputs are numbers, the x and y coordinates.

Setx command

```
SETX xcor
```

moves the turtle horizontally from its old position to a new absolute horizontal coordinate. The input is the new x coordinate.

Sety command

```
SETY ycor
```

moves the turtle vertically from its old position to a new absolute vertical coordinate. The input is the new y coordinate.

Setheading command

```
SETHEADING degrees
```

turns the turtle to a new absolute heading. The input is a number, the heading in degrees clockwise from the positive y axis.

Home command

```
HOME
```

moves the turtle to its starting position (the origin) and orientation. Equivalent to "`SETPOS {0, 0}; SETHEADING 0`".

Arc command

```
ARC(angle, radius)
```

draws a circular arc centred on the turtle with the specified positive radius, starting at the turtle's heading and extending clockwise through the specified angle (counter-clockwise if angle is negative). The turtle does not move and the arc is drawn as if the turtle mode is `WINDOW` for all modes unless windowing is turned off.

Circle command

```
CIRCLE radius
```

draws a circle centred on the turtle with the positive radius specified. The turtle does not move and the circle is drawn as if the turtle mode is `WINDOW` for all modes unless windowing is turned off. Equivalent to `ARC(360, radius)`.

Arc2 command

```
ARC2(angle, radius)
```

moves the turtle around a circular arc that sweeps through the specified angle with the specified positive radius. The turtle always moves forwards: if angle is positive, then the turtle moves forwards in a clockwise direction; if angle is negative, then the turtle moves forwards in a counter-clockwise direction. At the end of the arc, the turtle's heading is increased by angle.

Circle2 command

```
CIRCLE2 radius
```

moves the turtle clockwise around a circle with the specified positive radius. The turtle ends in the same position in which it starts. Equivalent to `ARC2(360, radius)`.

Ellipticarc command

```
ELLIPTICARC(range, crosswise, inline, start)
```

draws an elliptic arc based on the turtle's position and heading. The turtle does not move. The center-point of the ellipse is the turtle's current position. The size and shape of the ellipse are determined by the specified positive crosswise and inline semi-axes. The crosswise semi-axis is the distance from the turtle to the ellipse in the direction perpendicular to the turtle's current heading. The inline semi-axis is the distance from the turtle to the ellipse in the direction in which the turtle is currently heading. Hence the turtle's heading determines the orientation of the ellipse. The elliptic arc starts at angle parameter start degrees and the angle parameter sweeps through range degrees. The elliptic arc is drawn clockwise if range is positive and counter-clockwise if range is negative.

Ellipse command

```
ELLIPSE(crosswise, inline)
```

draws an ellipse based on the turtle's position and heading. The turtle does not move. The center-point of the ellipse is the turtle's current position. The size and shape of the ellipse are determined by the specified positive crosswise and inline semi-axes. The crosswise semi-axis is the distance from the turtle to the ellipse in the direction perpendicular to the turtle's current heading. The inline semi-axis is the distance from the turtle to the ellipse in the direction in which the turtle is currently heading. Hence the turtle's heading determines the orientation of the ellipse. Equivalent to `ELLIPTICARC(360, crosswise, inline, 0)`.

18.3.9 Turtle Motion Queries

Pos query

```
POS
```

returns the turtle's current position, as a list of two numbers, the x and y coordinates.

Xcor query

```
XCOR
```

returns a number, the turtle's x coordinate.

Ycor query

```
YCOR
```

returns a number, the turtle's y coordinate.

Heading query

```
HEADING
```

returns a number, the turtle's heading in degrees.

Towards query

```
TOWARDS position
```

returns a number, the heading at which the turtle should be facing so that it would point from its current position to the position given as the input in the form of a list of two numbers, the x and y coordinates.

Distance query

```
DISTANCE position
```

returns a number, the distance the turtle must travel along a straight line to reach the position given as input in the form of a list of two numbers, the x and y coordinates.

As an example of using `TOWARDS` and `DISTANCE`, here is a somewhat convoluted way to angle the turtle towards, and then move it to, the position with coordinates $(1, 2)$:

```
setheading towards {1, 2};
forward distance {1, 2};
```

18.3.10 Turtle and Window Control

Showturtle command

```
SHOWTURTLE
```

makes the turtle visible (next time the picture is drawn).

Hideturtle command

```
HIDETURTLE
```

makes the turtle invisible (next time the picture is drawn).

Clean command

```
CLEAN
```

erases the graphics window. The turtle's state (position, heading, pen mode, etc.) is not changed.

Clearscreen command

```
CLEARSCREEN
```

erases *and closes* the graphics window, and sends the turtle to its initial position and heading. Like `HOME` and `CLEAN` together.

Setwindowsize command

```
SETWINDOWSIZE n
SETWINDOWSIZE(m, n)
SETWINDOWSIZE {m, n}
```

with a single numerical argument n sets the size of the graphics window so that $-|n| \leq x, y \leq |n|$; with two numerical arguments m, n or with a single list argument `{m, n}` in which m and n are numerical it sets the size of the graphics

window so that $-|m| \leq x \leq |m|, -|n| \leq y \leq |n|$. The default window size is $-|100| \leq x, y \leq |100|$.

Wrap command

```
WRAP
```

tells the turtle to enter wrap mode. From now on, if the turtle is asked to move past the boundary of the graphics window, it will "wrap around" and reappear at the opposite edge of the window. The top edge wraps to the bottom edge, while the left edge wraps to the right edge. (So the window is topologically equivalent to a torus.) This is the turtle's initial mode. Compare `WINDOW` and `FENCE`.

Window command

```
WINDOW
```

tells the turtle to enter window mode. From now on, if the turtle is asked to move past the boundary of the graphics window, it will move offscreen. The visible graphics window is considered as just part of an infinite graphics plane; the turtle can be anywhere on the plane. (If you lose the turtle, `HOME` will bring it back to the center of the window.) Compare `WRAP` and `FENCE`.

Fence command

```
FENCE
```

tells the turtle to enter fence mode. From now on, if the turtle is asked to move past the boundary of the graphics window, it will move as far as it can and then stop at the edge with an "out of bounds" error message. Compare `WRAP` and `WINDOW`.

Setturtlemode command

```
SETTURTLEMODE mode
```

sets the turtle (windowing) mode to one of `WRAP`, `FENCE`, `WINDOW`, with meaning as above, or `FALSE`, meaning that (like Turtle) there are no constraints on where the turtle draws.

Fill command

```
FILL
```

fills the region of the graphics window bounded by the lines that have just been drawn, i.e. the current curve if the pen is down or the last curve if the pen is up (or the pen colour or size has been changed). The fill colour is the current pen colour and the pen size is ignored. The curve is implicitly closed but the turtle is not moved. For example, the following code draws a filled blue triangle and a filled green circle:

```
clearscreen;
setpencolor blue;
pendown; setxy(0, 20); setxy(20, 0); fill;
penup; setxy(50, 50);
setpencolor green;
pendown; circle(20); fill;
draw;
```

Note that filling may cause the *default* pen (and hence fill) colour to change, but if the pen colour has been set explicitly then it will not change.

Filled command

```
FILLED(colour, commands...)
```

executes the commands in the order written, remembering all points visited, and then draws the resulting curve, *starting and ending* with the turtle's initial position, filled with the specified colour; see Colours. The pen size, whether the pen is up or down, and the pen colour are all ignored. The command arguments should be commands or lists of commands that move the turtle or draw curves. For example, the following code draws the same filled blue triangle and filled green circle as in the previous example:

```
clearscreen; penup;
filled(blue, setxy(0, 20), setxy(20, 0));
setxy(50, 50);
filled(green, circle(20));
draw;
```

Note that the sequence of commands used by `FILLED` cannot be generated directly using a loop construct such as `FOR`, whereas with `FILL` it can. However, the command arguments to `FILLED` can be calls of procedures that can contain

arbitrary code, e.g.

```
procedure shape;
   for i := 1 : 4 do << forward 80; arc2(-90, 40) >>;

clearscreen; penup;
setxy(40, 80); setheading(-90);
filled(false, shape());
draw;
```

Label command

```
LABEL text
```

takes a printable item or list of printable items as input and prints it on the graphics window with the top left-hand corner of the label at the turtle's position. The items in a list are concatenated with no additional spacing. Beware that long labels will just fall off the edge of the graphics window. Multi-line labels can be produced by including newline characters encoded as `\n` within the text, e.g. `"This is a\nmulti-line label"`. (The newline is recognised by Gnuplot, not by REDUCE.)

Setlabelfont command

```
SETLABELFONT face
SETLABELFONT size
SETLABELFONT(face, size)
SETLABELFONT {face, size}
SETLABELFONT false
```

sets the face and/or size of the label font. If the face is specified then it should be the only or first input and must be an identifier or string, e.g. `"Arial"`. If the size is specified then it should be the only or second input and must be a positive integer. If only one of the face and size is set then the other reverts to the default, not the previous value set. Alternatively, the single input can be a list of the form `{face, size}`, or `false` to revert to the default. The inputs must specify a font in a way that is accepted by Gnuplot but the details of font setting depend on the Gnuplot terminal in use. The defaults for the wxt terminal are face Sans and size 10. For the canvas terminal (and hence on Web REDUCE) setting the label font face is ignored.

Setlabelcolor command

```
SETLABELCOLOR colour
```

sets the label foreground colour; see Colours.

18.3.11 Turtle and Window Queries

Shownp query

```
SHOWNP
```

returns TRUE if the turtle is shown (visible), FALSE if the turtle is hidden. See SHOWTURTLE and HIDETURTLE.

Note that generally in LogoTurtle TRUE/FALSE values returned by query commands can be used to facilitate programming LogoTurtle by writing code such as the following:

```
if shownp = true then ...
```

Windowsize query

```
WINDOWSIZE
```

returns the current size of the graphics window as a list of the form $\{x_{max}, y_{max}\}$.

Turtlemode query

```
TURTLEMODE
```

returns the word WRAP, FENCE, WINDOW or FALSE depending on the current turtle mode.

Labelfont query

```
LABELFONT
```

returns a list of the current label font face and size if both are set, or whichever of the face or size is set, or false indicating that no label font information is set. Unset font information reverts to the Gnuplot default.

Labelcolor query

```
LABELCOLOR
```

returns the current label foreground colour; see Colours.

18.3.12 Pen and Background Control

The turtle carries a pen that can draw pictures. At any time the pen can be UP (in which case moving the turtle does not change what's on the graphics screen) or DOWN (in which case the turtle leaves a trace). Initially, the pen is UP.

Pendown command

```
PENDOWN
```

sets the pen's position to `DOWN`, i.e. lowers it so that the turtle draws when it moves.

Penup command

```
PENUP
```

sets the pen's position to `UP`, i.e. raises it so that the turtle moves without drawing.

Setpencolor command

```
SETPENCOLOR colour
```

sets the pen colour; see Colours.

Setpalette command

```
SETPALETTE(colournumber, colour)
```

sets the actual colour corresponding to a given colour number. The first argument must be an integer n such that $8 \leq n \leq 15$. (LogoTurtle keeps the first 8 colours constant.) The second argument may be either an RGB list of three nonnegative numbers not greater than 100 specifying the percent saturation of red, green, and blue in the desired colour, or an identifier or string representing a colour in any way

that is acceptable to Gnuplot, such as a colour name or hexadecimal number, e.g. `red`, `"red"` or `"#FF0000"`. See Colours for further details.

Setpensize command

```
SETPENSIZE size
```

sets the thickness of the pen. The input is a positive integer representing a multiple of the default thickness, or `FALSE`, meaning unspecified, which is equivalent to 1 but slightly more efficient.

Setbackground command

```
SETBACKGROUND colour
```

sets the screen background colour; see Colours. Currently, however, this command requires the `GNUTERM` environment variable to be set to the Gnuplot terminal type. This is because in Gnuplot the background is a property of the terminal, so the terminal type is required as part of the command to set the background. Unless you already specify the appropriate Gnuplot terminal type, you can find it by running Gnuplot interactively, when it will report something like

```
Terminal type set to 'wxt'
```

In this case, the correct value to assign to the `GNUTERM` environment variable would be `wxt` (without any quotes).

18.3.13 Pen and Background Queries

Pendownp query

```
PENDOWNP
```

returns the identifier `TRUE` if the pen is down, `FALSE` if it's up.

Pencolor query

```
PENCOLOR
```

returns the pen colour; see Colours.

Palette query

```
PALETTE colournumber
```

returns the colour associated with the given number; see Colours. Colournumber must be a nonnegative integer not greater than 15.

Pensize query

```
PENSIZE
```

returns a positive integer specifying the thickness of the pen as a multiple of the default thickness, or `false`, meaning unspecified, which is equivalent to 1 but slightly more efficient.

Background query

```
BACKGROUND
```

returns the graphics background colour; see Colours.

18.3.14 Saving and Loading Pictures

Savepict command

```
SAVEPICT identifier
```

saves the current plot and labels to internal storage under the specified identifier without changing them. The saved data can be restored as the current plot and labels using `LOADPICT`.

Loadpict command

```
LOADPICT(identifiers)
```

retrieves the plots and labels saved under the specified identifiers, which must have been saved by `SAVEPICT` commands, merges them in the order specified, and makes the result current. (The order is only significant in that it determines the colour of each plot if it is set automatically by Gnuplot, and if plots or labels overlap then later plots and labels overlay earlier ones and so hide them.) The previous current plot and labels are lost if not saved using `SAVEPICT`.

Chapter 19

Tracing in REDUCE

19.1 Introduction

The package `rtrace` provides portable tracing facilities for REDUCE programming.[1] These include

- entry-exit tracing of procedures,
- assignment tracing of procedures,
- tracing of rules when they fire.

In contrast to conventional Lisp-level tracing, values are printed in algebraic style whenever possible if the switch `rtrace` is on, which it is by default. The output has been specially tailored for the needs of algebraic-mode programming. Most features can be applied without explicitly modifying the target program, and they can be turned on and off dynamically at run time. If the switch `rtrace` is turned off then values are printed in conventional Lisp style, and the result should be similar to the tracing provided by the underlying Lisp system.

To make the facilities available, load the package using the command

```
load_package rtrace;
```

Alternatively, the package can be set up to auto load by putting appropriate code in your REDUCE initialisation file. An example is provided in the file `reduce.rc` in the `rtrace` source directory.

[1] This code was written by Herbert Melenk and Francis J. Wright.

19.2 RTrace versus RDebug

The `rtrace` package is a modification (by FJW) of the `rdebug` package (written by HM, and included in the `rtrace` source directory). The modifications are as follows. The procedure-tracing facilities in `rdebug` rely upon the low-level tracing facilities in PSL; in `rtrace` these low-level facilities have been (partly) re-implemented portably. The names of the tracing commands that have been re-implemented portably have been changed to avoid conflicting with those provided by the underlying Lisp system by preceding them with the letter "r", and they provide a generalized interface that supports algebraic mode better. An additional set of rule tracing facilities for inactive rules has been provided. Beware that the `rtrace` package is still experimental!

This package is intended to be portable, and has been tested with both CSL- and PSL-based REDUCE. However, it is intended not as a replacement for `rdebug` but as a partial re-implementation of `rdebug` that works with CSL-REDUCE, and it is assumed that PSL users will continue to use `rdebug`. It should, in principle, be possible to use both. Any `rtrace` functions with the same names as `rdebug` functions should either be identical or compatible; `rtrace` should be loaded after `rdebug` in order to retain any enhancements provided by `rtrace`. Perhaps at some future time the two packages should be merged. However, note that `rtrace` currently provides *only tracing* (hence the name) and does not support break points. (The current version also does not support conditional tracing.)

19.3 Procedure Tracing: RTR, UNRTR

Tracing of one or more procedures is initiated by the command `rtr`:

```
rtr <proc1>, <proc2>, ..., <procn>;
```

and cancelled by the command `unrtr`:

```
unrtr <proc1>, <proc2>, ..., <procn>;
```

Every time a traced procedure is executed, a message is printed when the procedure is entered or exited. The entry message displays the actual procedure arguments equated to the dummy parameter names, and the exit message displays the value returned by the procedure. Recursive calls are marked by a level number. Here is a (simplistic) example, using first the default algebraic display and second conventional Lisp display:

```
algebraic procedure power(x, n);
   if n = 0 then 1 else x*power(x, n-1)$
```

```
rtr power;

(power)

power(x+1, 2);

Enter (1) power
   x:  x + 1$
   n:  2$
Enter (2) power
   x:  x + 1$
   n:  1$
Enter (3) power
   x:  x + 1$
   n:  0$
Leave (3) power = 1$
Leave (2) power = x + 1$
Leave (1) power = x**2 + 2*x + 1$

 2
x  + 2*x + 1

off rtrace;

power(x+1, 2);

Enter (1) power
   x:  (plus x 1)
   n:  2
Enter (2) power
   x:  (plus x 1)
   n:  1
Enter (3) power
   x:  (plus x 1)
   n:  0
Leave (3) power = 1
Leave (2) power = (!*sq ((((x . 1) . 1) . 1) . 1)
t)
Leave (1) power = (!*sq ((((x . 2) . 1) ((x . 1) .
2) . 1) . 1) t)

 2
x  + 2*x + 1
```

```
on rtrace;

unrtr power;

(power)
```

Many algebraic-mode operators are implemented as internal procedures with different names. If an internal procedure with the specified name does not exist then `rtrace` tracing automatically applies to the appropriate internal procedure and returns a list of the names of the internal procedures, e.g.

```
rtr int;

(simpint)
```

This facility is an extension of the `rdebug` package.

Tracing of *compiled* procedures by the `rtrace` package is not completely reliable, in that recursive calls may not be traced. This is essentially because tracing works only when the procedure is called by name and not when it is called directly via an internal compiled pointer. It may not be possible to avoid this restriction in a portable way. Also, arguments of compiled procedures are not displayed using the names given to them in the source code, because these names are no longer available. Instead, they are displayed using the names `Arg1`, `Arg2`, etc.

19.4 Assignment Tracing: RTRST, UNRTRST

One often needs information about the internal behaviour of a procedure, especially if it is a longer piece of code. For an interpreted procedure declared in an `rtrst` command:

```
rtrst <proc1>, <proc2>, ..., <procn>;
```

all explicit assignments executed (as either the symbolic-mode `setq` or the algebraic-mode `setk`) inside these procedures are displayed during procedure execution. All procedure tracing (assignment and entry-exit) is removed by the command `unrtrst` (or `unrtr`, for which it is just a synonym):

```
unrtrst <proc1>, <proc2>, ..., <procn>;
```

Assignment tracing is not possible if a procedure is compiled, either because it was loaded from a "fasl" file or image, or because it was compiled as it was read

in as source code. This is because assignment tracing works by modifying the interpreted code of the procedure, which must therefore be available.

Applying `rtr` to a procedure that has been declared in an `rtrst` command, or vice versa, toggles the type of tracing applied (and displays an explanatory message).

Note that when a program contains a `for` loop, REDUCE translates this to a sequence of Lisp instructions. When using `rtrst`, the printout is driven by the "unfolded" code. When the code contains a `for each ... in` statement, the name of the control variable is internally used to keep the remainder of the list during the loop, and you will see the corresponding assignments in the trace rather than the individual values in the loop steps, e.g.

```
procedure fold u;
   for each x in u sum x$

rtrst fold;

(fold)

fold {z, z*y, y};
```

produces the following output (using CSL-REDUCE):

```
Enter (1) fold
   u:  {z,y*z,y}$
x := [z,y*z,y]$
G0 := 0$
G0 := z$
x := [y*z,y]$
G0 := z*(y + 1)$
x := [y]$
G0 := y*z + y + z$
x := []$
Leave (1) fold = y*z + y + z$

y*z + y + z

unrtrst fold;

(fold)
```

In this example, the printed assignments for `x` show the various stages of the loop. The variable `G0` is an internally generated place-holder for the sum, and may have

a slightly different name depending on the underlying Lisp systems.

19.5 Tracing Active Rules: TRRL, UNTRRL

The command `trrl` initiates tracing when they fire of individual rules or rule lists that have been activated using `let`.

```
trrl <rl1>, <rl2>, ..., <rln>;
```

where each of the $< rl_i >$ is:

- a rule or rule list;
- the name of a rule or rule list (that is, a non-indexed variable which is bound to a rule or rule list);
- an operator name, representing the rules assigned to this operator.

The specified rules are (re-) activated in REDUCE such that each of them prints a report every time it fires. The report is composed of the name of the rule or the name of the rule list together with the number of the rule in the list, the form matching the left side ("input") and the resulting right side ("output"). For an explicitly given rule or rule list, `trrl` assigns a unique generated name.

Note, however, that `trrl` does not trace rules with constant expressions on the left, on the assumption that they are not particularly interesting. [This behaviour may be made user-controllable in a future version.]

The command `untrrl` removes the tracing from rules:

```
untrrl <rl1>, <rl2>, ..., <rln>;
```

where each of the $< rl_i >$ is:

- a rule or rule list;
- the name of a rule or rule list (that is, a non-indexed variable which is bound to a rule or rule list or a unique name generated by `trrl`);
- an operator name, representing the rules assigned to this operator.

The rules are reactivated in their original form. Alternatively you can use the command `clearrules` to remove the rules totally from the system. Please do not modify the rules between `trrl` and `untrrl` – the result may be unpredictable.

Here are two simple examples that show tracing via the rule name and via the operator name:

```
trigrules := {sin(~x)^2 => 1 - cos(x)^2};

                      2              2
trigrules := {sin(~x)  => 1 - cos(x)  }

let trigrules;
trrl trigrules;

1 - sin(x)^2;

Rule trigrules.1: sin(x)**2 => 1 - cos(x)**2$

      2
cos(x)

untrrl trigrules;
trrl sin;

1 - sin(x)^2;

Rule sin.23: sin(x)**2 => 1 - cos(x)**2$

      2
cos(x)

untrrl sin;
clearrules trigrules;
```

19.6 Tracing Inactive Rules: TRRLID, UNTRRLID

The command `trrlid` initiates tracing of individual rule lists that have been assigned to variables, but have not been activated using `let`:

```
trrlid <rlid1>, <rlid2>, ..., <rlidn>;
```

where each of the $< rlid_i >$ is an identifier of a rule list (that is, a non-indexed variable which is bound to a rule list). It is assumed that they will be activated later, either via a `let` command or by using the `where` operator. When they are activated and fire, tracing output will be as if they had been traced using `trrl`. The command `untrrlid` clears the tracing. This facility is an extension of the `rdebug` package.

Here is a simple example that continues the example above:

```
trrlid trigrules;

1 - sin(x)^2 where trigrules;

Rule trigrules.1: sin(x)**2 => 1 - cos(x)**2$

      2
cos(x)

untrrlid trigrules;
```

19.7 Output Control: RTROUT

The trace output (only) can be redirected to a separate file by using the command `rtrout`, followed by a file name in string quotes. A second call of `rtrout` closes any current output file and opens a new one. The file name `NIL` (without string quotes) closes any current output file and causes the trace output to be redirected to the standard output device.

The `rdebug` variables `trlimit` and `trprinter!*` are not implemented in `rtrace`. If you want to select Lisp-style tracing then turn off the switch `rtrace`:

```
off rtrace;
```

Chapter 20

User Contributed Packages

The complete REDUCE system includes a number of packages contributed by users that are provided as a service to the user community. Questions regarding these packages should be directed to their individual authors.

All such packages have been precompiled as part of the installation process. However, many must be specifically loaded before they can be used. (Those that are loaded automatically are so noted in their description.) You should also consult the user notes for your particular implementation for further information on whether this is necessary. If it is, the relevant command is `load_package`, which takes a list of one or more package names as argument, for example:

```
load_package algint;
```

although this syntax may vary from implementation to implementation.

Nearly all these packages come with separate documentation and test files (except those noted here that have no additional documentation), which is included, along with the source of the package, in the REDUCE system distribution. These items should be studied for any additional details on the use of a particular package.

The packages available in the current release of REDUCE are as follows:

20.1 APPLYSYM: Infinitesimal Symmetries of Differential Equations

This package provides programs APPLYSYM, QUASILINPDE and DETRAFO for applying infinitesimal symmetries of differential equations, the generalization of special solutions and the calculation of symmetry and similarity variables. They use the package CRACK.

Author: Thomas Wolf

In this paper the programs APPLYSYM, QUASILINPDE and DETRAFO are described which aim at the utilization of infinitesimal symmetries of differential equations. The purpose of QUASILINPDE is the general solution of quasilinear PDEs. This procedure is used by APPLYSYM for the application of point symmetries for either

- calculating similarity variables to perform a point transformation which lowers the order of an ODE or effectively reduces the number of explicitly occuring independent variables in a PDE(-system) or for
- generalizing given special solutions of ODEs / PDEs with new constant parameters.

The program DETRAFO performs arbitrary point- and contact transformations of ODEs / PDEs and is applied if similarity and symmetry variables have been found. The program APPLYSYM is used in connection with the program LIEPDE for formulating and solving the conditions for point- and contact symmetries. The procedure LIEPDE is also described below (see also [Wol93]). The actual problem solving is done in all these programs through a call to the package CRACK for solving overdetermined PDE-systems.

Before using the procedures LIEPDE, APPLYSYM, QUASILINPDE and DETRAFO, it is necessary to load the packages:

```
load_package liepde, applysym;
```

20.1.1 Introduction and overview of the symmetry method

The investigation of infinitesimal symmetries of differential equations (DEs) with computer algebra programs attrackted considerable attention over the last years. Corresponding programs are available in all major computer algebra systems. In a review article by W. Hereman [Her95] about 200 references are given, many of them describing related software.

One reason for the popularity of the symmetry method is the fact that Sophus Lie's method [Lie75, Lie67] is the most widely used method for computing exact solu-

tions of non-linear DEs. Another reason is that the first step in this method, the formulation of the determining equation for the generators of the symmetries, can already be very cumbersome, especially in the case of PDEs of higher order and/or in case of many dependent and independent variables. Also, the formulation of the conditions is a straight forward task involving only differentiations and basic algebra - an ideal task for computer algebra systems. Less straight forward is the automatic solution of the symmetry conditions which is the strength of the program `LIEPDE` (for a comparison with another program see [Wol93]).

The novelty described in this paper are programs aiming at the final third step: Applying symmetries for

- calculating similarity variables to perform a point transformation which lowers the order of an ODE or effectively reduces the number of explicitly occuring independent variables of a PDE(-system) or for
- generalizing given special solutions of ODEs/PDEs with new constant parameters.

Programs which run on their own but also allow interactive user control are indispensible for these calculations. On one hand the calculations can become quite lengthy, like variable transformations of PDEs (of higher order, with many variables). On the other hand the freedom of choosing the right linear combination of symmetries and choosing the optimal new symmetry- and similarity variables makes it necessary to 'play' with the problem interactively.

The focus in this paper is directed on questions of implementation and efficiency, no principally new mathematics is presented.

In the following subsections a review of the first two steps of the symmetry method is given as well as the third, i.e. the application step is outlined. Each of the remaining sections is devoted to one procedure.

The first step: Formulating the symmetry conditions

To obey classical Lie-symmetries, differential equations

$$H_A = 0 \tag{20.1}$$

for unknown functions $y^\alpha, \quad 1 \leq \alpha \leq p$ of independent variables $x^i, \quad 1 \leq i \leq q$ must be forminvariant against infinitesimal transformations

$$\tilde{x}^i = x^i + \varepsilon\xi^i, \qquad \tilde{y}^\alpha = y^\alpha + \varepsilon\eta^\alpha \tag{20.2}$$

in first order of ε. To transform the equations (20.1) by (20.2), derivatives of y^α must be transformed, i.e. the part linear in ε must be determined. The correspond-

ing formulas are (see e.g. [Olv86, Ste89])

$$\tilde{y}^{\alpha}_{j_1\dots j_k} = y^{\alpha}_{j_1\dots j_k} + \varepsilon\eta^{\alpha}_{j_1\dots j_k} + O(\varepsilon^2)$$

$$\eta^{\alpha}_{j_1\dots j_{k-1}j_k} = \frac{D\eta^{\alpha}_{j_1\dots j_{k-1}}}{Dx^k} - y^{\alpha}_{ij_1\dots j_{k-1}}\frac{D\xi^i}{Dx^k} \tag{20.3}$$

where D/Dx^k means total differentiation w.r.t. x^k and from now on lower latin indices of functions y^α, (and later u^α) denote partial differentiation w.r.t. the independent variables x^i, (and later v^i). The complete symmetry condition then takes the form

$$XH_A = 0 \bmod H_A = 0 \tag{20.4}$$

$$X = \xi^i\frac{\partial}{\partial x^i} + \eta^\alpha\frac{\partial}{\partial y^\alpha} + \eta^\alpha_m\frac{\partial}{\partial y^\alpha_m} + \eta^\alpha_{mn}\frac{\partial}{\partial y^\alpha_{mn}} + \ldots + \eta^\alpha_{mn\dots p}\frac{\partial}{\partial y^\alpha_{mn\dots p}}. \tag{20.5}$$

where mod $H_A = 0$ means that the original PDE-system is used to replace some partial derivatives of y^α to reduce the number of independent variables, because the symmetry condition (20.4) must be fulfilled identically in x^i, y^α and all partial derivatives of y^α.

For point symmetries, ξ^i, η^α are functions of x^j, y^β and for contact symmetries they depend on x^j, y^β and y^β_k. We restrict ourself to point symmetries as those are the only ones that can be applied by the current version of the program `APPLYSYM` (see below). For literature about generalized symmetries see [Her95].

Though the formulation of the symmetry conditions (20.4), (20.5), (20.3) is straightforward and handled in principle by all related programs [Her95], the computational effort to formulate the conditions (20.4) may cause problems if the number of x^i and y^α is high. This can partially be avoided if at first only a few conditions are formulated and solved such that the remaining ones are much shorter and quicker to formulate.

A first step in this direction is to investigate one PDE $H_A = 0$ after another, as done in [CHW91]. Two methods to partition the conditions for a single PDE are described by Bocharov/Bronstein [BB89] and Stephani [Ste89].

In the first method only those terms of the symmetry condition $XH_A = 0$ are calculated which contain at least a derivative of y^α of a minimal order m. Setting coefficients of these u-derivatives to zero provides symmetry conditions. Lowering the minimal order m successively then gradually provides all symmetry conditions.

The second method is even more selective. If H_A is of order n then only terms of the symmetry condition $XH_A = 0$ are generated which contain n'th order derivatives of y^α. Furthermore these derivatives must not occur in H_A itself. They can

therefore occur in the symmetry condition (20.4) only in $\eta^\alpha_{j_1\ldots j_n}$, i.e. in the terms

$$\eta^\alpha_{j_1\ldots j_n}\frac{\partial H_A}{\partial y^\alpha_{j_1\ldots j_n}}.$$

If only coefficients of n'th order derivatives of y^α need to be accurate to formulate preliminary conditions then from the total derivatives to be taken in (20.3) only that part is performed which differentiates w.r.t. the highest y^α-derivatives. This means, for example, to form only $y^\alpha_{mnk}\partial/\partial y^\alpha_{mn}$ if the expression, which is to be differentiated totally w.r.t. x^k, contains at most second order derivatives of y^α.

The second method is applied in `LIEPDE`. Already the formulation of the remaining conditions is speeded up considerably through this iteration process. These methods can be applied if systems of DEs or single PDEs of at least second order are investigated concerning symmetries.

The second step: Solving the symmetry conditions

The second step in applying the whole method consists in solving the determining conditions (20.4), (20.5), (20.3) which are linear homogeneous PDEs for ξ^i, η^α. The complete solution of this system is not algorithmic any more because the solution of a general linear PDE-system is as difficult as the solution of its non-linear characteristic ODE-system which is not covered by algorithms so far.

Still algorithms are used successfully to simplify the PDE-system by calculating its standard normal form and by integrating exact PDEs if they turn up in this simplification process [Wol93]. One problem in this respect, for example, concerns the optimization of the symbiosis of both algorithms. By that we mean the ranking of priorities between integrating, adding integrability conditions and doing simplifications by substitutions - all depending on the length of expressions and the overall structure of the PDE-system. Also the extension of the class of PDEs which can be integrated exactly is a problem to be pursuit further.

The program `LIEPDE` which formulates the symmetry conditions calls the program `CRACK` to solve them. This is done in a number of successive calls in order to formulate and solve some first order PDEs of the overdetermined system first and use their solution to formulate and solve the next subset of conditions as described in the previous subsection. Also, `LIEPDE` can work on DEs that contain parametric constants and parametric functions. An ansatz for the symmetry generators can be formulated. For more details see [Wol93] or [BW92].

The procedure `LIEPDE` is called through
`LIEPDE(`*`problem,symtype,flist,inequ`*`);`
All parameters are lists.

The first parameter specifies the DEs to be investigated:
problem has the form {*equations, ulist, xlist*} where

equations is a list of equations, each has the form `df(ui,..)=...` where the LHS (left hand side) `df(ui,..)` is selected such that

- The RHS (right h.s.) of an equations must not include the derivative on the LHS nor a derivative of it.
- Neither the LHS nor any derivative of it of any equation may occur in any other equation.
- Each of the unknown functions occurs on the LHS of exactly one equation.

ulist is a list of function names, which can be chosen freely.

xlist is a list of variable names, which can be chosen freely.

Equations can be given as a list of single differential expressions and then the program will try to bring them into the 'solved form' `df(ui,..)=...` automatically. If equations are given in the solved form then the above conditions are checked and execution is stopped it they are not satisfied. An easy way to get the equations in the desired form is to use

`FIRST SOLVE({`*eq1,eq2,...*`},{`*one highest derivative for each function u*`})`

(see the example of the Karpman equations in `LIEPDE.TST`). The example of the Burgers equation in `LIEPDE.TST` demonstrates that the number of symmetries for a given maximal order of the infinitesimal generators depends on the derivative chosen for the LHS.

The second parameter *symtype* of `LIEPDE` is a list { } that specifies the symmetry to be calculated. *symtype* can have the following values and meanings:

`{"point"}` Point symmetries with $\xi^i = \xi^i(x^j, u^\beta),\ \eta^\alpha = \eta^\alpha(x^j, u^\beta)$ are determined.

`{"contact"}` Contact symmetries with $\xi^i = 0,\ \eta = \eta(x^j, u, u_k)$ are determined ($u_k = \partial u/\partial x^k$), which is only applicable if a single equation (20.1) with an order > 1 for a single function u is to be investigated. (The *symtype* `{"contact"}` is equivalent to `{"general", 1}` (see below) apart from the additional checks done for `{"contact"}`.)

`{"general",`*order*`}` where *order* is an integer > 0. Generalized symmetries $\xi^i = 0, \eta^\alpha = \eta^\alpha(x^j, u^\beta, \ldots, u^\beta_K)$ of a specified order are determined (where $_K$ is a multiple index representing *order* many indices.)
NOTE: Characteristic functions of generalized symmetries ($= \eta^\alpha$ if $\xi^i = 0$) are equivalent if they are equal on the solution manifold. Therefore, all dependences of characteristic functions on the substituted derivatives and their derivatives are dropped. For example, if the heat equation is given as

$u_t = u_{xx}$ (i.e. u_t is substituted by u_{xx}) then `{"general", 2}` would not include characteristic functions depending on u_{tx} or u_{xxx}.
THEREFORE:
If you want to find *all* symmetries up to a given order then either

- avoid using $H_A = 0$ to substitute lower order derivatives by expressions involving higher derivatives, or
- increase the order specified in *symtype*.

For an illustration of this effect see the two symmetry determinations of the Burgers equation in the file `LIEPDE.TST`.

`{xi!_`*`x1`*`=...,..., eta!_`*`u1`*`=...,...}` It is possible to specify an ansatz for the symmetry. Such an ansatz must specify all ξ^i for all independent variables and all η^α for all dependent variables in terms of differential expressions which may involve unknown functions/constants. The dependences of the unknown functions have to be declared in advance by using the `DEPEND` command. For example,

```
DEPEND f, t, x, u$
```

specifies f to be a function of t, x, u. If one wants to have f as a function of derivatives of $u(t, x)$, say f depending on u_{txx}, then one <u>*cannot*</u> write

```
DEPEND f, df(u,t,x,2)$
```

but instead must write

```
DEPEND f, u!`1!`2!`2$
```

assuming *xlist* has been specified as `{t,x}`. Because t is the first variable and x is the second variable in *xlist* and u is differentiated oncs wrt. t and twice wrt. x we therefore use `u!`1!`2!`2`. The character `!` is the escape character to allow special characters like ‘ to occur in an identifier.

For generalized symmetries one usually sets all $\xi^i = 0$. Then the η^α are equal to the characteristic functions.

The third parameter *flist* of `LIEPDE` is a list { } that includes

- all parameters and functions in the equations which are to be determined such that symmetries exist (if any such parameters/functions are specified in *flist* then the symmetry conditions formulated in `LIEPDE` become non-linear conditions which may be much harder for `CRACK` to solve with many cases and subcases to be considered.)
- all unknown functions and constants in the ansatz `xi!_..` and `eta!_..` if that has been specified in *symtype*.

The fourth parameter *inequ* of `LIEPDE` is a list { } that includes all non-vanishing expressions which represent inequalities for the functions in flist.

The result of `LIEPDE` is a list with 3 elements, each of which is a list:

$$\{\{con_1, con_2, \ldots\}, \{\texttt{xi\_...} = \ldots, \ldots, \texttt{eta\_...} = \ldots, \ldots\}, \{flist\}\}.$$

The first list contains remaining unsolved symmetry conditions con_i. It is the empty list {} if all conditions have been solved. The second list gives the symmetry generators, i.e. expressions for ξ_i and η_j. The last list contains all free constants and functions occuring in the first and second list.

The third step: Application of infinitesimal symmetries

If infinitesimal symmetries have been found then the program `APPLYSYM` can use them for the following purposes:

1. Calculation of one symmetry variable and further similarity variables. After transforming the DE(-system) to these variables, the symmetry variable will not occur explicitly any more. For ODEs this has the consequence that their order has effectively been reduced.
2. Generalization of a special solution by one or more constants of integration.

Both methods are described in the following section.

20.1.2 Applying symmetries with `APPLYSYM`

The first mode: Calculation of similarity and symmetry variables

In the following we assume that a symmetry generator X, given in (20.5), is known such that ODE(s)/PDE(s) $H_A = 0$ satisfy the symmetry condition (20.4). The aim is to find new dependent functions $u^\alpha = u^\alpha(x^j, y^\beta)$ and new independent variables $v^i = v^i(x^j, y^\beta), \quad 1 \le \alpha, \beta \le p, \ 1 \le i, j \le q$ such that the symmetry generator $X = \xi^i(x^j, y^\beta)\partial_{x^i} + \eta^\alpha(x^j, y^\beta)\partial_{y^\alpha}$ transforms to

$$X = \partial_{v^1}. \tag{20.6}$$

Inverting the above transformation to $x^i = x^i(v^j, u^\beta), y^\alpha = y^\alpha(v^j, u^\beta)$ and setting $H_A(x^i(v^j, u^\beta), y^\alpha(v^j, u^\beta), \ldots) = h_A(v^j, u^\beta, \ldots)$ this means that

$$\begin{aligned} 0 &= XH_A(x^i, y^\alpha, y^\beta_j, \ldots) \mod H_A = 0 \\ &= Xh_A(v^i, u^\alpha, u^\beta_j, \ldots) \mod h_A = 0 \\ &= \partial_{v^1} h_A(v^i, u^\alpha, u^\beta_j, \ldots) \mod h_A = 0. \end{aligned}$$

Consequently, the variable v^1 does not occur explicitly in h_A. In the case of an ODE(-system) ($v^1 = v$) the new equations $0 = h_A(v, u^\alpha, du^\beta/dv, \ldots)$ are

then of lower total order after the transformation $z = z(u^1) = du^1/dv$ with now $z, u^2, \ldots u^p$ as unknown functions and u^1 as independent variable.

The new form (20.6) of X leads directly to conditions for the symmetry variable v^1 and the similarity variables $v^i|_{i\neq 1}, u^\alpha$ (all functions of x^k, y^γ):

$$Xv^1 = 1 = \xi^i(x^k, y^\gamma)\partial_{x^i}v^1 + \eta^\alpha(x^k, y^\gamma)\partial_{y^\alpha}v^1 \tag{20.7}$$

$$Xv^j|_{j\neq 1} = Xu^\beta = 0 = \xi^i(x^k, y^\gamma)\partial_{x^i}u^\beta + \eta^\alpha(x^k, y^\gamma)\partial_{y^\alpha}u^\beta \tag{20.8}$$

The general solutions of (20.7), (20.8) involve free functions of $p+q-1$ arguments. From the general solution of equation (20.8), $p + q - 1$ functionally independent special solutions have to be selected ($v^2, \ldots, v^p$ and $u^1, \ldots, u^q$), whereas from (20.7) only one solution v^1 is needed. Together, the expressions for the symmetry and similarity variables must define a non-singular transformation $x, y \to u, v$.

Different special solutions selected at this stage will result in different resulting DEs which are equivalent under point transformations but may look quite differently. A transformation that is more difficult than another one will in general only complicate the new DE(s) compared with the simpler transformation. We therefore seek the simplest possible special solutions of (20.7), (20.8). They also have to be simple because the transformation has to be inverted to solve for the old variables in order to do the transformations.

The following steps are performed in the corresponding mode of the program `APPLYSYM`:

- The user is asked to specify a symmetry by selecting one symmetry from all the known symmetries or by specifying a linear combination of them.
- Through a call of the procedure `QUASILINPDE` (described in a later section) the two linear first order PDEs (20.7), (20.8) are investigated and, if possible, solved.
- From the general solution of (20.7) 1 special solution is selected and from (20.8) $p + q - 1$ special solutions are selected which should be as simple as possible.
- The user is asked whether the symmetry variable should be one of the independent variables (as it has been assumed so far) or one of the new functions (then only derivatives of this function and not the function itself turn up in the new DE(s)).
- Through a call of the procedure `DETRAFO` the transformation $x^i, y^\alpha \to v^j, u^\beta$ of the DE(s) $H_A = 0$ is finally done.
- The program returns to the starting menu.

The second mode: Generalization of special solutions

A second application of infinitesimal symmetries is the generalization of a known special solution given in implicit form through $0 = F(x^i, y^\alpha)$. If one knows a symmetry variable v^1 and similarity variables $v^r, u^\alpha, \quad 2 \le r \le p$ then v^1 can be shifted by a constant c because of $\partial_{v^1} H_A = 0$ and therefore the DEs $0 = H_A(v^r, u^\alpha, u^\beta_j, \ldots)$ are unaffected by the shift. Hence from

$$0 = F(x^i, y^\alpha) = F(x^i(v^j, u^\beta), y^\alpha(v^j, u^\beta)) = \bar{F}(v^j, u^\beta)$$

follows that

$$0 = \bar{F}(v^1 + c, v^r, u^\beta) = \bar{F}(v^1(x^i, y^\alpha) + c, v^r(x^i, y^\alpha), u^\beta(x^i, y^\alpha))$$

defines implicitly a generalized solution $y^\alpha = y^\alpha(x^i, c)$.

This generalization works only if $\partial_{v^1} \bar{F} \neq 0$ and if $\bar{F}$ does not already have a constant additive to v^1.

The method above needs to know $x^i = x^i(u^\beta, v^j), \; y^\alpha = y^\alpha(u^\beta, v^j)$ and $u^\alpha = u^\alpha(x^j, y^\beta), v^\alpha = v^\alpha(x^j, y^\beta)$ which may be practically impossible. Better is, to integrate x^i, y^α along X:

$$\frac{d\bar{x}^i}{d\varepsilon} = \xi^i(\bar{x}^j(\varepsilon), \bar{y}^\beta(\varepsilon)), \qquad \frac{d\bar{y}^\alpha}{d\varepsilon} = \eta^\alpha(\bar{x}^j(\varepsilon), \bar{y}^\beta(\varepsilon)) \tag{20.9}$$

with initial values $\bar{x}^i = x^i, \bar{y}^\alpha = y^\alpha$ for $\varepsilon = 0$. (This ODE-system is the characteristic system of (20.8).)

Knowing only the finite transformations

$$\bar{x}^i = \bar{x}^i(x^j, y^\beta, \varepsilon), \quad \bar{y}^\alpha = \bar{y}^\alpha(x^j, y^\beta, \varepsilon) \tag{20.10}$$

gives immediately the inverse transformation $\bar{x}^i = \bar{x}^i(x^j, y^\beta, \varepsilon), \quad \bar{y}^\alpha = \bar{y}^\alpha(x^j, y^\beta, \varepsilon)$ just by $\varepsilon \to -\varepsilon$ and renaming $x^i, y^\alpha \leftrightarrow \bar{x}^i, \bar{y}^\alpha$.

The special solution $0 = F(x^i, y^\alpha)$ is generalized by the new constant ε through

$$0 = F(x^i, y^\alpha) = F(x^i(\bar{x}^j, \bar{y}^\beta, \varepsilon), y^\alpha(\bar{x}^j, \bar{y}^\beta, \varepsilon))$$

after dropping the $^-$.

The steps performed in the corresponding mode of the program `APPLYSYM` show features of both techniques:

- The user is asked to specify a symmetry by selecting one symmetry from all the known symmetries or by specifying a linear combination of them.
- The special solution to be generalized and the name of the new constant have to be put in.

- Through a call of the procedure QUASILINPDE, the PDE (20.7) is solved which amounts to a solution of its characteristic ODE system (20.9) where $v^1 = \varepsilon$.
- QUASILINPDE returns a list of constant expressions

$$c_i = c_i(x^k, y^\beta, \varepsilon), \quad 1 \le i \le p + q \tag{20.11}$$

  which are solved for $x^j = x^j(c_i, \varepsilon), \quad y^\alpha = y^\alpha(c_i, \varepsilon)$ to obtain the generalized solution through

$$0 = F(x^j, y^\alpha) = F(x^j(c_i(x^k, y^\beta, 0), \varepsilon), y^\alpha(c_i(x^k, y^\beta, 0), \varepsilon)).$$

- The new solution is availabe for further generalizations w.r.t. other symmetries.

If one would like to generalize a given special solution with m new constants because m symmetries are known, then one could run the whole program m times, each time with a different symmetry or one could run the program once with a linear combination of m symmetry generators which again is a symmetry generator. Running the program once adds one constant but we have in addition $m - 1$ arbitrary constants in the linear combination of the symmetries, so m new constants are added. Usually one will generalize the solution gradually to make solving (20.9) gradually more difficult.

Syntax

The call of APPLYSYM is APPLYSYM({*de*, *fun*, *var*}, {*sym*, *cons*});

- *de* is a single DE or a list of DEs in the form of a vanishing expression or in the form $\ldots = \ldots$.
- *fun* is the single function or the list of functions occuring in *de*.
- *var* is the single variable or the list of variables in *de*.
- *sym* is a linear combination of all symmetries, each with a different constant coefficient, in form of a list of the ξ^i and η^α: {xi_...=...,...,eta_...=...,...}, where the indices after 'xi_' are the variable names and after 'eta_' the function names.
- *cons* is the list of constants in *sym*, one constant for each symmetry.

The list that is the first argument of APPLYSYM is the same as the first argument of LIEPDE and the second argument is the list that LIEPDE returns without its first element (the unsolved conditions). An example is given below.

What APPLYSYM returns depends on the last performed modus. After modus 1 the return is
{{*newde*, *newfun*, *newvar*}, *trafo*}
where

- *newde* lists the transformed equation(s)
- *newfun* lists the new function name(s)
- *newvar* lists the new variable name(s)
- *trafo* lists the transformations $x^i = x^i(v^j, u^\beta), y^\alpha = y^\alpha(v^j, u^\beta)$

After modus 2, APPLYSYM returns the generalized special solution.

Example: A second order ODE

Weyl's class of solutions of Einsteins field equations consists of axialsymmetric time independent metrics of the form

$$\mathrm{d}s^2 = e^{-2U}\left[e^{2k}\left(\mathrm{d}\rho^2 + \mathrm{d}z^2\right) + \rho^2\mathrm{d}\varphi^2\right] - e^{2U}\mathrm{d}t^2, \tag{20.12}$$

where U and k are functions of ρ and z. If one is interested in generalizing these solutions to have a time dependence then the resulting DEs can be transformed such that one longer third order ODE for U results which contains only ρ derivatives [Kub]. Because U appears not alone but only as derivative, a substitution

$$g = dU/d\rho \tag{20.13}$$

lowers the order and the introduction of a function

$$h = \rho g - 1 \tag{20.14}$$

simplifies the ODE to

$$0 = 3\rho^2 h\, h'' - 5\rho^2\, h'^2 + 5\rho\, h\, h' - 20\rho\, h^3 h' - 20\, h^4 + 16\, h^6 + 4\, h^2. \tag{20.15}$$

where $' = d/d\rho$. Calling LIEPDE through

```
depend h,r;
prob:={{-20*h**4+16*h**6+3*r**2*h*df(h,r,2)+5*r*h*df(h,r)
        -20*h**3*r*df(h,r)+4*h**2-5*r**2*df(h,r)**2},
       {h}, {r}};
sym:=liepde(prob, {"point"},{},{});
```

gives

```
sym := {{},

                        3                          2
        {xi_r= - c10*r  - c11*r, eta_h=c10*h*r },

        {c10,c11}}.
```

All conditions have been solved because the first element of `sym` is $\{\}$. The two existing symmetries are therefore

$$-\rho^3\partial_\rho + h\rho^2\,\partial_h \qquad \text{and} \qquad \rho\partial_\rho. \tag{20.16}$$

Corresponding finite transformations can be calculated with `APPLYSYM` through

```
newde:=applysym(prob,rest first sym);
```

The interactive session is given below with the user input following the prompt '?'. (Empty lines have been deleted.)

```
Do you want to find similarity and symmetry variables (1)
or generalize a special solution with new parameters  (2)
or exit the program                                   (3) ?1
```

We enter '1' because we want to reduce dependencies by finding similarity variables and one symmetry variable and then doing the transformation such that the symmetry variable does not explicitly occur in the DE.

```
----------------------   The 1.  symmetry is:
         3
xi_r= - r
         2
eta_h=h*r
----------------------   The 2.  symmetry is:
xi_r= - r
----------------------
Which single symmetry or linear combination of symmetries
do you want to apply?
Enter an expression with 'sy_(i)' for the i'th
symmetry. Terminate input with '$' or ';'.
sy_(1);
```

We could have entered 'sy_(2);' or a combination of both as well with the calculation running then differently.

```
The symmetry to be applied in the following is
            3            2
{xi_r =  - r ,eta_h = h*r }
Terminate the following input with $ or ;  .
Enter the name of the new dependent variable
```

```
(which will get an index attached): u;
Enter the name of the new independent variables:
(which will get an index attached): v;
```

This was the input part, now the real calculation starts.

```
The ODE/PDE (-system) under investigation is :
                    2            2  2               3
0 = 3*df(h,r,2)*h*r  - 5*df(h,r) *r  - 20*df(h,r)*h *r
                              6       4      2
     + 5*df(h,r)*h*r + 16*h  - 20*h  + 4*h
for the function(s) : h.
It will be looked for a new dependent variable u
and an independent variable v such that the transformed
de(-system) does not depend on u or v.
1. Determination of the similarity variable
                             2
The quasilinear PDE:  0 = r *(df(u_,h)*h - df(u_,r)*r).
The equivalent characteristic system:
                 3
0= - df(u_,r)*r
       2
0= - r *(df(h,r)*r + h)
for the functions: h(r)  u_(r).
```

The PDE is equation (20.8).

```
The general solution of the PDE is given through
0 = ff(u_,h*r)
with arbitrary function ff(..).
A suggestion for this function ff provides:
0 =  - h*r + u_
Do you like this choice? (Y or N) y
```

For the following calculation only a single special solution of the PDE is necessary and this has to be specified from the general solution by choosing a special function `ff`. (This function is called `ff` to prevent a clash with names of user variables/functions.) In principle any choice of `ff` would work, if it defines a non-singular coordinate transformation, i.e. here r must be a function of $u\_$. If we have q independent variables and p functions of them then `ff` has $p + q$ arguments. Because of the condition $0 =$`ff` one has essentially the freedom of choosing a function of $p + q - 1$ arguments freely. This freedom is also necessary to select $p + q - 1$ different functions `ff` and to find as many functionally independent solutions $u\_$ which all become the new similarity variables. q of them become the new functions u^α and $p - 1$ of them the new variables $v^2, \ldots, v^p$. Here we have $p = q = 1$ (one single ODE).

Though the program could have done that alone, once the general solution `ff(..)`

is known, the user can interfere here to enter a simpler solution, if possible.

```
2. Determination of the symmetry variable
                                          2             3
The quasilinear PDE:  0 = df(u_,h)*h*r  - df(u_,r)*r  - 1.
The equivalent characteristic system:
              3
0=df(r,u_) + r
                2
0=df(h,u_) - h*r
for the functions: r(u_)  h(u_)  .
New attempt with a different independent variable
The equivalent characteristic system:
              2
0=df(u_,h)*h*r  - 1
   2
0=r *(df(r,h)*h + r)
for the functions: r(h)  u_(h)  .
The general solution of the PDE is given through
                  2  2       2
             - 2*h *r *u_ + h
0 = ff(h*r,--------------------)
                    2
with arbitrary function ff(..).
A suggestion for this function ff(..) yields:
      2        2
     h *( - 2*r *u_ + 1)
0 = ---------------------
              2
Do you like this choice? (Y or N) y
```

Similar to above.

```
The suggested solution of the algebraic system which will
do the transformation is:
                         sqrt(v)*sqrt(2)
{h=sqrt(v)*sqrt(2)*u,r=-----------------}
                               2*v
Is the solution ok? (Y or N) y
In the intended transformation shown above the dependent
variable is u and the independent variable is v.
The symmetry variable is v, i.e. the transformed expression
will be free of v.
Is this selection of dependent and independent variables ok?
                                                    (Y or N) n
```

We so far assumed that the symmetry variable is one of the new variables, but, of course we also could choose it to be one of the new functions. If it is one of the functions then only derivatives of this function occur in the new DE, not

the function itself. If it is one of the variables then this variable will not occur explicitly.

In our case we prefer (without strong reason) to have the function as symmetry variable. We therefore answered with 'no'. As a consequence, u and v will exchange names such that still all new functions have the name u and the new variables have name v:

```
Please enter a list of substitutions. For example, to
make the variable, which is so far call u1, to an
independent variable v2 and the variable, which is
so far called v2, to an dependent variable u1,
enter: `{u1=v2, v2=u1};'{u=v,v=u};

The transformed equation which should be free of u:
                  3  6        2  3
0=3*u  *v - 16*u  *v  - 20*u  *v  + 5*u
     2v         v           v          v
Do you want to find similarity and symmetry variables (1)
or generalize a special solution with new parameters  (2)
or exit the program                                   (3) :
```

We stop here. The following is returned from our `APPLYSYM` call:

```
                                  3  6
{{{3*df(u,v,2)*v - 16*df(u,v) *v

                 2  3
    - 20*df(u,v) *v  + 5*df(u,v)},

  {u},

  {v}},

            1                   2*u*v
 {r=-----------------,h=-----------------
     sqrt(u)*sqrt(2)     sqrt(u)*sqrt(2)

           }}
```

The use of `APPLYSYM` effectively provided us the finite transformation

$$\rho = (2\,u)^{-1/2}, \qquad h = (2\,u)^{1/2}\,v. \tag{20.17}$$

and the new ODE

$$0 = 3u''v - 16u'^3v^6 - 20u'^2v^3 + 5u' \tag{20.18}$$

where $u = u(v)$ and $' = d/dv$. Using one symmetry we reduced the 2. order ODE (20.15) to a first order ODE (20.18) for u' plus one integration. The second symme-

try can be used to reduce the remaining ODE to an integration too by introducing a variable w through $v^3 d/dv = d/dw$, i.e. $w = -1/(2v^2)$. With

$$p = du/dw \tag{20.19}$$

the remaining ODE is

$$0 = 3\,w\,\frac{dp}{dw} + 2\,p\,(p+1)(4\,p+1)$$

with solution

$$\tilde{c}w^{-2}/4 = \tilde{c}v^4 = \frac{p^3(p+1)}{(4\,p+1)^4}, \quad \tilde{c} = const.$$

Writing (20.19) as $p = v^3(du/dp)/(dv/dp)$ we get u by integration and with (20.17) further a parametric solution for ρ, h:

$$\rho = \left(\frac{3c_1^2(2p-1)}{p^{1/2}(p+1)^{1/2}} + c_2\right)^{-1/2} \tag{20.20}$$

$$h = \frac{(c_2 p^{1/2}(p+1)^{1/2} + 6c_1^2 p - 3c_1^2)^{1/2} p^{1/2}}{c_1(4p+1)} \tag{20.21}$$

where $c_1, c_2 = const.$ and $c_1 = \tilde{c}^{1/4}$. Finally, the metric function $U(p)$ is obtained as an integral from (20.13),(20.14).

Limitations of APPLYSYM

Restrictions of the applicability of the program APPLYSYM result from limitations of the program QUASILINPDE described in a section below. Essentially this means that symmetry generators may only be polynomially non-linear in x^i, y^α. Though even then the solvability can not be guaranteed, the generators of Lie-symmetries are mostly very simple such that the resulting PDE (20.22) and the corresponding characteristic ODE-system have good chances to be solvable.

Apart from these limitations implied through the solution of differential equations with CRACK and algebraic equations with SOLVE the program APPLYSYM itself is free of restrictions, i.e. if once new versions of CRACK, SOLVE would be available then APPLYSYM would not have to be changed.

Currently, whenever a computational step could not be performed the user is informed and has the possibility of entering interactively the solution of the unsolved algebraic system or the unsolved linear PDE.

20.1.3 Solving quasilinear PDEs

The content of `QUASILINPDE`

The generalization of special solutions of DEs as well as the computation of similarity and symmetry variables involve the general solution of single first order linear PDEs. The procedure `QUASILINPDE` is a general procedure aiming at the general solution of PDEs

$$a_1(w_i,\phi)\phi_{w_1} + a_2(w_i,\phi)\phi_{w_2} + \ldots + a_n(w_i,\phi)\phi_{w_n} = b(w_i,\phi) \tag{20.22}$$

in n independent variables $w_i, i = 1 \ldots n$ for one unknown function $\phi = \phi(w_i)$.

1. The first step in solving a quasilinear PDE (20.22) is the formulation of the corresponding characteristic ODE-system

$$\frac{dw_i}{d\varepsilon} = a_i(w_j,\phi) \tag{20.23}$$
$$\frac{d\phi}{d\varepsilon} = b(w_j,\phi) \tag{20.24}$$

   for ϕ, w_i regarded now as functions of one variable ε.

   Because the a_i and b do not depend explicitly on ε, one of the equations (20.23),(20.24) with non-vanishing right hand side can be used to divide all others through it and by that having a system with one less ODE to solve. If the equation to divide through is one of (20.23) then the remaining system would be

$$\frac{dw_i}{dw_k} = \frac{a_i}{a_k}, \quad i = 1,2,\ldots k-1,k+1,\ldots n \tag{20.25}$$
$$\frac{d\phi}{dw_k} = \frac{b}{a_k} \tag{20.26}$$

   with the independent variable w_k instead of ε. If instead we divide through equation (20.24) then the remaining system would be

$$\frac{dw_i}{d\phi} = \frac{a_i}{b}, \quad i = 1,2,\ldots n \tag{20.27}$$

   with the independent variable ϕ instead of ε.

   The equation to divide through is chosen by a subroutine with a heuristic to find the "simplest" non-zero right hand side (a_k or b), i.e. one which

   - is constant or
   - depends only on one variable or
   - is a product of factors, each of which depends only on one variable.

One purpose of this division is to reduce the number of ODEs by one. Secondly, the general solution of (20.23), (20.24) involves an additive constant to ε which is not relevant and would have to be set to zero. By dividing through one ODE we eliminate ε and lose the problem of identifying this constant in the general solution before we would have to set it to zero.

2. To solve the system (20.25), (20.26) or (20.27), the procedure `CRACK` is called. Although being designed primarily for the solution of overdetermined PDE-systems, `CRACK` can also be used to solve simple not overdetermined ODE-systems. This solution process is not completely algorithmic. Improved versions of `CRACK` could be used, without making any changes of `QUASILINPDE` necessary.

   If the characteristic ODE-system can not be solved in the form (20.25), (20.26) or (20.27) then successively all other ODEs of (20.23), (20.24) with non-vanishing right hand side are used for division until one is found such that the resulting ODE-system can be solved completely. Otherwise the PDE can not be solved by `QUASILINPDE`.

3. If the characteristic ODE-system (20.23), (20.24) has been integrated completely and in full generality to the implicit solution

$$0 = G_i(\phi, w_j, c_k, \varepsilon), \quad i, k = 1, \ldots, n+1, \quad j = 1, \ldots, n \tag{20.28}$$

   then according to the general theory for solving first order PDEs, ε has to be eliminated from one of the equations and to be substituted in the others to have left n equations. Also the constant that turns up additively to ε is to be set to zero. Both tasks are automatically fulfilled, if, as described above, ε is already eliminated from the beginning by dividing all equations of (20.23), (20.24) through one of them.

   On either way one ends up with n equations

$$0 = g_i(\phi, w_j, c_k), \quad i, j, k = 1 \ldots n \tag{20.29}$$

   involving n constants c_k.

   The final step is to solve (20.29) for the c_i to obtain

$$c_i = c_i(\phi, w_1, \ldots, w_n) \qquad i = 1, \ldots n. \tag{20.30}$$

   The final solution $\phi = \phi(w_i)$ of the PDE (20.22) is then given implicitly through

$$0 = F(c_1(\phi, w_i), c_2(\phi, w_i), \ldots, c_n(\phi, w_i))$$

   where F is an arbitrary function with n arguments.

Syntax

The call of `QUASILINPDE` is
`QUASILINPDE`(*de*, *fun*, *varlist*);

- *de* is the differential expression which vanishes due to the PDE $de = 0$ or, *de* may be the differential equation itself in the form $\ldots = \ldots$.
- *fun* is the unknown function.
- *varlist* is the list of variables of *fun*.

The result of `QUASILINPDE` is a list of general solutions

$$\{sol_1, sol_2, \ldots\}.$$

If `QUASILINPDE` can not solve the PDE then it returns $\{\}$. Each solution sol_i is a list of expressions

$$\{ex_1, ex_2, \ldots\}$$

such that the dependent function (ϕ in (20.22)) is determined implicitly through an arbitrary function F and the algebraic equation

$$0 = F(ex_1, ex_2, \ldots).$$

Examples

Example 1:
To solve the quasilinear first order PDE

$$1 = xu_{,x} + uu_{,y} - zu_{,z}$$

for the function $u = u(x, y, z)$, the input would be

```
depend u,x,y,z;
de:=x*df(u,x)+u*df(u,y)-z*df(u,z) - 1;
varlist:={x,y,z};
QUASILINPDE(de,u,varlist);
```

In this example the procedure returns

$$\{\{x/e^u, ze^u, u^2 - 2y\}\},$$

i.e. there is one general solution (because the outer list has only one element which itself is a list) and u is given implicitly through the algebraic equation

$$0 = F(x/e^u, ze^u, u^2 - 2y)$$

with arbitrary function F.
Example 2:
For the linear inhomogeneous PDE

$$0 = yz_{,x} + xz_{,y} - 1, \quad \text{for} \quad z = z(x, y)$$

QUASILINPDE returns the result that for an arbitrary function F, the equation

$$0 = F\left(\frac{x+y}{e^z}, e^z(x-y)\right)$$

defines the general solution for z.
Example 3:
For the linear inhomogeneous PDE (3.8) from [Kam59]

$$0 = xw_{,x} + (y+z)(w_{,y} - w_{,z}), \quad \text{for} \quad w = w(x, y, z)$$

QUASILINPDE returns the result that for an arbitrary function F, the equation

$$0 = F\left(w,\ y+z,\ \ln(x)(y+z) - y\right)$$

defines the general solution for w, i.e. for any function f

$$w = f\left(y+z,\ \ln(x)(y+z) - y\right)$$

solves the PDE.

Limitations of QUASILINPDE

One restriction on the applicability of QUASILINPDE results from the program CRACK which tries to solve the characteristic ODE-system of the PDE. So far CRACK can be applied only to polynomially non-linear DE's, i.e. the characteristic ODE-system (20.25),(20.26) or (20.27) may only be polynomially non-linear, i.e. in the PDE (20.22) the expressions a_i and b may only be rational in w_j, ϕ.

The task of CRACK is simplified as (20.28) does not have to be solved for w_j, ϕ. On the other hand (20.28) has to be solved for the c_i. This gives a second restriction coming from the REDUCE function SOLVE. Though SOLVE can be applied to polynomial and transzendential equations, again no guarantee for solvability can be given.

20.1.4 Transformation of DEs

The content of DETRAFO

Finally, after having found the finite transformations, the program APPLYSYM calls the procedure DETRAFO to perform the transformations. DETRAFO can also be

used alone to do point- or higher order transformations which involve a considerable computational effort if the differential order of the expression to be transformed is high and if many dependent and independent variables are involved. This might be especially useful if one wants to experiment and try out different coordinate transformations interactively, using DETRAFO as standalone procedure.

To run DETRAFO, the old functions y^α and old variables x^i must be known explicitly in terms of algebraic or differential expressions of the new functions u^β and new variables v^j. Then for point transformations the identity

$$dy^\alpha = \left(y^\alpha{}_{,v^i} + y^\alpha{}_{,u^\beta}\, u^\beta{}_{,v^i}\right) dv^i \tag{20.31}$$

$$= y^\alpha{}_{,x^j}\, dx^j \tag{20.32}$$

$$= y^\alpha{}_{,x^j} \left(x^j{}_{,v^i} + x^j{}_{,u^\beta}\, u^\beta{}_{,v^i}\right) dv^i \tag{20.33}$$

provides the transformation

$$y^\alpha{}_{,x^j} = \frac{dy^\alpha}{dv^i} \cdot \left(\frac{dx^j}{dv^i}\right)^{-1} \tag{20.34}$$

with $det\left(dx^j/dv^i\right) \neq 0$ because of the regularity of the transformation which is checked by DETRAFO. Non-regular transformations are not performed.

DETRAFO is not restricted to point transformations. In the case of contact- or higher order transformations, the total derivatives dy^α/dv^i and dx^j/dv^i then only include all v^i- derivatives of u^β which occur in

$$y^\alpha = y^\alpha(v^i, u^\beta, u^\beta{}_{,v^j}, \ldots)$$
$$x^k = x^k(v^i, u^\beta, u^\beta{}_{,v^j}, \ldots).$$

Syntax

The call of DETRAFO is

DETRAFO({*ex*$_1$, *ex*$_2$, …, *ex*$_m$},
{*ofun*$_1$ =*fex*$_1$, *ofun*$_2$ =*fex*$_2$, …,*ofun*$_p$ =*fex*$_p$},
{*ovar*$_1$ =*vex*$_1$, *ovar*$_2$ =*vex*$_2$, …, *ovar*$_q$ =*vex*$_q$},
{*nfun*$_1$, *nfun*$_2$, …, *nfun*$_p$},
{*nvar*$_1$, *nvar*$_2$, …, *nvar*$_q$});

where m, p, q are arbitrary.

- The *ex*$_i$ are differential expressions to be transformed.
- The second list is the list of old functions *ofun* expressed as expressions *fex* in terms of new functions *nfun* and new independent variables *nvar*.

- Similarly the third list expresses the old independent variables *ovar* as expressions *vex* in terms of new functions *nfun* and new independent variables *nvar*.
- The last two lists include the new functions *nfun* and new independent variables *nvar*.

Names for *ofun, ovar, nfun* and *nvar* can be arbitrarily chosen.

As the result `DETRAFO` returns the first argument of its input, i.e. the list

$$\{ex_1, ex_2, \ldots, ex_m\}$$

where all ex_i are transformed.

Limitations of `DETRAFO`

The only requirement is that the old independent variables x^i and old functions y^α must be given explicitly in terms of new variables v^j and new functions u^β as indicated in the syntax. Then all calculations involve only differentiations and basic algebra.

20.2 ASSIST: Useful Utilities for Various Applications

ASSIST contains a large number of additional general purpose functions that allow a user to better adapt REDUCE to various calculational strategies and to make the programming task more straightforward and more efficient.

Author: Hubert Caprasse

20.2.1 Introduction

The package ASSIST contains an appreciable number of additional general purpose operators which allow one to better adapt REDUCE to various calculational strategies, to make the programming task more straightforward and, sometimes, more efficient.

In contrast with all other packages, ASSIST does not aim to provide either a new facility to compute a definite class of mathematical objects or to extend the base of mathematical knowledge of REDUCE. The operators it contains should be useful independently of the nature of the application which is considered. They were initially written while applying REDUCE to specific problems in theoretical physics. Most of them were designed in such a way that their applicability range is broad. Though it was not the primary goal, efficiency has been sought whenever possible.

The source code in ASSIST contains many comments concerning the meaning and use of the supplementary operators available in the algebraic mode. These comments, hopefully, make the code transparent and allow a thorough exploitation of the package. The present documentation contains a non-technical description of it and describes the various new facilities it provides.

20.2.2 Survey of the Available New Facilities

An elementary help facility is available, independent of the help facility of REDUCE itself. It includes two operators:

`assist` is a operator which takes no argument. If entered, it returns the informations required for a proper use of `assisthelp`.
`assisthelp` takes one argument.

i. If the argument is the identifier `assist`, the operator returns the information necessary to retrieve the names of all the available operators.

ii. If the argument is an integer equal to one of the section numbers of the present documentation. The names of the operators described in that section are obtained.

There is, presently, no possibility to retrieve the number and the type of the arguments of a given operator.

The package contains several modules. Their content reflects closely the various categories of facilities listed below. Some operators do already exist inside the Core of REDUCE. However, their range of applicability is *extended*.

- Control of Switches:

  `switches switchorg`

- Operations on Lists and Bags:

  `mklist kernlist algnlist length`
  `position frequency sequences split`
  `insert insert_keep_order merge_list`
  `first second third rest reverse last`
  `belast cons ( . ) append appendn`
  `remove delete delete_all delpair`
  `member elmult pair depth mkdepth_one`
  `repfirst represt asfirst aslast asrest`
  `asflist asslist restaslist substitute`
  `bagprop putbag clearbag bagp baglistp`
  `alistp abaglistp listbag`

- Operations on Sets:

  `mkset setp union intersect diffset symdiff`

- General Purpose Utility Functions:

  `list_to_ids mkidn mkidnew dellastdigit detidnum`
  `oddp followline == randomlist mkrandtabl`
  `permutations cyclicpermlist perm_to_num`
  `num_to_perm combnum combinations symmetrize`
  `remsym sortnumlist sortlist algsort extremum`
  `gcdnl`
  `depatom funcvar implicit explicit remnoncom`
  `korderlist simplify checkproplist extractlist`

- Properties and Flags:

  `putflag putprop displayprop displayflag`
  `clearflag clearprop`

- Control Statements, Control of Environment:

`nordp` `depvarp` `alatomp` `alkernp` `precp`
`show` `suppress` `clearop` `clearfunctions`

- Handling of Polynomials:

  `alg_to_symb` `symb_to_alg`
  `distribute` `leadterm` `redexpr` `monom`
  `lowestdeg` `divpol` `splitterms` `splitplusminus`

- Handling of Transcendental Functions:

  `trigexpand` `hypexpand` `trigreduce` `hypreduce`

- Coercion from Lists to Arrays and converse:

  `list_to_array` `array_to_list`

- Handling of n-dimensional Vectors:

  `sumvect` `minvect` `scalvect` `crossvect` `mpvect`

- Handling of Grassmann Operators:

  `putgrass` `remgrass` `grassp` `grassparity`
  `ghostfactor`

- Handling of Matrices:

  `unitmat` `mkidm` `baglmat` `coercemat`
  `submat` `matsubr` `matsubc` `rmatextr` `rmatextc`
  `hconcmat` `vconcmat` `tpmat` `hermat`
  `seteltmat` `geteltmat`

- Control of the HEPHYS package:

  `remvector` `remindex` `mkgam`

In the following all these operators are described.

20.2.3 Control of Switches

The two available operators i.e. `switches`, `switchorg` have no argument and are called as if they were mere identifiers.

`switches` displays the actual status of the most frequently used switches when manipulating rational operators. The chosen switches are

exp, allfac, ezgcd, gcd, mcd, lcm, div, rat,
intstr, rational, precise, reduced, rationalize,
combineexpt, complex, revpri, distribute.

The selection is somewhat arbitrary but it may be changed in a trivial fashion by the user.

The new switch `distribute` allows one to put polynomials in a distributed form (see the description below of the new operators for manipulating them).

Most of the symbolic variables `!*exp`, `!*div`, ... which have either the value `t` or the value `nil` are made available in the algebraic mode so that it becomes possible to write conditional statements of the kind

```
if !*exp then do ......

if !*gcd then off gcd;
```

`SWITCHORG` resets the switches enumerated above to the status they had when **starting** REDUCE.

20.2.4 Manipulation of the List Structure

Additional operators for list manipulations are provided and some already defined operators in the kernel of REDUCE are modified to properly generalize them to the available new structure `bag`.

i. Generation of a list of length n with all its elements initialized to 0 and possibility to append to a list l a certain number of zero's to make it of length n:

```
mklist n ;     n is an integer

mklist(l,n);     l is List-like, n is an integer
```

ii. Generation of a list of sublists of length n containing p elements equal to 0 and q elements equal to 1 such that

$$p + q = n.$$

The operator `sequences` works both in algebraic and symbolic modes. Here is an example in the algebraic mode:

```
sequences 2 ; ==> {{0,0},{0,1},{1,0},{1,1}}
```

An arbitrary splitting of a list can be done. The operator `split` generates a list which contains the splitted parts of the original list.

```
split({a,b,c,d},{1,1,2}) ==> {{a},{b},{c,d}}
```

The operator `algnlist` constructs a list which contains n copies of a list bound to its first argument.

```
algnlist({a,b,c,d},2); ==> {{a,b,c,d},{a,b,c,d}}
```

The operator `kernlist` transforms any prefix of a kernel into the `list` prefix. The output list is a copy:

```
kernlist (<kernel>); ==> {<kernel arguments>}
```

iii. Four operators to delete elements are `delete`, `remove`, `delete_all` and `delpair`. The first two act as in symbolic mode, and the third eliminates from a given list *all* elements equal to its first argument. The fourth acts on a list of pairs and eliminates from it the *first* pair whose first element is equal to its first argument :

```
delete(x,{a,b,x,f,x}); ==> {a,b,f,x}

remove({a,b,x,f,x},3); ==> {a,b,f,x}

delete_all(x,{a,b,x,f,x}); ==> {a,b,f}

delpair(a,{{a,1},{b,2},{c,3}}; ==> {{b,2},{c,3}}
```

iv. The operator `elmult` returns an *integer* which is the *multiplicity* of its first argument inside the list which is its second argument. The operator `frequency` gives a list of pairs whose second element indicates the number of times the first element appears inside the original list:

```
elmult(x,{a,b,x,f,x}) ==> 2

frequency({a,b,c,a}); ==> {{a,2},{b,1},{c,1}}
```

v. The operator `insert` allows one to insert a given object into a list at the desired position.

The operators `insert_keep_order` and `merge_list` allow one to keep a given ordering when inserting one element inside a list or when merging two lists. Both have 3 arguments. The last one is the name of a binary boolean ordering function:

```
ll:={1,2,3}$

insert(x,ll,3); ==> {1,2,x,3}

insert_keep_order(5,ll,lessp); ==> {1,2,3,5}

merge_list(ll,ll,lessp); ==> {1,1,2,2,3,3}
```

Notice that `merge_list` will act correctly only if the two lists are well ordered themselves.

vi. Algebraic lists can be read from right to left or left to right. They *look* symmetrical. One would like to dispose of manipulation functions which reflect this. So, to the already defined functions `first` and `rest` are added the functions `last` and `belast`. `last` gives the last element of the list while `belast` gives the list *without* its last element.
Various additional functions are provided. They are:

`.` ("dot"), `position`, `depth`, `mkdepth_one`, `pair`, `appendn`, `repfirst`, `represt`

The token "dot" needs a special comment. It corresponds to several different operations.

1. If one applies it on the left of a list, it acts as the `cons` infix operator. Note however that blank spaces are required around the dot:

   ```
   4 . {a,b}; ==> {4,a,b}
   ```

2. If one applies it on the right of a list, it has the same effect as the `part` operator:

   ```
   {a,b,c}.2; ==> b
   ```

3. If one applies it to a 4-dimensional vectors, it acts as in the HEPHYS package.

`position` returns the *position* of the first occurrence of x in a list or a message if x is not present in it.

`depth` returns an *integer* equal to the number of levels where a list is found if and only if this number is the *same* for each element of the list otherwise it returns a message telling the user that the list is of *unequal depth*. The function `mkdepth_one` allows to transform any list into a list of depth equal to 1.

`pair` has two arguments which must be lists. It returns a list whose elements are *lists of two elements.* The n^{th} sublist contains the n^{th} element of the first list and the n^{th} element of the second list. These types of lists are

called *association lists* or short *alists* in the following. To test for these type of lists a boolean function `abaglistp` is provided. It will be discussed below.
`appendn` has *any* fixed number of lists as arguments. It generalizes the already existing function `append` which accepts only two lists as arguments. It may also be used for arbitrary kernels but, in that case, it is important to notice that *the concatenated object is always a list*.
`repfirst` has two arguments. The first one is any object, the second one is a list. It replaces the first element of the list by the object. It works like the symbolic mode (lisp) function `rplaca` except that the original list is not destroyed.
`represt` has also two arguments. It replaces the rest of the list by its first argument and returns the new list *without destroying* the original list. It is analogous to the symbolic mode (lisp) function `rplacd`. Here are examples:

```
ll:={{a,b}}$
lll:=ll.1; ==> {a,b}
ll.0; ==> list
0 . ll; ==> {0,{a,b}}

depth ll; ==> 2

pair(lll,lll); ==> {{a,a},{b,b}}

repfirst{new,ll); ==> {new}

ll3:=appendn(lll,lll,lll); ==> {a,b,a,b,a,b}

position(b,ll3); ==> 2

represt(new,ll3); ==> {a,new}
```

vii. The functions `asfirst`, `aslast`, `asrest`, `asflist`, `asslist`, and `restaslist` act on alists or on lists of lists of well defined depths and have two arguments. The first is the key object which one seeks to associate in some way with an element of the association list which is the second argument.
`asfirst` returns the pair whose first element is equal to the first argument.
`aslast` returns the pair whose last element is equal to the first argument.
`asrest` needs a *list* as its first argument. The function seeks the first sublist of a list of lists (which is its second argument) equal to its first argument and returns it.
`restaslist` has a *list of keys* as its first argument. It returns the collection

of pairs which meet the criterium of `asrest`.
`asflist` returns a list containing *all pairs* which satisfy the criteria of the function `asfirst`. So the output is also an association list.
`asslist` returns a list which contains *all pairs* which have their second element equal to the first argument.
Here are a few examples:

```
lp:={{a,1},{b,2},{c,3}}$

asfirst(a,lp); ==> {a,1}

aslast(1,lp); ==> {a,1}

asrest({1},lp); ==> {a,1}

restaslist({a,b},lp); ==> {{1},{2}}

lpp:=append(lp,lp)$

asflist(a,lpp); ==> {{a,1},{a,1}}

asslist(1,lpp); ==> {{a,1},{a,1}}
```

vii. The function `substitute` has three arguments. The first is the object to be substituted, the second is the object which must be replaced by the first, and the third is the list in which the substitution must be made. Substitution is made to all levels. It is a more elementary function than `sub` but its capabilities are less. When dealing with algebraic quantities, it is important to make sure that *all* objects involved in the function have either the prefix lisp or the standard quotient representation otherwise it will not properly work.

20.2.5 The Bag Structure and its Associated Functions

The list structure of REDUCE is very convenient for manipulating groups of objects which are, a priori, unknown. This structure is endowed with other properties such as "mapping" i.e. the fact that if `op` is an operator one gets, by default,

```
op({x,y}); ==> {op(x),op(y)}
```

It is not permitted to submit lists to the operations valid on rings so that, for example, lists cannot be indeterminates of polynomials.
Very frequently too, procedure arguments cannot be lists. At the other extreme,

so to say, one has the kernel structure associated with the algebraic declaration `operator` . This structure behaves as an "unbreakable" one and, for that reason, behaves like an ordinary identifier. It may generally be bound to all non-numeric procedure parameters and it may appear as an ordinary indeterminate inside polynomials.
The `BAG` structure is intermediate between a list and an operator. From the operator it borrows the property of being a kernel and, therefore, may be an indeterminate of a polynomial. From the list structure it borrows the property of being a *composite* object.

Definition:

A bag is an object endowed with the following properties:

1. It is a kernel, i.e. it is composed of an atomic prefix (its envelope) and its content (miscellaneous objects).
2. Its content may be handled in an analogous way as the content of a list. The important difference is that during these manipulations the name of the bag is *kept*.
3. Properties may be given to the envelope. For instance, one may declare it `noncom` or `symmetric`, etc.

Available Functions:

i. A default bag envelope
textttbag is defined. It is a reserved identifier. An identifier other than `list` or one which is already associated with a boolean function may be defined as a bag envelope through the command `putbag`. In particular, any operator may also be declared to be a bag. **When and only when** the identifier is not an already defined function then `putbag` set for it the property of an *operator prefix*. The command:

```
putbag id1,id2,....idn;
```

declares `id1,...,idn` as bag envelopes. Analogously, the command

```
clearbag id1,...idn;
```

eliminates the bag property on `id1,...,idn`.

ii. The boolean operator `bagp` detects the bag property. Here is an example:

```
aa:=bag(x,y,z)$

if bagp aa then "ok"; ==> ok
```

iii. The functions listed below may act both on lists or bags. Moreover, functions subsequently defined for *sets*SETS also work for a bag when its content is a set. Here is a list of the main ones:

`FIRST`, `second`, `last`, `rest`, `belast`, `depth`, `length`, `reverse`,
`member`, `append`, `.` (“`dot`”), `repfirst`, `represt`, ...

However, since they keep track of the envelope, they act somewhat differently. Remember that

the *name* of the *envelope* is *kept* by the operators
`first`, `second` and `last`.

Here are a few examples (more examples are given inside the test file):

```
putbag op; ==> t

aa:=op(x,y,z)$

first op(x,y,z); ==> op(x)

rest op(x,y,z); ==> op(y,z)

belast op(x,y,z); ==> op(x,y)

append(aa,aa); ==> op(x,y,z,x,y,z)

appendn(aa,aa,aa); ==> {x,y,z,x,y,z,x,y,z}

length aa; ==> 3

depth aa; ==> 1

member(y,aa); ==> op(y,z)
```

When “appending” two bags with *different* envelopes, the resulting bag gets the name of the one bound to the first parameter of `append`. When `appendn` is used, the output is always a list.
The function `length` gives the number of objects contained in the bag.

iv. The connection between the list and the bag structures is made easy thanks to `kernlist` which transforms a bag into a list and thanks to the coercion function `listbag` which transforms a list into a bag. This function has 2 arguments and is used as follows:

`listbag(`⟨*list*⟩`,`⟨*id*⟩`);` ==> ⟨*id*⟩(⟨*arg_list*⟩)

The identifier ⟨*id*⟩, if allowed, is automatically declared as a bag envelope or an error message is generated.

Finally, two boolean functions which work both for bags and lists are provided. They are `baglistp` and `abaglistp`. They return t or nil (in a conditional statement) if their argument is a bag or a list for the first one, or if their argument is a list of sublists or a bag containing bags for the second one.

20.2.6 Sets and their Manipulation Functions

Functions for sets exist at the level of symbolic mode. The package makes them available in algebraic mode but also *generalizes* them so that they can be applied to bag-like objects as well.

i. The constructor `mkset` transforms a list or bag into a set by eliminating duplicates.

```
mkset({1,a,a}); ==> {1,a}
mkset bag(1,a,1,a); ==> bag(1,a)
```

`setp` is a boolean function which recognizes set–like objects.

```
if setp {1,2,3} then ... ;
```

ii. The available functions are

`union`, `intersect`, `diffset`, `symdiff`.

They have two arguments which must be sets otherwise an error message is issued. Their meaning is transparent from their name. They respectively give the union, the intersection, the difference and the symmetric difference of two sets.

20.2.7 General Purpose Utility Functions

Functions in this sections have various purposes. They have all been used many times in applications in some form or another. The form given to them in this package is adjusted to maximize their range of applications.

i. The operators `mkidnew`, `dellastdigit`, `detidnum`, `list_to_ids` handle identifiers.

`mkidnew` has either 0 or 1 argument. It generates an identifier which has not yet been used before.

```
mkidnew(); ==> g0001

mkidnew(a); ==> ag0002
```

`dellastdigit` takes an integer as argument and strips from it its last digit.

```
dellastdigit 45; ==> 4
```

`detidnum` deletes the last digit from an identifier. It is a very convenient function when one wants to make a do loop starting from a set of indices $a_1, \ldots, a_n$.

```
detidnum a23; ==> 23
```

`list_to_ids` generalizes the function `mkid` to a list of atoms. It creates and intern an identifier from the concatenation of the atoms. The first atom cannot be an integer.

```
list_to_ids {a,1,id,10}; ==> a1id10
```

The boolean operator `oddp` detects odd integers.

The function `followline` is convenient when using the function `prin2`. It allows one to format output text in a much more flexible way than with the `write` statement.
Try the following examples :

```
<<prin2 2; prin2 5>>$ ==> ?

<<prin2 2; followline(5); prin2 5;>>; ==> ?
```

The infix operator == is a short and convenient notation for the `set` function. In fact it is a *generalization* of it to allow one to deal also with kernels:

```
operator op;

op(x):=abs(x)$

op(x) == x; ==> x

op(x); ==> x

abs(x); ==> x
```

The function `randomlist` generates a list of random numbers. It takes two arguments which are both integers. The first one indicates the range inside which the random numbers are chosen. The second one indicates how many numbers are to be generated. Its output is the list of generated numbers.

```
randomlist(10,5); ==> {2,1,3,9,6}
```

`mkrandtabl` generates a table of random numbers. This table is either a one or two dimensional array. The base of random numbers may be either an integer or a decimal number. In this last case, to work properly, the switch `rounded` must be ON. It has three arguments. The first is either a one integer or a two integer list. The second is the base chosen to generate the random numbers. The third is the chosen name for the generated array. In the example below a two-dimensional table of random integers is generated as array elements of the identifier `ar`.

```
mkrandtabl({3,4},10,ar); ==>

       *** array ar redefined

             {3,4}
```

The output is the dimension of the constructed array.

`permutations` gives the list of permutations of n objects. Each permutation is itself a list. `cyclicpermlist` gives the list of *cyclic* permutations. For both functions, the argument may also be a `bag`.

```
permutations {1,2} ==> {{1,2},{2,1}}

cyclicpermlist {1,2,3} ==>

        {{1,2,3},{2,3,1},{3,1,2}}
```

perm_to_num and num_to_perm allow to associate to a given permutation of n numbers or identifiers a number between 0 and $n! - 1$. The first function has the two permutated lists as its arguments and it returns an integer. The second one has an integer as its first argument and a list as its second argument. It returns the list of permutated objects.

```
perm_to_num({4,3,2,1},{1,2,3,4}) ==> 23

num_to_perm(23,{1,2,3,4}); ==> {4,3,2,1}
```

combnum gives the number of combinations of n objects taken p at a time. It has the two integer arguments n and p.

combinations gives a list of combinations on n objects taken p at a time. It has two arguments. The first one is a list (or a bag) and the second one is the integer p.

```
combinations({1,2,3},2) ==> {{2,3},{1,3},{1,2}}
```

remsym is a command that suppresses the effect of the REDUCE commands symmetric or antisymmetric .

symmetrize is a powerful function which generates a symmetric expression. It has 3 arguments. The first is a list (or a list of lists) containing the expressions which will appear as variables for a kernel. The second argument is the kernel-name and the third is a permutation function which exists either in algebraic or symbolic mode. This function may be constructed by the user. Within this package the two functions permutations and cyclicpermlist may be used. Examples:

```
ll:={a,b,c}$

symmetrize(ll,op,cyclicpermlist); ==>

        op(a,b,c) + op(b,c,a) + op(c,a,b)

symmetrize(list ll,op,cyclicpermlist); ==>

        op({a,b,c}) + op({b,c,a}) + op({c,a,b})
```

Notice that, taking for the first argument a list of lists gives rise to an expression where each kernel has a *list as argument*. Another peculiarity of this function is the fact that, unless a pattern matching is made on the operator op, it needs to be reevaluated. This peculiarity is convenient when op is an abstract operator if one wants to control the subsequent simplification process. Here is an illustration:

```
op(a,b,c):=a*b*c$

symmetrize(ll,op,cyclicpermlist); ==>

          op(a,b,c) + op(b,c,a) + op(c,a,b)

reval ws; ==>

          op(b,c,a) + op(c,a,b) + a*b*c

for all x let op(x,a,b)=sin(x*a*b);

symmetrize(ll,op,cyclicpermlist); ==>

           op(b,c,a) + sin(a*b*c) + op(a,b,c)
```

The functions `sortnumlist` and `sortlist` are functions which sort lists. They use the *bubblesort* and the *quicksort* algorithms.

`sortnumlist` takes as argument a list of numbers. It sorts it in increasing order.

`sortlist` is a generalization of the above function. It sorts the list according to any well defined ordering. Its first argument is the list and its second argument is the ordering function. The content of the list need not necessarily be numbers but must be such that the ordering function has a meaning.

`algsort` exploits the PSL `sort` function. It is intended to replace the two functions above.

```
l:={1,3,4,0}$  sortnumlist l; ==> {0,1,3,4}

ll:={1,a,tt,z}$ sortlist(ll,ordp); ==> {a,z,tt,1}

l:={-1,3,4,0}$  algsort(l,>); ==> {4,3,0,-1}
```

It is important to realise that using these functions for kernels or bags may be dangerous since they are destructive. If it is necessary, it is recommended to first apply `kernlist` to them to act on a copy.

The function `extremum` is a generalization of the already defined functions `min`, `max` to include general orderings. It is a 2 argument function. The first is the list and the second is the ordering function. With the list `ll` defined in the last example, one gets

```
extremum(ll,ordp); ==> 1
```

`GCDNL` takes a list of integers as argument and returns their gcd.

iii. There are four functions to identify dependencies. `funcvar` takes any expression as argument and returns the set of variables on which it depends. Constants are eliminated.

```
funcvar(e+pi+sin(log(y)); ==> {y}
```

`depatom` has an **atom** as argument. It returns it if it is a number or if no dependency has previously been declared. Otherwise, it returns the list of variables which the previous `DEPEND` declarations imply.

```
depend a,x,y;

depatom a; ==> {x,y}
```

The operators `explicit` and `implicit` make explicit or implicit the dependencies. This example shows how they work:

```
depend a,x; depend x,y,z;

explicit a; ==> a(x(y,z))

implicit ws; ==> a
```

These are useful when one wants to trace the names of the independent variables and (or) the nature of the dependencies.

`korderlist` is a zero argument function which displays the actual ordering.

```
korder x,y,z;

korderlist; ==> (x,y,z)
```

iv. A command `remnoncom` to remove the non-commutativity of operators previously declared non-commutative is available. Its use is like the one of the command `noncom`.

v. Filtering functions for lists.

`checkproplist` is a boolean function which checks if the elements of a list have a definite property. Its first argument is the list, its second argument is a boolean operator (`fixp`, `numberp`, ...) or an ordering function (as `ordp`).

`extractlist` extracts from the list given as its first argument the elements which satisfy the boolean function given as its second argument. For example:

```
if checkproplist({1,2},fixp) then "ok"; ==> ok

l:={1,a,b,"st")$

extractlist(l,fixp); ==> {1}

extractlist(l,stringp); ==> {st}
```

vi. Coercion.

Since lists and arrays have quite distinct behaviour and storage properties, it is interesting to coerce lists into arrays and vice-versa in order to fully exploit the advantages of both datatypes. The functions `array_to_list` and `list_to_array` are provided to do that easily. The first function has the array identifier as its unique argument. The second function has three arguments. The first is the list, the second is the dimension of the array and the third is the identifier which defines it. If the chosen dimension is not compatible with the the list depth, an error message is issued. As an illustration suppose that ar is an array whose components are 1,2,3,4. then

```
array_to_list ar; ==> {1,2,3,4}

list_to_array({1,2,3,4},1,arr}; ==>
```

generates the array arr with the components 1,2,3,4.

vii. Control of the HEPHYS package.

The commands `remvector` and `remindex` remove the property of being a 4-vector or a 4-index respectively.

The function `mkgam` allows to assign to any identifier the property of a Dirac gamma matrix and, eventually, to suppress it. Its interest lies in the fact that, during a calculation, it is often useful to transform a gamma matrix into an abstract operator and vice-versa. Moreover, in many applications in basic physics, it is interesting to use the identifier g for other purposes. It takes two arguments. The first is the identifier. The second must be chosen equal to `t` if one wants to transform it into a gamma matrix. Any other binding for this second argument suppresses the property of being a gamma matrix the identifier is supposed to have.

20.2.8 Properties and Flags

In spite of the fact that many facets of the handling of property lists is easily accessible in algebraic mode, it is useful to provide analogous functions *genuine* to the algebraic mode. The reason is that, altering property lists of objects, may easily

destroy the integrity of the system. The functions, which are here described, *do ignore* the property list and flags already defined by the system itself. They generate and track the *addtional properties and flags* that the user issues using them. They offer him the possibility to work on property lists so that he can design a programming style of the "conceptual" type.

i. We first consider "flags".

To a given identifier, one may associate another one linked to it "in the background". The three functions `putflag`, `displayflag` and `clearflag` handle them.

`putflag` has 3 arguments. The first one is the identifier or a list of identifiers, the second one is the name of the flag, and the third one is `t` (true) or 0 (zero). When the third argument is `t`, it creates the flag, when it is 0 it destroys it. In this last case, the function does return nil (not seen inside the algebraic mode).

```
putflag(z1,flag_name,t); ==> flag_name

putflag({z1,z2},flag1_name,t); ==> t

putflag(z2,flag1_name,0) ==>
```

`displayflag` allows one to extract flags. The previous actions give:

```
displayflag z1; ==>{flag_name,flag1_name}

displayflag z2 ; ==> {}
```

`clearflag` is a command which clears *all* flags associated with the identifiers $id_1, \ldots, id_n$.

ii. Properties are handled by similar operators. `putprop` has four arguments. The second argument is, here, the *indicator* of the property. The third argument may be *any valid expression*. The fourth one is also t or 0.

```
putprop(z1,property,x^2,t); ==> z1
```

In general, one enters

```
putprop(list(idp1,idp2,..),<propname>,<value>,t);
```

To display a specific property, one uses `displayprop` which takes two arguments. The first is the name of the identifier, the second is the indicator of the property.

```
                                                    2
displayprop(z1,property); ==> {property,x  }
```

Finally, `clearprop` is a nary commmand which clears *all* properties of the identifiers which appear as arguments.

20.2.9 Control Functions

Here we describe additional functions which improve user control on the environment.

i. The first set of functions is composed of unary and binary boolean functions. They are:

```
alatomp x;     x is anything.
alkernp x;     x is anything.
depvarp(x,v); x is anything.
```

(`v` is an atom or a kernel.) `alatomp` has the value `t` iff `x` is an integer or an identifier *after* it has been evaluated down to the bottom.

`alkernp` has the value `t` iff `x` is a kernel *after* it has been evaluated down to the bottom.

`depvarp` returns `t` iff the expression `x` depends on `v` at *any level*.

The above functions together with `precp` have been declared operator functions to ease the verification of their value.

`nordp` is equal to `not ordp`.

ii. The next functions allow one to *analyze* and to *clean* the environment of REDUCE created by the user while working **interactively**. Two functions are provided:

`show` allows the user to get the various identifiers already assigned and to see their type. `suppress` selectively clears the used identifiers or clears them all. It is to be stressed that identifiers assigned from the input of files are *ignored*. Both functions have one argument and the same options for this argument:

```
show (suppress) all
show (suppress) scalars
show (suppress) lists
show (suppress) saveids
            (for saved expressions)
show (suppress) matrices
```

```
show (suppress) arrays
show (suppress) vectors
            (contains vector, index and tvector)
show (suppress) forms
```

The option `all` is the most convenient for `show` but, with it, it may takes some time to get the answer after one has worked several hours. When entering REDUCE the option `all` for `show` gives:

```
show all; ==>

        scalars are: NIL
        arrays are: NIL
        lists are: NIL
        matrices are: NIL
        vectors are: NIL
        forms are: NIL
```

It is a convenient way to remind the various options. Here is an example which is valid when one starts from a fresh environment:

```
a:=b:=1$

show scalars; ==>  scalars are: (a b)

suppress scalars; ==> t

show scalars; ==>  scalars are: nil
```

iii. The `clear` command of the system does not do a complete cleaning of operators and functions. The following two commands do a more complete cleaning and, also, automatically takes into account the *user* flag and properties that the functions `putflag` and `putprop` may have introduced.

Their names are `clearop` and `clearfunctions`. `clearop` takes one operator as its argument.
`clearfunctions` is a nary command. If one issues

```
clearfunctions a1,a2, ... , an $
```

The functions with names `a1`, `a2`, ..., `an` are cleared. One should be careful when using this facility since the only functions which cannot be erased are those which are protected with the `lose` flag.

20.2.10 Handling of Polynomials

The module contains some utility functions to handle standard quotients and several new facilities to manipulate polynomials.

i. Two operators `alg_to_symb` and `symb_to_alg` allow one to change an expression which is in the algebraic standard quotient form into a prefix lisp form and vice-versa. This is done in such a way that the symbol `list` which appears in the algebraic mode disappears in the symbolic form (there it becomes a parenthesis "()") and it is reintroduced in the translation from a symbolic prefix lisp expression to an algebraic one. Here, is an example, showing how the wellknown lisp function `flattens` can be trivially transposed inside the algebraic mode:

```
algebraic procedure ecrase x;
lisp symb_to_alg
   flattens1 alg_to_symb algebraic x;

symbolic procedure flattens1 x;
% ll; ==> ((a b) ((c d) e))
% flattens1 ll; (a b c d e)
  if atom x then list x else
  if cdr x then
      append(flattens1 car x, flattens1 cdr x)
    else flattens1 car x;
```

gives, for instance,

```
ll:={a,{b,{c},d,e},{{{z}}}}$

ecrase ll; ==> {a, b, c, d, e, z}
```

The function `mkdepth_one` described above implements that functionality.

ii. `leadterm` and `redexpr` are the algebraic equivalent of the symbolic mode functions `lt` and `red`. They give, respectively, the *leading term* and the *reductum* of a polynomial. They also work for rational functions. Their interest lies in the fact that they do not require one to extract the main variable. They work according to the current ordering of the system:

```
pol:=x++y+z$

leadterm pol; ==> x
```

```
korder y,x,z;

leadterm pol; ==> y

redexpr pol; ==> x + z
```

By default, the representation of multivariate polynomials is recursive. It is justified since it is the one which takes the least memory. With such a representation, the function `leadterm` does not necessarily extract a true monom. It extracts a monom in the leading indeterminate multiplied by a polynomial in the other indeterminates. However, very often, one needs to handle true monoms separately. In that case, one needs a polynomial in *distributive* form. Such a form is provided by the package GROEBNER (H. Melenk et al.). The facility there is, however, much too involved in many applications and the necessity to load the package makes it interesting to construct an elementary facility to handle the distributive representation of polynomials. A new switch has been created for that purpose. It is called `distribute` and a new function `distribute` puts a polynomial in distributive form. With that switch set to on, `leadterm` returns true monoms.

`monom` transforms a polynomial into a list of monoms. It works *whatever the position of the switch* `distribute`.

`splitterms` is analoguous to `monom` except that it gives a list of two lists. The first sublist contains the positive terms while the second sublist contains the negative terms.

`splitplusminus` gives a list whose first element is the positive part of the polynomial and its second element is its negative part.

iii. Two complementary operators `lowestdeg` and `divpol` are provided. The first takes a polynomial as its first argument and the name of an indeterminate as its second argument. It returns the *lowest degree* in that indeterminate. The second function takes two polynomials and returns both the quotient and its remainder.

20.2.11 Handling of Transcendental Functions

The functions `trigreduce` and `trigexpand` and the equivalent ones for hyperbolic functions `hypreduce` and `hypexpand` make the transformations to multiple arguments and from multiple arguments to elementary arguments. Here is a simple example:

```
aa:=sin(x+y)$
```

```
trigexpand aa; ==> sin(x)*cos(y) + sin(y)*cos(x)

trigreduce ws; ==> sin(y + x)
```

When a trigonometric or hyperbolic expression is symmetric with respect to the interchange of `sin` (`sinh`) and `cos` (`cosh`), the application of `trigreduce` (`hypreduce`) may often lead to great simplifications. However, if it is highly asymmetric, the repeated application of `trigreduce` (`hypreduce`) followed by the use of `trigexpand` (`hypexpand`) will lead to *more* complicated but more symmetric expressions:

```
aa:=(sin(x)^2+cos(x)^2)^3$

trigreduce aa; ==> 1

bb:=1+sin(x)^3$

trigreduce bb; ==>

          - sin(3*x) + 3*sin(x) + 4
        ---------------------------
                     4

trigexpand ws; ==>

         3                    2
  sin(x)  - 3*sin(x)*cos(x)  + 3*sin(x) + 4
 ------------------------------------------
                      4
```

20.2.12 Handling of n-dimensional Vectors

Explicit vectors in euclidean space may be represented by list-like or bag-like objects of depth 1. The components may be bags but may *not* be lists. Functions are provided to do the sum, the difference and the scalar product. When the space-dimension is three there are also functions for the cross and mixed products.

`sumvect`, `minvect`, `scalvect`, and `crossvect` have two arguments. `mpvect` has three arguments. The following example is sufficient to explain how they work:

```
l:={1,2,3}$

ll:=list(a,b,c)$
```

```
sumvect(l,ll); ==> {a + 1,b + 2,c + 3}

minvect(l,ll); ==> { - a + 1, - b + 2, - c + 3}

scalvect(l,ll); ==> a + 2*b + 3*c

crossvect(l,ll); ==> { - 3*b + 2*c,
                       3*a - c, - 2*a + b}

mpvect(l,ll,l); ==> 0
```

20.2.13 Handling of Grassmann Operators

Grassman variables are often used in physics. For them the multiplication operation is associative, distributive but anticommutative. The core of REDUCE does not provide it. However, implementing it in full generality would almost certainly decrease the overall efficiency of the system. This small module together with the declaration of antisymmetry for operators is enough to deal with most calculations. The reason is, that a product of similar anticommuting kernels can easily be transformed into an antisymmetric operator with as many indices as the number of these kernels. Moreover, one may also issue pattern matching rules to implement the anticommutativity of the product. The functions in this module represent the minimum functionality required to identify them and to handle their specific features.

`putgrass` is a (nary) command which give identifiers the property of being the names of Grassmann kernels. `remgrass` removes this property.

`grassp` is a boolean function which detects grassmann kernels.

`GRASSPARITY` takes a monom as argument and gives its parity. If the monom is a simple grassmann kernel it returns 1.

`GHOSTFACTOR` has two arguments. Each one is a monom. It is equal to

```
(-1)**(grassparity u * grassparity v)
```

Here is an illustration to show how the above functions work:

```
putgrass eta; ==> t

if grassp eta(1) then "grassmann kernel"; ==>

                grassmann kernel
```

```
aa:=eta(1)*eta(2)-eta(2)*eta(1); ==>

        aa :=  - eta(2)*eta(1) + eta(1)*eta(2)

grassparity eta(1); ==> 1

grassparity (eta(1)*eta(2)); ==> 0

ghostfactor(eta(1),eta(2)); ==> -1

grasskernel:=
  {eta(~x)*eta(~y)
       => -eta y * eta x when nordp(x,y),
  (~x)*(~x) => 0 when grassp x};

exp:=eta(1)^2$

exp where grasskernel; ==> 0

aa where grasskernel; ==>  - 2*eta(2)*eta(1)
```

20.2.14 Handling of Matrices

This module provides functions for handling matrices more comfortably.

i. Often, one needs to construct some unit matrix of a given dimension. This construction is done by the system thanks to the command `unitmat`. It takes any number of arguments:

```
unitmat m1(n1), m2(n2), .....mi(ni) ;
```

where `m1`, `m2`,...,`mi` are names of matrices and `n1`, `n2`,...,`ni` are integers.

`mkidm` is a generalization of `mkid`. It allows one to connect two or several matrices. If `u` and `u1` are two matrices, one can go from one to the other:

```
matrix u(2,2);$  unitmat u1(2)$

u1; ==>

        [1  0]
        [    ]
        [0  1]
```

```
mkidm(u,1); ==>

          [1   0]
          [     ]
          [0   1]
```

This operators allows one to make loops on matrices like in the following illustration. If `u`, `u1`, `u2`,..., `u5` are matrices:

```
for i:=1:5 do u:=u-mkidm(u,i);
```

can be issued.

ii. The next functions map matrices on bag-like or list-like objects and conversely they generate matrices from bags or lists.

`coercemat` transforms the matrix `u` into a list of lists. The entry is

```
coercemat(u,id)
```

where `id` is equal to `list`, otherwise it transforms it into a bag of bags whose envelope is equal to `id`.

`baglmat` does the opposite job. The first argument is the bag-like or list-like object while the second argument is the matrix identifier. The input is

```
baglmat(bgl,u)
```

`bgl` becomes the matrix `u` . The transformation is not done if `u` is already the name of a previously defined matrix. This is to avoid accidental redefinition of that matrix.

ii. The operators `submat`, `matextr`, and `matextc` take parts of a given matrix.

`submat` has three arguments. The entry is

```
submat(u,nr,nc)
```

The first is the matrix name, and the other two are the row and column numbers. It gives the submatrix obtained from `u` by deleting the row `nr` and the column `nc`. When one of them is equal to zero only column `nc` or row `nr` is deleted.

`matextr` and `matextc` extract a row or a column and place it into a list-like or bag-like object. The entries are

```
matextr(u,vn,nr)

matextc(u,vn,nc)
```

where u is the matrix, vn is the "vector name", nr and nc are integers. If vn is equal to list the vector is returned as a list otherwise as a bag.

iii. Functions which manipulate matrices. They are matsubr, matsubc, hconcmat, vconcmat, tpmat, and hermat.

matsubr and matsubc substitute rows and columns. They have three arguments. Entries are:

```
matsubr(u,bgl,nr)

matsubc(u,bgl,nc)
```

The meaning of the variables u, nr, and nc is the same as above while bgl is a list-like or bag-like vector. Its length should be compatible with the dimensions of the matrix.

hconcmat and vconcmat concatenate two matrices. The entries are

```
hconcmat(u,v)

vconcmat(u,v)
```

The first function concatenates horizontally, the second one concatenates vertically. The dimensions must match.

tpmat makes the tensor product of two matrices. It is also an *infix* operator. The entry is

```
tpmat(u,v) or u tpmat V
```

hermat takes the hermitian conjuguate of a matrix. The entry is

```
 hermat(u,hu)
```

where
textttu is the identifier for the hermitian conjugate of matrix u. It should be *unassigned* for this function to work successfully. This is done on purpose to prevent accidental redefinition of an already used identifier.

iv. setelmat getelmat are functions of two integers. The first one resets the element (i, j) while the second one extracts an element identified by (i, j). They may be useful when dealing with matrices *inside procedures*.

20.3 ATENSOR: A REDUCE Program for Tensor Simplification

Simplification of tensor expression with taking into account multiterm linear identities, symmetry relations and renaming dummy indices. This problem is important for the calculation in the gravity theory, differential geometry, other fields where indexed objects arise.

The group algebra technique for permutation group is applied to construt a canonical subspace and the effective algorithm for the corresponding projection.

Authors: V. A. Ilyin and A. P. Kryukov

For more information, see [IK96]. Further documentation is available at https://reduce-algebra.sourceforge.io/extra-docs/atensor.pdf.

20.4 AVECTOR: A Vector Algebra and Calculus Package

This package provides REDUCE with the ability to perform vector algebra using the same notation as scalar algebra. The basic algebraic operations are supported, as are differentiation and integration of vectors with respect to scalar variables, cross product and dot product, component manipulation and application of scalar functions (e.g. cosine) to a vector to yield a vector result.

Author: David Harper

20.4.1 Introduction

This package ([Har89]) provides REDUCE with the ability to perform vector algebra using the same notation as scalar algebra. The basic algebraic operations are supported, as are differentiation and integration of vectors with respect to scalar variables, cross product and dot product, component manipulation and application of scalar functions, e.g., cosine) to a vector to yield a vector result.

A set of vector calculus operators are provided for use with any orthogonal curvilinear coordinate system. These operators are gradient, divergence, curl and del-squared (Laplacian). The Laplacian operator can take scalar or vector arguments.

Several important coordinate systems are pre-defined and can be invoked by name. It is also possible to create new coordinate systems by specifying the names of the coordinates and the values of the scale factors.

20.4.2 Vector declaration and initialisation

Any name may be declared to be a vector, provided that it has not previously been declared as a matrix or an array. To declare a list of names to be vectors use the `vec` command:

```
vec a,b,c;
```

declares the variables `a`, `b` and `c` to be vectors. If they have already been assigned (scalar) values, these will be lost.

When a vector is declared using the `vec` command, it does not have an initial value.

If a vector value is assigned to a scalar variable, then that variable will automatically be declared as a vector and the user will be notified that this has happened.

A vector may be initialised using the `avec` function which takes three scalar arguments and returns a vector made up from those scalars. For example

```
a := avec(a1, a2, a3);
```

sets the components of the vector `a` to `a1`, `a2` and `a3`.

20.4.3 Vector algebra

(In the examples which follow, `v`, `v1`, `v2`, etc. are assumed to be vectors while `s`, `s1`, `s2`, etc. are scalars.)

The scalar algebra operators +,-,* and / may be used with vector operands according to the rules of vector algebra. Thus multiplication and division of a vector by a scalar are both allowed, but it is an error to multiply or divide one vector by another.

`v := v1 + v2 - v3;`	Addition and subtraction
`v := s1*3*v1;`	Scalar multiplication
`v := v1/s;`	Scalar division
`v := -v1;`	Negation

Vector multiplication is carried out using the infix operators `dot` and `cross`. These are defined to have higher precedence than scalar multiplication and division.

`v := v1 cross v2;`	Cross product
`s := v1 dot v2;`	Dot product
`v := v1 cross v2 + v3;`	
`v := (v1 cross v2) + v3;`	

The last two expressions are equivalent due to the precedence of the `cross` operator.

The modulus of a vector may be calculated using the `VMOD` operator.

```
s := vmod v;
```

A unit vector may be generated from any vector using the `vmod` operator.

```
v1 := v/(vmod v);
```

Components may be extracted from any vector using index notation in the same way as an array.

`v := avec(ax, ay, az);`	
`v(0);`	yields ax
`v(1);`	yields ay
`v(2);`	yields az

It is also possible to set values of individual components. Following from above:

```
v(1) := b;
```

The vector v now has components ax, b, az.

Vectors may be used as arguments in the differentiation and integration routines in place of the dependent expression.

```
v := avec(x**2, sin(x), y);
df(v,x);                          yields (2*x, cos(x), 0)
int(v,x);                         yields (x**3/3, -cos(x), y*x)
```

Vectors may be given as arguments to monomial functions such as sin, log and tan. The result is a vector obtained by applying the function component-wise to the argument vector.

```
v := avec(a1, a2, a3);
sin(v);                    yields (sin(a1), sin(a2), sin(a3))
```

20.4.4 Vector calculus

The vector calculus operators div, grad and curl are recognised. The Laplacian operator is also available and may be applied to scalar and vector arguments.

```
v := grad s;      Gradient of a scalar field
s := div v;       Divergence of a vector field
v := curl v1;     Curl of a vector field
s := delsq s1;    Laplacian of a scalar field
v := delsq v1;    Laplacian of a vector field
```

These operators may be used in any orthogonal curvilinear coordinate system. The user may alter the names of the coordinates and the values of the scale factors. Initially the coordinates are x, y and z and the scale factors are all unity.

There are two special vectors : coords contains the names of the coordinates in the current system and hfactors contains the values of the scale factors.

The coordinate names may be changed using the coordinates command.

```
coordinates r,theta,phi;
```

This command changes the coordinate names to r, theta and phi.

The scale factors may be altered using the scalefactors operator.

```
scalefactors(1,r,r*sin(theta));
```

This command changes the scale factors to 1, r and r sin(theta).

Note that the arguments of scalefactors must be enclosed in parentheses.

This is not necessary with the `coordinates` command.

When vector differential operators are applied to an expression, the current set of coordinates are used as the independent variables and the scale factors are employed in the calculation. (See, for example, Batchelor G.K. 'An Introduction to Fluid Mechanics', Appendix 2.)

Several coordinate systems are pre-defined and may be invoked by name. To see a list of valid names enter

```
symbolic !*csystems;
```

and REDUCE will respond with something like

```
(cartesian spherical cylindrical)
```

To choose a coordinate system by name, use the command `getcsystem`.

To choose the Cartesian coordinate system :

```
getcsystem 'cartesian;
```

Note the quote which prefixes the name of the coordinate system. This is required because `getcsystem` (and its complement `putcsystem`) is a `symbolic` procedure which requires a literal argument.

REDUCE responds by typing a list of the coordinate names in that coordinate system. The example above would produce the response

```
(x y z)
```

whilst

```
getcsystem 'spherical;
```

would produce

```
(r theta phi)
```

Note that any attempt to invoke a coordinate system is subject to the same restrictions as the implied calls to `coordinates` and `SCALEFACTORS`. In particular, `getcsystem` fails if any of the coordinate names has been assigned a value and the previous coordinate system remains in effect.

A user-defined coordinate system can be assigned a name using the command `putcsystem`. It may then be re-invoked at a later stage using `getcsystem`.

Example 1

We define a general coordinate system with coordinate names X,Y,Z and scale factors h1,h2,h3 :

```
coordinates x,y,z;
scalefactors(h1,h2,h3);
putcsystem 'general;
```

This system may later be invoked by entering

```
getcsystem 'general;
```

20.4.5 Volume and Line Integration

Several functions are provided to perform volume and line integrals. These operate in any orthogonal curvilinear coordinate system and make use of the scale factors described in the previous section.

Definite integrals of scalar and vector expressions may be calculated using the `defint` function.

Example 2

To calculate the definite integral of $\sin(x)^2$ between 0 and 2π we enter

```
defint(sin(x)**2,x,0,2*pi);
```

This function is a simple extension of the `int` operator taking two extra arguments, the lower and upper bounds of integration respectively.

Definite volume integrals may be calculated using the `volintegral` function whose syntax is as follows :

`volintegral(`⟨*integrand:expression*⟩,
⟨*lower-bound:vector*⟩,⟨*upper-bound:vector*⟩)

Example 3

In spherical polar coordinates we may calculate the volume of a sphere by integrating unity over the range r=0 to `rr`, θ=0 to π, ϕ=0 to 2*π as follows :

```
vlb := avec(0,0,0);          Lower bound
vub := avec(rr,pi,2*pi);     Upper bound in r, θ, φ respectively
volintorder := (0,1,2);      The order of integration
volintegral(1,vlb,vub);
```

Note the use of the special vector `volintorder` which controls the order in which the integrations are carried out. This vector should be set to contain the

number 0, 1 and 2 in the required order. The first component of `volintorder` contains the index of the first integration variable, the second component is the index of the second integration variable and the third component is the index of the third integration variable.

Example 4

Suppose we wish to calculate the volume of a right circular cone. This is equivalent to integrating unity over a conical region with the bounds:

z = 0 to h	(h = the height of the cone)
r = 0 to p z	(p = ratio of base diameter to height)
phi = 0 to 2*pi	

We evaluate the volume by integrating a series of infinitesimally thin circular disks of constant z-value. The integration is thus performed in the order : d(ϕ) from 0 to 2π, dr from 0 to p*Z, dz from 0 to H. The order of the indices is thus 2, 0, 1.

```
volintorder := avec(2,0,1);
vlb := avec(0,0,0);
vub := avec(p*z,h,2*pi);
volintegral(1,vlb,vub);
```

(At this stage, we replace `p*h` by `rr`, the base radius of the cone, to obtain the result in its more familiar form.)

Line integrals may be calculated using the `lineint` and `deflineint` operators. Their general syntax is

`lineint`(⟨*vector-function*⟩,⟨*vector-curve*⟩,⟨*variable:kernel*⟩)
`deflineint`(⟨*vector-function*⟩,⟨*vector-curve*⟩,⟨*variable:kernel*⟩,
⟨*lower-bound*⟩,⟨*upper-bound*⟩)

where

⟨***vector-function***⟩ is any vector-valued expression;

⟨***vector-curve***⟩ is a vector expression which describes the path of integration in terms of the independent variable;

⟨***variable***⟩ is the independent variable;

⟨***lower-bound***⟩

⟨***upper-bound***⟩ are the bounds of integration in terms of the independent variable.

Example 5

In spherical polar coordinates, we may integrate round a line of constant theta ('latitude') to find the length of such a line. The vector function is thus the tangent to the 'line of latitude', (0,0,1) and the path is `(0,lat,phi)` where `phi` is the independent variable. We show how to obtain the definite integral, i.e. from $\phi = 0$ to 2π :

```
deflineint(avec(0,0,1),avec(0,lat,phi),phi,0,2*pi);
```

20.4.6 Defining new functions and procedures

Most of the procedures in this package are defined in symbolic mode and are invoked by the REDUCE expression-evaluator when a vector expression is encountered. It is not generally possible to define procedures which accept or return vector values in algebraic mode. This is a consequence of the way in which the REDUCE interpreter operates and it affects other non-scalar data types as well : arrays cannot be passed as algebraic procedure arguments, for example.

20.4.7 Acknowledgements

This package was written whilst the author was the U.K. Computer Algebra Support Officer at the University of Liverpool Computer Laboratory.

20.5 BIBASIS: A Package for Calculating Boolean Involutive Bases

Authors: Yuri A. Blinkov and Mikhail V. Zinin

20.5.1 Introduction

Involutive polynomial bases are redundant Gröbner bases of special structure with some additional useful features in comparison with reduced Gröbner bases [GB98]. Apart from numerous applications of involutive bases [Sei10] the involutive algorithms [Ger05] provide an efficient method for computing reduced Gröbner bases. A reduced Gröbner basis is a well-determined subset of an involutive basis and can be easily extracted from the latter without any extra reductions. All this takes place not only in rings of commutative polynomials but also in Boolean rings.

Boolean Gröbner basis already have already revealed their value and usability in practice. The first impressive demonstration of practicability of Boolean Gröbner bases was breaking the first HFE (Hidden Fields Equations) challenge in the public key cryptography done in [FJ03] by computing a Boolean Gröbner basis for the system of quadratic polynomials in 80 variables. Since that time the Boolean Gröbner bases application area has widen drastically and nowadays there is also a number of quite successful examples of using Gröbner bases for solving SAT problems.

During our research we had developed [GZ08b, GZ08a, GZB10] Boolean involutive algorithms based on Janet and Pommaret divisions and applied them to computation of Boolean Gröbner bases. Our implementation of both divisions has experimentally demonstrated computational superiority of the Pommaret division implementation. This package BIBASIS is the result of our thorough research in the field of Boolean Gröbner bases. BIBASIS implements the involutive algorithm based on Pommaret division in a multivariate Boolean ring.

In section 2 the Boolean ring and its peculiarities are shortly introduced. In section 3 we briefly argue why the involutive algorithm and Pommaret division are good for Boolean ring while the Buhberger's algorithm is not. And finally in section 4 we give the full description of BIBASIS package capabilities and illustrate it by examples.

20.5.2 Boolean Ring

Boolean ring perfectly goes with its name, it is a ring of *Boolean functions* of n variables, i.e mappings from $\{0,1\}^n$ to $\{0,1\}^n$. Considering these variables are $\mathbf{X} := \{x_1, \ldots, x_n\}$ and $\mathbb{F}_2$ is the finite field of two elements $\{0,1\}$, Boolean ring

can be regarded as the quotient ring

$$\mathbb{B}\left[\mathbf{X}\right] := \mathbb{F}_2[\mathbf{X}] \,/ < x_1^2 + x_1, \ldots, x_n^2 + x_n > .$$

Multiplication in $\mathbb{B}\left[\mathbf{X}\right]$ is *idempotent* and addition is *nilpotent*

$$\forall\, b \in \mathbb{B}\left[\mathbf{X}\right] \;:\; b^2 = b\,,\; b + b = 0.$$

Elements in $\mathbb{B}\left[\mathbf{X}\right]$ are *Boolean polynomials* and can be represented as finite sums

$$\sum_{j} \prod_{x \in\, \Omega_j \subseteq \mathbf{X}} x$$

of *Boolean monomials*. Each monomial is a conjunction. If set Ω is empty, then the corresponding monomial is the unity Boolean function 1. The sum of zero monomials corresponds to zero polynomial, i.e. is zero Boolean function 0.

20.5.3 Pommaret Involutive Algorithm

Detailed description of involutive algorithm can found in [Ger05]. Here we note that result of both involutive and Buhberger's algorithms depend on chosen monomial ordering. At that the ordering must be admissible, i.e.

$$m \neq 1 \Longleftrightarrow m \succ 1, \qquad m_1 \succ m_2 \Longleftrightarrow m_1 m \succ m_2 m \qquad \forall\, m, m_1, m_2.$$

But as one can easily check the second condition of admissibility does not hold for any monomial ordering in Boolean ring:

$$x_1 \succ x_2 \quad \xrightarrow{*x_1} \quad x_1 * x_1 \succ x_2 * x_2 \quad \longrightarrow \quad x_1 \prec x_1 x_2$$

Though $\mathbb{B}\left[\mathbf{X}\right]$ is a principal ideal ring, boolean singleton $\{p\}$ is not necessarily a Gröbner basis of ideal $< p >$, for example:

$$x_1, x_2 \in < x_1 x_2 + x_1 + x_2 > \subset \mathbb{B}\left[x_1, x_2\right].$$

That the reason why one cannot apply the Buhberger's algorithm directly in a Boolean ring, using instead a ring $\mathbb{F}_2[\mathbf{X}]$ and *the field binomials* $x_1^2 + x_1, \ldots, x_n^2 + x_n$.

The involutive algorithm based on Janet division has the same disadvantage unlike the Pommaret division algorithm as shown in [GZ08b]. The Pommaret division algorithm can be applied directly in a Boolean ring and admits effective data structures for monomial representation.

20.5.4 BIBASIS Package

The package BIBASIS implements the Pommaret division algorithm in a Boolean ring. The first step to using the package is to load it:

```
1: load_package bibasis;
```

The current version of the BIBASIS user interface consists only of 2 functions: `bibasis` and `bibasis_print_statistics`.

The `bibasis` is the function that performs all the computation and has the following syntax:

`bibasis(` ⟨*initial_polynomial_list*⟩,⟨*variables_list*⟩,
⟨*monomial_ordering*⟩,⟨*reduce_to_groebner*⟩);

Input:

- ⟨*initial_polynomial_list*⟩ is the list of polynomials containing the known basis of initial Boolean ideal. All given polynomials are treated modulo 2. See Example 1.
- ⟨*variables_list*⟩ is the list of independent variables in decreasing order.
- ⟨*monomial_ordering*⟩ is a chosen monomial ordering and the supported ones are:

  `lex` – pure lexicographical ordering;

  `deglex` – degree lexicographic ordering;

  `degrevlex` – degree reverse lexicographic.

  See Examples 2–4 to check that Gröbner (as well as involutive) basis depends on monomial ordering.
- ⟨*reduce_to_groebner*⟩ is a Boolean value, if it is `t` the output is the reduced Boolean Gröbner basis, if `nil`, then the reduced Boolean Pommaret basis. Examples 5,6 show distinctions between these two outputs.

Output:

- The list of polynomials which constitute the reduced Boolean Gröbner or Pommaret basis.

The syntax of `bibasis_print_statistics` is simple:

```
bibasis_print_statistics();
```

This function prints out a brief statistics for the last invocation of `bibasis` function. See Example 7.

20.5.5 Examples

Example 1:

```
1: load_package bibasis;
2: bibasis({x+2*y}, {x,y}, lex, t);
{x}
```

Example 2:

```
1: load_package bibasis;
2: variables :={x0,x1,x2,x3,x4}$
3: polynomials := {x0*x3+x1*x2,x2*x4+x0}$
4: bibasis(polynomials, variables, lex, t);
{x0 + x2*x4,x2*(x1 + x3*x4)}
```

Example 3:

```
1: load_package bibasis;
2: variables :={x0,x1,x2,x3,x4}$
3: polynomials := {x0*x3+x1*x2,x2*x4+x0}$
4: bibasis(polynomials, variables, deglex, t);
{x1*x2*(x3 + 1),
 x1*(x0 + x2),
 x0*(x2 + 1),
 x0*x3 + x1*x2,
 x0*(x4 + 1),
 x2*x4 + x0}
```

Example 4:

```
1: load_package bibasis;
2: variables :={x0,x1,x2,x3,x4}$
3: polynomials := {x0*x3+x1*x2,x2*x4+x0}$
4: bibasis(polynomials, variables, degrevlex, t);
{x0*(x1 + x3),
 x0*(x2 + 1),
 x1*x2 + x0*x3,
 x0*(x4 + 1),
 x2*x4 + x0}
```

Example 5:

```
1: load_package bibasis;
2: variables :={x,y,z}$
3: polinomials := {x, z}$
4: bibasis(polinomials, variables, degrevlex, t);
{x,z}
```

Example 6:

```
1: load_package bibasis;
2: variables :={x,y,z}$
3: polinomials := {x, z}$
4: bibasis(polinomials, variables, degrevlex, nil);
{x,z,y*z}
```

Example 7:

```
1: load_package bibasis;
2: variables :={u0,u1,u2,u3,u4,u5,u6,u7,u8,u9}$
3: polinomials := {u0*u1+u1*u2+u1+u2*u3+u3*u4+u4*u5
3:                  +u5*u6+u6*u7+u7*u8+u8*u9,
3:                 u0*u2+u1+u1*u3+u2*u4+u2+u3*u5
3:                  +u4*u6+u5*u7+u6*u8+u7*u9,
3:                 u0*u3+u1*u2+u1*u4+u2*u5+u3*u6
3:                  +u3+u4*u7+u5*u8+u6*u9,
3:                 u0*u4+u1*u3+u1*u5+u2+u2*u6+u3*u7
3:                  +u4*u8+u4+u5*u9,
3:                 u0*u5+u1*u4+u1*u6+u2*u3+u2*u7
3:                  +u3*u8+u4*u9+u5,
3:                 u0*u6+u1*u5+u1*u7+u2*u4+u2*u8
3:                  +u3+u3*u9+u6,
3:                 u0*u7+u1*u6+u1*u8+u2*u5+u2*u9
3:                  +u3*u4+u7,
3:                 u0*u8+u1*u7+u1*u9+u2*u6+u3*u5+u4+u8,
3:                 u0+u1+u2+u3+u4+u5+u6+u7+u8+u9+1}$
4: bibasis(polinomials, variables, degrevlex, t);
{u3*u6,
 u3*u7,
 u7*(u6 + 1),
 u3*u8,
 u6*u8 + u6 + u7,
 u7*u8,
 u3*(u9 + 1),
```

```
 u6*u9 + u7,
 u7*(u9 + 1),
 u8*u9 + u6 + u7 + u8,
 u0 + u3 + u6 + u9 + 1,
 u1 + u7,
 u2 + u7 + u8,
 u4 + u6 + u8,
 u5 + u6 + u7 + u8}
5: bibasis_print_statistics();
        Variables order = u0 > u1 > u2 > u3 > u4
 > u5 > u6 > u7 > u8 > u9
Normal forms calculated = 216
  Non-zero normal forms = 85
        Reductions made = 4488
Time: 270 ms
GC time: 0 ms
```

20.6 BOOLEAN: A Package for Boolean Algebra

This package supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions connected by the infix boolean operators **and**, **or**, **implies**, **equiv**, and the unary prefix operator **not**. **Boolean** allows you to simplify expressions built from these operators, and to test properties like equivalence, subset property etc.

Author: Herbert Melenk

20.6.1 Introduction

The package BOOLEAN supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions ("atomic parts", "leafs") connected by the infix boolean operators `and`, `or`, `implies`, `equiv`, and the unary prefix operator `not`. BOOLEAN allows you to simplify expressions built from these operators, and to test properties like equivalence, subset property etc. Also the reduction of a boolean expression by a partial evaluation and combination of its atomic parts is supported.

20.6.2 Entering boolean expressions

In order to distinguish boolean data expressions from boolean expressions in the REDUCE programming language (e.g. in an `if` statement), each expression must be tagged explicitly by an operator `boolean`. Otherwise the boolean operators are not accepted in the REDUCE algebraic mode input. The first argument of `boolean` can be any boolean expression, which may contain references to other boolean values.

```
boolean (a and b or c);
q := boolean(a and b implies c);
boolean(q or not c);
```

Brackets are used to override the operator precedence as usual. The leafs or atoms of a boolean expression are those parts which do not contain a leading boolean operator. These are considered as constants during the boolean evaluation. There are two pre-defined values:

- **true**, **t** or **1**
- **false**, **nil** or **0**

These represent the boolean constants. In a result form they are used only as **1** and **0**.

By default, a **boolean** expression is converted to a disjunctive normal form, that is a form where terms are connected by **or** on the top level and each term is set of leaf expressions, eventually preceded by **not** and connected by **and**. An operators **or** or **and** is omitted if it would have only one single operand. The result of the transformation is again an expression with leading operator **boolean** such that the boolean expressions remain separated from other algebraic data. Only the boolean constants **0** and **1** are returned untagged.

On output, the operators **and** and **or** are represented as `/\` and `\/`, respectively.

```
boolean(true and false);
    ->   0
boolean(a or not(b and c));
    -> boolean(not(b) \/ not(c) \/ a)
boolean(a equiv not c);
    -> boolean(not(a)/\c \/ a/\not(c))
```

20.6.3 Normal forms

The **disjunctive** normal form is used by default. It represents the "natural" view and allows us to represent any form free or parentheses. Alternatively a **conjunctive** normal form can be selected as simplification target, which is a form with leading operator **and**. To produce that form add the keyword **and** as an additional argument to a call of **boolean**.

```
boolean (a or b implies c);
                      ->
     boolean(not(a)/\not(b) \/ c)

boolean (a or b implies c, and);
                      ->
     boolean((not(a) \/ c)/\(not(b) \/ c))
```

Usually the result is a fully reduced disjunctive or conjuntive normal form, where all redundant elements have been eliminated following the rules

$a \wedge b \vee \neg a \wedge b \longleftrightarrow b$

$a \vee b \wedge \neg a \vee b \longleftrightarrow b$

Internally the full normal forms are computed as intermediate result; in these forms each term contains all leaf expressions, each one exactly once. This unreduced form is returned when you set the additional keyword **full**:

```
boolean (a or b implies c, full);
```

```
                    ->
boolean(a/\b/\c \/ a/\not(b)/\c \/ not(a)/\b/\c

          \/ not(a)/\not(b)/\c

          \/ not(a)/\not(b)/\not(c))
```

The keywords **full** and **and** may be combined.

20.6.4 Evaluation of a boolean expression

If the leafs of the boolean expression are algebraic expressions which may evaluate to logical values because the environment has changed (e.g. variables have been bound), you can re–investigate the expression using the operator `testbool` with the boolean expression as argument. This operator tries to evaluate all leaf expressions in REDUCE boolean style. As many terms as possible are replaced by their boolean values; the others remain unchanged. The resulting expression is contracted to a minimal form. The result **1** (= true) or **0** (=false) signals that the complete expression could be evaluated.

In the following example the leafs are built as numeric greater test. For using $>$ in the expressions the greater sign must be declared operator first. The error messages are meaningless.

```
operator >;
fm:=boolean(x>v or not (u>v));
         ->
     fm := boolean(not(u>v) \/ x>v)

v:=10$

testbool fm;

   ***** u - 10 invalid as number
   ***** x - 10 invalid as number

         ->
   boolean(not(u>10) \/ x>10)

x:=3$
testbool fm;

   ***** u - 10 invalid as number
```

```
        ->
   boolean(not(u>10))

x:=17$

testbool fm;

   ***** u - 10 invalid as number

        ->
    1
```

20.7 CALI: A Package for Computational Commutative Algebra

Author: Hans-Gert Gräbe

Key words: affine and projective monomial curves, affine and projective sets of points, analytic spread, associated graded ring, blowup, border bases, constructive commutative algebra, dual bases, elimination, equidimensional part, extended Gröbner factorizer, free resolution, Gröbner algorithms for ideals and module, Gröbner factorizer, ideal and module operations, independent sets, intersections, lazy standard bases, local free resolutions, local standard bases, minimal generators, minors, normal forms, pfaffians, polynomial maps, primary decomposition, quotients, symbolic powers, symmetric algebra, triangular systems, weighted Hilbert series, primality test, radical, unmixed radical.

20.7.1 Introduction

This package contains algorithms for computations in commutative algebra closely related to the Gröbner algorithm for ideals and modules. Its heart is a new implementation of the Gröbner algorithm[1] that allows the computation of syzygies, too. This implementation is also applicable to submodules of free modules with generators represented as rows of a matrix.

Moreover CALI contains facilities for local computations, using a modern implementation of Mora's standard basis algorithm, see [MPT89] and [Grä94b], that works for arbitrary term orders. The full analogy between modules over the local ring $k[x_v : v \in H]_{\mathbf{m}}$ and homogeneous (in fact H-local) modules over $k[x_v : v \in H]$ is reflected through the switch . Turn it on (Gröbner basis, the default) or off (local standard basis) to choose appropriate algorithms automatically. In v. 2.2 we present an unified approach to both cases, using reduction with bounded ecart for non Noetherian term orders, see [Grä95a] for details. This allows to have a common driver for the Gröbner algorithm in both cases.

CALI extends also the restricted term order facilities of the GROEBNER package, defining term orders by degree vector lists, and the rigid implementation of the sugar idea, by a more flexible *ecart* vector, in particular useful for local computations, see [Grä94b].

The package was designed mainly as a symbolic mode programming environment extending the build-in facilities of REDUCE for the computational approach to problems arising naturally in commutative algebra. An algebraic mode interface accesses (in a more rigid frame) all important features implemented symbolically

[1] The data representation even for polynomials is different from that given in the GROEBNER package distributed with REDUCE (and rests on ideas in the DIPOLY package).

and thus should be favored for short sample computations.

On the other hand, tedious computations are strongly recommended to be done symbolically since this allows considerably more flexibility and avoids unnecessary translations of intermediate results from CALI's internal data representation to the algebraic mode and vice versa. Moreover, one can easily extend the package with new symbolic mode scripts, or do more difficult interactive computations. For all these purposes the symbolic mode interface offers substantially more facilities than the algebraic one.

For a detailed description of special symbolic mode procedures one should consult the source code and the comments therein. In this manual we can give only a brief description of the main ideas incorporated into the package CALI. We concentrate on the data structure design and the description of the more advanced algorithms. For sample computations from several fields of commutative algebra the reader may consult also the *cali.tst* file.

As main topics CALI contains facilities for

- defining rings, ideals and modules,
- computing Gröbner bases and local standard bases,
- computing syzygies, resolutions and (graded) Betti numbers,
- computing (now also weighted) Hilbert series, multiplicities, independent sets, and dimensions,
- computing normal forms and representations,
- computing sums, products, intersections, quotients, stable quotients, elimination ideals etc.,
- primality tests, computation of radicals, unmixed radicals, equidimensional parts, primary decompositions etc. of ideals and modules,
- advanced applications of Gröbner bases (blowup, associated graded ring, analytic spread, symmetric algebra, monomial curves etc.),
- applications of linear algebra techniques to zero dimensional ideals, as e.g. the FGLM change of term orders, border bases and affine and projective ideals of sets of points,
- splitting polynomial systems of equations mixing factorization and the Gröbner algorithm, triangular systems, and different versions of the extended Gröbner factorizer.

Below we will use freely without further explanation the notions common for text books and papers about constructive commutative algebra, assuming the reader to be familiar with the corresponding ideas and concepts. For further references see e.g. the text books [BWK93], [CLO92] and [Mis93] or the survey papers [Buc85], [Buc88] and [Rob89].

CALI should be loaded via

```
load_package cali;
```

Upon successful loading CALI responds with a message containing the version number and the last update of the distribution.

Feel free to contact me by email if You have problems to get CALI started. Also comments, hints, bug reports etc. are welcome.

CALI's Language Concept

From a certain point of view one of the major disadvantage of the current RLISP (and the underlying Lisp) language is the fact that it supports modularity and data encapsulation only in a rudimentary way. Since all parts of code loaded into a session are visible all the time, name conflicts between different packages may occur, will occur (even not issuing a warning message), and are hard to prevent, since packages are developed (and are still developing) by different research groups at different places and different time.

A (yet rudimentary) concept of REDUCE packages and modules indicates the direction into what the REDUCE designers are looking for a solution for this general problem.

CALI (2.0 and higher) follows a name concept for internal procedures to mimick data encapsulation at a semantical level. We hope this way on the one hand to resolve the conflicts described above at least for the internal part of CALI and on the other hand to anticipate a desirable future and already foregoing development of REDUCE towards a true modularity.

The package CALI is divided into several modules, each of them introducing either a single new data type together with basic facilities, constructors, and selectors or a collection of algorithms subject to a common problem. Each module contains *internal procedures*, conceptually hidden by this module, *local procedures*, designed for a CALI wide use, and *global procedures*, exported by CALI into the general (algebraic or symbolic) environment of REDUCE. A header *module cali* contains all (fluid) global variables and switches defined by the package CALI.

Along these lines the CALI procedures available in symbolic mode are divided into

three types with the following naming convention:

`module!=procedure`
: internal to the given module.

`module_procedure`
: exported by the given module into the local CALI environment.

`procedure!*`
: a global procedure usually having a semantically equivalent procedure (possibly with another parameter list) without trailing asterisk in algebraic mode.

There are also symbolic mode equivalents without trailing asterisk, if the algebraic procedure is not a *psopfn*, but a *symbolic operator*. They transfer data to CALI's internal structure and call the corresponding procedure with trailing asterisk. CALI 2.2 distinguishes between algebraic and symbolic calls of such a procedure. In symbolic mode such a procedure calls the corresponding procedure with trailing asterisk directly without data transfer.

CALI 2.2 follows also a more concise concept for global variables. There are three types of them:

True *fluid* global variables,
: that are part of the current data structure, as e.g. the current base ring and the degree vector. They are often locally rebound to be restored after interrupts.

Global variables, stored on the property list of the package name `cali`,
: that reflect the state of the computational model as e.g. the trace level, the output print level or the chosen version of the Gröbner basis algorithm. There are several such parameters in the module *dualbases* to serve the common dual basis driver with information for different applications.

Switches,
: that allow to choose different branches of algorithms. Note that this concept interferes with the second one. Different *versions* of algorithms, that apply different functions in a common driver, are *not* implemented through switches.

20.7.2 The Computational Model

This section gives a short introduction into the data type design of CALI at different levels. First (§1 and 2) we describe CALI's way of algorithmic translation of the abstract algebraic objects *ring of polynomials, ideal* and (finitely generated) *module*. Then (§3 and 4) we describe the algebraic mode interface of CALI and the switches and global variables to drive a session. In the next chapter we give a

more detailed overview of the basic (symbolic mode) data structures involved with CALI. We refer to the appendix for a short summary of the commands available in algebraic mode.

The Base Ring

A polynomial ring consists in CALI of the following data:

a list of variable names: All variables not occuring in the list of ring names are treated as parameters. Computations are executed denominatorfree, but the results are valid only over the corresponding parameter *field* extension.

a term order and a term order tag: They describe the way in which the terms in each polynomial (and polynomial vector) are ordered.

an ecart vector: A list of positive integers corresponding to the variable names.

A *base ring* may be defined (in algebraic mode) through the command

`setring` ⟨*ring*⟩

with ⟨*ring*⟩ ::= { vars, tord, tag [, ecart] } resp.

```
setring(vars, tord, tag [,ecart])
```

This sets the global (symbolic) variable `cali!=basering`. Here `vars` is the list of variable names, `tord` a (possibly empty) list of weight lists, the *degree vectors*, and `tag` the tag LEX or REVLEX. Optionally one can supply `ecart`, a list of positive integers of the same length as `vars`, to set an ecart vector different from the default one (see below).

The degree vectors must have the same length as `vars`. If $(w_1 \ \ldots \ w_k)$ is the list of degree vectors then

$$\begin{array}{lcll} x^a < x^b & :\Leftrightarrow & \text{either} & w_j(x^a) = w_j(x^b) \quad \text{for } j < i \quad \text{and} \\ & & & w_i(x^a) < w_i(x^b) \\ & & \text{or} & w_j(x^a) = w_j(x^b) \quad \text{for all } j \quad \text{and} \\ & & & x^a <_{lex} x^b \text{ resp. } x^a <_{revlex} x^b \end{array}$$

Here $<_{lex}$ resp. $<_{revlex}$ denote the *lexicographic* (tag=LEX) resp. *reverse lexicographic* (tag=REVLEX) term orders[2] with respect to the variable order given in

[2]The definition of the revlex term order changed for version 2.2.

`vars`, i.e.

$$x^a < x^b \quad :\Leftrightarrow \quad \exists\, j\, \forall\, i < j \;:\; a_i = b_i \quad \text{and} \quad a_j < b_j \text{ (lex.)}$$

or

$$x^a < x^b \quad :\Leftrightarrow \quad \exists\, j\, \forall\, i > j \;:\; a_i = b_i \quad \text{and} \quad a_j > b_j \text{ (revlex.)}$$

Every term order can be represented in such a way, see [MR88].

During the ring setting the term order will be checked to be Noetherian (i.e. to fulfill the descending chain condition) provided the switch is on (the default). The same applies turning *noetherian on*: If the term order of the underlying base ring isn't Noetherian the switch can't be turned over. Hence, starting from a non Noetherian term order, one should define *first* a new ring and *then* turn the switch on.

Useful term orders can be defined by the procedures

`degreeorder` ⟨*vars*⟩

that returns $tord = \{\{1, \ldots, 1\}\}$.

`localorder` ⟨*vars*⟩

that returns $tord = \{\{-1, \ldots, -1\}\}$ (a non Noetherian term order for computations in local rings).

`eliminationorder(` ⟨*vars*⟩,⟨*elimvars*⟩)

that returns a term order for elimination of the variables in ⟨*elimvars*⟩, a subset of all ⟨*vars*⟩. It's recommended to combine it with the tag revlex.

`blockorder(` ⟨*vars*⟩,⟨*vars,integerlist*⟩)

that returns the list of degree vectors for the block order with block lengths given in the ⟨*integerlist*⟩. Note that these numbers should sum up to the length of the variable list supplied as the first argument.

Examples:

```
vars:={x,y,z};
tord:=degreeorder vars;
       % Returns {{1,1,1}}.

setring(vars,tord,lex);
       % GRADLEX in the groebner package.

% or

setring({a,b,c,d},{},lex);
```

```
       % LEX in the groebner package.
% or

vars:={a,b,c,x,y,z};
tord:=eliminationorder(vars,{x,y,z});
tord:=reverse blockorder(vars,{3,3});
        % Return both {{0,0,0,1,1,1},{1,1,1,0,0,0}}.
setring(vars,tord,revlex);
```

The base ring is initialized with

```
{{t,x,y,z},{{1,1,1,1}},revlex,{1,1,1,1}}
```

i.e. $S = k[t,x,y,z]$ supplied with the degreewise reverse lexicographic term order.

`getring` $\langle m \rangle$

returns the ring attached to the object with the identifier $\langle m \rangle$. E.g.,

`setring getring` $\langle m \rangle$

(re)sets the base ring to the base ring of the formerly defined object (ideal or module) $\langle m \rangle$.

`getring()`

returns the currently active base ring.

CALI defines also an *ecart vector*, attaching to each variable a positive weight with respect to that homogenizations and related algorithms are executed. It may be set optionally by the user during the *setring* command. (Default: If the term order is a (positive) degree order then the ecart is the first degree vector, otherwise each ecart equals 1).

The ecart vector is used in several places for efficiency reason (Gröbner basis computation with the sugar strategy) or for termination (local standard bases). If the input is homogeneous the ecart vector should reflect this homogeneity rather than the first degree vector to obtain the best possible performance. For a discussion of local computations with encoupled ecart vector see [Grä94b]. In general the ecart vector is recommended to be chosen in such a way that the input examples become close to be homogeneous. *Homogenizations* and *Hilbert series* are computed with respect to this ecart vector. `getecart()` returns the ecart vector currently set.

Ideals and Modules

If $S = k[x_v,\ v \in H]$ is a polynomial ring, a matrix M of size $r \times c$ defines a map

$$f\ :\ S^r \longrightarrow S^c$$

by the following rule

$$f(v) := v \cdot M \qquad \text{for } v \in S^r.$$

There are two modules, connected with such a map, $im\ f$, the submodule of S^c generated by the rows of M, and $\operatorname{coker} f\ (= S^c/im\ f)$. Conceptually we will identify M with $im\ f$ for the basic algebra, and with $\operatorname{coker} f$ for more advanced topics of commutative algebra (Hilbert series, dimension, resolution etc.) following widely accepted conventions.

With respect to a fixed basis $\{e_1, \ldots, e_c\}$ one can define module term orders on S^c, Gröbner bases of submodules of S^c etc. They generalize the corresponding notions for ideal bases. See [Eis95] or [MM86] for a detailed introduction to this area of computational commutative algebra. This allows to define joint facilities for both ideals and submodules of free modules. Moreover computing syzygies the latter come in in a natural way.

CALI handles ideal and module bases in a unique way representing them as rows of a *dpmat* (**d**istributive **p**olynomial **mat**rix). It attaches to each unit vector e_i a monomial x^{a_i}, the i-th *column degree* and represents the rows of a dpmat M as lists of module terms $x^a e_i$, sorted with respect to a *module term order*, that may be roughly[3] described as

$$x^a e_i < x^b e_j \quad :\Leftrightarrow \quad \text{either} \quad x^a x^{a_i} < x^b x^{a_j} \text{ in } S$$

$$\text{or} \quad x^a x^{a_i} = x^b x^{a_j} \text{ and } i < j \text{ (lex.) resp. } i > j \text{ (revlex.)}$$

Every dpmat M has its own column degrees (no default !). They are managed through a global (symbolic) variable `cali!=degrees`.

`getdegrees` ⟨*m*⟩

returns the column degrees of the object with identifier ⟨*m*⟩.

`getdegrees`()

returns the current setting of `cali!=degrees`.

[3] The correct definition is even more difficult.

`setdegrees` ⟨*list of monomials*⟩

sets `cali!=degrees` correspondingly. Use this command before executing `setmodule` to give a dpmat prescribed column degrees since cali!=degrees has no default value and changes during computations. A good guess is to supply the empty list (i.e. all column degrees are equal to $\mathbf{x}^0$). Be careful defining modules without prescribed column degrees.

To distinguish between *ideals* and *modules* the former are represented as a *dpmat* with $c = 0$ (and hence without column degrees). If $I \subset S$ is such an ideal one has to distinguish between the ideal I (with $c = 0$, allowing special ideal operations as e.g. ideal multiplication) and the submodule I of the free one dimensional module S^1 (with $c = 1$, allowing matrix operations as e.g. transposition, matrix multiplication etc.). *ideal2mat* converts an (algebraic) list of polynomials into an (algebraic) matrix column whereas *mat2list* collects all matrix entries into a list.

The Algebraic Mode Interface

Corresponding to CALI's general philosophy explained in the introduction the algebraic mode interface translates algebraic input into CALI's internal data representation, calls the corresponding symbolic functions, and retranslates the result back into algebraic mode. Since Gröbner basis computations may be very tedious even on small examples, one should find a well balance between the storage of results computed earlier and the unavoidable time overhead and memory request associated with the management of these results.

Therefore CALI distinguishes between *free* and *bounded* identifiers. Free identifiers stand only for their value whereas to bounded identifiers several internal information is attached to their property list for later use.

After the initialization of the *base ring* bounded identifiers for ideals or modules should be declared via

`setmodule`(⟨*name*⟩,⟨*matrix value*⟩)

resp.

`setideal`(⟨*name*⟩,⟨*list of polynomials*⟩)

This way the corresponding internal representation (as *dpmat*) is attached to ⟨*name*⟩ as the property *basis*, the prefix form as its value and the current base ring as the property *ring*.

Performing any algebraic operation on objects defined this way their ring will be compared with the current base ring (including the term order). If they are different an error message occurs. If `m` is a valid name, after resetting the base ring

```
setmodule(m1,m)
```

reevaluates `m` with respect to the new base ring (since the *value* of `m` is its prefix form) and assigns the reordered dpmat to `m1` clearing all information previously computed for `m1` (`m1` and `m` may coincide).

All computations are performed with respect to the ring $S = k[x_v \in \texttt{vars}]$ over the field k. Nevertheless by efficiency reasons *base coefficients* are represented in a denominator free way as standard forms. Hence the computational properties of the base coefficient domain depend on the *dmode* and also on auxiliary variables, contained in the expressions, but not in the variable list. They are assumed to be parameters.

Best performance will be obtained with integer or modular domain modes, but one can also try *Algebraic numbers* as coefficients as e.g. generated by `sqrt` or the ARNUM package. To avoid an unnecessary slow-down connected with the management of simplified algebraic expressions there is a *switch hardzerotest* (default: off) that may be turned on to force an additional simplification of algebraic coefficients during each zero test. It should be turned on only for domain modes without canonical representations as e.g. mixtures of arnums and square roots. We remind the general zero decision problem for such domains.

Alternatively, CALI offers the possibility to define a set of algebraic substitution rules that will affect CALI's base coefficient arithmetic only.

`setrules` ⟨*rule list*⟩

transfers the (algebraic) ⟨*rule list*⟩ into the internal representation stored at the `cali` value `rules`.

In particular, `setrules {}` clears the rules previously set.

`getrules()`

returns the internal CALI rules list in algebraic form.

We recommend to use `setrules` for computations with algebraic numbers since they are better adapted to the data structure of CALI than the algebraic numbers provided by the ARNUM package. Note, that due to the zero decision problem complicated `setrules` based computations may produce wrong results if base coefficient's pseudo division is involved (as e.g. with `dp_pseudodivmod`). In this case we recommend to enlarge the variable set and add the defining equations of the algebraic numbers to the equations of the problem[4].

The standard domain (Integer) doesn't allow denominators for input. `setideal` clears automatically the common denominator of each input expression whereas a

[4] A *qring* facility for the computation over quotient rings will be incorporated into future versions.

polynomial matrix with true rational coefficients will be rejected by `setmodule`.

One can save/initialize ideal and module bases together with their accompanying data (base ring, degrees) to/from a file:

`savemat`(⟨*m*⟩,⟨*name*⟩)

resp.

`initmat` ⟨*name*⟩

execute the file transfer from/to disk files with the specified file ⟨*name*⟩. e.g.

```
savemat(m,"myfile");
```

saves the base ring and the ideal basis of `m` to the file "`myfile`" whereas

```
setideal(m,initmat "myfile");
```

sets the current base ring (via a call to `setring`) to the base ring of `m` saved at "`myfile`" and then recovers the basis of `m` from the same file.

Switches and Global Variables

There are several switches, (fluid) global variables, a trace facility, and global parameters on the property list of the package name `cali` to control CALI's computations.

Switches

`bcsimp` (Default:on)
: On: Cancel out gcd's of base coefficients.

`detectunits` (Default: off)
: On: replace polynomials of the form $\langle monomial \rangle * \langle polynomial\ unit \rangle$ by $\langle monomial \rangle$ during interreductions and standard basis computations.

  Affects only local computations.

`factorprimes` (Default: on)
: On: Invoke the Gröbner factorizer during computation of isolated primes. Note that REDUCE lacks a modular multivariate factorizer, hence for modular prime decomposition computations this switch has to be turned off.

factorunits (Default: off)
On: factor polynomials and remove polynomial unit factors during interreductions and standard basis computations.

Affects only local computations.

hardzerotest (Default: off)
On: try an additional algebraic simplification of base coefficients at each base coefficient's zero test. Useful only for advanced base coefficient domains without canonical REDUCE representation. May slow down the computation drastically.

lexefgb (Default: off) On: Use the pure lexicographic term order and *zerosolve* during reduction to dimension zero in the *extended Gröbner factorizer*. This is a single, but possibly hard task compared to the degrevlex invocation of *zerosolve1*. See [Grä95b] for a discussion of different zero dimensional solver strategies.

Noetherian (Default: on)
On: choose algorithms for Noetherian term orders.

Off: choose algorithms for local term orders.

red_total (Default: on)
On: compute total normal forms, i.e. apply reduction (Noetherian term orders) or reduction with bounded ecart (non Noetherian term orders to tail terms of polynomials, too.

Off: Do only top reduction.

Tracing

Different to v. 2.1 now intermediate output during the computations is controlled by the value of the `trace` and `printterms` entries on the property list of the package name `cali`. The former value controls the intensity of the intermediate output (Default: 0, no tracing), the latter the number of terms printed in such intermediate polynomials (Default: all).

`setcalitrace` ⟨*n*⟩

Changes the trace intensity. Set $n = 2$ for a sparse tracing (a dot for each reduction step). Other good suggestions are the values 30 or 40 for tracing the Gröbner algorithm or $n > 70$ for tracing the normal form algorithm. The higher n the more intermediate information will be given.

`setcaliprintterms` ⟨*n*⟩

Sets the number of terms that are printed in intermediate polynomials. Note that this does not affect the output of whole *dpmats*. The output of polynomials with more than n terms ($n > 0$) breaks off and continues with ellipses.

`clearcaliprintterms()`

Clears the `printterms` value forcing full intermediate output (according to the current trace level).

Global Variables

`cali!=basering` The currently active base ring initialized e.g. by `setring`.

`cali!=degrees` The currently active module component degrees initialized e.g. by `setdegrees`.

`cali!=monset` A list of variable names considered as non zero divisors during Gröbner basis computations initialized e.g. by `setmonset`. Useful e.g. for binomial ideals defining monomial varieties or other prime ideals.

Entries on the Property List of `cali`

This approach is new for v. 2.2. Information concerning the state of the computational model as e.g. trace intensity, base coefficient rules, or algorithm versions are stored as values on the property list of the identifier (package name) `cali`. This concerns

`trace` and `printterms`

see above.

`efgb`

Changed by the *switch lexefgb*.

`groeb!=rf`

Reduction function invoked during the Gröbner algorithm. It can be changed with *gbtestversion* $< n >$ ($n = 1, 2, 3$, default is 1).

`hf!=ff`

Variant for the computation of the Hilbert series numerator. It can be changed with *hftestversion* $< n >$ ($n = 1, 2$, default is 1).

`rules`

Algebraic "replaceby" rules introduced to CALI with the `setrules` command.

`evlf`, `varlessp`, `sublist`, `varnames`, `oldborderbasis`, `oldring`, `oldbasis`

see *module lf*, implementing the dual bases approach.

20.7.3 Basic Data Structures

In the following we describe the data structure layers underlying the dpmat representation in CALI and some important (symbolic) procedures to handle them. We refer to the source code and the comments therein for a more complete survey about the procedures available for different data types.

The Coefficient Domain

Base coefficients as implemented in the *module bcsf* are standard forms in the variables outside the variable list of the current ring. All computations are executed "denominator free" over the corresponding quotient field, i.e. gcd's are canceled out without request. To avoid this set the `bcsimp` off.[5] In the given implementation we use the s.f. procedure `qremf` for effective divisibility test. We had some trouble with it under `on factor`.

Additionally it is possible to supply the parameters occuring as base coefficients with a (global) set of algebraic rules.[6]

`setrules!*` $\langle r \rangle$

converts an algebraic mode rules list r as e.g. used in WHERE statements into the internal CALI format.

The Base Ring

The *base ring* is defined by its `name list`, the `degree matrix` (a list of lists of integers), the `ring tag` (LEX or REVLEX), and the `ecart`. The name list contains a phantom name `cali!=mk` for the module component at place 0.

The *module ring* exports among others the following selectors: `ring_names`, `ring_degrees`, `ring_tag`, `ring_ecart`, the symbolic mode test function `ring_isnoetherian` and the transfer procedures from/to an (appropriate, printable by `mathprint`) algebraic prefix form `ring_from_a` (including extensive tests of the supplied parameters for consistency) and `ring_2a`.

[5] This induces a rapid base coefficient's growth and doesn't yield **Z**-Gröbner bases in the sense of [GTZ88] since the S-pair criteria are different.

[6] This is different from the `let` rule mechanism since they must be present in symbolic mode. Hence for a simultaneous application of the same rules in algebraic mode outside CALI they must additionally be declared in the usual way.

The following procedures allow to define a base ring:

`ring_define`(⟨*name list*⟩, ⟨*degree matrix*⟩, ⟨*ring tag*⟩, ⟨*ecart*⟩

combines the given parameters to a ring.

`setring!*` ⟨*ring*⟩

sets `cali!=basering` and checks for consistency with the `Noetherian`. It also sets through `setkorder` the current variable list as main variables. It is strongly recommended to use `setring!* ...` instead of `cali!=basering := ...`.

The procedures `degreeorder!*`, `localorder!*`, `eliminationorder!*`, and `blockorder!*` define term order matrices in full analogy to algebraic mode.

There are three ring constructors for special purposes:

`ring_sum`(⟨a⟩,⟨b⟩)

returns a ring, that is constructed in the following way: Its variable list is the union of the (disjoint) lists of the variables of the rings ⟨a⟩ and ⟨b⟩ (in this order) whereas the degree list is the union of the (appropriately shifted) degree lists of ⟨b⟩ and ⟨a⟩ (in this order). The ring tag is that of ⟨a⟩. Hence it returns (essentially) the ring $b \bigoplus a$ if b has a degree part (e.g. useful for elimination problems, introducing "big" new variables) and the ring $a \bigoplus b$ if b has no degree part (introducing "small" new variables).

`ring_rlp`(⟨r⟩,⟨u⟩)

⟨u⟩ is a subset of the names of the ring ⟨r⟩. Returns the ring ⟨r⟩, but with a term order "first degrevlex on ⟨u⟩, then the order on ⟨r⟩".

`ring_lp`(⟨r⟩,⟨u⟩)

As `ring_rlp`, but with a term order "first lex on ⟨u⟩, then the order on ⟨r⟩".

Example:

```
vars:='(x y z)
setring!*
  ring_define(vars,degreeorder!* vars,'lex,'(1 1 1));
                   % GRADLEX in the groebner package.
```

Monomials

The current version uses a place-driven exponent representation closely related to a vector model. This model handles term orders on S and module term orders on S^c in a unique way. The zero component of the exponent list of a monomial contains its module component (> 0) or 0 (ring element). All computations are executed with respect to a *current ring* (`cali!=basering`) and *current (monomial) weights* of the free generators $e_i, i = 1, \ldots, c$, of S^c (`cali!=degrees`). For efficiency reasons every monomial has a precomputed degree part that should be reevaluated if `cali!=basering` (i.e. the term order) or `cali!=degrees` were changed. `cali!=degrees` contains the list of column degrees of the current module as an assoc. list and will be set automatically by (almost) all dpmat procedure calls. Since monomial operations use the degree list that was precomputed with respect to fixed column degrees (and base ring)

> **watch carefully for `cali!=degrees` programming at the monomial or dpoly level !**

As procedures there are selectors for the module component, the exponent and the degree parts, comparison procedures, procedures for the management of the module component and the degree vector, monomial arithmetic, transfer from/to prefix form, and more special tools.

Polynomials and Polynomial Vectors

CALI uses a distributive representation as a list of terms for both polynomials and polynomial vectors, where a *term* is a dotted pair

$$(\langle \textit{monomial} \rangle \ . \ \langle \textit{base coefficient} \rangle)$$

The *ecart* of a polynomial (vector) $f = \sum t_i$ with (module) terms t_i is defined as

$$\max(\mathrm{ec}(t_i)) - \mathrm{ec}(lt(t_i)),$$

see [Grä94b]. Here $\mathrm{ec}(t_i)$ denotes the ecart of the term t_i, i.e. the scalar product of the exponent vector of t_i (including the monomial weight of the module generator) with the ecart vector of the current base ring.

As procedures there are selectors, dpoly arithmetic including the management of the module component, procedures for reordering (and reevaluating) polynomials wrt. new term order degrees, for extracting common base coefficient or monomial factors, for transfer from/to prefix form and for homogenization and dehomogenization (wrt. the current ecart vector).

Two advanced procedures use ideal theory ingredients:

`dp_pseudodivmod`($\langle g \rangle$,$\langle f \rangle$)

returns a dpoly list $\{q, r, z\}$ such that $z \cdot g = q \cdot f + r$ and z is a dpoly unit (i.e. a scalar for Noetherian term orders). For non Noetherian term orders the necessary modifications are described in [Grä95a].

g, f and r belong to the same free module or ideal.

`dpgcd`($\langle a \rangle$,$\langle b \rangle$)

computes the gcd of two dpolys a and b by the syzygy method: The syzygy module of $\{a, b\}$ is generated by a single element $[-b_0 \quad a_0]$ with $a = ga_0, b = gb_0$, where g is the gcd of a and b. Since it uses dpoly pseudo-division it may work not properly with `setrules`.

Base Lists

Ideal bases are one of the main ingredients for dpmats. They are represented as lists of *base elements* and contain together with each dpoly entry the following information:

- a number (the row number of the polynomial vector in the corresponding dpmat).
- the dpoly, its ecart (as the main sort criterion), and length.
- a representation part, that may contain a representation of the given dpoly in terms of a certain fixed basis (default: empty).

The representation part is managed during normal form computations and other row arithmetic of dpmats appropriately with the following procedures:

`bas_setrelations` $\langle b \rangle$

sets the relation part of the base element i in the base list $\langle b \rangle$ to e_i.

`bas_removerelations` $\langle b \rangle$

removes all relations, i.e. replaces them with the zero polynomial vector.

`bas_getrelations` $\langle b \rangle$

gets the relation part of $\langle b \rangle$ as a separate base list.

Further there are procedures for selection and construction of base elements and for the manipulation of lists of base elements as e.g. sorting, renumbering, reordering, simplification, deleting zero base elements, transfer from/to prefix form, homogenization and dehomogenization.

Dpoly Matrices

Ideals and matrices, represented as *dpmat*s, are the central data type of the CALI package, as already explained above. Every dpmat m combines the following information:

- its size (`dpmat_rows` m,`dpmat_cols` m),
- its base list (`dpmat_list` m) and
- its column degrees as an assoc. list of monomials (`dpmat_coldegs` m). If this list is empty, all degrees are assumed to be equal to x^0.
- New in v. 2.2 there is a *gb-tag* (`dpmat_gbtag` m), indicating that the given base list is already a Gröbner basis (under the given term order).

The *module dpmat* contains selectors, constructors, and the algorithms for the basic management of this data structure as e.g. file transfer, transfer from/to algebraic prefix forms, reordering, simplification, extracting row degrees and leading terms, dpmat matrix arithmetic, homogenization and dehomogenization.

The modules *matop* and *quot* collect more advanced procedures for the algebraic management of dpmats.

Extending the REDUCE Matrix Package

In v. 2.2 minors, Jacobian matrix, and Pfaffians are available for general REDUCE matrices. They are collected in the *module calimat* and allow to define procedures in more generality, especially allowing variable exponents in polynomial expressions. Such a generalization is especially useful for the investigation of whole classes of examples that may be obtained from a generic one by specialization. In the following $\langle m \rangle$ is a matrix given in algebraic prefix form.

`matjac`($\langle m \rangle$,$\langle l \rangle$)

returns the Jacobian matrix of the ideal $\langle m \rangle$ (given as an algebraic mode list) with respect to the variable list $\langle l \rangle$.

`minors`($\langle m \rangle$,$\langle k \rangle$)

returns the matrix of k-minors of the matrix m.

`ideal_of_minors`($\langle m \rangle$,$\langle k \rangle$)

returns the ideal of the k-minors of the matrix m.

`pfaffian` $\langle m \rangle$

returns the pfaffian of a skewsymmetric matrix m.

`ideal_of_pfaffians`(⟨*m*⟩,⟨*k*⟩)

returns the ideal of the $2k$-pfaffians of the skewsymmetric matrix m.

`random_linear_form`(⟨*vars*⟩,⟨*bound*⟩)

returns a random linear form in algebraic prefix form in the supplied variables $vars$ with integer coefficients bounded by the supplied $bound$.

`singular_locus!*`(⟨*m*⟩,⟨*c*⟩)

returns the singular locus of m (as dpmat). m must be an ideal of codimension c given as a list of polynomials in prefix form. `singular_locus` computes the ideal generated by the corresponding Jacobian and m itself.

20.7.4 About the Algorithms Implemented in CALI

Below we give a short explanation of the main algorithmic ideas of CALI and the way they are implemented and may be accessed (symbolically).

Normal Form Algorithms

For v. 2.2 we completely revised the implementation of normal form algorithms due to the insight obtained from our investigations of normal form procedures for local term orders in [Grä95a] and [Grä94b]. It allows a common handling of Noetherian and non Noetherian term orders already on this level thus making superfluous the former duplication of reduction procedures in the modules *red* and *mora* as in v. 2.1.

Normal form algorithms reduce polynomials (or polynomial vectors) with respect to a given finite set of generators of an ideal or module. The result is not unique except for a total normal form with respect to a Gröbner basis. Furthermore different reduction strategies may yield significant differences in computing time.

CALI reduces by first matching, usually keeping base lists sorted with respect to the sort predicate *red_better*. In v. 2.2 we sort solely by the dpoly length, since the introduction of *red_TopRedBE*, i.e. reduction with bounded ecart, guarantees termination also for non Noetherian term orders. Overload red_better for other reduction strategies.

Reduction procedures produce for a given ideal basis $B \subset S$ and a polynomial $f \in S$ a (pseudo) normal form $h \in S$ such that $h \equiv u \cdot f \ mod \ B$ where $u \in S$ is a polynomial unit, i.e. a (polynomially represented) non zero domain element in the Noetherian case (pseudodivision of f by B) or a polynomial with a scalar as leading term in the non Noetherian case. Following up the reduction steps one can

even produce a presentation of $h - u \cdot f$ as a polynomial combination of the base elements in B.

More general, given for $f_i \in B$ and f representations $f_i = \sum r_{ik} e_k = R_i \cdot E^T$ and $f = R \cdot E^T$ as polynomial combinations wrt. a fixed basis E one can produce such a presentation also for h. For this purpose the dpoly f and its representation are collected into a base element and reduced simultaneously by the base list B, that collects the base elements and their representations.

The main procedures of the newly designed reduction package are the following:

`red_TopRedBE`(⟨*bas*⟩,⟨*model*⟩)

Top reduction with bounded ecart of the base element $model$ by the base list bas, i.e. only reducing the top term and only with base elements with ecart bounded by that of $model$.

`red_TopRed`(⟨*bas*⟩,⟨*model*⟩)

Top reduction of $model$, but without restrictions.

`red_TailRed`(⟨*bas*⟩,⟨*model*⟩)

Make tail reduction on $model$, i.e. top reduction on the tail terms. For convergence this uses reduction with bounded ecart for non Noetherian term orders and full reduction otherwise.

There is a common `red_TailRedDriver` that takes a top reduction function as parameter. It can be used for experiments with other top reduction procedure combinations.

`red_TotalRed`(⟨*bas*⟩,⟨*model*⟩)

A terminating total reduction, i.e. for Noetherian term orders the classical one and for local term orders using tail reduction with bounded ecart.

`red_Straight` ⟨*bas*⟩

Reduce (with `red_TailRed`) the tails of the polynomials in the base list bas.

`red_TopInterreduce` ⟨*bas*⟩

Reduces the base list bas with red_TopRed until it has pairwise incomparable leading terms, computes correct representation parts, but does no tail reduction.

`red_Interreduce` ⟨*bas*⟩

Does top and, if the switch `red_total` is on, also tail interreduction on the base list bas.

Usually, e.g. for ideal generation problems, there is no need to care about the multiplier u. If nevertheless one needs its value, the base element f may be prepared with `red_prepare` to collect this information in the 0-slot of its representation part. Extract this information with `red_extract`.

`red_redpol(`⟨*bas*⟩`,`⟨*model*⟩`)`

combines this tool with a total reduction of the base element $model$ and returns a dotted pair

`(`⟨*reduced model*⟩ `.` ⟨*dpoly unit multiplier*⟩`)`

Advanced applications call the interfacing procedures

`interreduce!*` ⟨*m*⟩

that returns an interreduced basis of the dpmat m.

`mod!*(`⟨*f*⟩`,`⟨*m*⟩`)`

that returns the dotted pair $(h.u)$ where h is the pseudo normal form of the dpoly f modulo the dpmat m and u the corresponding polynomial unit multiplier.

`normalform!*(`⟨*a*⟩`,`⟨*b*⟩`)`

that returns $\{a_1, r, z\}$ with $a_1 = z * a - r * b$ where the rows of the dpmat a_1 are the normalforms of the rows of the dpmat a with respect to the dpmat b.

For local standard bases the ideal generated by the basic polynomials may have components not passing through the origin. Although they do not contribute to the ideal in $Loc(S) = S_{\mathbf{m}}$ they usually heavily increase the necessary computational effort. Hence for local term orders one should try to remove polynomial units as soon as they are detected. To remove them from base elements in an early stage of the computation one can either try the (cheap) test, whether $f \in S$ is of the form ⟨*monomial*⟩ * ⟨*polynomial unit*⟩ or factor f completely and remove polynomial unit factors. For base elements this may be done with `bas_detectunits` or `bas_factorunits`.

Moreover there are two switches `detectunits` and `factorunits`, both off by default, that force such automatic simplifications during more advanced computations.

The procedure `deleteunits!*` tries explicitely to factor the basis polynomials of a dpmat and to remove polynomial units occuring as one of the factors.

The Gröbner and Standard Basis Algorithms

There is now a unique *module groeb* that contains the Gröbner resp. standard basis algorithms with syzygy computation facility and related topics. There are common procedures (working for both Noetherian and non Noetherian term orders)

`gbasis!*` $\langle m \rangle$

that returns a minimal Gröbner or standard basis of the dpmat m,

`syzygies!*` $\langle m \rangle$

that returns an interreduced basis of the first syzygy module of the dpmat m and

`syzygies1!*` $\langle m \rangle$

that returns a (not yet interreduced) basis of the syzygy module of the dpmat m.

These procedures start the outer Gröbner engine (now also common for both Noetherian and non Noetherian term orders)

`groeb_stbasis`($\langle m \rangle$,$\langle mgb \rangle$,$\langle ch \rangle$,$\langle syz \rangle$)

that returns, applied to the dpmat m, three dpmats g, c, s with

g — the minimal reduced Gröbner basis of m if $mgb = t$,

c — the transition matrix $g = c \cdot m$ if $ch = t$, and

s — the (not yet interreduced) syzygy matrix of m if $syz = t$.

The next layer manages the preparation of the representation parts of the base elements to carry the syzygy information, calls the *general internal driver*, and extracts the relevant information from the result of that computation. The general internal driver branches according to different reduction functions into several versions. Upto now there are three different strategies for the reduction procedures for the S-polynomial reduction (different versions may be chosen via *gbtestversion*):

1. Total reduction with local simplifier lists. For local term orders this is (almost) Mora's first version for the tangent cone (the default).
2. Total reduction with global simplifier list. For local term orders this is (almost) Mora's SimpStBasis, see [MPT89].
3. Total reduction with bounded ecart.

The first two versions (almost) coincide for Noetherian term orders. The third version reduces only with bounded ecart, thus forcing more pairs to be treated than necessary, but usually less expensive to be reduced. It is not yet well understood, whether this idea is of practical importance.

groeb_lazystbasis calls the lazy standard basis driver instead, that implements Mora's lazy algorithm, see [MPT89]. As *groeb_homstbasis*, the computation of Gröbner and standard bases via homogenization (Lazard's approach), it is not fully integrated into the algebraic interface. Use

`homstbasis!*` ⟨*m*⟩

for the invocation of the homogenization approach to compute a standard basis of the dpmat m and

`lazystbasis!*` ⟨*m*⟩

for the lazy algorithm.

Experts commonly agree that the classical approach is better for "computable" examples, but computations done by the author on large examples indicate, that both approaches are in fact independent.

The pair list management uses the sugar strategy, see [GMN+91], with respect to the current ecart vector. If the input is homogeneous and the ecart vector reflects this homogeneity then pairs are sorted by ascending degree. Hence no superfluous base elements will be computed in this case. In general the sugar strategy performs best if the ecart vector is chosen to make the input close to be homogeneous.

There is another global variable `cali!=monset` that may contain a list of variable names (a subset of the variable names of the current base ring). During the "pure" Gröbner algorithm (without syzygy and representation computations) common monomial factors containing only these variables will be canceled out. This shortcut is useful if some of the variables are known to be non zero divisors as e.g. in most implicitation problems.

`setmonset!*` ⟨*vars*⟩

initializes `cali!=monset` with a given list of variables $vars$.

The Gröbner tools as e.g. pair criteria, pair list update, pair management and S-polynomial construction are available.

`groeb_mingb` ⟨*m*⟩

extracts a minimal Gröbner basis from the dpmat m, removing base elements with leading terms, divisible by other leading terms.

`groeb_minimize`(⟨*bas*⟩,⟨*syz*⟩)

minimizes the dpmat pair (bas, syz) deleting superfluous base elements from bas using syzygies from syz containing unit entries.

The Gröbner Factorizer

If $\bar{k}$ is the algebraic closure of k, $B := \{f_1, \ldots, f_m\} \subset S$ a finite system of polynomials, and $C := \{g_1, \ldots, g_k\}$ a set of side conditions define the *relative set of zeroes*

$$Z(B, C) := \{a \in \bar{k}^n : \forall\ f \in B\ f(a) = 0 \text{ and } \forall g \in C\ g(a) \neq 0\}.$$

Its Zariski closure is the zero set of $I(B) :< \prod C >$.

The Gröbner factorizer solves the following problem:

> *Find a collection* (B_α, C_α) *of Gröbner bases* B_α *and side conditions* C_α *such that*
> $$Z(B, C) = \bigcup_\alpha Z(B_\alpha, C_\alpha).$$

The *module groebf* and the *module triang* contain algorithms related to that problem, triangular systems, and their generalizations as described in [Grä94a] and [Grä95b]. V. 2.2 thus heavily extends the algorithmic possibilities that were implemented in former releases of CALI.

Note that, different to v. 2.1, we work with constraint *lists*.

`groebfactor!*`(⟨*bas*⟩,⟨*con*⟩)

returns for the dpmat ideal bas and the constraint list con (of dpolys) a minimal list of $(dpmat, constraint\ list)$ pairs with the desired property.

During a preprocessing it splits the submitted basis bas by a recursive factorization of polynomials and interreduction of bases into a (reduced) list of smaller subproblems consisting of a partly computed Gröbner basis, a constraint list, and a list of pairs not yet processed. The main procedure forces the next subproblem to be processed until another factorization is possible. Then the subproblem splits into subsubproblems, and the subproblem list will be updated. Subproblems are kept sorted with respect to their expected dimension *easydim* forcing this way a *depth first* recursion. Returned and not yet interreduced Gröbner bases are, after interreduction, subject to another call of the preprocessor since interreduced polynomials may factor anew.

`listgroebfactor!*` ⟨*l*⟩

processes a whole list of dpmats (without constraints) at once and strips off constraints at the end.

Using the (ordinary) Gröbner factorizer even components of different dimension may keep gluing together. The *extended Gröbner factorizer* involves a postprocessing, that guarantees a decomposition into puredimensional components, given by triangular systems instead of Gröbner bases. Triangular systems in positive dimension must not be Gröbner bases of the underlying ideal. They should be preferred, since they are more simple but contain all information about the (quasi) prime component that they represent. The complete Gröbner basis of the corresponding component can be obtained by an easy stable quotient computation, see [Grä95b]. We refer to the same paper for the definition of *triangular systems* in positive dimension, that is consistent with our approach.

`extendedgroebfactor!*`($\langle bas \rangle$,$\langle c \rangle$)
`extendedgroebfactor1!*`($\langle bas \rangle$,$\langle c \rangle$)

return a list of results $\{b_i, c_i, v_i\}$ in algebraic prefix form such that b_i is a triangular set wrt. the variables v_i and c_i is a list of constraints, such that $b_i :< \prod c_i >$ is the (puredimensional) recontraction of the zerodimensional ideal $b_i \bigotimes_k k(v_i)$. For the first version the recontraction is not computed, hence the output may be not minimal. The second version computes recontractions to decide superfluous components already during the algorithm. Note that the stable quotient computation involved for that purpose may drastically slow down the whole attempt.

The postprocessing involves a change to dimension zero and invokes (zero dimensional) triangular system computations from the *module triang*. In a first step `groebf_zeroprimes1` incorporates the square free parts of certain univariate polynomials into these systems and strips off the constraints (since relative sets of zeroes in dimension zero are Zariski closed), using a splitting approach analogous to the Gröbner factorizer. In a second step, according to the switch `lexefgb`, either `zerosolve!*` or `zerosolve1!*` converts these intermediate results into lists of triangular systems in prefix form. If `lexefgb` is `off` (the default), the zero dimensional term order is degrevlex and `zerosolve1!*`, the "slow turn to lex" is involved, with `lexefgb` on the pure lexicographic term order and `zerosolve!*`, Möllers original approach, see [Möl93], are used. Note that for this term order we need only a single Gröbner basis computation at this level.

A third version, `zerosolve2!*`, mixes the first approach with the FGLM change of term orders. It is not incorporated into the extended Gröbner factorizer.

Basic Operations on Ideals and Modules

Gröbner and local standard bases are the heart of several basic algorithms in ideal theory, see e.g. [BWK93, 6.2.]. CALI offers the following facilities:

`submodulep!*`($\langle m \rangle$,$\langle n \rangle$)

tests the dpmat m for being a submodule of the dpmat n reducing the basis elements of m with respect to n. The result will be correct provided n is a Gröbner basis.

`modequalp!*`($\langle m \rangle$,$\langle n \rangle$)

= submodulep!*(m,n) and submodulep!*(n,m).

`eliminate!*`($\langle m \rangle$,$\langle$*variable list*$\rangle$)

computes the elimination ideal/module eliminating the variables in the given variable list (a subset of the variables of the current base ring). Changes temporarily the term order to degrevlex.

`matintersect!*` $\langle l \rangle$

computes the intersection of the dpmats in the dpmat list l along [BWK93, 6.20].[7]

CALI offers several quotient algorithms. They rest on the computation of quotients by a single element of the following kind: Assume $M \subset S^c, v \in S^c, f \in S$. Then there are

the *module quotient* $M : (v) = \{g \in S \mid gv \in M\}$,

the *ideal quotient* $M : (f) = \{w \in S^c \mid fw \in M\}$, and

the *stable quotient* $M : (f)^\infty = \{w \in S^c \mid \exists n : f^n w \in M\}$.

CALI uses the elimination approach [CLO92, 4.4.] and [BWK93, 6.38] for their computation:

`matquot!*`($\langle M \rangle$,$\langle f \rangle$)

returns the module or ideal quotient $M : (f)$ depending on f.

`matqquot!*`($\langle M \rangle$,$\langle f \rangle$)

returns the stable quotient $M : (f)^\infty$.

`matquot!*` calls the pseudo division with remainder

`dp_pseudodivmod!*`($\langle g \rangle$,$\langle f \rangle$)

that returns a dpoly list $\{q, r, z\}$ such that $z \cdot g = q \cdot f + r$ with a dpoly unit z. (g, f and r must belong to the same free module). This is done uniformly for noetherian and local term orders with an extended normal form algorithm as described in [Grä95a].

[7] This can be done for ideals and modules in an unique way. Hence `idealintersect!*` has been removed in v. 2.1.

In the same way one defines the quotient of a module by another module (both embedded in a common free module S^c), the quotient of a module by an ideal, and the stable quotient of a module by an ideal. Algorithms for their computation can be obtained from the corresponding algorithms for a single element as divisor either by the generic element method [Eis95] or as an intersection [BWK93, 6.31]. CALI offers both approaches (X=1 or 2 below) at the symbolic level, but for true quotients only the latter one is integrated into the algebraic mode interface.

`idealquotientX!`$\star(\langle M\rangle,\langle I\rangle)$

returns the ideal quotient $M : I$ of the dpmat M by the dpmat ideal I.

`modulequotientX!`$\star(\langle M\rangle,\langle N\rangle)$

returns the module quotient $M : N$ of the dpmat M by the dpmat N.

`annihilatorX!`$\star\ \langle M\rangle$

returns the annihilator of $\operatorname{coker} M$, i.e. the module quotient $S^c : M$, if M is a submodule of S^c.

`matstabquot!`$\star(\langle M\rangle,\langle I\rangle)$

returns the stable quotient $M : I^\infty$ (only by the general element method).

Monomial Ideals

Monomial ideals occur as ideals of leading terms of (ideal's) Gröbner bases and also as components of leading term modules of submodules of free modules, see [Grä93], and reflect some properties of the original ideal/module. Several parameters of the original ideal or module may be read off from it as e.g. dimension and Hilbert series.

The *module moid* contains the corresponding algorithms on monomial ideals. Monomial ideals are lists of monomials, kept sorted by descending lexicographic order as proposed in [BS92].

`moid_primes` $\langle u\rangle$

returns the minimal primes (as a list of lists of variable names) of the monomial ideal u using an adaption of the algorithm, proposed in [BS92] for the computation of the codimension.

`indepvarsets!`$\star\ \langle m\rangle$

returns (based on `moid_primes`) the list of strongly independent sets of m, see [KW88] and [Grä93] for definitions.

`dim!`$\star\ \langle m\rangle$

returns the dimension of $\operatorname{coker} m$ as the size of the largest independent set.

`codim!*` $\langle m \rangle$

returns the codimension of $\operatorname{coker} m$.

`easyindepset!*` $\langle m \rangle$

returns a maximal with respect to inclusion independent set of m.

`easydim!*` $\langle m \rangle$

is a fast dimension algorithm (based on `easyindepset`), that will be correct if m is (radically) unmixed. Since it is significantly faster than the general dimension algorithm[8], it should be used, if all maximal independent sets are known to be of equal cardinality (as e.g. for prime or unmixed ideals, see [Grä93]).

Hilbert Series

CALI v. 2.2 now offers also *weighted Hilbert series*, i.e. series that may reflect multihomogeneity of ideals and modules. For this purpose a weighted Hilbert series has a list of (integer) degree vectors as second parameter, and the ideal(s) of leading terms are evaluated wrt. these weights. For the output and polynomial arithmetic, involved during the computation of the Hilbert series numerator, the different weight levels are mapped onto the first variables of the current ring. If w is such a weight vector list and I is a monomial ideal in the polynomial ring $S = k[x_v : v \in V]$ we get (using multi exponent notation)

$$H(S/I,t) := \sum_{\alpha} |\{x^a \notin I : w(a) = \alpha\}| \cdot t^{\alpha} = \frac{Q(t)}{\prod_{v \in V} \left(1 - t^{w(x_v)}\right)}$$

for a certain polynomial Hilbert series numerator $Q(t)$. $H(R/I,t)$ is known to be a rational function with pole order at $t = 1$ equal to $dim\ R/I$. Note that *WeightedHilbertSeries* returns a *reduced* rational function where the gcd of numerator and denominator is canceled out.

(Non weighted) Hilbert series call the weighted Hilbert series procedure with a single weight vector, the ecart vector of the current ring.

The Hilbert series numerator $Q(t)$ is computed using (the obvious generalizations to the weighted case of) the algorithms in [BS92] and [BCRT93]. Experiments suggest that the former is better for few generators of high degree whereas the latter has to be preferred for many generators of low degree. Choose the version with *hftestversion* n, $n = 1, 2$. Bayer/Stillman's approach ($n = 1$) is the default. In the following m is a dpmat and Gröbner basis.

[8] This algorithm is of linear time as opposed to the problem to determine the dimension of an arbitrary monomial ideal, that is known to be NP-hard in the number of variables, see [BS92].

`hf_whilb(`$\langle m \rangle$`,`$\langle w \rangle$`)`

returns the weighted Hilbert series numerator $Q(t)$ of m according to the version chosen with *hftestversion*.

`WeightedHilbertSeries!*(`$\langle m \rangle$`,`$\langle w \rangle$`)`

returns the weighted Hilbert series reduced rational function of m as s.q.

`HilbertSeries!*(`$\langle m \rangle$`,`$\langle w \rangle$`)`

returns the Hilbert series reduced rational function of m wrt. the ecart vector of the current ring as s.q.

`hf_whilb3(`$\langle u \rangle$`,`$\langle w \rangle$`)`
and
`hf_whs_from_resolution(`$\langle u \rangle$`,`$\langle w \rangle$`)`

compute the weighted Hilbert series numerator and the corresponding reduced rational function from (the column degrees of) a given resolution u.

`degree!*` $\langle m \rangle$

returns the value of the numerator of the reduced Hilbert series of m at $t = 1$. i.e. the sum of its coefficients. For the standard ecart this is the degree of $\operatorname{coker} m$.

Resolutions

Resolutions of ideals and modules, represented as lists of dpmats, are computed via repeated syzygy computation with minimization steps between them to get minimal bases and generators of syzygy modules. Note that the algorithms apply simultaneously to both Noetherian and non Noetherian term orders. For compatibility reasons with further releases v. 2.2 introduces a second parameter to bound the number of syzygy modules to be computed, since Hilbert's syzygy theorem applies only to regular rings.

`Resolve!*(`$\langle m \rangle$`,`$\langle d \rangle$`)`

computes a minimal resolution of the dpmat m, i.e. a list of dpmats $\{s_0, s_1, s_2, \ldots\}$, where s_k is the k-th syzygy module of m, upto part s_d.

`BettiNumbers!*` $\langle c \rangle$
and
`GradedBettiNumbers!*` $\langle c \rangle$

returns the Betti numbers resp. the graded Betti numbers of the resolution c, i.e. the list of the lengths resp. the degree lists (according to the ecart) themselves of the dpmats in c.

Zero Dimensional Ideals and Modules

There are several algorithms that either force the reduction of a given problem to dimension zero or work only for zero dimensional ideals or modules. The *module odim* offers such algorithms. It contains, e.g.

`dimzerop!`$\star$ $\langle m \rangle$

that tests a dpmat m for being zero dimensional.

`getkbase!`$\star$ $\langle u \rangle$

that returns a (monomial) k-vector space basis of $Coker\ m$ provided m is a Gröbner basis.

`odim_borderbasis` $\langle m \rangle$

that returns a border basis, see [MMM91], of the zero dimensional dpmat m as a list of base elements.

`odim_parameter` $\langle m \rangle$

that returns a parameter of the dpmat m, i.e. a variable $x \in vars$ such that $k[x] \bigcap Ann\ S^c/m = (0)$, or *nil* if m is zero dimensional.

`odim_up`($\langle a \rangle$,$\langle m \rangle$)

that returns an univariate polynomial (of smallest possible degree if m is a gbasis) in the variable a, that belongs to the zero dimensional dpmat ideal m, using Buchberger's approach [Buc85].

Primary Decomposition and Related Algorithms

The algorithms of the *module prime* implement the ideas of [GTZ88] with modifications along [Kre87] and their natural generalizations to modules as e.g. explained in [Rut92]. Version 2.2.1 fixes a serious bug detecting superfluous embedded primary components, see section 20.7.7, and contains now a second primary decomposition algorithm, based on ideal separation, as standard. For a discussion about embedded primes and the ideal separation approach, see [Grä97].

CALI contains also algorithms for the computation of the unmixed part of a given module and the unmixed radical of a given ideal (along the same lines). We followed the stepwise recursion decreasing dimension in each step by 1 as proposed in (the final version of) [GTZ88] rather than the "one step" method described in [BWK93] since handling leading coefficients, i.e. standard forms, depending on several variables is a quite hard job for REDUCE[9].

In the following procedures m must be a Gröbner basis.

[9] `prime!=decompose2` implements this strategy in the symbolic mode layer.

`zeroradical!*` ⟨*m*⟩

returns the radical of the zero dimensional ideal m, using squarefree decomposition of univariate polynomials.

`zeroprimes!*` ⟨*m*⟩

computes as in [GTZ88] the list of prime ideals of $Ann\ F/M$ if m is zero dimensional, using the (sparse) general position argument from [KW88].

`zeroprimarydecomposition!*` ⟨*m*⟩

computes the primary components of the zero dimensional dpmat m using prime splitting with the prime ideals of $Ann\ F/M$. It returns a list of pairs with first entry the primary component and second entry the corresponding associated prime ideal.

`isprime!*` ⟨*m*⟩

a (one step) primality test for ideals, extracted from [GTZ88].

`isolatedprimes!*` ⟨*m*⟩

computes (only) the isolated prime ideals of $Ann\ F/M$.

`radical!*` ⟨*m*⟩

computes the radical of the dpmat ideal m, reducing as in [GTZ88] to the zero dimensional case.

`easyprimarydecomposition!*` ⟨*m*⟩

computes the primary components of the dpmat m, if it has no embedded components. The algorithm uses prime splitting with the isolated prime ideals of $Ann\ F/M$. It returns a list of pairs as described in `zeroprimarydecomposition!*`.

`primarydecomposition!*` ⟨*m*⟩

computes the primary components of the dpmat m along the lines of [GTZ88]. It returns a list of two-element lists as described above in `zeroprimarydecomposition!*`.

`unmixedradical!*` ⟨*m*⟩

returns the unmixed radical, i.e. the intersection of the isolated primes of top dimension, associated to the dpmat ideal m.

`eqhull!*` ⟨*m*⟩

returns the equidimensional hull, i.e. the intersection of the top dimensional primary components of the dpmat m.

Advanced Algorithms

The *module scripts* just under further development offers some advanced topics of the Gröbner bases theory. It introduces the new data structure of a *map* between base rings:

A ring map

$$\phi \ : \ R \longrightarrow S$$

for $R = k[r_i], S = k[s_j]$ is represented in symbolic mode as a list

$$\{preimage\_ring\ R,\ image\_ring\ S, subst\_list\},$$

where `subst_list` is a substitution list $\{r_1 = \phi_1(s), r_2 = \phi_2(s), \ldots\}$ in algebraic prefix form, i.e. looks like `(list (equal var image) ...)`.

The central tool for several applications is the computation of the preimage $\phi^{-1}(I) \subset R$ of an ideal $I \subset S$ either under a polynomial map ϕ or its closure in R under a rational map ϕ, see [BWK93, 7.69 and 7.71].

`preimage!*`(⟨*m*⟩,⟨*map*⟩)

computes the preimage of the ideal m in algebraic prefix form under the given polynomial map and sets the current base ring to the preimage ring. Returns the result also in algebraic prefix form.

`ratpreimage!*`(⟨*m*⟩,⟨*map*⟩)

computes the closure of the preimage of the ideal m in algebraic prefix form under the given rational map and sets the current base ring to the preimage ring. Returns the result also in algebraic prefix form.

Derived applications are

`affine_monomial_curve!*`(⟨*l*⟩,⟨*vars*⟩)

l is a list of integers, $vars$ a list of variable names of the same length as l. The procedure sets the current base ring and returns the defining ideal of the affine monomial curve with generic point $(t^i \ : \ i \in l)$ computing the corresponding preimage.

`analytic_spread!*` ⟨*m*⟩

Computes the analytic spread of M, i.e. the dimension of the exceptional fiber $\mathcal{R}(M)/m\mathcal{R}(M)$ of the blowup along M over the irrelevant ideal m of the current base ring.

`assgrad!`$\star$($\langle$*M*$\rangle$,$\langle$*N*$\rangle$,$\langle$*vars*$\rangle$)

Computes the associated graded ring

$$gr_R(N) := (R/N \oplus N/N^2 \oplus \ldots) = \mathcal{R}(N)/N\mathcal{R}(N)$$

over the ring $R = S/M$, where M and N are dpmat ideals defined over the current base ring S. $\langle$*vars*$\rangle$ is a list of new variable names one for each generator of N. They are used to create a second ring T with degree order corresponding to the ecart of the row degrees of N and a ring map

$$\phi : S \oplus T \longrightarrow S.$$

It returns a dpmat ideal J such that $(S \oplus T)/J$ is a presentation of the desired associated graded ring over the new current base ring $S \oplus T$.

`blowup!`$\star$($\langle$*M*$\rangle$,$\langle$*N*$\rangle$,$\langle$*vars*$\rangle$)

Computes the blow up $\mathcal{R}(N) := R[N \cdot t]$ of N over the ring $R = S/M$, where M and N are dpmat ideals defined over the current base ring S. `vars` is a list of new variable names one for each generator of N. They are used to create a second ring T with degree order corresponding to the ecart of the row degrees of N and a ring map

$$\phi : S \oplus T \longrightarrow S.$$

It returns a dpmat ideal J such that $(S \oplus T)/J$ is a presentation of the desired blowup ring over the new current base ring $S \oplus T$.

`proj_monomial_curve!`$\star$($\langle$*l*$\rangle$,$\langle$*vars*$\rangle$)

l is a list of integers, $vars$ a list of variable names of the same length as l. The procedure set the current base ring and returns the defining ideal of the projective monomial curve with generic point $(s^{d-i} \cdot t^i \ : \ i \in l)$ in R, where $d = max\{x \ : \ x \in l\}$, computing the corresponding preimage.

`sym!`$\star$($\langle$*M*$\rangle$,$\langle$*vars*$\rangle$)

Computes the symmetric algebra $Sym(M)$ where M is a dpmat ideal defined over the current base ring S. $\langle$*vars*$\rangle$ is a list of new variable names one for each generator of M. They are used to create a second ring R with degree order corresponding to the ecart of the row degrees of N and a ring map

$$\phi : S \oplus R \longrightarrow S.$$

It returns a dpmat ideal J such that $(S \oplus R)/J$ is the desired symmetric algebra over the new current base ring $S \oplus R$.

There are several other applications:

`minimal_generators!*` $\langle m \rangle$

returns a set of minimal generators of the dpmat m inspecting the first syzygy module.

`nzdp!*`($\langle f \rangle$,$\langle m \rangle$)

tests whether the dpoly f is a non zero divisor on $\operatorname{coker} m$. m must be a Gröbner basis.

`symbolic_power!*`($\langle m \rangle$,$\langle d \rangle$)

returns the dth symbolic power of the prime dpmat ideal m as the equidimensional hull of the dth true power. (Hence applies also to unmixed ideals.)

`varopt!*` $\langle m \rangle$

finds a heuristically optimal variable order by the approach in [BGK86] and returns the corresponding list of variables.

Dual Bases

For the general ideas underlying the dual bases approach see e.g. [MMM91]. This paper explains, that constructive problems from very different areas of commutative algebra can be formulated in a unified way as the computation of a basis for the intersection of the kernels of a finite number of linear functionals generating a dual S-module. Our implementation honours this point of view, presenting two general drivers *dualbases* and *dualhbases* for the computation of such bases (even as submodules of a free module $M = S^m$) with affine resp. projective dimension zero.

Such a collection of N linear functionals

$$L : M = S^m \longrightarrow k^N$$

should be given through values $\{[e_i, L(e_i)], i = 1, \ldots, m\}$ on the generators e_i of M and an evaluation function `evlf([p,L(p)],x)`, that evaluates $L(p \cdot x)$ from $L(p)$ for $p \in M$ and the variable $x \in S$.

dualbases starts with a list of such generator/value constructs generating M and performs Gaussian reduction on expressions $[p \cdot x, L(p \cdot x)]$, where p was already processed, $L(p) \neq 0$, and $x \in S$ is a variable. These elements are processed in ascending order wrt. the term order on M. This guarantees both termination and that the resulting basis of $ker\ L$ is a Gröbner basis. The N values of L are attached to N variables, that are ordered linearly. Gaussian elimination is executed wrt. this variable order.

To initialize the dual bases driver one has to supply the basic generator/value list (through the parameter list; for ideals just the one element list containing the gen-

erator $[1 \in S, L(1)]$), the evaluation function, and the linear algebra variable order. The latter are supplied via the property list of `cali` as properties `evlf` and `varlessp`. Different applications need more entries on the property list of `cali` to manage the communication between the driver and the calling routine.

dualhbases realizes the same idea for (homogeneous) ideals and modules of (projective) dimension zero. It produces zerodimensional "slices" with ascending degree until it reaches a supremum supplied by the user, see [MMM91] for details.

Applications concern affine and projective defining ideals of a finite number of points[10] and two versions (with and without precomputed border basis) of term order changes for zerodimensional ideals and modules as first described in [FGLM93].

`affine_points!*` $\langle m \rangle$

m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in affine space with coordinates given by the rows of m. Note that m may contain parameters. In this case k is treated as rational function field.

`change_termorder!*`($\langle m \rangle$,$\langle r \rangle$)
and
`change_termorder1!*`($\langle m \rangle$,$\langle r \rangle$)

m is a Gröbner basis of a zero dimensional ideal wrt. the current base ring. These procedures change the current ring to r and compute the Gröbner basis of m wrt. the new ring r. The former uses a precomputed border basis.

`proj_points!*` $\langle m \rangle$

m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in projective space with homogeneous coordinates given by the rows of m. Note that m may as for `affine_points` contain parameters.

20.7.5 A Short Description of Procedures Available in Algebraic Mode

Here we give a short description, ordered alphabetically, of **algebraic** procedures offered by CALI in the algebraic mode interface[11].

[10] This substitutes the "brute force" method computing the corresponding intersections directly as it was implemented in v. 2.1. The new approach is significantly faster. The old stuff is available as `affine_points1!*` and `proj_points1!*`.

[11] It does **not** contain switches, get... procedures, setting trace level and related stuff.

If not stated explicitely procedures take (algebraic mode) polynomial matrices ($c > 0$) or polynomial lists ($c = 0$) $m, m1, m2, \ldots$ as input and return results of the same type. $\langle gb \rangle$ stands for a bounded identifier[12], $\langle gbr \rangle$ for one with precomputed resolution. For the mechanism of *bounded identifier* see the section "Algebraic Mode Interface".

`affine_monomial_curve`($\langle l \rangle$,$\langle vars \rangle$)

l is a list of integers, $vars$ a list of variable names of the same length as l. The procedure sets the current base ring and returns the defining ideal of the affine monomial curve with generic point $(t^i \ : \ i \in l)$.

`affine_points` $\langle m \rangle$

m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in affine space with coordinates given by the rows of m. Note that m may contain parameters. In this case k is treated as rational function field.

`analytic_spread` $\langle m \rangle$

Computes the analytic spread of m.

`annihilator` $\langle m \rangle$

returns the annihilator of the dpmat $m \subseteq S^c$, i.e. $Ann\ S^c/M$.

`assgrad`($\langle M \rangle$,$\langle N \rangle$,$\langle vars \rangle$)

Computes the associated graded ring $gr_R(N)$ over $R = S/M$, where S is the current base ring. $\langle vars \rangle$ is a list of new variable names, one for each generator of N. They are used to create a second ring T to return an ideal J such that $(S \oplus T)/J$ is the desired associated graded ring over the new current base ring $S \oplus T$.

`bettiNumbers` $\langle gbr \rangle$

extracts the list of Betti numbers from the resolution of gbr.

`blowup`($\langle M \rangle$,$\langle N \rangle$,$\langle vars \rangle$)

Computes the blow up $\mathcal{R}(N)$ of N over the ring $R = S/M$, where S is the current base ring. `vars` is a list of new variable names, one for each generator of N. They are used to create a second ring T to return an ideal J such that $(S \oplus T)/J$ is the desired blowup ring over the new current base ring $S \oplus T$.

[12] Different to v. 2.1 a Gröbner basis will be computed automatically, if necessary.

`change_termorder`(⟨*m*⟩,⟨*r*⟩)
and
`change_termorder1`(⟨*m*⟩,⟨*r*⟩)

Change the current ring to r and compute the Gröbner basis of m wrt. the new ring r by the FGLM approach. The former uses internally a precomputed border basis.

`codim` ⟨*gb*⟩

returns the codimension of S^c/gb.

`degree` ⟨*gb*⟩

returns the multiplicity of gb as the sum of the coefficients of the (classical) Hilbert series numerator.

`degsfromresolution` ⟨*gbr*⟩

returns the list of column degrees from the minimal resolution of gbr.

`deleteunits` ⟨*m*⟩

factors each basis element of the dpmat ideal m and removes factors that are polynomial units. Applies only to non Noetherian term orders.

`dim` ⟨*gb*⟩

returns the dimension of S^c/gb.

`dimzerop` ⟨*gb*⟩

tests whether S^c/gb is zerodimensional.

`directsum`(⟨*m1*⟩,⟨*m2*⟩,...)

returns the direct sum of the modules $m1, m2, \ldots$, embedded into the direct sum of the corresponding free modules.

`dpgcd`(⟨*f*⟩,⟨*g*⟩)

returns the gcd of two polynomials f and g, computed by the syzygy method.

`easydim` ⟨*m*⟩
and
`easyindepset` ⟨*m*⟩

If the given ideal or module is unmixed (e.g. prime) then all maximal strongly independent sets are of equal size and one can look for a maximal with respect to inclusion rather than size strongly independent set. These procedures don't test the input for being a Gröbner basis or unmixed, but construct a maximal with respect to inclusion independent set of the basic leading terms resp. detect from this (an approximation for) the dimension.

`easyprimarydecomposition` ⟨*m*⟩

a short primary decomposition using ideal separation of isolated primes of m, that yields true results only for modules without embedded components. Returns a list of $\{component, associated\ prime\}$ pairs.

`eliminate`(⟨*m*⟩,⟨*variable list*⟩)

computes the elimination ideal/module eliminating the variables in the given variable list (a subset of the variables of the current base ring). Changes temporarily the term order to degrevlex.

`eqhull` ⟨*m*⟩

returns the equidimensional hull of the dpmat m.

`extendedgroebfactor`(⟨*m*⟩,⟨*c*⟩)
and
`extendedgroebfactor1`(⟨*m*⟩,⟨*c*⟩)

return for a polynomial ideal m and a list of (polynomial) constraints c a list of results $\{b_i, c_i, v_i\}$, where b_i is a triangular set wrt. the variables v_i and c_i is a list of constraints, such that $Z(m, c) = \bigcup Z(b_i, c_i)$. For the first version the output may be not minimal. The second version decides superfluous components already during the algorithm.

`gbasis` ⟨*gb*⟩

returns the Gröbner resp. local standard basis of gb.

`getkbase` ⟨*gb*⟩

returns a k-vector space basis of S^c/gb, consisting of module terms, provided gb is zerodimensional.

`getleadterms` ⟨*gb*⟩

returns the dpmat of leading terms of a Gröbner resp. local standard basis of gb.

`GradedBettinumbers` ⟨*gbr*⟩

extracts the list of degree lists of the free summands in a minimal resolution of gbr.

`groebfactor` (⟨*m*⟩[,⟨*c*⟩])

returns for the dpmat ideal m and an optional constraint list c a (reduced) list of dpmats such that the union of their zeroes is exactly $Z(m, c)$. Factors all polynomials involved in the Gröbner algorithms of the partial results.

`HilbertSeries` ⟨*gb*⟩

returns the Hilbert series of gb with respect to the current ecart vector.

`homstbasis` ⟨*m*⟩

computes the standard basis of m by Lazard's homogenization approach.

`ideal2mat` ⟨*m*⟩

converts the ideal (=list of polynomials) m into a column vector.

`ideal_of_minors`(⟨*mat*⟩,⟨*k*⟩)

computes the generators for the ideal of k-minors of the matrix mat.

`ideal_of_pfaffians`(⟨*mat*⟩,⟨*k*⟩)

computes the generators for the ideal of the $2k$-pfaffians of the skewsymmetric matrix mat.

`idealpower`(⟨*m*⟩,⟨*n*⟩)

returns the interreduced basis of the ideal power m^n with respect to the integer $n \geq 0$.

`idealprod`(⟨*m1*⟩,⟨*m2*⟩,...)

returns the interreduced basis of the ideal product $m1 \cdot m2 \cdot \ldots$ of the ideals $m1, m2, \ldots$.

`idealquotient`(⟨*m1*⟩,⟨*m2*⟩)

returns the ideal quotient $m1 : m2$ of the module $m1 \subseteq S^c$ by the ideal $m2$.

`idealsum`(⟨*m1*⟩,⟨*m2*⟩,...)

returns the interreduced basis of the ideal sum $m1 + m2 + \ldots$.

`indepvarsets` ⟨*gb*⟩

returns the list of strongly independent sets of gb with respect to the current term order, see [KW88] for a definition in the case of ideals and [Grä93] for submodules of free modules.

`initmat`(⟨*m*⟩,⟨*filename*⟩)

initializes the dpmat m together with its base ring, term order and column degrees from a file.

`interreduce` ⟨*m*⟩

returns the interreduced module basis given by the rows of m, i.e. a basis with pairwise indivisible leading terms.

`isolatedprimes` ⟨*m*⟩

returns the list of isolated primes of the dpmat m, i.e. the isolated primes of $Ann\ S^c/M$.

`isprime` ⟨*gb*⟩

tests the ideal gb to be prime.

`iszeroradical` ⟨*gb*⟩

tests the zerodimensional ideal gb to be radical.

`lazystbasis` ⟨*m*⟩

computes the standard basis of m by the lazy algorithm, see e.g. [MPT89].

`listgroebfactor` ⟨*in*⟩

computes for the list in of ideal bases a list out of Gröbner bases by the Gröbner factorization method, such that $\bigcup_{m\in in} Z(m) = \bigcup_{m\in out} Z(m)$.

`mat2list` ⟨*m*⟩

converts the matrix m into a list of its entries.

`matappend`(⟨*m1*⟩,⟨*m2*⟩,...)

collects the rows of the dpmats $m1, m2, \ldots$ to a common matrix. $m1, m2, \ldots$ must be submodules of the same free module, i.e. have equal column degrees (and size).

`mathomogenize`(⟨*m*⟩,⟨*var*⟩)[13]

returns the result obtained by homogenization of the rows of m with respect to the variable ⟨*var*⟩ and the current *ecart vector*.

`matintersect`(⟨*m1*⟩,⟨*m2*⟩,...)

returns the interreduced basis of the intersection $m1 \bigcap m2 \bigcap \ldots$.

`matjac`(⟨*m*⟩,⟨*variable list*⟩)

returns the Jacobian matrix of the ideal m with respect to the supplied variable list.

`matqquot`(⟨*m*⟩,⟨*f*⟩)

returns the stable quotient $m : (f)^\infty$ of the dpmat m by the polynomial $f \in S$.

`matquot`(⟨*m*⟩,⟨*f*⟩)

returns the quotient $m : (f)$ of the dpmat m by the polynomial $f \in S$.

`matstabquot`(⟨*m1*⟩,⟨*id*⟩)

returns the stable quotient $m1 : id^\infty$ of the dpmat $m1$ by the ideal id.

[13] Dehomogenize with `sub(z=1,m)` if z is the homogenizing variable.

`matsum`(⟨*m1*⟩,⟨*m2*⟩,...)

returns the interreduced basis of the module sum $m1+m2+\ldots$ in a common free module.

`minimal_generators` ⟨*m*⟩

returns a set of minimal generators of the dpmat m.

`minors`(⟨*m*⟩,⟨*n*⟩)

returns the matrix of minors of size $b \times b$ of the matrix m.

⟨*a*⟩ `mod` ⟨*b*⟩

computes the (true) normal form(s), i.e. a standard quotient representation, of a modulo the dpmat m. a may be either a polynomial or a polynomial list ($c = 0$) or a matrix ($c > 0$) of the correct number of columns.

`modequalp`(⟨*gb1*⟩,⟨*gb2*⟩)

tests, whether $gb1$ and $gb2$ are equal (returns YES or NO).

`modulequotient`(⟨*m1*⟩,⟨*m2*⟩)

returns the module quotient $m1 : m2$ of two dpmats $m1, m2$ in a common free module.

`normalform`(⟨*m1*⟩,⟨*m2*⟩)

returns a list of three dpmats $\{m3, r, z\}$, where $m3$ is the normalform of $m1$ modulo $m2$, z a scalar matrix of polynomial units (i.e. polynomials of degree 0 in the noetherian case and polynomials with leading term of degree 0 in the tangent cone case), and r the relation matrix, such that

$$m3 = z * m1 + r * m2.$$

`nzdp`(⟨*f*⟩,⟨*m*⟩)

tests whether the dpoly f is a non zero divisor on $\operatorname{coker} m$.

`pfaffian` ⟨*mat*⟩

returns the pfaffian of a skewsymmetric matrix mat.

`preimage`(⟨*m*⟩,⟨*map*⟩)

computes the preimage of the ideal m under the given polynomial map and sets the current base ring to the preimage ring.

`primarydecomposition` ⟨*m*⟩

returns the primary decomposition of the dpmat m as a list of pairs of the form {⟨*component*⟩, ⟨*associated prime*⟩}.

`proj_monomial_curve(`⟨*l*⟩`,`⟨*vars*⟩`)`

l is a list of integers, $vars$ a list of variable names of the same length as l. The procedure sets the current base ring and returns the defining ideal of the projective monomial curve with generic point $(s^{d-i} \cdot t^i \ : \ i \in l)$ in R where $d = max\{x \ : \ x \in l\}$.

`proj_points` ⟨*m*⟩

m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in projective space with homogeneous coordinates given by the rows of m. Note that m may as for `affine_points` contain parameters.

`radical` ⟨*m*⟩

returns the radical of the dpmat ideal m.

`random_linear_form(`⟨*vars*⟩`,`⟨*bound*⟩`)`

returns a random linear form in the variables ⟨*vars*⟩ with integer coefficients less than the supplied `bound`.

`ratpreimage(`⟨*m*⟩`,`⟨*map*⟩`)`

computes the closure of the preimage of the ideal m under the given rational map and sets the current base ring to the preimage ring.

`resolve(`⟨*m*⟩`[,`⟨*d*⟩`])`

returns the first d members of the minimal resolution of the bounded identifier m as a list of matrices. If the resolution has less than d non zero members, only those are collected. (Default: $d = 100$)

`savemat(`⟨*m*⟩`,`⟨*file name*⟩`)`

save the dpmat m together with the settings of it base ring, term order and column degrees to a file.

`setgbasis` ⟨*m*⟩

declares the rows of the bounded identifier m to be already a Gröbner resp. local standard basis thus avoiding a possibly time consuming Gröbner or standard basis computation.

`sive(`⟨*m*⟩`,`⟨*variable list*⟩`)`

sieves out all base elements with leading terms having a factor contained in the specified variable list (a subset of the variables of the current base ring). Useful for elimination problems solved “by hand”.

`singular_locus`(⟨*M*⟩,⟨*c*⟩)

returns the defining ideal of the singular locus of $Spec\ S/M$ where M is an ideal of codimension c, adding to M the generators of the ideal of the c-minors of the Jacobian of M.

`submodulep`(⟨*m*⟩,⟨*gb*⟩)

tests, whether m is a submodule of gb (returns YES or NO).

`sym`(⟨*M*⟩,⟨*vars*⟩)

Computes the symmetric algebra $Sym(M)$ where M is an ideal defined over the current base ring S. ⟨*vars*⟩ is a list of new variable names, one for each generator of M. They are used to create a second ring R to return an ideal J such that $(S \oplus R)/J$ is the desired symmetric algebra over the new current base ring $S \oplus R$.

`symbolic_power`(⟨*m*⟩,⟨*d*⟩)

returns the dth symbolic power of the prime dpmat ideal m.

`syzygies` ⟨*m*⟩

returns the first syzygy module of the bounded identifier m.

`tangentcone` ⟨*gb*⟩

returns the tangent cone part, i.e. the homogeneous part of highest degree with respect to the first degree vector of the term order from the Gröbner basis elements of the dpmat gb. The term order must be a degree order.

`unmixedradical` ⟨*m*⟩

returns the unmixed radical of the dpmat ideal m.

`varopt` ⟨*m*⟩

finds a heuristically optimal variable order, see [BGK86].

```
vars:=varopt m;
setring(vars,\{\},lex);
setideal(m,m);
```

changes to the lexicographic term order with heuristically best performance for a lexicographic Gröbner basis computation.

`WeightedHilbertSeries`(⟨*m*⟩,⟨*w*⟩)

returns the weighted Hilbert series of the dpmat m. Note that m is not a bounded identifier and hence not checked to be a Gröbner basis. w is a list of integer weight vectors.

`zeroprimarydecomposition` ⟨*m*⟩

returns the primary decomposition of the zerodimensional dpmat m as a list of $\{component, associated\ prime\}$ pairs.

`zeroprimes` ⟨*m*⟩

returns the list of primes of the zerodimensional dpmat m.

`zeroradical` ⟨*gb*⟩

returns the radical of the zerodimensional ideal gb.

`zerosolve` ⟨*m*⟩
and
`zerosolve1` ⟨*m*⟩
and
`zerosolve2` ⟨*m*⟩

Returns for a zerodimensional ideal a list of triangular systems that cover $Z(m)$. `zerosolve` needs a pure lex. term order for the "fast" turn to lex. as described in [Möl93], `zerosolve1` is the "slow" turn to lex. as described in [Grä95b], and `zerosolve2` incorporated the FGLM term order change into `zerosolve1`.

20.7.6 The CALI Module Structure

name	subject	data type	representation
cali	Header module, contains global variables, switches etc.	—	—
bcsf	Base coefficient arithmetic	base coeff.	standard forms
ring	Base ring setting, definition of the term order	base ring	special type RING
mo	monomial arithmetic	monomials	(exp. list . degree list)
dpoly	Polynomial and vector arithmetic	dpolys	list of terms
bas	Operations on base lists	base list	list of base elements
dpmat	Operations on polynomial matrices, the central data type of CALI	dpmat	special type DPMAT
red	Normal form algorithms	—	—
groeb	Gröbner basis algorithm and related ones	—	—
groebf	the Gröbner factorizer and its extensions	—	—
matop	Operations on (lists of) dpmats that correspond to ideal/module operations	—	—
quot	Different quotient algorithms	—	—
moid	Monomial ideal algorithms	monomial ideal	list of monomials
hf	weighted Hilbert series	–	–
res	Resolutions of dpmats	resolution	list of dpmats
intf	Interface to algebraic mode	—	—
odim	Algorithms for zerodimensional ideals and modules	—	—
prime	Primary decomposition and related questions	—	—
scripts	Advanced applications	—	—
calimat	Extension of the matrix package	—	—
lf	The dual bases approach	—	—
triang	(Zero dimensional) triangular systems	—	—

20.7.7 Changelog

New and Improved Facilities in v. 2.1

The major changes in v. 2.1 reflect the experience we've got from the use of CALI 2.0. The following changes are worth mentioning explicitely:

1. The algebraic rule concept was adapted to CALI. It allows to supply rule based coefficient domains. This is a more efficient way to deal with (easy) algebraic numbers than through the *arnum package*.

2. *listtest* and *listminimize* provide an unified concept for different list operations previously scattered in the source text.

3. There are several new quotient algorithms at the symbolic level (both the general element and the intersection approaches are available) and new features for the computation of equidimensional hull and equidimensional radical.

4. A new *module scripts* offers advanced applications of Gröbner bases.

5. Several advanced procedures initialize a Gröbner basis computation over a certain intermediate base ring or term order as e.g. *eliminate*, *resolve*, *matintersect* or all *primary decomposition* procedures. Interrupting a computation in v. 2.1 now restores the original values of CALI's global variables, since all intermediate procedures work with local copies of the global variables.[14] This doesn't apply to advanced procedures that change the current base ring as e.g. *blowup*, *preimage*, *sym* etc.

New and Improved Facilities in v. 2.2

Version 2.2 (beside bug fixes) incorporates several new facilities of constructive non linear algebra that we investigated the last two years, as e.g. dual bases, the Gröbner factorizer, triangular systems, and local standard bases. Essential changes concern the following topics:

1. The CALI modules *red* and *groeb* were rewritten and the *module mora* was removed. This is due to new theoretical insight into standard bases theory as e.g. described in [Grä94b] or [Grä95a]. The Gröbner basis algorithm is reorganized as a Gröbner driver with simplifier and base lists, that involves different versions of polynomial reduction according to the setting

[14] Note that recovering the base ring this way may cause some trouble since the intermediate ring, installed with *setring*, changed possibly the internal variable order set by *setkorder*.

via *gbtestversion*. It applies now to both noetherian and non noetherian term orders in a unified way.

The switches `binomial` and `lazy` were removed.

2. The Gröbner factorizer was thoroughly revised, extended along the lines explained in [Grä94a], and collected into a separate *module groebf*. It now allows a list of constraints also in algebraic mode. Two versions of an *extended Gröbner factorizer* produce *triangular systems*, i.e. a decomposition into quasi prime components, see [Grä95b], that are well suited for further (numerical) evaluation. There is also a version of the Gröbner factorizer that allows a list of problems as input. This is especially useful, if a system is splitted with respect to a "cheap" (e.g. degrevlex) term order and the pieces are recomputed with respect to a "hard" (e.g. pure lex) term order.

   The extended Gröbner factorizer involves, after change to dimension zero, the computation of *triangular systems*. The corresponding module *triang* extends the facilities for zero dimensional ideals and modules in the *module odim*.

3. A new *module lf* implements the *dual bases* approach as described in [MMM91]. On this basis there are new implementations of procedures `affine_points` and `proj_points` that are significantly faster than the old ones. The linear algebra *change of term orders* [FGLM93] is available, too. There are two versions, one with precomputed *border basis*, the other with conventional normal forms.

4. *dpmat*s now have a *gb-tag* that indicates, whether the given ideal or module basis is already a Gröbner basis. This avoids certain Gröbner basis recomputations especially during advanced algorithms as e.g. prime decomposition. In the algebraic interface Gröbner bases are computed automatically when needed rather than to issue an error message as in v. 2.1. So one can call `modequalp` or `dim` etc. not having computed Gröbner bases in advance. Note that such automatic computation can be avoided with *setgbasis*.

5. Hilbert series are now *weighted Hilbert series*, since e.g. for blow up rings the generating ideal is multigraded. Usual Hilbert series are computed as in v. 2.1 with respect to the *ecart vector*. Weighted Hilbert series accept a list of (integer) weight lists as second parameter.

6. There are some name and conceptual changes to existing procedures and variables to have a more concise semantic concept. This concerns

   - *Tracing* (the trace parameter is now stored on the property list of `cali` and should be set with *setcalitrace*),
   - choosing different versions of the Gröbner algorithm (through *gbtestversion*) and the Hilbert series computation (through *hftestversion*),

- some names (*mat2list* replaced *flatten*, *HilbertSeries* replaced *hilbseries*) and
- parameter lists of some local and internal procedures

7. The *revlex term order* is now the reverse lexicographic term order on the **reversely** ordered variables. This is consistent with other computer algebra systems (e.g. SINGULAR or AXIOM)[15] and implies the same order on the variables for deglex and degrevlex term orders (this was the main reason to change the definition).

8. Ideals of minors, pfaffians and related stuff are now implemented as extension of the internal `matrix` package and collected into a separate *module calimat*. Thus they allow more general expressions, especially with variable exponents, as general REDUCE matrices do. So one can define generic ideals as e.g. ideals of minors or pfaffians of matrices, containing generic expressions as elements. They must be specified for further use in CALI substituting general exponents by integers.

New and Improved Facilities in v. 2.2.1

The main change concerns the primary decomposition algorithm, where I fixed a serious bug for deciding which embedded primes are really embedded[16]. During that remake I incorporated also the Gröbner factorizer to compute isolated primes. Since REDUCE has no multivariate *modular* factorizer, the switch *factorprimes* may be turned off to switch to the former algorithm.

Some minor bugs were fixed as well, e.g., the bug that made *radical* crashing.

[15] But different to the currently distibuted `groebner` package in REDUCE. Note that the computations in [Grä94a] were done *before* these changes.

[16] That there must be a bug was pointed out to me by Shimoyama Takeshi who compared different p.d. implementations. The bug is due to an incorrect test for embedded primes: A (superfluous) primary component may contain none of the isolated primary components, but their intersection! Note that neither [GTZ88] nor [BWK93] comment on that. Details of the implementation will appear in [Grä97].

20.8 CAMAL: Calculations in Celestial Mechanics

This packages implements in REDUCE the Fourier transform procedures of the CAMAL package for celestial mechanics.

Author: John P. Fitch

It is generally accepted that special purpose algebraic systems are more efficient than general purpose ones, but as machines get faster this does not matter. An experiment has been performed to see if using the ideas of the special purpose algebra system CAMAL(F) it is possible to make the general purpose system REDUCE perform calculations in celestial mechanics as efficiently as CAMAL did twenty years ago. To this end a prototype Fourier module is created for REDUCE, and it is tested on some small and medium-sized problems taken from the CAMAL test suite. The largest calculation is the determination of the Lunar Disturbing Function to the sixth order. An assessment is made as to the progress, or lack of it, which computer algebra has made, and how efficiently we are using modern hardware.

20.8.1 Introduction

A number of years ago there emerged the divide between general-purpose algebra systems and special purpose one. Here we investigate how far the improvements in software and more predominantly hardware have enabled the general systems to perform as well as the earlier special ones. It is similar in some respects to the Possion program for MACSYMA [Fat74] which was written in response to a similar challenge.

The particular subject for investigation is the Fourier series manipulator which had its origins in the Cambridge University Institute for Theoretical Astronomy, and later became the F subsystem of CAMAL [Bar67a, Fit83]. In the late 1960s this system was used for both the Delaunay Lunar Theory [Del86, Bar67b] and the Hill Lunar Theory [Bou72], as well as other related calculations. Its particular area of application had a number of peculiar operations on which the general speed depended. These are outlined below in the section describing how CAMAL worked. There have been a number of subsequent special systems for celestial mechanics, but these tend to be restricted to the group of the originator.

The main body of the paper describes an experiment to create within the REDUCE system a sub-system for the efficient manipulation of Fourier series. This prototype program is then assessed against both the normal (general) REDUCE and the extant CAMAL results. The tests are run on a number of small problems typical of those for which CAMAL was used, and one medium-sized problem, the calculation of the Lunar Disturbing Function. The mathematical background to this problem is also presented for completeness. It is important as a problem as it is the first stage

in the development of a Delaunay Lunar Theory.

The paper ends with an assessment of how close the performance of a modern REDUCE on modern equipment is to the (almost) defunct CAMAL of eighteen years ago.

20.8.2 How CAMAL Worked

The Cambridge Algebra System was initially written in assembler for the Titan computer, but later was rewritten a number of times, and matured in BCPL, a version which was ported to IBM mainframes and a number of microcomputers. In this section a brief review of the main data structures and special algorithms is presented.

CAMAL Data Structures

CAMAL is a hierarchical system, with the representation of polynomials being completely independent of the representations of the angular parts.

The angular part had to represent a polynomial coefficient, either a sine or cosine function and a linear sum of angles. In the problems for which CAMAL was designed there are 6 angles only, and so the design restricted the number, initially to six on the 24 bit-halfword TITAN, and later to eight angles on the 32-bit IBM 370, each with fixed names (usually u through z). All that is needed is to remember the coefficients of the linear sum. As typical problems are perturbations, it was reasonable to restrict the coefficients to small integers, as could be represented in a byte with a guard bit. This allowed the representation to pack everything into four words.

```
[ NextTerm, Coefficient, Angles0-3, Angles4-7 ]
```

The function was coded by a single bit in the `Coefficient` field. This gives a particularly compact representation. For example the Fourier term $\sin(u - 2v + w - 3x)$ would be represented as

```
[ NULL, "1"|0x1, 0x017e017d, 0x00000000 ]
or
[ NULL, "1"|0x1, 1:-2:1:-3, 0:0:0:0 ]
```

where `"1"` is a pointer to the representation of the polynomial 1. In all this representation of the term took 48 bytes. As the complexity of a term increased the store requirements to no grow much; the expression $(7/4)ae^3f^5\cos(u-2v+3w-4x+5y+6z)$ also takes 48 bytes. There is a canonicalisation operation to ensure that the leading angle is positive, and $\sin(0)$ gets removed. It should be noted that

$\cos(0)$ is a valid and necessary representation.

The polynomial part was similarly represented, as a chain of terms with packed exponents for a fixed number of variables. There is no particular significance in this except that the terms were held in *increasing* total order, rather than the decreasing order which is normal in general purpose systems. This had a number of important effects on the efficiency of polynomial multiplication in the presence of a truncation to a certain order. We will return to this point later. Full details of the representation can be found in [Fit75].

The space administration system was based on explicit return rather than garbage collection. This meant that the system was sometimes harder to write, but it did mean that much attention was focussed on efficient reuse of space. It was possible for the user to assist in this by marking when an expression was needed no longer, and the compiler then arranged to recycle the space as part of the actual operation. This degree of control was another assistance in running of large problems on relatively small machines.

Automatic Linearisation

In order to maintain Fourier series in a canonical form it is necessary to apply the transformations for linearising products of sine and cosines. These will be familiar to readers of the REDUCE test program as

$$\cos\theta\cos\phi \Rightarrow (\cos(\theta+\phi)+\cos(\theta-\phi))/2, \tag{20.35}$$

$$\cos\theta\sin\phi \Rightarrow (\sin(\theta+\phi)-\sin(\theta-\phi))/2, \tag{20.36}$$

$$\sin\theta\sin\phi \Rightarrow (\cos(\theta-\phi)-\cos(\theta+\phi))/2, \tag{20.37}$$

$$\cos^2\theta \Rightarrow (1+\cos(2\theta))/2, \tag{20.38}$$

$$\sin^2\theta \Rightarrow (1-\cos(2\theta))/2. \tag{20.39}$$

In CAMAL these transformations are coded directly into the multiplication routines, and no action is necessary on the part of the user to invoke them. Of course they cannot be turned off either.

Differentiation and Integration

The differentiation of a Fourier series with respect to an angle is particularly simple. The integration of a Fourier series is a little more interesting. The terms like $\cos(nu+\ldots)$ are easily integrated with respect to u, but the treatment of terms independent of the angle would normally introduce a secular term. By convention in Fourier series these secular terms are ignored, and the constant of integration is taken as just the terms independent of the angle in the integrand. This is equivalent

to the substitution rules

$$\sin(n\theta) \Rightarrow -(1/n)\cos(n\theta)$$
$$\cos(n\theta) \Rightarrow (1/n)\sin(n\theta)$$

In CAMAL these operations were coded directly, and independently of the differentiation and integration of the polynomial coefficients.

Harmonic Substitution

An operation which is of great importance in Fourier operations is the *harmonic substitution*. This is the substitution of the sum of some angles and a general expression for an angle. In order to preserve the format, the mechanism uses the translations

$$\sin(\theta + A) \Rightarrow \sin(\theta)\cos(A) + \cos(\theta)\sin(A)$$
$$\cos(\theta + A) \Rightarrow \cos(\theta)\cos(A) - \sin(\theta)\sin(A)$$

and then assuming that the value A is small it can be replaced by its expansion:

$$\begin{aligned}\sin(\theta + A) \Rightarrow &\sin(\theta)\{1 - A^2/2! + A^4/4! \ldots\} + \\ &\cos(\theta)\{A - A^3/3! + A^5/5! \ldots\} \\ \cos(\theta + A) \Rightarrow &\cos(\theta)\{1 - A^2/2! + A^4/4! \ldots\} - \\ &\sin(\theta)\{A - A^3/3! + A^5/5! \ldots\}\end{aligned}$$

If a truncation is set for large powers of the polynomial variables then the series will terminate. In CAMAL the `HSUB` operation took five arguments; the original expression, the angle for which there is a substitution, the new angular part, the expression part (A in the above), and the number of terms required.

The actual coding of the operation was not as expressed above, but by the use of Taylor's theorem. As has been noted above the differentiation of a harmonic series is particularly easy.

Truncation of Series

The main use of Fourier series systems is in generating perturbation expansions, and this implies that the calculations are performed to some degree of the small quantities. In the original CAMAL all variables were assumed to be equally small (a restriction removed in later versions). By maintaining polynomials in increasing

maximum order it is possible to truncate the multiplication of two polynomials. Assume that we are multiplying the two polynomials

$$A = a_0 + a_1 + a_2 + \ldots$$
$$B = b_0 + b_1 + b_2 + \ldots$$

If we are generating the partial answer

$$a_i(b_0 + b_1 + b_2 + \ldots)$$

then if for some j the product $a_i b_j$ vanishes, then so will all products $a_i b_k$ for $k > j$. This means that the later terms need not be generated. In the product of $1 + x + x^2 + x^3 + \ldots + x^{10}$ and $1 + y + y^2 + y^3 + \ldots + y^{10}$ to a total order of 10 instead of generating 100 term products only 55 are needed. The ordering can also make the merging of the new terms into the answer easier.

20.8.3 Towards a CAMAL Module

For the purposes of this work it was necessary to reproduce as many of the ideas of CAMAL as feasible within the REDUCE framework and philosophy. It was not intended at this stage to produce a complete product, and so for simplicity a number of compromises were made with the "no restrictions" principle in REDUCE and the space and time efficiency of CAMAL. This section describes the basic design decisions.

Data Structures

In a fashion similar to CAMAL a two level data representation is used. The coefficients are the standard quotients of REDUCE, and their representation need not concern us further. The angular part is similar to that of CAMAL, but the ability to pack angle multipliers and use a single bit for the function are not readily available in Standard LISP, so instead a longer vector is used. Two versions were written. One used a balanced tree rather than a linear list for the Fourier terms, this being a feature of CAMAL which was considered but never coded. The other uses a simple linear representation for sums. The angle multipliers are held in a separate vector in order to allow for future flexibility. This leads to a representation as a vector of length 6 or 4;

```
Version1:  [ BalanceBits, Coeff, Function, Angles,
                  LeftTree, RightTree ]
Version2:  [ Coeff, Function, Angles, Next ]
```

where the `Angles` field is a vector of length 8, for the multipliers. It was decided to forego packing as for portability we do not know how many to pack into a small

integer. The tree system used is AVL, which needs 2 bits to maintain balance information, but these are coded as a complete integer field in the vector. We can expect the improvements implicit in a binary tree to be advantageous for large expressions, but the additional overhead may reduce its utility for smaller expressions.

A separate vector is kept relating the position of an angle to its print name, and on the property list of each angle the allocation of its position is kept. So long as the user declares which variables are to be treated as angles this mechanism gives flexibility which was lacking in CAMAL.

Linearisation

As in the CAMAL system the linearisation of products of sines and cosines is done not by pattern matching but by direct calculation at the heart of the product function, where the transformations (1) through (3) are made in the product of terms function. A side effect of this is that there are no simple relations which can be used from within the Fourier multiplication, and so a full addition of partial products is required. There is no need to apply linearisations elsewhere as a special case. Addition, differentiation and integration cannot generate such products, and where they can occur in substitution the natural algorithm uses the internal multiplication function anyway.

Substitution

Substitution is the main operation of Fourier series. It is useful to consider three different cases of substitutions.

1. Angle Expression for Angle:
2. Angle Expression + Fourier Expression for Angle:
3. Fourier Expression for Polynomial Variable.

The first of these is straightforward, and does not require any further comment. The second substitution requires a little more care, but is not significantly difficult to implement. The method follows the algorithm used in CAMAL, using TAYLOR series. Indeed this is the main special case for substitution.

The problem is the last case. Typically many variables used in a Fourier series program have had a WEIGHT assigned to them. This means that substitution must take account of any possible WEIGHTs for variables. The standard code in REDUCE does this in effect by translating the expression to prefix form, and recalculating the value. A Fourier series has a large number of coefficients, and so this operations are repeated rather too often. At present this is the largest problem area

with the internal code, as will be seen in the discussion of the Disturbing Function calculation.

20.8.4 Integration with REDUCE

The Fourier module needs to be seen as part of REDUCE rather than as a separate language. This can be seen as having internal and external parts.

Internal Interface

The Fourier expressions need to co-exist with the normal REDUCE syntax and semantics. The prototype version does this by (ab)using the module method, based in part on the TPS code [PB90]. Of course Fourier series are not constant, and so are not really domain elements. However by asserting that Fourier series form a ring of constants REDUCE can arrange to direct basic operations to the Fourier code for addition, subtraction, multiplication and the like.

The main interface which needs to be provided is a simplification function for Fourier expressions. This needs to provide compilation for linear sums of angles, as well as constructing sine and cosine functions, and creating canonical forms.

User Interface

The creation of `hdiff` and `hint` functions for differentiation disguises this. An unsatisfactory aspect of the interface is that the tokens `sin` and `cos` are already in use. The prototype uses the operator form

```
fourier sin(u)
```

to introduce harmonically represented sine functions. An alternative of using the tokens `f_sin` and `f_cos` is also available.

It is necessary to declare the names of the angles, which is achieved with the declaration

```
harmonic theta, phi;
```

At present there is no protection against using a variable as both an angle and a polynomial variable. This will nooed to be done in a user-oriented version.

20.8.5 The Simple Experiments

The REDUCE test file contains a simple example of a Fourier calculation, determining the value of $(a_1 \cos(wt) + a_3 \cos(3wt) + b_1 \sin(wt) + b_3 \sin(3wt))^3$. For the purposes of this system this is too trivial to do more than confirm the correct answers.

The simplest non-trivial calculation for a Fourier series manipulator is to solve Kepler's equation for the eccentric anomoly E in terms of the mean anomoly u, and the eccentricity of an orbit e, considered as a small quantity

$$E = u + e \sin E$$

The solution procedes by repeated approximation. Clearly the initial approximation is $E_0 = u$. The n^{th} approximation can be written as $u + A_n$, and so A_n can be calculated by

$$A_k = e \sin(u + A_{k-1})$$

This is of course precisely the case for which the HSUB operation is designed, and so in order to calculate $E_n - u$ all one requires is the code

```
bige := fourier 0;
for k:=1:n do <<
  wtlevel k;
  bige:=fourier e * hsub(fourier(sin u), u, u, bige, k);
>>;
write "Kepler Eqn solution:", bige$
```

It is possible to create a regular REDUCE program to simulate this (as is done for example in Barton and Fitch[BF72], page 254). Comparing these two programs indicates substantial advantages to the Fourier module, as could be expected.

Solving Kepler's Equation

Order	REDUCE	Fourier Module
5	9.16	2.48
6	17.40	4.56
7	33.48	8.06
8	62.76	13.54
9	116.06	21.84
10	212.12	34.54
11	381.78	53.94
12	692.56	82.96
13	1247.54	125.86
14	2298.08	187.20
15	4176.04	275.60
16	7504.80	398.62
17	13459.80	569.26
18	***	800.00
19	***	1116.92
20	***	1536.40

These results were with the linear representation of Fourier series. The tree representation was slightly slower. The ten-fold speed-up for the 13th order is most satisfactory.

20.8.6 A Medium-Sized Problem

Fourier series manipulators are primarily designed for large-scale calculations, but for the demonstration purposes of this project a medium problem is considered. The first stage in calculating the orbit of the Moon using the Delaunay theory (of perturbed elliptic motion for the restricted 3-body problem) is to calculate the energy of the Moon's motion about the Earth — the Hamiltonian of the system. This is the calculation we use for comparisons.

Mathematical Background

The full calculation is described in detail in [Bro96], but a brief description is given here for completeness, and to grasp the extent of the calculation.

Referring to the figure 1 which gives the cordinate system, the basic equations are

$$S = (1-\gamma^2)\cos(f+g+h-f'-g'-h') + \gamma^2\cos(f+g-h+f'+g'+h') \tag{20.40}$$

$$r = a(1-e\cos E) \tag{20.41}$$

$$l = E - e\sin E \tag{20.42}$$

$$a = \frac{r\mathbf{d}E}{\mathbf{d}l} \tag{20.43}$$

$$\frac{r^2\mathbf{d}f}{\mathbf{d}l} = a^2(1-e^2)^{\frac{1}{2}} \tag{20.44}$$

$$R = m'\frac{a^2}{a'^3}\frac{a'}{r'}\left\{\left(\frac{r}{a}\right)^2\left(\frac{a'}{r'}\right)^2 P_2(S) + \left(\frac{a}{a'}\right)\left(\frac{r}{a}\right)^3\left(\frac{a'}{r'}\right)^3 P_3(S)+\ldots\right\} \tag{20.45}$$

There are similar equations to (20.41) to (20.44) for the quantities r', a', e', l', E' and f' which refer to the position of the Sun rather than the Moon. The problem is to calculate the expression R as an expansion in terms of the quantities e, e', γ, a/a', l, g, h, l', g' and h'. The first three quantities are small quantities of the first order, and a/a' is of second order.

The steps required are

1. Solve the Kepler equation (20.42)
2. Substitute into (20.41) to give r/a in terms of e and l.
3. Calculate a/r from (20.43) and f from (20.44)
4. Substitute for f and f' into S using (20.40)
5. Calculate R from S, a'/r' and r/a

The program is given in the Appendix.

Results

The Lunar Disturbing function was calculated by a direct coding of the previous sections' mathematics. The program was taken from Barton and Fitch [BF72] with just small changes to generalise it for any order, and to make it acceptable for REDUCE 3.4. The Fourier program followed the same pattern, but obviously used the `HSUB` operation as appropriate and the harmonic integration. It is very similar to the CAMAL program in [BF72].

The disturbing function was calculated to orders 2, 4 and 6 using Cambridge LISP on an HLH Orion 1/05 (Intergraph Clipper), with the three programs α) REDUCE 3.4, β) REDUCE 3.4 + Camal Linear Module and γ) REDUCE 3.4 + Camal AVL Module. The timings for CPU seconds (excluding garbage collection time) are summarised the following table:

Order of DDF	REDUCE	Camal Linear	Camal Tree
2	23.68	11.22	12.9
4	429.44	213.56	260.64
6	> 7500	3084.62	3445.54

If these numbers are normalised so REDUCE calculating the DDF is 100 units for each order the table becomes

Order of DDF	REDUCE	Camal Linear	Camal Tree
2	100	47.38	54.48
4	100	49.73	60.69
6	100	< 41.13	< 45.94

From this we conclude that a doubling of speed is about correct, and although the balanced tree system is slower as the problem size increases the gap between it and the simpler linear system is narrowing.

It is disappointing that the ratio is not better, nor the absolute time less. It is worth noting in this context that Jefferys claimed that the sixth order DDF took 30s on a CDC6600 with TRIGMAN in 1970 [Jef70], and Barton and Fitch took about 1s for the second order DDF on TITAN with CAMAL [BF72]. A closer look at the relative times for individual sections of the program shows that the substitution case of replacing a polynomial variable by a Fourier series is only marginally faster than the simple REDUCE program. In the DDF program this operation is only used once in a major form, substituting into the Legendre polynomials, which have been previously calculated by Rodrigues formula. This suggests that we replace this with the recurrence relationship.

Making this change actually slows down the normal REDUCE by a small amount but makes a significant change to the Fourier module; it reduces the run time for the 6th order DDF from 3084.62s to 2002.02s. This gives some indication of the problems with benchmarks. What is clear is that the current implementation of substitution of a Fourier series for a polynomial variable is inadequate.

20.8.7 Conclusion

The Fourier module is far from complete. The operations necessary for the solution of Duffing's and Hill's equations are not yet written, although they should not cause much problem. The main defficiency is the treatment of series truncation; at present it relies on the REDUCE `wtlevel` mechanism, and this seems too coarse for efficient truncation. It would be possible to re-write the polynomial manipulator as well, while retaining the REDUCE syntax, but that seems rather more than one would hope.

The real failure so far is the large time lag between the REDUCE-based system on a modern workstation against a mainframe of 25 years ago running a special system. The CAMAL Disturbing function program could calculate the tenth order with a maximum of 32K words (about 192Kbytes) whereas this system failed to calculate the eigth order in 4Mbytes (taking 2000s before failing). I have in my archives the output from the standard CAMAL test suite, which includes a sixth order DDF on an IBM 370/165 run on 2 June 1978, taking 22.50s and using a maximum of 15459 words of memory for heap — or about 62Kbytes. A rough estimate is that the Orion 1/05 is comparable in speed to the 360/165, but with more real memory and virtual memory.

However, a simple Fourier manipulator has been created for REDUCE which performs between twice and three times the speed of REDUCE using pattern matching. It has been shown that this system is capable of performing the calculations of celestial mechanics, but it still seriously lags behind the efficiency of the specialist systems of twenty years before. It is perhaps fortunate that it was not been possible to compare it with a modern specialist system.

There is still work to do to provide a convenient user interface, but it is intended to develop the system in this direction. It would be pleasant to have again a system of the efficiency of CAMAL(F).

I would like to thank Codemist Ltd for the provision of computing resources for this project, and David Barton who taught be so much about Fourier series and celstial mechanics. Thank are also due to the National Health Service, without whom this work and paper could not have been produced.

Appendix: The DDF Function

```
array p(n/2+2);
harmonic u,v,w,x,y,z;
weight e=1, b=1, d=1, a=1;

%% Generate Legendre Polynomials to sufficient order
for i:=2:n/2+2 do <<
```

```
  p(i):=(h*h-1)^i;
  for j:=1:i do p(i):=df(p(i),h)/(2j)
>>;

%%%%%%%%%%%%%%%% Step1: Solve Kepler equation
bige := fourier 0;
for k:=1:n do <<
  wtlevel k;
  bige:=fourier e * hsub(fourier(sin u), u, u, bige, k);
>>;

%% Ensure we do not calculate things of
%% too high an order
wtlevel n;

%%%%%%%%%%%%%%%% Step 2: Calculate r/a
%%%%%%%%%%%%%%%%         in terms of e and l
dd:=-e*e; hh:=3/2; j:=1; cc := 1;
for i:=1:n/2 do <<
  j:=i*j; hh:=hh-1; cc:=cc+hh*(dd^i)/j
>>;
bb:=hsub(fourier(1-e*cos u), u, u, bige, n);
aa:=fourier 1+hdiff(bige,u);
ff:=hint(aa*aa*fourier cc,u);

%%%%%%%%%%%%%%%% Step 3: a/r and f
uu := hsub(bb,u,v); uu:=hsub(uu,e,b);
vv := hsub(aa,u,v); vv:=hsub(vv,e,b);
ww := hsub(ff,u,v); ww:=hsub(ww,e,b);

%%%%%%%%%%%%%%%% Step 4: Substitute f and f' into S
yy:=ff-ww; zz:=ff+ww;
xx:=hsub(fourier((1-d*d)*cos(u)),u,u-v+w-x-y+z,yy,n)+
    hsub(fourier(d*d*cos(v)),v,u+v+w+x+y-z,zz,n);

%%%%%%%%%%%%%%%% Step 5: Calculate R
zz:=bb*vv; yy:=zz*zz*vv;

on fourier;
for i := 2:n/2+2 do <<
  wtlevel n+4-2i; p(i) := hsub(p(i), h, xx) >>;

wtlevel n;
for i:=n/2+2 step -1 until 3 do
```

```
    p(n/2+2):=fourier(a*a)*zz*p(n/2+2)+p(i-1);
yy*p(n/2+2);
```

20.9 CANTENS: A Package for Manipulations and Simplifications of Indexed Objects

This package creates an environment which allows the user to manipulate and simplify expressions containing various indexed objects like tensors, spinors, fields and quantum fields.

Author: Hubert Caprasse

20.9.1 Introduction

CANTENS is a package that creates an environment inside REDUCE which allows the user to manipulate and simplify expressions containing various indexed objects like tensors, spinors, fields and quantum fields. Briefly said, it allows him

- to define generic indexed quantities which can eventually depend implicitly or explicitly on any number of variables;
- to define one or several affine or metric (sub-)spaces, and to work within them without difficulty;
- to handle dummy indices and simplify adequatly expressions which contain them.

Beside the above features, it offers the user:

1. Several invariant elementary tensors which are always used in the applications involving the use of indexed objects like `delta`, `epsilon`, `eta`, and the generalized delta function.
2. The possibility to define any metric and to make it bloc-diagonal if he wishes to.
3. The capability to symmetrize or antisymmetrize any expression.
4. The possibility to introduce any kind of symmetry (even partial symmetries) for the indexed objects.
5. The choice to work with commutative, non-commutative or anticommutative indexed objects.

In this package, one cannot find algorithms or even specific objects (i.e. like the covariant derivative or the SU(3) group structure constants) which are of used either in nuclear and particle physics. The objective of the package is simply to allow the user to easily formulate *his algorithms* in the *notations he likes most*. The package

is also conceived so as to minimize the number of new commands. However, the large number of new capabilities inherently implies that quite a substantial number of new functions and commands must be used. On the other hand, in order to avoid too many error or warning messages the package assumes, in many cases, that the user is reponsible of the consistency of its inputs. The author is aware that the package is still perfectible and he will be grateful to all people who shall spare some time to communicate bugs or suggest improvements.

The documentation below is separated into four sections. In the first one, the space(s) properties and definitions are described.

In the second one, the commands to geberate and handle generic indexed quantities (called abusively tensors) are illustrated. The manipulation and control of free and dummy indices is discussed.

In the third one, the special tensors are introduced and their properties discussed especially with respect to their ability to work simultaneously within several subspaces.

The last section, which is also the most important, is devoted entirely to the simplification function `canonical`. This function originates from the package DUMMY and has been substantially extended . It takes account of all symmetries, make dummy summations and seeks a "canonical" form for any tensorial expression. Without it, the present package would be much less useful.

When CANTENS is loaded, the packages ASSIST and DUMMY are also loaded.

20.9.2 Handling of space(s)

One can work either in a *single* space environment or in a multiple space environment. After the package is loaded, the single space environment is set and a unique space is defined. It is euclidian, and has a symbolic dimension equal to `dim`. The single space environment is determined by the switch `onespace` which is turned on. One can verify the above assertions as follows `wholespace_dim`:

```
onespace ?; => yes

wholespace_dim ?; => dim

signature ?; => 0
```

One can introduce a pseudoeuclidian metric for the above space by the command `signature` and verify that the signature is indeed 1:

```
signature 1;
```

```
signature ?; => 1
```

In principle the signature may be set to any positive integer. However, presently, the package cannot handle signatures larger than 1. One gets the Minkowski-like space metric

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

which corresponds to the convention of high energy physicists. It is possible to change it into the astrophysicists convention using the command `global_sign`:

```
global_sign ?; => 1

global_sign (-1);

global_sign ?; => -1
```

This means that the actual metric is now $(-1, 1, 1, 1)$. The space dimension may, of course, be assigned at will using the function `wholespace_dim`. Below, it is assigned to 4:

```
wholespace_dim 4; ==> 4
```

When the switch `onespace` is turned off, the system *assumes* that this default space is non-existent and, therefore, that the user is going to define the space(s) in which he wants to work. Unexpected error messages will occur if it is not done. Once the switch is turned off many more functions become active. A few of them are available in the algebraic mode to allow the user to properly conctruct and control the properties of the various (sub-)spaces he is going to define and, also, to assign symbolic indices to some of them.

`define_spaces` is the space constructor and `wholespace` is a reserved identifier which is meant to be the name of the global space if subspaces are introduced. Suppose we want to define a unique space, we can choose for its any name but choosing `wholespace` will be more efficient. On the other hand, it leaves open the possibility to introduce subspaces in a more transparent way. So one writes, for instance,:

```
define_spaces wholespace=

        {6,signature=1,indexrange=0 .. 5}; ==>t
```

The arguments inside the list, assign respectively the dimension, the signature and

the range of the numeric indices which is allowed. Notice that the range starts from 0 and not from 1. This is made to conform with the usual convention for spaces of signature equal to 1. However, this is not compulsory. Notice that the declaration of the indexrange may be omitted if this is the only defined space. There are two other options which may replace the signature option, namely `euclidian` and `affine`. They have both an obvious significance.

In the subsequent example, an eleven dimension global space is defined and two subspaces of this space are specified. Notice that no indexrange has been declared for the entire space. However, the indexrange declaration is compulsory for subspaces otherwise the package will improperly work when dealing with numeric indices.

```
define_spaces wholespace={11,signature=1}; ==> t

define_spaces mink=

          {4,signature=1,indexrange=0 .. 3}; ==> t

define_spaces eucl=

          {6,euclidian,indexrange=4 .. 9}; ==> t
```

To remind ones the space context in which one is working, the use of the function `show_spaces` is required. Its output is an *algebraic value* from which the user can retrieve all the informations displayed. After the declarations above, this function gives:

```
show_spaces(); ==>

          {{wholespace,11,signature=1}

           {mink,4,signature=1,indexrange=0..3},

           {eucl,6,euclidian,indexrange=4..9}}
```

If an input error is made or if one wants to change the space framework, one cannot directly redefine the relevant space(s). For instance, the input

```
define_spaces eucl=

           {7,euclidian,indexrange=4 .. 9}; ==>

     *** Warning: eucl cannot be (or is already)
```

```
                    defined as space identifier
          t
```

which aims to fill all dimensions present in `wholespace` tells that the space `eucl` cannot be redefined. To redefine it effectively, one is to *remove* the existing definition first using the function `rem_spaces` which takes any number of space-names as its argument. Here is the illustration:

```
rem_spaces eucl; ==> t

show_spaces(); ==>

          {{wholespace,11,signature=1},

           {mink,4,signature=1,indexrange=0..3}}

define_spaces eucl=

          {7,euclidian,indexrange=4 .. 10}; ==> t

show_spaces(); ==>

          {{wholespace,11,signature=1},

           {mink,4,signature=1,indexrange=0..3},

           {eucl,7,euclidian,indexrange=4..10}}
```

Here, the user is entirely responsible of the coherence of his construction. The system does *NOT* verify it but will incorrectly run if there is a mistake at this level.

When two spaces are direct product of each other (as the color and Minkowski spaces in quantum chromodynamics), it is not necessary to introduce the global space `wholespace`.

"Tensors" and symbolic indices can be declared to belong to a specific space or subspace. It is in fact an essential ingredient of the package and make it able to handle expressions which involve quantities belonging to several (sub-)spaces or to handle bloc-diagonal "tensors". This will be discussed in the next section. Here, we just mention how to declare that some set of symbolic indices belong to a specific (sub-)space or how to declare them to belong to any space. The relevant command is `mk_ids_belong_space` whose syntax is

`mk_ids_belong_space(`⟨*list of indices*⟩`,`⟨*space*⟩ | ⟨*subspace identifier*⟩`)`

For example, within the above declared spaces one could write:

```
mk_ids_belong_space({a0,a1,a2,a3},mink); ==> t

mk_ids_belong_space({x,y,z,u,v},eucl); ==> t
```

The command `mk_ids_belong_anyspace` allows to remake them usable either in `wholespace` if it is defined or in anyone among the defined spaces. For instance, the declaration:

```
mk_ids_belong_anyspace a1,a2; ==> t
```

tells that a1 and a2 belong either to `mink` or to `eucl` or to `wholespace`.

20.9.3 Generic tensors and their manipulation

Definition

The generic tensors handled by CANTENS are objects much more general than usual tensors. The reason is that they are not supposed to obey well defined transformation properties under a change of coordinates. They are only indexed quantities. The indices are either contravariantly (upper indices) or covariantly (lower indices) placed. They can be symbolic or numeric. When a given index is found both in one upper and in one lower place, it is supposed to be summed over all space-coordinates it belongs to viz. it is a *dummy* index and *automatically recognized* as such. So they are supposed to obey the summation rules of tensor calculus. This why and only why they are called tensors. Moreover, aside from indices they may also depend implicitly or explicitly on any number of *variables*. Within this definition, tensors may also be spinors, they can be non-commutative or anticommutative, they may also be algebra generators and represent fields or quantum fields.

Implications of `TENSOR` declaration

The procedure `tensor` which takes an arbitrary number of identifiers as argument defines them as operator-like objects which admit an arbitrary number of indices. Each component has a formal character and may or may not belong to a specific (sub-)space. Numeric indices are also allowed. The way to distinguish upper and lower indices is the same as the one in the package EXCALC e.g. $-a$ is a lower index and a is an upper index. A special printing function has been created so as to mimic as much as possible the way of writing such objects on a sheet of paper. Let us illustrate the use of `tensor`:

```
tensor te; ==> t

te(3,a,-4,b,-c,7); ==>
                           3 a   b   7
                      te
                             4   c


te(3,a,{x,y},-4,b,-c,7); ==>

           3 a   b   7
        te             (x,y)
               4   c


te(3,a,-4,b,{u,v},-c,7); ==>

           3 a   b   7
        te             (u,v)
               4   c


te({x,y}); ==>  te(x,y)
```

Notice that the system distinguishes indices from variables on input solely on the basis that the user puts variables *inside a list*.

The dependence can also be declared implicit through the REDUCE command `depend` which is generalized so as to allow to declare a tensor to depend on another tensor irrespective of its components. It means that only *one* declaration is enough to express the dependence with respect to *all its components*. A simple example:

```
tensor te,x;

depend te,x;

df(te(a,-b),x(c)); ==>

             a    c
        df(te   ,x )
               b
```

Therefore, when *all* objects are tensors, the dependence declaration is valid for all indices.

One can also avoid the trouble to place the explicit variables inside a list if one declare them as variables through the command `make_variables`. This property can also be removed[17] using `remove_variables`:

```
make_variables x,y; ==> t

te(x,y); ==> te(x,y)


te(x,y,a); ==>

           a
       te  (x,y)


remove_variables x; ==> t


te(x,y,a); ==>

           x a
       te    (y)
```

If one does that one must be careful not to substitute a number to such declared variables because this number would be considered as an index and no longer as a variable. So it is only useful for *formal* variables.

A tensor can be easily eliminated using the function `rem_tensor`. It has the syntax

```
rem_tensor t1,t2,t3 ....;
```

Dummy indices recognition

For all individual tensors met by the evaluator, the system will analyse the written indices and will detect those which must be considered dummy according to the usual rules of tensor calculus. Those indices will be given the `dummy` property and will no longer be allowed to play the role of *free* indices unless the user

[17] One important feature of this package is its *reversibility* viz. it gives the user the means to erase its previous operations at any time. So, most functions described below do possess "removing" action companions.

removes this dummy property. In that way, the system checks immediately the consistency of an input. Three functions are at the disposal of the user to control dummy indices. They are `dummy_indices`, `rem_dummy_indices` The following illustrates their use as well as the behaviour of the system:

```
dummy_indices(); ==> {} % In a fresh environment

te(a,b,-c,-a); ==>

          a b
        te
              c a

dummy_indices(); ==> {a}

te(a,b,-c,a); ==>

***** ((c)(a b a)) are inconsistent lists of indices

          % a cannot be found twice as an upper index

te(a,b,-b,-a); ==>

          a b
       te
              b a

dummy_indices(); ==> {b,a}

te(d,-d,d); ==>

  ***** ((d)(d d)) are inconsistent lists of indices

dummy_indices(); ==> {d,b,a}

rem_dummy_indices d; ==> t

dummy_indices(); ==> {b,a}

te(d,d); ==>

          d d
        te              %  This is allowed again.
```

```
dummy_indices(); ==> {b,a}

rem_dummy_indices(); ==> t

dummy_indices(); ==> {}
```

Other verifications of coherence are made when space specifications are introduced both in the ON and OFF `onespace` environment. We shall discuss them later.

Substitutions, assignments and rewriting rules

The user must be able to manipulate and give specific characteristics to the generic tensors he has introduced. Since tensors are essentially REDUCE operators, the usual commands of the system are available. However, some limitations are implied by the fact that indices and, especially numeric indices, must always be properly recognized before any substitution or manipulation is done. We have gathered below a set of examples which illustrate all the "delicate" points. First, the substitutions:

```
sub(a=-c,te(a,b)); ==>

           b
       te
         c

sub(a=-1,te(a,b)); ==>

           b
       te
         1

sub(a=-0,te(a,b)); ==>

         0 b
       te    % sub has replaced -0 by 0. wrong!

sub(a=-!0,te(a,b)); ==>

           b
       te    % right
         0
```

The substitution of an index by -0 is the *only one* case where there is a problem.

The operator `sub` replaces -0 by 0 because it does not recognize 0 as an index of course. Such a recognition is context dependent and implies a modification of `sub` for this *single* exceptional case. Therefore,we have opted, not do do so and to use the index 0 which is simply !0 instead of 0.

Second, the assignments. Here, we advise the user to rely on the operator==[18] instead of the operator `:=`. Again, the reason is to avoid the problem raised above in the case of substitutions. `:=` does not evaluate its left hand side so that -0 is not recognized as an index and simplified to 0 while the == operator evaluates both its left and right hand sides and does recognize it. The disadvantage of == is that it demands that a second assignement on a given component be made only after having suppressed *explicitly* the first assignement. This is done by the function `rem_value_tens` which can be applied on any component. We stress, however, that if one is willing to use -!0 instead of -0 as the lower 0 index, the use of := is perfectly legitimate:

```
te({x,y},a,-0)==x*y*te(a,-0); ==>

          a
        te   *x*y
            0

te({x,y},a,-0); ==>

          a
        te   *x*y
            0

te({x,y},a,0); ==>


           a 0
        te    (x,y)


te({x,y},a,-0)==x*y*te(a,-0); ==>

              a
    ***** te    *x*y invalid as setvalue kernel
                0

rem_value_tens te({x,y},a,-0);
```

[18] See the ASSIST documentation for its description.

```
te({x,y},a,-0); ==>

          a
       te    (x,y)
            0

te({x,y},a,-0)==(x+y)*te(a,-0); ==>

      a
    te   *(x + y)
        0
```

In the elementary application below, the use of a tensor avoids the introduction of two different operators and makes the calculation more readable.

```
te(1)==sin th * cos phi; ==> cos(phi)*sin(th)

te(-1)==sin th * cos phi; ==> cos(phi)*sin(th)

te(2)==sin th * sin phi; ==> sin(phi)*sin(th)

te(-2)==sin th * sin phi; ==> sin(phi)*sin(th)

te(3)==cos th ; ==> cos(th)

te(-3)==cos th ; ==> cos(th)

for i:=1:3 sum te(i)*te(-i); ==>

     2        2          2           2        2
cos(phi) *sin(th)  + cos(th)  + sin(phi) *sin(th)

rem_value_tens te;

te(2); ==>

         2
       te
```

There is no difference in the manipulation of numeric indices and numeric *tensor* indices. The function `rem_value_tens` when applied to a tensor prefix suppresses the value of *all its components*. Finally, there is no "interference" with i as a dummy index and i as a numeric index in a loop.

Third, rewriting rules. They are either global or local and can be used as in REDUCE. Again, here, the -0 index problem exists each time a substitution by the index -0 must be made in a template.

```
% LET:

    let te({x,y},-0)=x*y;

    te({x,y},-0); ==> x*y

    te({x,y},+0); ==>

              0
           te  (x,y)


    te({x,u},-0); ==>

           te  (x,u)
              0

% FOR ALL .. LET:

    for all x,a let te({x},a,-b)=x*te(a,-b);

    te({u},1,-b); ==>

             1
           te   *u
                b

    te({u},c,-b); ==>

             c
           te   *u
                b

    te({u},b,-b); ==>

             b
           te   *u
                b

    te({u},a,-a); ==>
```

```
              a
          te    (u)
               a

   for all x,a clear te({x},a,-b);

   te({u},c,-b); ==>

              c
          te    (u)
               b

   for all a,b let te({x},a,-b)=x*te(a,-b);

   te({x},c,-b); ==>

             c
          te   *x
              b

   te({x},a,-a); ==>

             a
          te   *x
              a

 % The index -0 problem:

   te({x},a,-0); ==> % -0 becomes +0 in the template

              a
          te    (x)  %  the rule does not apply.
               0

   te({x},0,-!0); ==>

             0
          te   *x      % here it applies.
              0

% WHERE:
```

```
rul:={te(~a) => sin a}; ==>

                    a
        rul := {te  => sin(a)}


te(1) where rul; ==> sin(1)

te(1); ==>

           1
         te


% with variables:

rul1:={te(~a,{~x,~y}) => x*y*sin(a)}; ==>


                    ~a
        rul1 := {te   (~x,~y) => x*y*sin(a)}


te(a,{x,y}) where rul1; ==> sin(a)*x*y


te({x,y},a) where rul1; ==> sin(a)*x*y


rul2:={te(-~a,{~x,~y}) => x*y*sin(-a)};


      rul2 := {te   (~x,~y) => x*y*sin(-a)}
                  ~a

te(-a,{x,y}) where rul2; ==> -sin(a)*x*y


te({x,y},-a) where rul2; ==> -sin(a)*x*y
```

Notice that the position of the list of variables inside the rule may be chosen at will. It is an irrelevant feature of the template. This may be confusing, so, we advise to

write the rules not as above but placing the list of variables *in front of all indices* since it is in that canonical form which it is written by the simplification function of individual tensors.

Behaviour under space specifications

The characteristics and the behaviour of generic tensors described up to now are independent of all space specifications. They are complete as long as we confine to the default space which is active when starting CANTENS. However, as soon as some space specification is introduced, it has some consequences one the generic tensor properties. This is true both when `onespace` is switched ON or OFF. Here we shall describe how to deal with these features.

When `onespace` is ON, if the space dimension is set to an integer, numeric indices of any generic tensors are forced to be less or equal that integer if the signature is 0 or less than that integer if the signature is equal to 1. The following illustrates what happens.

```
on onespace;

wholespace_dim 4; ==> 4

signature 0; ==> 0

te(3,a,-b,7); ==> ***** numeric indices out of range

te(3,a,-b,3); ==>


          3 a   3
       te
             b


te(4,a,-b,4); ==>

          4 a   4
       te
             b

sub(a=5,te(3,a,-b,3));

             ==> ***** numeric indices out of range
```

```
signature 1; ==> 1

% Now indices range from 0 to 3:

te(4,a,-b,4);

             ==> ***** numeric indices out of range

te(0,a,-b,3); ==>


          0 a   3
        te
              b
```

When `onespace` is OFF, many more possibilities to control the input or to give specific properties to tensors are open. For instance, it is possible to declare that a tensor belongs to one of them. It is also possible to declare that some indices belongs to one of them. It is even possible to do that for *numeric* indices thanks to the declaration indexrange included optionally in the space definition generated by `define_spaces`. First, when `onespace` is OFF, the run equivalent to the previous one is like the following:

```
off onespace;

define_spaces wholespace={6,signature=1); ==> t

show_spaces(); ==> {{wholespace,6,signature=1}}

make_tensor_belong_space(te,wholespace);

                                    ==> wholespace

te(4,a,-b,6); ==>

               ***** numeric indices out of range

te(4,a,-b,5); ==>


          4 a   5
        te
              b
```

```
rem_spaces wholespace;

define_spaces wholespace={4,euclidean}; ==> t

te(a,5,-b); ==> ***** numeric indices out of range

te(a,4,-b); ==>


          a 4
        te
              b

define_spaces eucl={1,signature=0}; ==> t

show_spaces(); ==>

  {{wholespace,5,signature=1},

           {eucl,1,signature=0}}

make_tensor_belong_space(te,eucl); ==> eucl

te(1); ==>

          1
        te

te(2); ==> ***** numeric indices out of range

te(0); ==>

          0
        te
```

In the run, the new function `make_tensor_belong_space` has been used. One may be surprised that `te(0)` is allowed in the end of the previous run and, indeed, it is incorrect that the system allows *two* different components to `te`. This is due to an incomplete definition of the space. When one deals with spaces of integer dimensions, if one wants to control numeric indices correctly *when* `onespace` is switched off *one must also give the indexrange*. So the previous run must be corrected to

```
define_spaces eucl=
```

```
            {1,signature=0,indexrange=1 .. 1}; ==> t

make_tensor_belong_space(te,eucl); ==> eucl

te(0); ==>

  ***** numeric indices do not belong to (sub)-space

te(1); ==>

         1
       te

te(2); ==>

  ***** numeric indices do not belong to (sub)-space
```

Notice that the error message has also changed accordingly. So, now one can even constrain the 0 component to belong to an euclidian space.

Let us go back to symbolic indices. By default, any symbolic index belongs to the global space or to all defined partial spaces. In many cases, this is, of course, not consistent. So, the possibility exists to declare that one or several indices belong to a specific (sub-)space. To this end, one is to use the function `mk_ids_belong_space`. Its syntax is

`mk_ids_belong_space`(⟨*list of indices*⟩, ⟨*(sub-)space identifier*⟩)

The function `mk_ids_belong_anyspace` whose syntax is the same do the reverse operation.

Combined with the declaration `make_tensor_belong_space`, it allows to express all problems which involve tensors belonging to different spaces and do the dummy summations correctly. One can also define a tensor which has a "bloc-diagonal" structure. All these features are illustrated in the next sections which describe specific tensors and the properties of the extended function `canonical`.

20.9.4 Specific tensors

The means provided in the two previous subsection to handle generic tensors already allow to construct any specific tensor we may need. That the package contains a certain number of them is already justified on the level of conviviality. However, a more important justification is that some basic tensors are so universaly and

frequently used that a careful programming of these improves considerably the robustness and the efficiency of most calculations. The choice of the set of specific tensors is not clearcut. We have tried to keep their number to a minimum but, experience, may lead us extend it without dificulty. So, up to now, the list of specific tensors is:

- `delta` tensor,
- `eta` Minkowski tensor,
- `epsilon` tensor,
- `del` generalised delta tensor,
- `metric` generic tensor metric.

It is important to realize that the typewriter font names in the list are *keywords* for the corresponding tensors and do not necessarily correspond to their *actual names*. Indeed, the choice of the names of particular tensors is left to the user. When startting CANTENS specific tensors are NOT available. They must be activated by the user using the function `make_partic_tens` whose syntax is:

`make_partic_tens(`⟨*tensor name*⟩ , ⟨*keyword*⟩);

The name chosen may be the same as the keyword. As we shall see, it is never needed to define more than one `delta` tensor but it is often needed to define several `epsilon` tensors. Hereunder, we describe each of the above tensors especially their behaviour in a multi-space environment.

DELTA tensor

It is the simplest example of a bloc-diagonal tensor we mentioned in the previous section. It can also work in a space which is a direct product of two spaces. Therefore, one never needs to introduce more than one such tensor. If one is working in a graphic environment, it is advantageous to choose the keyword as its name. Here we choose `delt`. We illustrate how it works when the switch `onespace` is successively switched ON and OFF.

```
on onespace;

make_partic_tens(delt,delta); ==> t

delt(a,b); ==>

    ***** bad choice of indices for DELTA tensor

% order of upper and lower indices irrelevant:
```

```
delt(a,-b); ==>

            a
        delt
            b

delt(-b,a); ==>

            a
        delt
            b

delt(-a,b); ==>

            b
        delt
            a


wholespace_dim ?; ==> dim

delt(1,-5); ==> 0

% dummy summation done:

delt(-a,a); ==> dim


wholespace_dim 4; ==> 4

delt(1,-5); ==> ***** numeric indices out of range


wholespace_dim 3; ==> 3

delt(-a,a); ==> 3
```

There is a peculiarity of this tensor, viz. it can serve to represent the Dirac *delta function* when it has no indices and an explicit variable dependency as hereunder

```
delt({x-y}) ==> delt(x-y)
```

Next we work in the context of several spaces:

```
off onespace;

define_spaces wholespace={5,signature=1}; ==> t

% we need to assign delta to wholespace
% when it exists:

make_tensor_belong_space(delt,wholespace);

delt(a,-a); ==> 5

delt(0,-0); ==>1

rem_spaces wholespace; ==> t

define_spaces wholespace={5,signature=0}; ==> t

delt(a,-a); ==> 5

delt(0,-a); ==>

    ***** bad value of indices for DELTA tensor
```

The checking of consistency of chosen indices is made in the same way as for generic tensor. In fact, all the previous functions which act on generic tensors may also affect, in the same way, a specific tensor. For instance, it was compulsory to explicitly tell that we want `delt` to belong to the wholespace `make_tensor_belong_space`, otherwise `delt` would remain defined on the *default space*. In the next sample run, we display the bloc-diagonal property of the delta tensor.

```
    onespace ?; ==> no

    rem_spaces wholespace; ==> t

    define_spaces wholespace={10,signature=1}$

    define_spaces d1={5,euclidian}$

    define_spaces d2={2,euclidian}$


    mk_ids_belong_space({a},d1); ==> t
```

```
    mk_ids_belong_space({b},d2); ==> t

% c belongs to wholespace so:

    delt(c,-b); ==>

                c
           delt
                b

    delt(c,-c); ==> 10


    delt(b,-b); ==> 2

    delt(a,-a); ==> 5

% this is especially important:

    delt(a,-b); ==> 0
```

The bloc-diagonal property of `delt` is made active under two conditions. The first is that the system knows to which space it belongs, the second is that indices must be declared to belong to a specific space. To enforce the same property on a generic tensor, we have to make the `make_bloc_diagonal` declaration:

```
    make_bloc_diagonal t1,t2, ...;
```

and to make it active, one proceeds as in the above run. Starting from a fresh environment, the following sample run is illustrative:

```
off onespace;

define_spaces wholespace={6,signature=1}$

define_spaces mink={4,signature=1,indexrange=0 .. 3}$

define_spaces eucl={3,euclidian,indexrange=4 .. 6}$

tensor te;

make_tensor_belong_space(te,eucl); ==> eucl
```

```
% the key declaration:

make_bloc_diagonal te; ==> t


% bloc-diagonal property activation:

mk_ids_belong_space({a,b,c},eucl); ==> t

mk_ids_belong_space({m1,m2},mink); ==> t


te(a,b,m1); ==> 0

te(a,b,m2); ==> 0

% bloc-diagonal property suppression:

mk_ids_belong_anyspace a,b,c,m1,m2; ==> t


te(a,b,m2); ==>

          a b m2
        te
```

ETA Minkowski tensor

The use of `make_partic_tens` with the keyword `eta` allows to create a Minkowski diagonal metric tensor in a one or multi-space context either with the convention of high energy physicists or in the convention of astrophysicists. Any `eta`-like tensor is assumed to work within a space of signature 1. Therefore, if the space whose metric, it is supposed to describe has a signature 0, an error message follows if one is working in an ON `onespace` context and a warning when in an OFF `onespace` context. Illustration:

```
on onespace;

make_partic_tens(et,eta); ==> t

signature 0; ==> 0;

et(-b,-a); ==>
```

```
   *****  signature must be equal to 1 for ETA tensor

off onespace;

et(a,b); ==>

   ***  ETA tensor not properly assigned to a space

% it is then evaluated to zero:

    0

on onespace;

signature 1; ==> 1

et(-b,-a); ==>

       et
         a b
```

Since `et(a,-a)` is evaluated to the corresponding `delta` tensor, one cannot define properly an `eta` tensor without a simultaneous introduction of a `delta` tensor. Otherwise one gets the following message:

```
et(a,-a); ==> *****  no name found for (delta)
```

So we need to issue, for instance,

```
make_partic_tens(delta,delta); ==> t
```

The value of its diagonal elements depends on the chosen global sign. The next run illustrates this:

```
global_sign ?; ==> 1

et(0,0); ==> 1

et(3,3); ==>  - 1

global_sign(-1); ==> -1
```

```
et(0,0); ==>  - 1

et(3,3); ==> 1
```

The tensor is of course symmetric . Its indices are checked in the same way as for a generic tensor. In a multi_space context, the `eta` tensor must belong to a well defined space of signature 1:

```
off onespace;

define_spaces wholespace={4,signature=1}$

make_tensor_belong_space(et,wholespace)$

et(a,-a); ==> 4
```

If the space to which `et` belongs to is a subspace, one must also take care to give a space-identity to dummy indices which may appear inside it. In the following run, the index `a` belongs to `wholespace` if it is not told to the system that it is a dummy index of the space `mink`:

```
make_tensor_belong_anyspace et; ==> t

rem_spaces wholespace; ==> t

define_spaces wholespace={8,signature=1}; ==> t

define_spaces mink={5,signature=1}; ==> t

make_tensor_belong_space(et,mink); ==> mink

% a sits in wholespace:

et(a,-a); ==> 8

mk_ids_belong_space({a},mink); ==> t

% a sits in mink:

et(a,-a); ==>  5
```

EPSILON tensors

It is an antisymmetric tensor which is the invariant tensor for the unitary group transformations in n-dimensional complex space which are continuously connected to the identity transformation. The number of their indices are always stricty equal to the number of space dimensions. So, to each specific space is associated a specific `epsilon` tensor. Its properties are also dependent on the signature of the space. We describe how to define and manipulate it in the context of a unique space and, next, in a multi-space context.

Switch `onespace` is on

The use of `make_partic_tens` places it, by default, in an euclidian space if the signature is 0 and in a Minkowski-type space if the signature is 1. For higher signatures it is not constructed. For a space of symbolic dimension, the number of its indices is not constrained. When it appears inside an expression, its indices are *all* currently upper or lower indices. However, the system allows for mixed positions of the indices. In that case, the output of the system is changed compared to the input only to place all contravariant indices to the left of the covariant ones.

```
make_partic_tens(eps,epsilon); ==> t

eps(a,d,b,-g,e,-f); ==>

              a d b e
        - eps
                     g f

eps(a,d,b,-f,e,-f); ==> 0

% indices have all the same variance:

eps(-b,-a); ==>

        - eps
             a b


signature ?; ==> 0

eps(1,2,3,4); ==> 1

eps(-1,-2,-3,-4); ==> 1
```

```
wholespace_dim 3; ==> 3

eps(-1,-2,-3); ==> 1

eps(-1,-2,-3,-4); ==>

       ***** numeric indices out of range

eps(-1,-2,-3,-3); ==>

       ***** bad number of indices for (eps) tensor

eps(a,b); ==>

       ***** bad number of indices for (eps) tensor

eps(a,b,c); ==>

          a b c
       eps


eps(a,b,b); ==> 0
```

When the signature is equal to 1, it is known that there exists two *conventions* which are linked to the chosen value 1 or -1 of the $(0, 1, \ldots, n)$ component. So, the sytem does evaluate all components in terms of the $(0, 1, \ldots, n)$ upper index component. It is left to the user to assign it to 1 or -1.

```
signature 1; ==> 1

eps(0,1,2); ==>

          0 1 2
       eps

eps(-0,-1,-2); ==>

          0 1 2
       eps

wholespace_dim 4; ==> 4
```

```
    eps(0,1,2,3); ==>

              0 1 2 3
          eps

    eps(-0,-1,-2,-3); ==>

                 0 1 2 3
           - eps

% change of the global_sign convention:

    global_sign(-1);

    wholespace_dim 3; ==> 3

% compare with second input:

    eps(-0,-1,-2); ==>

                 0 1 2
           - eps
```

Switch onespace is off

As already said, several epsilon tensors may be defined. They *must* be assigned to a well defined (sub-)space otherwise the simplifying function `canonical` will not properly work. The set of epsilon tensors defined associated to their space-name may be retrieved using the function `show_epsilons`. An important word of caution here. The output of this function does NOT show the epsilon tensor one may have defined in the ON `onespace` context. This is so because the default space has *NO* name. Starting from a fresh environment, the following run illustrates this point:

```
    show_epsilons(); ==> {}

    onespace ?; ==> yes

    make_partic_tens(eps,epsilon); ==> t

    show_epsilons(); ==> {}
```

To make the `epsilon` tensor defined in the single space environment visible in

the multi-space environment, one needs to associate it to a space. For example:

```
off onespace;

define_spaces wholespace={7,signature=1}; ==> t

show_epsilons(); ==> {}   % still invisible

make_tensor_belong_space(eps,wholespace); ==>

                                          wholespace

show_epsilons(); ==> {{eps,wholespace}}
```

Next, let us define an *additional* `epsilon`-type tensor:

```
define_spaces eucl={3,euclidian}; ==> t

make_partic_tens(ep,epsilon); ==>

        *** Warning: ep MUST belong to a space
              t

make_tensor_belong_space(ep,eucl); ==> eucl


show_epsilons(); ==> {{ep,eucl},{eps,wholespace}}

% We show that it is indeed working inside eucl:

ep(-1,-2,-3); ==> 1

ep(1,2,3); ==> 1

ep(a,b,c,d); ==>

      ***** bad number of indices for  (ep)  tensor

ep(1,2,4); ==>

      ***** numeric indices out of range
```

As previously, the discrimation between symbolic indices may be introduced by assigning them to one or another space :

```
rem_spaces wholespace;

define_spaces wholespace={dim,signature=1}; ==> t

mk_ids_belong_space({e1,e2,e3},eucl); ==> t

mk_ids_belong_space({a,b,c},wholespace); ==> t

ep(e1,e2,e3); ==>

         e1 e2 e3
       ep           % accepted

ep(e1,e2,z); ==>

         e1 e2 z
       ep            % accepted because z
                     % not attached to a space.

ep(e1,e2,a);==>

    ***** some indices are not in the space of ep


eps(a,b,c); ==>

          a b c
       eps           % accepted because *symbolic*
                     % space dimension.
```

epsilon-like tensors can also be defined on disjoint spaces. The subsequent sample run starts from the environment of the previous one. It suppresses the space wholespace as well as the space-assignment of the indices a,b,c. It defines the new space mink. Next, the previously defined eps tensor is attached to this space. ep remains unchanged and e1,e2,e3 still belong to the space eucl.

```
rem_spaces wholespace; ==> t

make_tensor_belong_anyspace eps; ==> t

show_epsilons(); ==> {{ep,eucl}}

show_spaces(); ==> {{eucl,3,signature=0}}
```

```
mk_ids_belong_anyspace a,b,c; ==> t

define_spaces mink={4,signature=1}; ==> t

show_spaces(); ==>

       {{eucl,3,signature=0},

        {mink,4,signature=1}}

make_tensor_belong_space(eps,mink); ==> mink

show_epsilons(); ==> {{eps,mink},{ep,eucl}}

eps(a,b,c,d); ==>

       a b c d
    eps

eps(e1,b,c,d); ==>

   ***** some indices are not in the space of eps

ep(e1,b,c,d); ==>

   ***** bad number of indices for  (ep)  tensor

ep(e1,b,c); ==>

         b c e1
       ep


ep(e1,e2,e3); ==>

         e1 e2 e3
       ep
```

DEL generalized delta tensor

The generalized delta function comes from the contraction of two epsilons. It is totally antisymmetric. Suppose its name has been chosen to be gd, that the space to which it is attached has dimension n while the name of the chosen delta tensor

is δ, then one can define it as follows:

$$\mathrm{gd}^{a_1,a_2,\ldots,a_n}_{b_1,b_2,\ldots,b_n} = \begin{vmatrix} \delta^{a_1}_{b_1} & \delta^{a_1}_{b_2} & \ldots & \delta^{a_1}_{b_n} \\ \delta^{a_2}_{b_1} & \delta^{a_2}_{b_2} & \ldots & \delta^{a_2}_{b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \delta^{a_n}_{b_1} & \delta^{a_n}_{b_1} & \ldots & \delta^{a_n}_{b_1} \end{vmatrix}$$

It is, in general uneconomical to explicitly write that determinant except for particular *numeric* values of the indices or when almost all upper and lower indices are recognized as dummy indices. In the sample run below, `gd` is defined as the generalized delta function in the default space. The main automatic evaluations are illustrated. The indices which are summed over are always simplified:

```
     onespace ? ==> yes

     make_partic_tens(delta,delta); ==> t

     make_partic_tens(gd,del); ==> t

% immediate simplifications:

     gd(1,2,-3,-4); ==> 0

     gd(1,2,-1,-2); ==> 1

     gd(1,2,-2,-1); ==> -1   % antisymmetric

     gd(a,b,-a,-b);

        ==> dim*(dim - 1) % summed over dummy indices

     gd(a,b,c,-a,-d,-e); ==>

                   b c
                 gd    *(dim - 2)
                   d e

     gd(a,b,c,-a,-d,-c); ==>

                       b      2
                 delta  *(dim  - 3*dim + 2)
                       d

% no simplification:
```

```
gd(a,b,c,-d,-e,-f); ==>

              a b c
            gd
              d e f
```

One can force evaluation in terms of the determinant in all cases. To this end, the switch `exdelt` is provided. It is initially `OFF`. Switching it `ON` will most often give inconveniently large outputs:

```
on exdelt;

gd(a,b,c,-d,-e,-f); ==>

     a       b       c          a       b       c
delta  *delta  *delta   - delta  *delta  *delta
     d       e       f          d       f       e

        a       b       c
 - delta  *delta  *delta
        e       d       f

        a       b       c
 + delta  *delta  *delta
        e       f       d

        a       b       c
 + delta  *delta  *delta
        f       d       e

        a       b       c
 - delta  *delta  *delta
        f       e       d
```

In a multi-space environment, it is never necessary to define several such tensor. The reason is that `canonical` uses it always from the contraction of a pair of `epsilon`-like tensors. Therefore the control of indices is already done, the space-dimension in which `del` is working is also well defined.

METRIC tensors

Very often, one has to define a specific metric. The metric-type of tensors include all generic properties. The first one is their symmetry, the second one is their equality to the delta tensor when they get mixed indices, the third one is their optional bloc-diagonality. So, a metric (generic) tensor is generated by the declaration

make_partic_tens(⟨*tensor-name*⟩,metric);

By default, when one is working in a multi-space environment, it is defined in wholespace One uses the usual means of REDUCE to give it specific values. In particular, the metric 'delta' tensor of the euclidian space can be defined that way. Implicit or explicit dependences on variables are allowed. Here is an illustration in the single space environment:

```
make_partic_tens(g,metric); ==> t

make_partic_tens(delt,delta); ==> t

onespace ?; ==> yes

g(a,b); ==>

           a b
          g

g(b,a); ==>

           a b
          g

g(a,b,c); ==>

  ***** bad choice of indices for a METRIC tensor


g(a,b,{x,y}); ==>

            a b
          g    (x,y)

g(a,-b,{x,z}); ==>
```

```
     a
delt
     b
```

```
let g({x,y},1,1)=1/2(x+y);
```

```
g({x,y},1,1); ==>
```

```
 x + y
-------
   2
```

```
rem_value_tens g({x,y},1,1);
```

```
g({x,y},1,1); ==>
```

```
  1 1
g    (x,y)
```

20.9.5 The simplification function `canonical`

Tensor expressions

Up to now, we have described the behaviour of individual tensors and how they simplify themselves whenever possible. However, this is far from being sufficient. In general, one is to deal with objects which involve several tensors together with various dummy summations between them. We define a tensor expression as an arbitrary multivariate polynomial. The indeterminates of such a polynomial may be either an indexed object, an operator, a variable or a rational number. A tensor-type indeterminate cannot appear to a degree larger than one except if it is a trace. The following is a tensor expression:

```
aa:= delt({x - y})*delt(a, - g)*delt(d, - g)

  *delt(g, -r)

  *eps( - d, - e, - f)*eps(a,b,c)*op(x,y) + 1; ==>
```

```
                          a      d      g
aa := delt(x - y)*delt   *delt  *delt  *eps
                          g      g      r     d e f


            a b c
        *eps     *op(x,y) + 1
```

In the above expression, `delt` and `eps` are, respectively, the `delta` and the `epsilon` tensors, `op` is an operator. and `delt(x-y)` is the Dirac delta function. Notice that the above expression is not cohérent since the first term has a variance while the second term is a scalar. Moreover, the dummy index `g` appears *three* times in the first term. In fact, on input, each factor is simplified and each factor is checked for coherence not more. Therefore, if a dummy summation appears inside one factor, it will be done whenever possible. Hereunder `delt(a,-a)` is summed over:

```
sub(g=a,aa); ==>

                        a      d              a b c
       delt(x - y)*delt   *delt  *eps     *eps
                        r      a     d e f

                   *op(x,y)*dim + 1
```

The use of `canonical`

`canonical` is an offspring of the function with the same name of the package DUMMY. It applies to tensor expressions as defined above. When it acts, this functions has several features which are worth to realise:

1. It tracks the free indices in each term and checks their identity. It identifies and verify the coherence of the various dummy index summations.
2. Dummy indices summations are done on tensor products whenever possible since it recognises the particular tensors defined above or defined by the user.
3. It seeks a canonical form for the various simplified terms, makes the comparison between them. In that way it maximises simplifications and generates a canonical form for the output polynomial.

Its capabilities have been extended in four directions:

- It is able to work within *several* spaces.

- It manages correctly expressions where formal tensor *derivatives* are present[19].
- It takes into account all symmetries even if partial.
- As its parent function, it can deal with non-commutative and anticommutative indexed objects. So, Indexed objects may be spinors or quantum fields.

We describe most of these features in the rest of this documentation.

Check of tensor indices

Dummy indices for individual tensors are kept in the memory of the system. If they are badly distributed over several tensors, it is `canonical` which gives an error message:

```
tensor te,tf; ==> t

bb:=te(a,b,b)*te(-b); ==>

                    a b b
       bb := te *te
                b

canonical bb; ==>

***** ((b)(a b b)) are inconsistent lists of indices

aa:=te(b,-c)*tf(b,-c); ==>

               b     b
       aa := te   *tf     % b and c are free.
                  c     c

canonical aa; ==>

         b     b
       te   *tf
           c     c

bb:=te(a,c,b)*te(-b)*tf(a)$
```

[19] In DUMMY it does not take them into account

```
canonical bb; ==>


          a c     b   a
        te     *te *tf
              b

delt(a,-a); ==> dim  % a is now a dummy index


canonical bb; ==>

        ***** wrong use of indices (a)
```

The message of canonical is clear, the first sublist contains the list of all lower indices, and the second one the list of all upper indices. The index `b` is repeated *three* times. In the second example, `b` and `c` are considered as free indices, so they may be repeated. The last example shows the interference between the check on individual tensors and the one of canonical. The use of `a` as dummy index inside `delt` does no longer allow `a` to be used as a free index in expression `bb`. To be usable, one must explicitly remove it as dummy index using `rem_dummy_indices`. In the fourth example there are no problems as `b` and `c` are both *free* indices. `canonical` checks that in a tensor polynomial all do possess the *same* variance:

```
aa:=te(a,c)+x^2; ==>

                a c    2
        aa := te    + x

canonical aa; ==>

        ***** scalar added with tensor(s)

aa:=te(a,b)+tf(a,c); ==>

                a b     a c
        aa := te    + tf

canonical aa; ==>

       ***** mismatch in free indices :  ((a c) (a b))
```

In the message the first two lists of incompatible indices are explicitly indicated. So, it is not an exhaustive message and a more complete correction may be needed. Of course, no message of that kind appears if the indices are inside ordinary operators[20]

```
dummy_names b; ==> t

cc:=op(b)*op(a,b,b); ==> cc := op(a,b,b)*op(b)

canonical cc; ==> op(a,b,b)*op(b)

clear_dummy_names; ==> t
```

Position and renaming of dummy indices

For a specific tensor, contravariant dummy indices are place in front of covariant ones. This already leads to some useful simplifications. For instance:

```
pp:=te(a,-a)+te(-a,a)+1; ==>

                a     a
    pp := te     + te     + 1
            a           a

canonical pp; ==>

          a
    2*te     + 1
           a

pp:=te(a,-a)+te(-b,b); ==>

                b     a
    pp := te     + te
            b           a


canonical pp; ==>

          a
    2*te
           a
```

[20] This is the case inside the DUMMY package.

```
pp:=te(r,a,c,d,-a,f)+te(r,-b,c,d,b,f); ==>

                r   c d b f     r a c d   f
      pp := te            + te
                  b                     a

canonical pp; ==>


          r a c d   f
     2*te
                a
```

In the second and third example, there is also a renaming of the dummy variable `b` whih becomes `a`. There is a loophole at this point. For some expressions one will never reach a stable expression. This is the case for the following very simple monom:

```
tensor nt; ==> t

a1:=nt(-a,d)*nt(-c,a); ==>

           d     a
       nt   *nt
         a     c

canonical a1; ==>


              a d
       nt   *nt
         c a

a12:=a1-canonical a1; ==>

                d     a           a d
       a12 := nt   *nt    - nt   *nt
                a     c       c a


canonical a12; ==>
```

```
          d     a           a d
- nt    *nt    + nt    *nt    % changes sign.
     a     c        c a
```

In the above example, no canonical form can be reached. When applied twice on the tensor monom `a1` it gives back `a1`!

No change of dummy index position is allowed if a tensor belongs to an `affine` space. With the tensor polynomial `pp` introduced above one has:

```
off onespace;

define_spaces aff={dd,affine}; ==> t

make_tensor_belong_space(te,aff); ==> aff

mk_ids_belong_space({a,b},aff); ==> t

canonical pp; ==>

          r   c d a f     r a c d   f
       te            + te
           a                     a
```

The renaming of `b` has been made however.

Contractions and summations with particular tensors

This is a central part of the extension of `canonical`. The required contractions and summations can be done in a multi-space environment as well in a single space environment.

The case of `delta`

Dummy indices are recognized contracted and summed over whenever possible:

```
aa:=delt(a,-b)*delt(b,-c)*delt(c,-a) + 1; ==>

                 a      b      c
       aa := delt  *delt  *delt   + 1
                 b      c      a

canonical aa; ==> dim + 1
```

```
aa:=delt(a,-b)*delt(b,-c)*delt(c,-d)*te(d,e)$

canonical aa; ==>

          a e
       te
```

`canonical` will not attempt to make contraction with dummy indices included inside ordinary operators:

```
operator op;

aa:=delt(a,-b)*op(b,b)$


canonical aa; ==>

           a
      delt  *op(b,b)
           b

dummy_names b; ==> t

canonical aa; ==>

          a
    delta  *op(b,b)
          b
```

The case of `eta`

First, we introduce `eta`:

```
make_partic_tens(eta,eta); ==> t

signature 1; ==> 1 % necessary

aa:=delta(a,-b)*eta(b,c); ==>

                  a     b c
       aa := delt  *eta
                  b
```

```
canonical aa; ==>

          a c
       eta


canonical(eta(a,b)*eta(-b,c)); ==>

          a c
       eta


canonical(eta(a,b)*eta(-b,-c)); ==>

           a
       delt
           c

canonical(eta(a,b)*eta(-b,-a)); ==> dim

canonical (eta(-a,-b)*te(d,-e,f,b)); ==>

         d   f
       te
           e   a


aa:=eta(a,b)*eta(-b,-c)*te(-a,c)+1; ==>

                         a b     c
       aa := eta   *eta   *te    + 1
                 b c          a

canonical aa; ==>

         a
       te    + 1
           a

aa:=eta(a,b)*eta(-b,-c)*delta(-a,c)+

    1+eta(a,b)*eta(-b,-c)*te(-a,c)$
```

```
canonical aa; ==>

          a
        te    + dim + 1
            a
```

Let us add a generic metric tensor:

```
aa:=g(a,b)*g(-b,-d); ==>

                  a b
       aa := g   *g
              b d

canonical aa; ==>

           a
       delt
           d

aa:=g(a,b)*g(c,d)*eta(-c,-e)*eta(e,f)*te(-f,g); ==>

                       e f  a b  c d     g
       aa := eta   *eta   *g   *g   *te
                c e                   f

canonical aa; ==>

        a b   d g
       g   *te
```

The case of `epsilon`

The epsilon tensor plays an important role in many contexts. `canonical` realises the contraction of two epsilons if and only if they belong to the same space. The proper use of `canonical` on expressions which contains it requires a preliminary definition of the tensor `DEL`. When the signature is 0; the contraction of two epsilons gives a `del`-like tensor. When the signature is equal to 1, it is equal to *minus* a `del`-like tensor. Here we choose 1 for the signature and we work in a single space. We define the `del` tensor:

```
on onespace;

wholespace_dim dim; ==> dim
```

```
make_partic_tens(gd,del); ==> t

signature 1; ==> 1
```

We define the `epsilon` tensor and show how `canonical` contracts expression containing *two*[21] of them:

```
aa:=eps(a,b)*eps(-c,-d); ==>

                        a b
       aa := eps    *eps
                 c d

canonical aa; ==>

             a b
        - gd
             c d


aa:=eps(a,b)*eps(-a,-b); ==>

                        a b
       aa := eps    *eps
                 a b

canonical aa; ==> dim*( - dim + 1)

on exdelt;

gd(-a,-b,a,b); ==> dim*(dim - 1)

aa:=eps(a,b,c)*eps(-b,-d,-e)$

canonical aa; ==>

            a      c               a      c
       delt  *delt  *dim - 2*delt  *delt  -
            d      e               d      e
```

[21] No contractions are done on expressions containing three or more epsilons which sit in the *same* space. We are not sure whether it is useful to be more general than we are presently.

```
                a      c              a       c
      - delt   *delt  *dim + 2*delt   * delt
                e      d              e       d
```

Several expressions which contain the epsilon tensor together with other special tensors are given below as examples to treat with `canonical`:

```
aa:=eps( - b, - c)*eta(a,b)*eta(a,c); ==>

                    a b    a c
       eps   *eta   *eta
          b c

canonical aa; ==> 0

aa:=eps(a,b,c)*te(-a)*te(-b); ==> % te is generic.

                   a b c
       aa := eps     *te *te
                        a   b

canonical aa; ==> 0

tensor tf,tg;

aa:=eps(a,b,c)*te(-a)*tf(-b)*tg(-c)

    + eps(d,e,f)*te(-d)*tf(-e)*tg(-f); ==>

 canonical aa; ==>

              a b c
       2*eps     *te *tf *tg
                    a   b   c

aa:=eps(a,b,c)*te(-a)*tf(-c)*tg(-b)

    + eps(d,e,f)*te(-d)*tf(-e)*tg(-f)$

canonical aa; ==> 0
```

Since `canonical` is able to work inside several spaces, we can introduce also several epsilons and make the relevant simplifications on each (sub)-spaces. This is the goal of the next illustration.

```
off onespace;

define_spaces wholespace=

                  {dim,signature=1}; ==> t

define_spaces subspace=

                   {3,signature=0}; ==> t

show_spaces(); ==>

       {{wholespace,dim,signature=1},

        {subspace,3,signature=0}}

make_partic_tens(eps,epsilon); ==> t

make_partic_tens(kap,epsilon); ==> t

make_tensor_belong_space(eps,wholespace);

                         ==> wholespace

make_tensor_belong_space(kap,subspace);

                         ==> subspace

show_epsilons(); ==>

             {{eps,wholespace},{kap,subspace}}

off exdelt;

aa:=kap(a,b,c)*kap(-d,-e,-f)*eps(i,j)*eps(-k,-l)$

canonical aa; ==>

            a b c    i j
        - gd      *gd
            d e f    k l
```

If there are no index summation, as in the expression above, one can develop both terms into the delta tensor with `exdelt` switched ON. In fact, the previous calcu-

lation is correct *only if there are no dummy index* inside the two gd's. If some of the indices are dummy, then we must take care of the respective spaces in which the two gd tensors are considered. Since, the tensor themselves do not belong to a given space, the space identification can only be made through the indices. This is enough since the delta-like tensor is bloc-diagonal. With aa the result of the above illustration, one gets, for example,:

```
mk_ids_belong_space({a,b,c,d,e,f},wholespace)$

mk_ids_belong_space({i,j,k,l},subspace)$

sub(d=a,e=b,k=i,aa); ==>

             c      j          2
       2*delt  *delt  *( - dim  + 3*dim - 2)
             f      l

sub(k=i,l=j,aa); ==>
                                a b c
                        - 6*gd
                                d e f
```

canonical and symmetries

Most of the time, indexed objects have some symmetry property. When this property is either full symmetry or antisymmetry, there is no difficulty to implement it using the declarations symmetric or antisymmetric of REDUCE. However, most often, indexed objects are neither fully symmetric nor fully antisymmetric: they have *partial* or *mixed* symmetries . In the DUMMY package, the declaration symtree allows to impose such type of symmetries on operators. This command has been improved and extended to apply to tensors. In order to illustrate it, we shall take the example of the wellknown Riemann tensor in general relativity. Let us remind the reader that this tensor has four indices. It is separately *antisymmetric* with respect to the interchange of the first two indices and with respect to the interchange of the last two indices. It is *symmetric* with respect to the interchange of the first two and the last two indices. In the illustration below, we show how to express this and how canonical is able to recognize mixed symmetries:

```
tensor r; ==> t

symtree(r,{!+,{!-,1,2},{!-,3,4}});

rem_dummy_indices a,b,c,d; % free indices
```

```
ra:=r(b,a,c,d); ==>

              b a c d
       ra := r

canonical ra; ==>

           a b c d
        - r

ra:=r(c,d,a,b); ==>
                                 c d a b
                        ra := r

canonical ra; ==>

        a b c d
       r


canonical r(-c,-d,a,b); ==>

        a b
       r
            c d

r(-c,-c,a,b); ==>  0


ra:=r(-c,-d,c,b); ==>

                   c b
       ra := r
              c d

canonical ra; ==>

           b c
        - r
              c d
```

In the last illustration, contravariant indices are placed in front of covariant indices and the contravariant indices are transposed. The superposition of the two partial

symmetries gives a minus sign.

There exists an important (though natural) restriction on the use of SYMTREE which is linked to the algorithm itself: Integer used to localize indices must start from 1, be *contiguous* and monotoneously increasing. For instance, one is not allow to introduce

```
symtree(r,{!*,{!+,1,3},{!*,2,4}});

symtree(r,{!*,{!+,1,2},{!*,4,5}};

symtree(r,{!*,{!-,1,3},{!*,2}});
```

but the subsequent declarations are allowed:

```
symtree(r,{!*,{!+,1,2},{!*,3,4}});

symtree(r,{!*,{!+,1,2},{!*,3,4,5}});

symtree(r,{!*,{!-,1,2},{!*,3}});
```

The first declaration endows `r` with a *partial* symmetry with respect to the first two indices.

A side effect of `symtree` is to restrict the number of indices of a generic tensor. For instance, the second declaration in the above illustrations makes `r` depend on 5 indices as illustrated below:

```
symtree(r,{!*,{!+,1,2},{!*,3,4,5}});

canonical r(-b,-a,d,c); ==>

     ***** Index '5' out of range for

           ((minus b) (minus a) d c) in nth

canonical r(-b,-a,d,c,e); ==>

          d c e
    r                   % correct
     a b

canonical r(-b,-a,d,c,e,g); ==>

          d c e
```

```
          r          % The sixth index is forgotten!
           a b
```

Finally, the function `remsym` applied on any tensor identifier removes all symmetry properties.

Another related question is the frequent need to symmetrize a tensor polynomial. To fulfill it, the function `symmetrize` of the package ASSIST has been improved and generalised. For any kernel (which may be either an operator or a tensor) that function generates

- the sum over the cyclic permutations of indices,
- the symetric or antisymetric sums over all permutations of the indices.

Moreover, if it is given a list of indices, it generates a new list which contains sublists which contain the relevant permutations of these indices

```
symmetrize(te(x,y,z,{v}),te,cyclicpermlist); ==>


        x y z          y z x          z x y
    te       (v) + te       (v) + te       (v)

symmetrize(te(x,y),te,permutations); ==>

          x y     y x
       te    + te

symmetrize(te(x,y),te,permutations,perm_sign); ==>

          x y     y x
       te    - te

symmetrize(te(y,x),te,permutations,perm_sign); ==>

          x y     y x
     - te    + te
```

If one wants to symmetrise an expression which is not a kernel, one can also use `symmetrize` to obtain the desired result as the next example shows:

```
ex:=te(a,-b,c)*te1(-a,-d,-e); ==>

               a   c
```

```
       ex := te     *te1
                b      a d e

ll:=list(b,c,d,e)$ % the chosen relevant indices

lls:=symmetrize(ll,list,cyclicpermlist); ==>

   lls := {{b,c,d,e},{c,d,e,b},{d,e,b,c},{e,b,c,d}}

% The sum over the cyclic permutations is:

excyc:=for each i in lls sum

            sub(b=i.1,c=i.2,d=i.3,e=i.4,ex);  ==>


          a   c              a   d
excyc := te     *te1     + te     *te1
            b      a d e       c      a e b


            a   e              a   b
        + te     *te1      + te      *te1
              d      a b c       e       a c d
```

`canonical` and tensor derivatives

Only ordinary (partial) derivatives are fully correctly handled by `canonical`. This is enough, to explicitly construct covariant derivatives. We recognize here that extensions should still be made. The subsequent illustrations show how `canonical` does indeed manage to find the canonical form and simplify expressions which contain derivatives. Notice, the use of the (modified) `depend` command.

```
on onespace;

tensor te,x; ==> t

depend te,x;


aa:=df(te(a,-b),x(-b))-df(te(a,-c),x(-c))$
```

```
canonical aa; ==> 0

make_partic_tens(eta,eta); ==>  t

signature 1;

aa:=df(te(a,-b),x(-b))$

aa:=aa*eta(-a,-d);

                  a
      aa := df(te   ,x )*eta
                   b  b     a d

canonical aa; ==>

              a  a
      df(te   ,x )
           d
```

In the last example, after contraction, the covariant dummy index `b` has been changed into the contravariant dummy index `a`. This is allowed since the space is metric.

20.10 CDE: A Package for Integrability of PDEs

Author: Raffaele Vitolo

We describe CDE, a REDUCE package devoted to differential-geometric computations on Differential Equations (DEs, for short).

We will give concrete recipes for computations in the geometry of differential equations: higher symmetries, conservation laws, Hamiltonian operators and their Schouten bracket, recursion operators. All programs discussed here are shipped together with the CDE sources, inside the REDUCE sources. The mathematical theory on which computations are based can be found in refs. [BCD+99, KKV04]. We invite the interested reader to have a look at the website [gde] which contains useful resources in the above mathematical area. There is also a book on integrable systems and CDE [KVV18] with more examples and more detailed explanations about the mathematical part.

20.10.1 Introduction: why CDE?

CDE is a REDUCE package for differential-geometric computations for DEs. The package aims at defining differential operators in total derivatives and computing with them. Such operators are called *$\mathcal{C}$-differential operators* (see [BCD+99]).

CDE depends on the REDUCE package CDIFF for constructing total derivatives. CDIFF was developed by Gragert and Kersten for symmetry computations in DEs, and later extended by Roelofs and Post.

There are many software packages that can compute symmetries and conservation laws; many of them run on Mathematica or Maple. Those who run on REDUCE were written by M. C. Nucci [Nuc92, Nuc96], F. Oliveri (ReLie, [Oli]), F. Schwartz (SPDE, 20.56), T. Wolf (APPLYSYM (20.1) and CONLAW in the official REDUCE distribution, [Wol02a, Wol95, BW95, BW92]).

The development of CDE started from the idea that a computer algebra tool for the investigation of integrability-related structures of PDEs still does not exist in the public domain. We are only aware of a Mathematica package that may find recursion operators under quite restrictive hypotheses [BH10].

CDE is especially designed for computations of integrability-related structures (such as Hamiltonian, symplectic and recursion operators) for systems of differential equations with an arbitrary number of independent or dependent variables. On the other hand CDE is also capable of (generalized) symmetry and conservation laws computations. The aim of this guide is to introduce the reader to computations of integrability related structures using CDE.

The current version of CDE, 3.0, has the following features:

1. It is able to do standard computations in integrable systems like determining systems for generalized symmetries and conservation laws. However, CDE has not been programmed with this purpose in mind.

2. CDE is able to compute linear overdetermined systems of partial differential equations whose solutions are Hamiltonian, symplectic or recursion operators. Such equations may be solved by different techniques; one of the possibilities is to use CRACK, a REDUCE package for solving overdetermined systems of PDEs [WB].

3. CDE can compute linearization (or Fréchet derivatives) of vector functions and adjoints of differential operators.

4. CDE can do calculations on supermanifolds. In particular it can compute variational derivatives of superdensities, linearization of superfunctions, adjoint of superdifferential operators. Some of the features are still undocumented as they will be published in forthcoming papers.

5. CDE is able to compute Schouten brackets between local multivectors. This can be used *eg* to check Hamiltonianity of an operator or to check their compatibility.

6. CDE can calculate the Schouten bracket of weakly nonlocal differential operators; these are distinguished pseudodifferential operators in one independent variable. The algorithm has been published in [CLV20], while a user guide is being written and will appear soon (interested readers can ask the author of CDE for details).

At the moment the papers [FPV14, FPV16, KKVV09, KVV12, PV15, SV14] have been written using CDE, and more research by CDE on integrable systems is in progress.

The readers are warmly invited to send questions, comments, etc., both on the computations and on the technical aspects of installation and configuration of REDUCE, to the author of this document.

Acknowledgements. I'd like to thank Paul H.M. Kersten, who explained to me how to use the original CDIFF package for several computations of interest in the Geometry of Differential Equations. When I started writing CDE I was substantially helped by A.C. Norman in understanding many features of Reduce which were deeply hidden in the source code and not well documented. This also led to writing a manual of Reduce's internals for programmers [NV]. Moreover, I'd like to thank the developers of the REDUCE mailing list for their prompt replies with solutions to my problems. On the mathematical side, I would like to thank J.S. Krasil'shchik and A.M. Verbovetsky for constant support and stimulating discussions which led me to write the software. Thanks are also due to B.A. Dubrovin,

M. Casati, E.V. Ferapontov, P. Lorenzoni, M. Marvan, V. Novikov, A. Savoldi, A. Sergyeyev, M.V. Pavlov for many interesting discussions.

20.10.2 Jet space of even and odd variables, and total derivatives

The mathematical theory for jets of even (*ie* standard) variables and total derivatives can be found in [BCD+99, Olv93].

Let us consider the space $\mathbb{R}^n \times \mathbb{R}^m$, with coordinates (x^λ, u^i), $1 \leq \lambda \leq n$, $1 \leq i \leq m$. We say x^λ to be *independent variables* and u^i to be *dependent variables*. Let us introduce the *jet space* $J^r(n, m)$. This is the space with coordinates (x^λ, u^i_σ), where u^i_σ is defined as follows. If $s\colon \mathbb{R}^n \to \mathbb{R}^m$ is a differentiable function, then

$$u^i_\sigma \circ s(x) = \frac{\partial^{|\sigma|}(u^i \circ s)}{(\partial x^1)^{\sigma_1} \cdots (\partial x^n)^{\sigma_n}}.$$

Here $\sigma = (\sigma_1, \ldots, \sigma_n) \in \mathbb{N}^n$ is a multiindex. We set $|\sigma| = \sigma_1 + \cdots + \sigma_n$. If $\sigma = (0, \ldots, 0)$ we set $u^i_\sigma = u^i$.

CDE is first of all a program which is able to create a *finite order jet space* inside REDUCE. To this aim, issue the command

```
load_package cde;
```

Then, CDE needs to know the variables and the maximal order of derivatives. The input can be organized as in the following example:

```
indep_var:={x,t}$
dep_var:={u,v}$
total_order:=10$
```

Here

- `indep_var` is the list of independent variables;
- `dep_var` is the list of dependent variables;
- `total_order` is the maximal order of derivatives.

Two more parameters can be set for convenience:

```
statename:="jetuv_state.red"$
resname:="jetuv_res.red"$
```

These are the name of the output file for recording the internal state of the program `cde.red` (and for debugging purposes), and the name of the file containing results of the computation.

The main routine in `cde.red` is called as follows:

```
cde({indep_var,dep_var,{},total_order},{})$
```

Here the two empty lists are placeholders; they are of interest for computations with odd variables/differential equations. The function `cde` defines derivative symbols of the type:

```
u_x,v_t,u_2xt,v_xt,v_2x3t,...
```

Note that the symbol `v_tx` does not exist in the jet space. Indeed, introducing all possible permutations of independent variables in indices would increase the complexity and slow down every computation.

Two lists generated by CDE can be useful: `all_der_id` and `all_odd_id`, which are, respectively, the lists of identifiers of all even and odd variables.

Other lists are generated by CDE, but they are accessible in REDUCE symbolic mode only. Please check the file `global.txt` to know the names of the lists.

It can be useful to inspect the output generated by the function `cde` and the above lists in particular. All that data can be saved by the function:

```
save_cde_state(statename)$
```

CDE has a few procedures involving the jet space, namely:

- `jet_fiber_dim(jorder)` returns the number of derivative coordinates u^i_σ with $|\sigma|$ equal to `jorder`;
- `jet_dim(jorder)` returns the number of derivative coordinates u^i_σ with $0 \leq |\sigma|$ and $|\sigma|$ equal to `jorder`;
- `selectvars(par,orderofder,depvars,vars)` returns all derivative coordinates (even if `par=0`, odd if `par=1`) of order `orderofder` of the list of dependent variables `depvars` which belong to the set of derivative coordinates `vars`.

The function `cde` defines total derivatives truncated at the order `total_order`. Their coordinate expressions are of the form

$$D_\lambda = \frac{\partial}{\partial x^\lambda} + u^i_{\sigma\lambda}\frac{\partial}{\partial u^i_\sigma}, \tag{20.46}$$

where σ is a multiindex.

The total derivative of an argument φ is invoked as follows:

```
td(phi,x,2);
td(phi,x,t,3);
```

the syntax closely follows REDUCE's syntax for standard derivatives `df`; the above expression translates to $D_x D_x \varphi$, or $D_{\{2,0\}}\varphi$ in multiindex notation.

When in total derivatives there is a coefficient of order higher than maximal this is replaced by the identifier `letop`, which is a function that depends on independent variables. If such a function (or its derivatives) appears during computations it is likely that we went too close to the highest order variables that we defined in the file. All results of computations are scanned for the presence of such variables by default, and if the presence of `letop` is detected the computation is stopped with an error message. This usually means that we need to extend the order of the jet space, just by increasing the number `total_order`.

Note that in the folder containing all examples there is also a shell script, `rrr.sh` (works only under `bash`, a GNU/Linux command interpreter) which can be used to run reduce on a given CDE program. When an error message about `letop` is issued the script reruns the computation with a new value of `total_order` one unity higher than the previous one.

The function `check_letop` checks an expression for the presence of `letop`. If you wish to switch off this kind of check in order to increase the speed, the switch `checkord` must be set off:

```
off checkord;
```

The computation of total derivatives of a huge expression can be extremely time and resources consuming. In some cases it is a good idea to disable the expansion of the total derivative and leave an expression of the type $D_\sigma \varphi$ as indicated. This is achieved by the command

```
noexpand_td();
```

If you wish to restore the default behaviour, do

```
expand_td();
```

CDE can also compute on jets of supermanifolds. The theory can be found in [IVV04, KKV04, KV11]. The input can be organized as follows:

```
indep_var:={x,t}$
dep_var:={u,v}$
odd_var:={p,q}
total_order:=10$
```

Here `odd_var` is the list of odd variables. The call

```
cde({indep_var,dep_var,odd_var,total_order},{})$
```

will create the jet space of the supermanifold described by the independent variables and the even and odd dependent variables, up to the order `total_order`. Total derivatives truncated at the order `total_order` will also include odd derivatives:

$$D_\lambda = \frac{\partial}{\partial x^\lambda} + u^i_{\sigma\lambda}\frac{\partial}{\partial u^i_\sigma} + p^i_{\sigma\lambda}\frac{\partial}{\partial p^i_\sigma}, \tag{20.47}$$

where σ is a multiindex. The considerations on expansion and `letop` apply in this case too.

Odd variables can appear in anticommuting products; this is represented as

```
ext(p,p_2xt),ext(p_x,q_t,q_x2t),...
```

where `ext(p_2xt,p) = - ext(p,p_2xt)` and the variables are arranged in a unique way terms of an internal ordering. Indeed, the internal representation of odd variables and their products (not intended for normal users!) is

```
ext(3,23),ext(1,3,5),...
```

as all odd variables and their derivatives are indexed by integers. Note that `p` and `ext(p)` are just the same. The odd product of two expressions φ and ψ is achieved by the CDIFF function

```
super_product(phi,psi);
```

The derivative of an expression φ with respect to an odd variable p is achieved by

```
df_odd(phi,p);
```

20.10.3 Differential equations in even and odd variables

We now give the equation in the form of one or more derivatives equated to right-hand side expressions. The left-hand side derivatives are called *principal*, and the remaining derivatives are called *parametric*[22]. Parametric coordinates are coordinates on the equation manifold and its differential consequences, and principal coordinates are determined by the differential equation and its differential consequences. For scalar evolutionary equations with two independent variables parametric derivatives are of the type $(u, u_x, u_{xx}, \ldots)$. Note that the system must be

[22] This terminology dates back to Riquier, see [Mar09]

in passive orthonomic form; this also means that there will be no nontrivial integrability conditions between parametric derivatives. (Lines beginning with % are comments for REDUCE.) The input is formed as follows (Burger's equation).

```
% left-hand side of the differential equation
principal_der:={u_t}$
% right-hand side of the differential equation
de:={u_2x+2*u*u_x}$
```

Systems of PDEs are input in the same way: of course, the above two lists must have the same length. See 20.10.11 for an example.

The main routine in `cde.red` is called as follows:

```
cde({indep_var,dep_var,{},total_order},
   {principal_der,de,{},{}})$
```

Here the three empty lists are placeholders; they are important for computations with odd variables. The function `cde` computes principal and parametric derivatives of even and odd variables, they are stored in the lists `all_parametric_der`, `all_principal_der`, `all_parametric_odd`, `all_principal_odd`.

The function `cde` also defines total derivatives truncated at the order `total_order` and restricted on the (even and odd) equation; this means that total derivatives are tangent to the equation manifold. Their coordinate expressions are of the form

$$D_\lambda = \frac{\partial}{\partial x^\lambda} + \sum_{u^i_\sigma \text{ parametric}} u^i_{\sigma\lambda} \frac{\partial}{\partial u^i_\sigma} + \sum_{p^i_\sigma \text{ parametric}} p^i_{\sigma\lambda} \frac{\partial}{\partial p^i_\sigma}, \tag{20.48}$$

where σ is a multiindex. It can happen that $u^i_{\sigma\lambda}$ (or $p^i_{\sigma\lambda}$) is principal and must be replaced with differential consequences of the equation. Such differential consequences are called *primary differential consequences*, and are computed; in general they will depend on other, possibly new, differential consequences, and so on. Such newly appearing differential consequences are called *secondary differential consequences*. If the equation is in passive orthonomic form, the system of all differential consequences (up to the maximal order `total_order`) must be solvable in terms of parametric derivatives only. The function *cde* automatically computes all necessary and sufficient differential consequences which are needed to solve the system. The solved system is available in the form of REDUCE let-rules in the variables `repprincparam_der` and `repprincparam_odd`.

The syntax and properties (expansion and `letop`) of total derivatives remain the same. For exmaple:

```
td(u,t);
```

returns

```
u_2x+2*u*u_x;
```

It is possible to deal with mixed systems on eve and odd variables. For example, in the case of Burgers equation we can input the linearized equation as a PDE on a new odd variable as follows (of course, in addition to what has been defined before):

```
odd_var:={q}$
principal_odd:={q_t}$
de_odd:={q_2x + 2*u_x*q + 2*u*q_x}$
```

The main routine in `cde.red` is called as follows:

```
cde({indep_var,dep_var,odd_var,total_order},
   {principal_der,de,principal_odd,de_odd})$
```

20.10.4 Calculus of variations

CDE can compute variational derivatives of any function (usually a Lagrangian density) or superfunction $\mathcal{L}$. We have the following coordinate expression

$$\frac{\delta \mathcal{L}}{\delta u^i} = (-1)^{|\sigma|} D_\sigma \frac{\partial \mathcal{L}}{\partial u^i_\sigma}, \quad \frac{\delta \mathcal{L}}{\delta p^i} = (-1)^{|\sigma|} D_\sigma \frac{\partial \mathcal{L}}{\partial p^i_\sigma} \tag{20.49}$$

which translates into the CDE commands

```
pvar_df(0,lagrangian_dens,ui);
pvar_df(1,lagrangian_dens,pi);
```

where

- the first argument can be 0 or 1 and is the parity of the variable `ui` or `pi`;
- `lagrangian_dens` is $\mathcal{L}$;
- `ui` or `pi` are the given dependent variables.

The Euler operator computes variational derivatives with respect to all even and odd variables in the jet space, and arranges them in a list of two lists, the list of even variational derivatives and the list of odd variational derivatives. The command is

```
euler_df(lagrangian_dens);
```

All the above is used in the definition of Schouten brackets, as we will see in Subsection 20.10.6.

20.10.5 $\mathcal{C}$-differential operators

Linearizing (or taking the Fréchet derivative) of a vector function that defines a differential equation yields a differential operator in total derivatives. This operator can be restricted to the differential equation, which may be regarded as a differential constraint; the kernel of the restricted operator is the space of all symmetries (including higher or generalized symmetries) [BCD+99, Olv93].

The formal adjoint of the linearization operator yields by restriction to the corresponding differential equation a differential operator whose kernel contains all characteristic vectors or generating functions of conservation laws [BCD+99, Olv93].

Such operators are examples of $\mathcal{C}$-differential operators. The (still incomplete) REDUCE implementation of the calculus of $\mathcal{C}$-differential operators is the subject of this section.

$\mathcal{C}$-differential operators

Let us consider the spaces

$$P = \{\varphi\colon J^r(n,m) \to \mathbb{R}^k\}, \qquad Q = \{\psi\colon J^r(n,m) \to \mathbb{R}^s\}.$$

A *$\mathcal{C}$-differential operator* $\Delta\colon P \to Q$ is defined to be a map of the type

$$\Delta(\varphi) = (\sum_{\sigma,i} a_i^{\sigma j} D_\sigma \varphi^i), \tag{20.50}$$

where $a_i^{\sigma j}$ are differentiable functions on $J^r(n,m)$, $1 \le i \le k$, $1 \le j \le s$. The *order* of δ is the highest length of σ in the above formula.

We may consider a generalization to *k-$\mathcal{C}$-differential operators* of the type

$$\Delta\colon P_1 \times \cdots \times P_h \to Q$$
$$\Delta(\varphi_1, \ldots, \varphi_h) = (\sum_{\sigma_1,\ldots,\sigma_h,i_1,\ldots,i_h} a_{i_1\cdots i_h}^{\sigma_1,\ldots,\sigma_h,\,j} D_{\sigma_1}\varphi_1^{i_1} \cdots D_{\sigma_h}\varphi_h^{i_h}), \tag{20.51}$$

where the enclosing parentheses mean that the value of the operator is a vector function in Q.

A $\mathcal{C}$-differential operator in CDE must be declared as follows:

```
mk_cdiffop(opname,num_arg,length_arg,length_target)
```

where

- `opname` is the name of the operator;

- `num_arg` is the number of arguments *eg* k in (20.51);
- `length_arg` is the list of lengths of the arguments: *eg* the length of the single argument of Δ (20.50) is k, and the corresponding list is `{k}`, while in (20.51) one needs a list of k items `{k_1,...,k_h}`, each corresponding to number of components of the vector functions to which the operator is applied;
- `length_target` is the numer of components of the image vector function.

The syntax for one component of the operator `opname` is

```
opname(j,i1,...,ih,phi1,...,phih)
```

The above operator will compute

$$\Delta(\varphi_1,\dots,\varphi_h) = \sum_{\sigma_1,\dots,\sigma_h} a_{i_1\cdots i_h}^{\sigma_1,\dots,\sigma_h,\,j} D_{\sigma_1}\varphi_1^{i_1}\cdots D_{\sigma_h}\varphi_h^{i_h}, \tag{20.52}$$

for fixed integer indices i_1,...,i_h and j.

There are several operations which involve differential operators. Obviously they can be summed and multiplied by scalars.

An important example of $\mathcal{C}$-differential operator is that of *linearization*, or *Fréchet derivative*, of a vector function

$$F\colon J^r(n,m) \to \mathbb{R}^k.$$

This is the operator

$$\ell_F\colon \varkappa \to P, \quad \varphi \mapsto \sum_{\sigma,i} \frac{\partial F^k}{\partial u_\sigma^i} D_\sigma \varphi^i,$$

where $\varkappa = \{\varphi\colon J^r(n,m) \to \mathbb{R}^m\}$ is the space of *generalized vector fields on jets* [BCD+99, Olv93].

Linearization can be extended to an operation that, starting from a k-$\mathcal{C}$-differential operator, generates a $k+1$-$\mathcal{C}$-differential operator as follows:

$$\ell_\Delta(p_1,\dots,p_k,\varphi) = \Big(\sum_{\sigma,\sigma_1,\dots,\sigma_k,i,i_1,\dots,i_k} \frac{\partial a_{i_1\cdots i_k}^{\sigma_1,\dots,\sigma_k,\,j}}{\partial u_\sigma^i} D_\sigma\varphi^i D_{\sigma_1}p_1^{i_1}\cdots D_{\sigma_k}p_k^{i_k}\Big)$$

(The above operation is also denoted by $\ell_{\Delta,p_1,\dots,p_k}(\varphi)$.)

At the moment, CDE is only able to compute the linearization of a vector function (Section 20.10.8).

Given a $\mathcal{C}$-differential operator Δ like in (20.50) we can define its *adjoint* as

$$\Delta^*((q_j)) = (\sum_{\sigma,i}(-1)^{|\sigma|}D_\sigma(a_i^{\sigma j}q_j)). \tag{20.53}$$

Note that the matrix of coefficients is transposed. Again, the coefficients of the adjoint operator can be found by computing $\Delta^*(x^\sigma e_j)$ for every basis vector e_j and every count x^σ, where $|\sigma| \leq r$, and r is the order of the operator. This operation can be generalized to $\mathcal{C}$-differential operators with h arguments.

At the moment, CDE can compute the adjoint of an operator with one argument (Section 20.10.8).

Now, consider two operators $\Delta\colon P \to Q$ and $\nabla\colon Q \to R$. Then the composition $\nabla \circ \Delta$ is again a $\mathcal{C}$-differential operator. In particular, if

$$\Delta(p) = (\sum_{\sigma,i} a_i^{\sigma j}D_\sigma p^i), \quad \nabla(q) = (\sum_{\tau,j} b_j^{\tau k}D_\tau q^j),$$

then

$$\nabla \circ \Delta(p) = (\sum_{\tau,j} b_j^{\tau k}D_\tau(\sum_{\sigma,i} a_i^{\sigma j}D_\sigma p^i))$$

This operation can be generalized to $\mathcal{C}$-differential operators with h arguments.

There is another important operation between $\mathcal{C}$-differential operators with h arguments: the *Schouten bracket* [BCD+99]. We will discuss it in next Subsection, in the context of another formalism, where it takes an easier form [KKV04].

20.10.6 $\mathcal{C}$-differential operators as superfunctions

In the papers [IVV04, KKV04] (and independently in [Get02]) a scheme for dealing with (skew-adjoint) variational multivectors was devised. The idea was that operators of the type (20.51) could be represented by homogeneous vector superfunctions on a supermanifold, where odd coordinates q^i_σ would correspond to total derivatives $D_\sigma\varphi^i$.

The isomorphism between the two languages is given by

$$\begin{aligned}\Big(\sum_{\sigma_1,\dots,\sigma_h,i_1,\dots,i_h} a_{i_1\cdots i_h}^{\sigma_1,\dots,\sigma_h,\,j} D_{\sigma_1}\varphi_1^{i_1}\cdots D_{\sigma_h}\varphi_h^{i_h}\Big)\\ \longrightarrow \Big(\sum_{\sigma_1,\dots,\sigma_h,i_1,\dots,i_h} a_{i_1\cdots i_h}^{\sigma_1,\dots,\sigma_h,\,j} q_{\sigma_1}^{i_1}\cdots q_{\sigma_h}^{i_h}\Big)\end{aligned} \tag{20.54}$$

where q^i_σ is the derivative of an odd dependent variable (and an odd variable itself).

A superfunction in CDE must be declared as follows:

```
mk_superfun(sfname,num_arg,length_arg,length_target)
```

where

- `sfname` is the name of the superfunction;
- `num_arg` is the degree of the superfunction *eg* h in (20.54);
- `length_arg` is the list of lengths of the arguments: *eg* the length of the single argument of Δ (20.50) is k, and the corresponding list is `{k}`, while in (20.51) one needs a list of k items `{k_1,...,k_h}`, each corresponding to number of components of the vector functions to which the operator is applied;
- `length_target` is the numer of components of the image vector function.

The above parameters of the operator `opname` are stored in the property list[23] of the identifier `opname`. This means that if one would like to know how many arguments has the operator `opname` the answer will be the output of the command

```
get('cdnarg,cdiff_op);
```

and the same for the other parameters.

The syntax for one component of the superfunction `sfname` is

```
sfname(j)
```

CDE is able to deal with $\mathcal{C}$-differential operators in both formalisms, and provides conversion utilities:

- `conv_cdiff2superfun(cdop,superfun)`
- `conv_superfun2cdiff(superfun,cdop)`

where in the first case a $\mathcal{C}$-differential operator `cdop` is converted into a vector superfunction `superfun` with the same properties, and conversely.

20.10.7 The Schouten bracket

We are interested in the operation of Schouten bracket between *variational multivectors* [IVV04]. These are differential operators with h arguments in $\varkappa$ with

[23] The property list is a lisp concept, see [NV] for details.

values in densities, and whose image is defined up to total divergencies:

$$\Delta\colon \varkappa \times \cdots \times \varkappa \to \{J^r(n,m) \to \lambda^n T^*\mathbb{R}^\kappa\}/\bar{d}(\{J^r(n,m) \to \lambda^{n-1} T^*\mathbb{R}^\kappa\}) \quad (20.55)$$

It is known [Get02, KKV04] that the Schouten bracket between two variational multivectors A_1, A_2 can be computed in terms of their corresponding superfunction by the formula

$$[A_1, A_2] = \left[\frac{\delta A_1}{\delta u^j}\frac{\delta A_2}{\delta p_j} + \frac{\delta A_2}{\delta u^j}\frac{\delta A_1}{\delta p_j}\right] \quad (20.56)$$

where $\delta/\delta u^i$, $\delta/\delta p_j$ are the variational derivatives and the square brackets at the right-hand side should be understood as the equivalence class up to total divergencies.

If the operators A_1, A_2 are compatible, *ie* $[A_1, A_2] = 0$, the expression (20.56) must be a total derivative. This means that:

$$[A_1, A_2] = 0 \quad \Leftrightarrow \quad \mathcal{E}\left(\frac{\delta A_1}{\delta u^j}\frac{\delta A_2}{\delta p_j} + \frac{\delta A_2}{\delta u^j}\frac{\delta A_1}{\delta p_j}\right) = 0. \quad (20.57)$$

If A_1 is an h-vector and A_2 is a k-vector the formula (20.56) produces a $(h+k-1)$-vector, or a $\mathcal{C}$-differential operator with $h+k-1$ arguments. If we would like to check that this multivector is indeed a total divergence, we should apply the Euler operator, and check that it is zero. This procedure is considerably simpler than the analogue formula with operators (see for example [KKV04]). All this is computed by CDE:

```
schouten_bracket(biv1,biv2,tv12),
```

where `biv1` and `biv2` are bivectors, or $\mathcal{C}$-differential operators with 2 arguments, and `tv12` is the result of the computation, which is a three-vector (it is automatically declared to be a superfunction). Examples of this computation are given in Section 20.10.12.

20.10.8 Computing linearization and its adjoint

Currently, CDE supports linearization of a vector function, or a $\mathcal{C}$-differential operator with 0 arguments. The computation is performed in odd coordinates.

Suppose that we would like to linearize the vector function that defines the (dispersionless) Boussinesq equation [KKV06]:

$$\begin{cases} u_t - u_x v - u v_x - \sigma v_{xxx} = 0 \\ v_t - u_x - v v_x = 0 \end{cases} \quad (20.58)$$

where σ is a constant. Then a jet space with independent variables `x,t`, dependent variables `u,v` and odd variables *in the same number as dependent variables* `p,q` must be created:

```
indep_var:={x,t}$
dep_var:={u,v}$
odd_var:={p,q}$
total_order:=8$
cde({indep_var,dep_var,odd_var,total_order},{})$
```

The linearization of the above system and its adjoint are, respectively

$$\ell_{\text{Bou}} = \begin{pmatrix} D_t - vD_x - v_x & -u_x - uD_x - \sigma D_{xxx} \\ -D_x & D_t - v_x - vD_x \end{pmatrix},$$
$$\ell^*_{\text{Bou}} = \begin{pmatrix} -D_t + vD_x & D_x \\ uD_x + \sigma D_{xxx} & -D_t + vD_x \end{pmatrix}$$

Let us introduces the vector function whose zeros are the Boussinesq equation:

```
f_bou:={u_t - (u_x*v + u*v_x + sig*v_3x),
        v_t - (u_x + v*v_x)};
```

The following command assigns to the identifier `lbou` the linearization $\mathcal{C}$-differential operator ℓ_{Bou} of the vector function `f_bou`

```
ell_function(f_bou,lbou);
```

moreover, a superfunction `lbou_sf` is also defined as the vector superfunction corresponding to ℓ_{Bou}. Indeed, the following sequence of commands:

```
2: lbou_sf(1);

 - p*v_x + p_t - p_x*v - q*u_x - q_3x*sig - q_x*u

3: lbou_sf(2);

 - p_x - q*v_x + q_t - q_x*v
```

shows the vector superfunction corresponding to ℓ_{Bou}. To compute the value of the $(1,1)$ component of the matrix ℓ_{Bou} applied to an argument `psi` do

```
lbou(1,1,psi);
```

In order to check that the result is correct one could define the linearization as a $\mathcal{C}$-differential operator and then check that the corresponding superfunctions are

the same:

```
mk_cdiffop(lbou2,1,{2},2);
for all phi let lbou2(1,1,phi)
          = td(phi,t) - v*td(phi,x) - v_x*phi;
for all phi let lbou2(1,2,phi)
          = - u_x*phi - u*td(phi,x) - sig*td(phi,x,3);
for all phi let lbou2(2,1,phi)
          = - td(phi,x);
for all phi let lbou2(2,2,phi)
          = td(phi,t) - v*td(phi,x) - v_x*phi;

conv_cdiff2superfun(lbou2,lbou2_sf);
lbou2_sf(1) - lbou_sf(1);
lbou2_sf(2) - lbou_sf(2);
```

the result of the two last commands must be zero.

The formal adjoint of `lbou` can be computed and assigned to the identifier `lbou_star` by the command

```
adjoint_cdiffop(lbou,lbou_star);
```

Again, the associated vector superfunction `lbou_star_sf` is computed, with values

```
4: lbou_star_sf(1);

 - p_t + p_x*v + q_x

5: lbou_star_sf(2);

p_3x*sig + p_x*u - q_t + q_x*v
```

Again, the above operator can be checked for correctness.

Once the linearization and its ajdoint are computed, in order to do computations with symmetries and conservation laws such operator must be restricted to the corresponding equation. This can be achieved with the following steps:

1. compute linearization of a PDE of the form $F = 0$ and its adjoint, and save them in the form of a vector superfunction;
2. start a new computation with the given *even* PDE as a constraint on the (even) jet space;

3. load the superfunctions of item 1;

4. restrict them to the even PDE.

Only the last step needs to be explained. If we are considering, *eg* the Boussinesq equation, then u_t and its differential consequences (*ie* the principal derivatives) are not automatically expanded to the right-hand side of the equation and its differential consequences. At the moment this step is not fully automatic. More precisely, only principal derivatives which appear as coefficients in total derivatives can be replaced by their expression. The lists of such derivatives with the corresponding expressions are `repprincparam_der` and `repprincparam_odd` (see Section 20.10.3). They are in the format of REDUCE's replacement list and can be used in let-rules. If the linearization or its adjoint happen to depend on another principal derivative this must be computed separately. A forthcoming release of REDUCE will automatize this procedure.

However, note that for evolutionary equations this step is trivial, as the restriction of linearization and its adjoint on the given PDE will only affect total derivatives which are restricted by CDE to the PDE.

20.10.9 Higher symmetries

In this section we show the computation of (some) higher [BCD+99] (or generalized, [Olv93]) symmetries of Burgers'equation $B = u_t - u_{xx} + 2uu_x = 0$.

We provide two ways to solve the equations for higher symmetries. The first possibility is to use dimensional analysis. The idea is that one can use the scale symmetries of Burgers'equation to assign "gradings" to each variable appearing in the equation (in other words, one can use dimensional analisys). As a consequence, one could try different ansatz for symmetries with polynomial generating functions. For example, it is possible to require that they are sum of monomials of given degrees. This ansatz yields a simplification of the equations for symmetries, because it is possible to solve them in a "graded" way, *i.e.*, it is possible to split them into several equations made by the homogeneous components of the equation for symmetries with respect to gradings.

In particular, Burgers'equation translates into the following dimensional equation:

$$[u_t] = [u_{xx}], \quad [u_{xx}] = [2uu_x].$$

By the rules $[u_z] = [u] - [z]$ and $[uv] = [u] + [v]$, and choosing $[x] = -1$, we have $[u] = 1$ and $[t] = -2$. This will be used to generate the list of homogeneous monomials of given grading to be used in the ansatz about the structure of the generating function of the symmetries.

The file for the above computation is `bur_hsy1.red` and the results of the computation are in `results/bur_hsy1_res.red`.

Another possibility to solve the equation for higher symmetries is to use a PDE solver that is especially devoted to overdetermined systems, which is the distinguishing feature of systems coming from the symmetry analysis of PDEs. This approach is described below. The file for the above computation is `bur_hsy2.red` and the results of the computation are in `results/bur_hsy2_res.red`.

Setting up the jet space and the differential equation. After loading CDE:

```
indep_var:={x,t}$
dep_var:={u}$
deg_indep_var:={-1,-2}$
deg_dep_var:={1}$
total_order:=10$
```

Here the new lists are scale degrees:

- `deg_indep_var` is the list of scale degrees of the independent variables;
- `deg_dep_var` is the list of scale degrees of the dependent variables;

We now give the equation and call CDE:

```
principal_der:={u_t}$
de:={u_2x+2*u*u_x}$
cde({indep_var,dep_var,{},total_order},
   {principal_der,de,{},{}})$
```

Solving the problem via dimensional analysis. Higher symmetries of the given equation are functions `sym` depending on parametric coordinates up to some jet space order. We assume that they are graded polynomials of all parametric derivatives. In practice, we generate a linear combination of graded monomials with arbitrary coefficients, then we plug it in the equation of the problem and find conditions on the coefficients that fulfill the equation. To construct a good ansatz, it is required to make several attempts with different gradings, possibly including independent variables, etc.. For this reason, ansatz-constructing functions are especially verbose. In order to use such functions they must be initialized with the following command:

```
cde_grading(deg_indep_var,deg_dep_var,{})$
```

Note the empty list at the end; it playe a role only for computations involving odd variables.

We need one operator `equ` whose components will be the equation of higher symmetries and its consequences. Moreover, we need an operator `c` which will play the role of a vector of constants, indexed by a counter `ctel`:

```
ctel:=0;
operator c,equ;
```

We prepare a list of variables ordered by scale degree:

```
l_grad_var:=der_deg_ordering(0,all_parametric_der)$
```

The function `der_deg_ordering` is defined in `cde.red`. It produces the given list using the list `all_parametric_der` of all parametric derivatives of the given equation up to the order `total_order`. The first two parameters can assume the values 0 or 1 and say that we are considering even variables and that the variables are of parametric type.

Then, due to the fact that *all parametric variables have positive scale degree* then we prepare the list `ansatz` of all graded monomials of scale degree from 0 to 5

```
gradmon:=graded_mon(1,5,l_grad_var)$
gradmon:={1} . gradmon$
ansatz:=for each el in gradmon join el$
```

More precisely, the command `graded_mon` produces a list of monomials of degrees from `i` to `j`, formed from the list of graded variables `l_grad_var`; the second command adds the zero-degree monomial; and the last command produces a single list of all monomials.

Finally, we assume that the higher symmetry is a graded polynomial obtained from the above monomials (so, it is independent of x and t!)

```
sym:=(for each el in ansatz sum (c(ctel:=ctel+1)*el))$
```

Next, we define the equation $\ell_B(\mathtt{sym}) = 0$. Here, ℓ_B stands for the linearization (Section 20.10.8). A function `sym` that fulfills the above equation, on account of $B = 0$, is an higher symmetry.

We **cannot** define the linearization as a $\mathcal{C}$-differential operator in this way:

```
bur:={u_t - (2*u*u_x+u_2x)};
ell_function(bur,lbur);
```

as the linearization is performed with respect to parametric derivatives only! This means that the linearization has to be computed beforehand in a free jet space, then it may be used here.

So, the right way to go is

```
mk_cdiffop(lbur,1,{1},1);
for all phi let lbur(1,1,phi)
       = td(phi,t)-td(phi,x,2)-2*u*td(phi,x)-2*u_x*phi;
```

Note that for evolutionary equations the restriction of the linearization to the equation is equivalent to just restricting total derivatives, which is automatic in CDE.

The equation becomes

```
equ 1:=lbur(1,1,sym);
```

At this point we initialize the equation solver. This is a part of the CDIFF package called `integrator.red` (see the original documentation inside the folder `packages/cdiff` in REDUCE's source code). In our case the above package will solve a large sparse linear system of algebraic equations on the coefficients of `sym`.

The list of variables, to be passed to the equation solver:

```
vars:=append(indep_var,all_parametric_der);
```

The number of initial equation(s):

```
tel:=1;
```

Next command initializes the equation solver. It passes

- the equation vector `equ` togeher with its length `tel` (*i.e.*, the total number of equations);
- the list of variables with respect to which the system *must not* split the equations, *i.e.*, variables with respect to which the unknowns are not polynomial. In this case this list is just {};
- the constants'vector `c`, its length `ctel`, and the number of negative indexes if any; just `0` in our example;
- the vector of free functions `f` that may appear in computations. Note that in `{f,0,0 }` the second `0` stands for the length of the vector of free functions. In this example there are no free functions, but the command needs the presence of at least a dummy argument, `f` in this case. There is also a last zero which is the negative length of the vector f, just as for constants.

```
initialize_equations(equ,tel,{},{c,ctel,0},{f,0,0});
```

Run the procedure `splitvars_opequ` on the first component of `equ` in order to obtain equations on coefficiens of each monomial.

```
tel:=splitvars_opequ(equ,1,1,vars);
```

Note that `splitvars_opequ` needs to know the indices of the first and the last equation in `equ`, and here we have only one equation as `equ(1)`. The output `tel` is the final number of splitted equations, starting just after the initial equation `equ(1)`.

Next command tells the solver the total number of equations obtained after running `splitvars`.

```
put_equations_used tel;
```

This command solves the equations for the coefficients. Note that we have to skip the initial equations!

```
for i:=2:tel do integrate_equation i;
```

The output is written in the result file by the commands

```
off echo$
off nat$
out <<resname>>;
sym:=sym;
write ";end;";
shut <<resname>>;
on nat$
on echo$
```

The command `off nat` turns off writing in natural notation; results in this form are better only for visualization, not for writing or for input into another computation. The command «`resname`» forces the evaluation of the variable `resname` to its string value. The commands `out` and `shut` are for file opening and closing. The command `sym:=sym` is evaluated only on the right-hand side.

One more example file is available; it concerns higher symmetries of the KdV equation. In order to deal with symmetries explicitely depending on x and t it is possible to use REDUCE and CDE commands in order to have `sym = x*`(something of degree 3) `+ t*`(something of degree 5) + (something of degree 2); this yields scale symmetries. Or we could use `sym = x*`(something of degree 1) `+ t*`(something of degree 3) + (something of degree 0); this yields Galilean boosts.

Solving the problem using CRACK. CRACK is a PDE solver which is devoted mostly to the solution of overdetermined PDE systems [BW95, WB]. Several mathematical problems have been solved by the help of CRACK, like finding symmetries [Wol95, BW92] and conservation laws [Wol02a]. The aim of CDE is to provide a tool for computations with total derivatives, but it can be used to compute symmetries too. In this subsection we show how to interface CDE with CRACK in order to find higher (or generalized) symmetries for the Burgers'equation. To do that, after loading CDE and introducing the equation, we define the linearization of the equation `lbur`.

We introduce the new unknown function '`ansatz`'. We assume that the function depends on parametric variables of order not higher than 3. The variables are selected by the function `selectvars` of CDE as follows:

```
even_vars:=for i:=0:3 join
          selectvars(0,i,dep_var,all_parametric_der)$
```

In the arguments of `selectvars`, `0` means that we want even variables, `i` stands for the order of variables, `dep_var` stands for the dependent variables to be selected by the command (here we use all dependent variables), `all_parametric_der` is the set of variables where the function will extract the variables with the required properties. In the current example we wish to get all higher symmetries depending on parametric variables of order not higher than 3.

The dependency of `ansatz` from the variables is given with the standard REDUCE command `depend`:

```
for each el in even_vars do depend(ansatz,el)$
```

The equation to be solved is the equation `lbur(ansatz)=0`, hence we give the command

```
total_eq:=lbur(1,1,ansatz)$
```

The above command will issue an error if the list `{total_eq}` depends on the flag variable `letop`. In this case the computation has to be redone within a jet space of higher order.

The equation `ell_b(ansatz)=0` is polynomial with respect to the variables of order higher than those appearing in `ansatz`. For this reason, its coefficients can be put to zero independently. This is the reason why the PDEs that determine symmetries are overdetermined. To tell this to CRACK, we issue the command

```
split_vars:=diffset(all_parametric_der,even_vars)$
```

The list `split_vars` contains variables which are in the current CDE jet space but *not* in `even_vars`.

Then, we load the package CRACK and get results.

```
load_package crack;
crack_results:=crack(total_eq,{},{ansatz},split_vars);
```

The results are in the variable `crack_results`:

```
{{{},
{ansatz=(2*c_12*u_x + 2*c_13*u*u_x + c_13*u_2x
 + 6*c_8*u**2*u_x + 6*c_8*u*u_2x + 2*c_8*u_3x
 + 6*c_8*u_x**2)/2},{c_8,c_13,c_12},
{}}}$
```

So, we have three symmetries; of course the generalized symmetry corresponds to `c_8`. Remember to check *always* the output of CRACK to see if any of the symbols `c_n` is indeed a free function depending on some of the variables, and not just a constant.

20.10.10 Local conservation laws

In this section we will find (some) local conservation laws for the KdV equation $F = u_t - u_{xxx} + uu_x = 0$. Concretely, we have to find non-trivial 1-forms $f = f_x dx + f_t dt$ on $F = 0$ such that $\bar{d}f = 0$ on $F = 0$. "Triviality" of conservation laws is a delicate matter, for which we invite the reader to have a look in [BCD+99].

The files containing this example are `kdv_lcl1,kdv_lcl2` and the corresponding results and debug files.

We suppose that the conservation law has the form $\omega = f_x dx + f_t dt$. Using the same `ansatz` as in the previous example we assume

```
fx:=(for each el in ansatz sum (c(ctel:=ctel+1)*el))$
ft:=(for each el in ansatz sum (c(ctel:=ctel+1)*el))$
```

Next we define the equation $\bar{d}(\omega) = 0$, where $\bar{d}$ is the total exterior derivative restricted to the equation.

```
equ 1:=td(fx,t)-td(ft,x)$
```

After solving the equation as in the above example we get

```
fx := c(3)*u_x + c(2)*u + c(1)$
```

```
ft := (2*c(8) + 2*c(3)*u*u_x + 2*c(3)*u_3x + c(2)*u**2 +
2*c(2)*u_2x)/2$
```

Unfortunately it is clear that the conservation law corresponding to `c(3)` is trivial, because it is just the KdV equation. Here this fact is evident; how to get rid of less evident trivialities by an 'automatic' mechanism? We considered this problem in the file `kdv_lcl2`, where we solved the equation

```
equ 1:=fx-td(f0,x);
equ 2:=ft-td(f0,t);
```

after having loaded the values `fx` and `ft` found by the previous program. In order to do that we have to introduce two new counters:

```
operator cc,equ;
cctel:=0;
```

We make the following ansatz on `f0`:

```
f0:=(for each el in ansatz sum (cc(cctel:=cctel+1)*el))$
```

After solving the system, issuing the commands

```
fxnontriv := fx-td(f0,x);
ftnontriv := ft-td(f0,t);
```

we obtain

```
fxnontriv := c(2)*u$
ftnontriv := (c(2)*(u**2 + 2*u_2x))/2$
```

This mechanism can be easily generalized to situations in which the conservation laws which are found by the program are difficult to treat by pen and paper. However, we will present another approach to the computation of conservation laws in subsection 20.10.15.

20.10.11 Local Hamiltonian operators

In this section we will show how to compute local Hamiltonian operators for Korteweg–de Vries, Boussinesq and Kadomtsev–Petviashvili equations. It is interesting to note that we will adopt the same computational scheme for all equations, even if the latter is not in evolutionary form and it has more than two independent variables. This comes from a new mathematical theory which started in [KKV04] for evolution equations and was later extended to general differential equations in

[KKVV09].

Korteweg–de Vries equation. Here we will find local Hamiltonian operators for the KdV equation $u_t = u_{xxx} + uu_x$. A necessary condition for an operator to be Hamiltonian is that it sends generating functions (or characteristics, according with [Olv93]) of conservation laws to higher (or generalized) symmetries. As it is proved in [KKV04], this amounts at solving $\bar{\ell}_{KdV}(\texttt{phi}) = 0$ over the equation

$$\begin{cases} u_t = u_{xxx} + uu_x \\ p_t = p_{xxx} + up_x \end{cases}$$

or, in geometric terminology, find the shadows of symmetries on the ℓ^*-covering of the KdV equation, with the further condition that the shadows must be linear in the p-variables. Note that the second equation (in odd variables!) is just the adjoint of the linearization of the KdV equation applied to an odd variable.

The file containing this example is `kdv_lho1`.

We stress that the linearization $\bar{\ell}_{KdV}(\texttt{phi}) = 0$ is the equation

```
td(phi,t)-u*td(phi,x)-u_x*phi-td(phi,x,3)=0
```

but the total derivatives are lifted to the ℓ^* covering, hence they contain also derivatives with respect to p's. We can define a linearization operator `lkdv` as usual.

In order to produce an ansatz which is a superfunction of one odd variable (or a linear function in odd variables) we produce two lists: the list `l_grad_var` of all even variables collected by their gradings and a similar list `l_grad_odd` for odd variables:

```
l_grad_var := der_deg_ordering(0,all_parametric_der)$
l_grad_odd := {1} .
              der_deg_ordering(1,all_parametric_odd)$
gradmon := graded_mon(1,10,l_grad_var)$
gradmon := {1} . gradmon$
```

We need a list of graded monomials which are linear in odd variables. The function `mkalllinodd` produces all monomials which are linear with respect to the variables from `l_grad_odd`, have (monomial) coefficients from the variables in `l_grad_var`, and have total scale degrees from 1 to 6. Such monomials are then converted to the internal representation of odd variables.

```
linodd:=mkalllinodd(gradmon,l_grad_odd,1,6)$
```

Note that all odd variables have positive scale degrees thanks to our initial choice `deg_odd_var:=1;`. Finally, the ansatz for local Hamiltonian operators:

```
sym:=(for each el in linext sum (c(ctel:=ctel+1)*el))$
```

After having set

```
equ 1:=lkdv(1,1,sym);
```

and having initialized the equation solver as before, we do `splitext`

```
tel:=splitext_opequ(equ,1,1);
```

in order to split the polynomial equation with respect to the `ext` variables, then `splitvars`

```
tel2:=splitvars_opequ(equ,2,tel,vars);
```

in order to split the resulting polynomial equation in a list of equations on the coefficients of all monomials.

Now we are ready to solve all equations:

```
put_equations_used tel;
for i:=2:tel do integrate_equation i;
end;
```

Note that we want *all* equations to be solved!

The results are the two well-known Hamiltonian operators for the KdV. After integration the function `sym` becomes

```
sym := (c(5)*p*u_x + 2*c(5)*p_x*u +
        3*c(5)*p_3x + 3*c(2)*p_x)/3$
```

Of course, the results correspond to the operators

$$p_x \to D_x,$$
$$\frac{1}{3}(3p_{3x} + 2up_x + u_x p) \to \frac{1}{3}(3D_{xxx} + 2uD_x + u_x).$$

Note that each operator is multiplied by one arbitrary real constant, `c(5)` and `c(2)`.

The same problem can be approached using CRACK (see file `kdv_lho2.red`). An ansatz is constructed by the following instructions:

```
even_vars:=for i:=0:3 join
            selectvars(0,i,dep_var,all_parametric_der)$
odd_vars:=for i:=0:3 join
```

```
            selectvars(1,i,odd_var,all_parametric_odd)$
ext_vars:=replace_oddext(odd_vars)$

ctemp:=0$
ansatz:=for each el in ext_vars sum
        mkid(s,ctemp:=ctemp+1)*el$
```

Note that we have

```
ansatz := p*s1 + p_2x*s3 + p_3x*s4 + p_x*s2$
```

Indeed, we are looking for a third-order operator whose coefficients depend on variables of order not higher than 3. This last property has to be introduced by

```
unk:=for i:=1:ctemp collect mkid(s,i)$
for each ell in unk do
 for each el in even_vars do depend ell,el$
```

Then, we introduce the linearization (lifted on the cotangent covering)

```
operator ell_f$
for all sym let ell_f(sym)=
   td(sym,t) - u*td(sym,x) - u_x*sym - td(sym,x,3)$
```

and the equation to be solved, together with the usual test that checks for the nedd to enlarge the jet space:

```
total_eq:=ell_f(ansatz)$
```

Finally, we split the above equation by collecting all coefficients of odd variables:

```
system_eq:=splitext_list({total_eq})$
```

and we feed CRACK with the equations that consist in asking to the above coefficients to be zero:

```
load_package crack;
crack_results:=crack(system_eq,{},unk,
   diffset(all_parametric_der,even_vars));
```

The results are the same as in the previous section:

```
crack_results := {{{},
{s4=(3*c_17)/2,s3=0,s2=c_16 + c_17*u,s1=(c_17*u_x)/2},
```

```
{c_17,c_16},
{}}}$
```

Boussinesq equation. There is no conceptual difference when computing for systems of PDEs with respect to the previous computations for scalar equations. We will look for Hamiltonian structures for the dispersionless Boussinesq equation (20.58).

We will proceed by dimensional analysis. Gradings can be taken as

$$[t] = -2, \quad [x] = -1, \quad [v] = 1, \quad [u] = 2, \quad [p] = 1, \quad [q] = 2$$

where p, q are the two odd coordinates. We have the ℓ^*_{Bou} covering equation

$$\begin{cases} -p_t + vp_x + q_x = 0 \\ up_x + \sigma p_{xxx} - q_t + vq_x = 0 \\ u_t - u_x v - uv_x - \sigma v_{xxx} = 0 \\ v_t - u_x - vv_x = 0 \end{cases}$$

We have to find Hamiltonian operators as shadows of symmetries on the above covering. At the level of source file (`bou_lho1`) the input data is:

```
indep_var:={x,t}$
dep_var:={u,v}$
odd_var:={p,q}$
deg_indep_var:={-1,-2}$
deg_dep_var:={2,1}$
deg_odd_var:={1,2}$
total_order:=8$
principal_der:={u_t,v_t}$
de:={u_x*v+u*v_x+sig*v_3x,u_x+v*v_x}$
principal_odd:={p_t,q_t}$
de_odd:={v*p_x+q_x,u*p_x+sig*p_3x+v*q_x}$
```

The ansatz for the components of the Hamiltonian operator, of scale degree between 1 and 6, is

```
linodd:=mkalllinodd(gradmon,l_grad_odd,1,6)$
phi1:=(for each el in linodd sum (c(ctel:=ctel+1)*el))$
phi2:=(for each el in linodd sum (c(ctel:=ctel+1)*el))$
```

and the equation for shadows of symmetries is (`lbou2` is taken from Section 20.10.8)

```
equ 1:=lbou2(1,1,phi1) + lbou2(1,2,phi2);
```

```
equ 2:=lbou2(2,1,phi1) + lbou2(2,2,phi2);
```

After the usual procedures for decomposing polynomials we obtain three local Hamiltonian operators:

```
phi1_odd := (2*c(31)*p*sig*v_3x + 2*c(31)*p*u*v_x
 + 2*c(31)*p*u_x*v + 6*c(31)*p_2x*sig*v_x
 + 4*c(31)*p_3x*sig*v + 6*c(31)*p_x*sig*v_2x
 + 4*c(31)*p_x*u*v + 2*c(31)*q*u_x + 4*c(31)*q_3x*sig
 + 4*c(31)*q_x*u + c(31)*q_x*v**2 + 2*c(16)*p*u_x
 + 4*c(16)*p_3x*sig + 4*c(16)*p_x*u
 + 2*c(16)*q_x*v + 2*c(10)*q_x)/2$

phi2_odd := (2*c(31)*p*u_x + 2*c(31)*p*v*v_x
 + 4*c(31)*p_3x*sig + 4*c(31)*p_x*u
 + c(31)*p_x*v**2 + 2*c(31)*q*v_x + 4*c(31)*q_x*v
 + 2*c(16)*p*v_x + 2*c(16)*p_x*v
 + 4*c(16)*q_x + 2*c(10)*p_x)/2$
```

There is a whole hierarchy of nonlocal Hamiltonian operators [KKV04].

Kadomtsev–Petviashvili equation. There is no conceptual difference in symbolic computations of Hamiltonian operators for PDEs in 2 independent variables and in more than 2 independent variables, regardless of the fact that the equation at hand is written in evolutionary form. As a model example, we consider the KP equation

$$u_{yy} = u_{tx} - u_x^2 - uu_{xx} - \frac{1}{12}u_{xxxx}. \tag{20.59}$$

Proceeding as in the above examples we input the following data:

```
indep_var:={t,x,y}$
dep_var:={u}$
odd_var:={p}$
deg_indep_var:={-3,-2,-1}$
deg_dep_var:={2}$
deg_odd_var:={1}$
total_order:=6$
principal_der:={u_2y}$
de:={u_tx-u_x**2-u*u_2x-(1/12)*u_4x}$
principal_odd:={p_2y}$
de_odd:={p_tx-u*p_2x-(1/12)*p_4x}$
```

and look for Hamiltonian operators of scale degree between 1 and 5:

```
linodd:=mkalllinodd(gradmon,l_grad_odd,1,5)$
phi:=(for each el in linodd sum (c(ctel:=ctel+1)*el))$
```

After solving the equation for shadows of symmetries in the cotangent covering

```
equ 1:=td(phi,y,2) - td(phi,x,t) + 2*u_x*td(phi,x)
 + u_2x*phi + u*td(phi,x,2) + (1/12)*td(phi,x,4);
```

we get the only local Hamiltonian operator

```
phi := c(13)*p_2x$
```

As far as we know there are no further local Hamiltonian operators.

Remark: the above Hamiltonian operator is already known in an evolutionary presentation of the KP equation [Kup94]. Our mathematical theory of Hamiltonian operators for general differential equations [KKVV09] allows us to formulate and solve the problem for any presentation of the KP equation. Change of coordinate formulae could also be provided.

20.10.12 Examples of Schouten bracket of local Hamiltonian operators

In this Section we will discuss examples of calculation of Schouten bracket in order to check the Hamiltonian property for $\mathcal{C}$-differential operators and/or the compatibility of two distinct Hamiltonian operators. This subject is treated in a much greater detail in the recent paper [Vit19], where many examples of Schouten bracket calculations with CDE have been described.

We observe that a package that is capable to calculate the Schouten bracket of weakly nonlocal operators (in one independent variable) is currently part of CDE, version 3.0. Documentation for the package is being written; interested readers may contact the author of CDE for questions.

Let $F = 0$ be a system of PDEs. Here $F \in P$, where P is the module (in the algebraic sense) of vector functions $P = \{J^r(n,m) \to \mathbb{R}^k\}$.

The Hamiltonian operators which have been computed in the previous Section are differential operators sending generating functions of conservation laws into generating functions of symmetries for the above system of PDEs:

$$H\colon \hat{P} \to \varkappa \tag{20.60}$$

- $\hat{P} = \{J^r(n,m) \to (\mathbb{R}^k)^* \otimes \wedge^n T^*\mathbb{R}^n\}$ is the space of covector-valued densities,

- $\varkappa = \{J^r(n,m) \to \mathbb{R}^m\}$ is the space of generalized vector fields on jets; generating functions of higher symmetries of the system of PDEs are elements of this space.

As the operators are mainly used to define a bracket operation and a Lie algebra structure on conservation laws, two properties are required: skew-adjointness $H^* = -H$ (corresponding with skew-symmetry of the bracket) and $[H, H] = 0$ (corresponding with the Jacobi property of the bracket).

In order to compute the two properties we proceed as follows. Skew-adjointness is checked by computing the adjoint and verifying that the sum with the initial operator is zero.

In the case of evolutionary equations, $P = \varkappa$, and Hamiltonian operators (20.60) can also be interpreted as *variational bivectors, ie*

$$\hat{H} \colon \hat{\varkappa} \times \hat{\varkappa} \to \wedge^n T^* \mathbb{R}^n \tag{20.61}$$

where the correspondence is given by

$$H(\psi) = (a^{ij\sigma} D_\sigma \psi_j) \quad \to \quad \hat{H}(\psi_1, \psi_2) = (a^{ij\sigma} D_\sigma \psi_{1\,j} \psi_{2\,i}) \tag{20.62}$$

In terms of the corresponding superfunctions:

$$H = a^{ik\,\sigma} p_{k\,\sigma} \quad \to \quad \hat{H} = a^{ik\,\sigma} p_{k\,\sigma} p_i.$$

Note that the product $p_{k\,\sigma} p_i$ is anticommutative since p's are odd variables.

After that a $\mathcal{C}$-differential operator of the type of H has been converted into a bivector it is possible to apply the formulae (20.56) and (20.57) in order to compute the Schouten bracket. This is what we will see in next section.

Bi-Hamiltonian structure of the KdV equation. We can do the above computations using KdV equation as a test case (see the file `kdv_lho3.red`).

Let us load the above operators:

```
operator ham1;
for all psi1 let ham1(psi1)=td(psi1,x);
operator ham2;
for all psi2 let ham2(psi2)=
 (1/3)*u_x*psi2 + td(psi2,x,3) + (2/3)*u*td(psi2,x);
```

We may convert the two operators into the corresponding superfunctions

```
conv_cdiff2superfun(ham1,sym1);
conv_cdiff2superfun(ham2,sym2);
```

The result of the conversion is

```
sym1(1) := {p_x};
sym2(2) := {(1/3)*p*u_x + p_3x + (2/3)*p_x*u};
```

Skew-adjointness is checked at once:

```
adjoint_cdiffop(ham1,ham1_star);
adjoint_cdiffop(ham2,ham2_star);
ham1_star_sf(1)+sym1(1);
ham2_star_sf(1)+sym2(1);
```

and the result of the last two commands is zero.

Then we shall convert the two superfunctions into bivectors:

```
conv_genfun2biv(sym1_odd,biv1);
conv_genfun2biv(sym2_odd,biv2);
```

The output is:

```
biv1(1) :=  - ext(p,p_x);
biv2(1) := - (1/3)*( - 3*ext(p,p_3x) - 2*ext(p,p_x)*u);
```

Finally, the three Schouten brackets $[\hat{H}_i, \hat{H}_j]$ are computed, with $i, j = 1, 2$:

```
schouten_bracket(biv1,biv1,sb11);
schouten_bracket(biv1,biv2,sb12);
schouten_bracket(biv2,biv2,sb22);
```

the result are well-known lists of zeros.

Bi-Hamiltonian structure of the WDVV equation. This subsection refers to the the example file `wdvv_biham1.red`. The simplest nontrivial case of the WDVV equations is the third-order Monge–Ampère equation, $f_{ttt} = f_{xxt}^2 - f_{xxx}f_{xtt}$ [Dub96]. This PDE can be transformed into hydrodynamic form,

$$a_t = b_x, \quad b_t = c_x, \quad c_t = (b^2 - ac)_x,$$

via the change of variables $a = f_{xxx}$, $b = f_{xxt}$, $c = f_{xtt}$. This system possesses two Hamiltonian formulations [FGMN97]:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}_t = A_i \begin{pmatrix} \delta H_i/\delta a \\ \delta H_i/\delta b \\ \delta H_i/\delta c \end{pmatrix}, \quad i = 1, 2$$

with the homogeneous first-order Hamiltonian operator

$$\hat{A}_1 = \begin{pmatrix} -\frac{3}{2}D_x & \frac{1}{2}D_x a & D_x b \\ \frac{1}{2}aD_x & \frac{1}{2}(D_x b + bD_x) & \frac{3}{2}cD_x + c_x \\ bD_x & \frac{3}{2}D_x c - c_x & (b^2 - ac)D_x + D_x(b^2 - ac) \end{pmatrix}$$

with the Hamiltonian $H_1 = \int c\,dx$, and the homogeneous third-order Hamiltonian operator

$$A_2 = D_x \begin{pmatrix} 0 & 0 & D_x \\ 0 & D_x & -D_x a \\ D_x & -aD_x & D_x b + bD_x + aD_x a \end{pmatrix} D_x,$$

with the nonlocal Hamiltonian

$$H_2 = -\int \left(\frac{1}{2} a \left({D_x}^{-1} b \right)^2 + {D_x}^{-1} b {D_x}^{-1} c \right) dx.$$

Both operators are of Dubrovin–Novikov type [DN83, DN84]. This means that the operators are homogeneous with respect to the grading $|D_x| = 1$. It follows that the operators are form-invariant under point transformations of the dependent variables, $u^i = u^i(\tilde{u}^j)$. Here and in what follows we will use the letters u^i to denote the dependent variables (a, b, c). Under such transformations, the coefficients of the operators transform as differential-geometric objects.

The operator A_1 has the general structure

$$A_1 = g_1^{ij} D_x + \Gamma_k^{ij} u_x^k$$

where the covariant metric $g_{1\,ij}$ is flat, $\Gamma_k^{ij} = g_1^{is}\Gamma_{sk}^j$ (here g_1^{ij} is the inverse matrix that represent the contravariant metric induced by $g_{1\,ij}$), and Γ_{sk}^j are the usual Christoffel symbols of $g_{1\,ij}$.

The operator A_2 has the general structure

$$A_2 = D_x \left(g_2^{ij} D_x + c_k^{ij} u_x^k \right) D_x, \tag{20.63}$$

where the inverse $g_{2\,ij}$ of the leading term transforms as a covariant pseudo-Riemannian metric. From now on we drop the subscript 2 for the metric of A_2. It was proved in [FPV14] that, if we set $c_{ijk} = g_{iq} g_{jp} c_k^{pq}$, then

$$c_{ijk} = \frac{1}{3}(g_{ik,j} - g_{ij,k})$$

and the metric fulfills the following identity:

$$g_{mk,n} + g_{kn,m} + g_{mn,k} = 0. \tag{20.64}$$

This means that the metric is a Monge metric [FPV14]. In particular, its coefficients are quadratic in the variables u^i. It is easy to input the two operators in CDE. Let us start by A_1: we may define its entries one by one as follows

```
operator a1;

for all psi let a1(1,1,psi) = - (3/2)*td(psi,x);
for all psi let a1(1,2,psi) = (1/2)*td(a*psi,x);
...
```

We could also use one specialized Reduce package for the computation of the Christoffel symbols, like `RedTen` or `GRG`. Assuming that the operators `gamma_hi(i,j,k)` have been defined equal to Γ_k^{ij} and computed in the system using the inverse matrix g_{ij} of the leading coefficient contravariant metric[24]

$$g^{ij} = \begin{pmatrix} -\frac{3}{2} & \frac{1}{2}a & b \\ \frac{1}{2}a & b & \frac{3}{2}c \\ b & \frac{3}{2}c & 2(b^2 - ac) \end{pmatrix}$$

then, provided we defined a list `dep_var` of the dependent variables, we could set

```
operator gamma_hi_con;
for all i,j let gamma_hi_con(i,j) =
(
 for k:=1:3 sum gamma_hi(i,j,k)
                  *mkid(part(dep_var,k),!_x)
)$
```

and

```
operator a1$
for all i,j,psi let a1(i,j,psi) =
gu1(i,j)*td(psi,x)+(for k:=1:3 sum gamma_hi_con(i,j)*psi
)$
```

The third order operator can be reconstructed as follows. Observe that the leading contravariant metric is

$$g^{ij} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -a \\ 1 & -a & 2b + a^2 \end{pmatrix}$$

Introduce the above matrix in REDUCE as `gu3`. Then set

```
gu3:=gl3**(-1)$
```

and define c_{ijk} as

[24] Indeed in the example file `wdvv_biham1.red` there are procedures for computing all those quantities.

```
operator c_lo$
for i:=1:3 do
 for j:=1:3 do
  for k:=1:3 do
  <<
   c_lo(i,j,k):=
    (1/3)*(df(gl3(k,i),part(dep_var,j))
    - df(gl3(j,i),part(dep_var,k)))$
  >>$
```

Then define c_k^{ij}

```
templist:={}$
operator c_hi$
for i:=1:ncomp do
 for j:=1:ncomp do
  for k:=1:ncomp do
   c_hi(i,j,k):=
    <<
     templist:=
      for m:=1:ncomp join
       for n:=1:ncomp collect
        gu3(n,i)*gu3(m,j)*c_lo(m,n,k)$
     templist:=part(templist,0):=plus
    >>$
```

Introduce the contracted operator

```
operator c_hi_con$
for i:=1:ncomp do
 for j:=1:ncomp do
  c_hi_con(i,j):=
   <<
    templist:=for k:=1:ncomp collect
     c_hi(i,j,k)*mkid(part(dep_var,k),!_x)$
    templist:=part(templist,0):=plus
   >>$
```

Finally, define the operator A_2

```
operator aa2$
for all i,j,psi let aa2(i,j,psi) =
td(
gu3(i,j)*td(psi,x,2)+c_hi_con(i,j)*td(psi,x)
```

```
,x)$
```

Now, we can test the Hamiltonian property of A_1, A_2 and their compatibility:

```
conv_cdiff2genfun(aa1,sym1)$
conv_cdiff2genfun(aa2,sym2)$

conv_genfun2biv(sym1,biv1)$
conv_genfun2biv(sym2,biv2)$

schouten_bracket(biv1,biv1,sb11);
schouten_bracket(biv1,biv2,sb12);
schouten_bracket(biv2,biv2,sb22);
```

Needless to say, the result of the last three command is a list of zeroes.

We observe that the same software can be used to prove the bi-Hamiltonianity of a 6-component WDVV system [PV15].

Schouten bracket of multidimensional operators. The formulae (20.56), (20.57) hold also in the case of multidimensional operators, *ie* operators with total derivatives in more than one independent variables. Here we give one Hamiltonian operator H and we give two more variational bivectors P_1, P_2; all operators are of Dubrovin–Novikov type (homogeneous). We check the compatibility by computing $[H, P_1]$ and $[H, P_2]$. Such computations are standard for the problem of computing the Hamiltonian cohomology of H.

This example has been provided by M. Casati. The file of the computation is `dn2d_sb1.red`. The dependent variables are p^1, p^2.

Let us set

$$H = \begin{pmatrix} D_x & 0 \\ 0 & D_y \end{pmatrix} \tag{20.65}$$

$$P_1 = \begin{pmatrix} P_1^{11} & P_1^{12} \\ P_1^{21} & P_1^{22} \end{pmatrix} \tag{20.66}$$

where

$$
\begin{aligned}
P_1^{11} =& 2\frac{\partial g}{\partial p^1}p_y^2 D_x + \frac{\partial g}{\partial p^1}p_{xy}^2 + \frac{\partial g}{\partial p^1 \partial p^2}p_x^2 p_y^2 + \frac{\partial g}{\partial^2 p^1}p_x^1 p_y^2 \\
P_1^{21} =& -fD_x^2 + gD_y^2 + \frac{\partial g}{\partial p^2}p_y^2 D_y - (\frac{\partial f}{\partial p^1}p_x^1 + 2\frac{\partial f}{\partial p^2}p_x^2)D_x \\
& - \frac{\partial f}{\partial^2 p^2}p_x^2 p_x^2 - \frac{\partial f}{\partial p^1 \partial p^2}p_x^1 p_x^2 - \frac{\partial f}{\partial p^2}p_2^2 x; \\
P_1^{12} =& fD_x^2 - gD_y^2 + \frac{\partial f}{\partial p^1}p_x^1 D_x - \Big(\frac{\partial g}{\partial p^2}p_y^2 + 2\frac{\partial g}{\partial p^1}p_y^1\Big)D_y \\
& - \frac{\partial g}{\partial^2 p^1}p_y^1 p_y^1 - \frac{\partial g}{\partial p^1 \partial p^2}p_y^1 p_y^2 - \frac{\partial g}{\partial p^1}p_{2y}^1; \\
P_1^{22} =& 2\frac{\partial f}{\partial p^2}p_x^1 D_y + \frac{\partial f}{\partial p^2}p_x^1 y + \frac{\partial f}{\partial p^1 \partial p^2}p_x^1 p_y^1 + \frac{\partial f}{\partial^2 p^2}p_x^1 p_y^2;
\end{aligned}
$$

and let $P_2 = P_1^T$. This is implemented as follows:

```
mk_cdiffop(aa2,1,{2},2)$
for all psi let aa2(1,1,psi) =
 2*df(g,p1)*p2_y*td(psi,x) + df(g,p1)*p2_xy*psi
 + df(g,p1,p2)*p2_x*p2_y*psi + df(g,p1,2)*p1_x*p2_y*psi;

for all psi let aa2(1,2,psi) =
 f*td(psi,x,2) - g*td(psi,y,2) + df(f,p1)*p1_x*td(psi,x)
 - (df(g,p2)*p2_y + 2*df(g,p1)*p1_y)*td(psi,y)
 - df(g,p1,2)*p1_y*p1_y*psi - df(g,p1,p2)*p1_y*p2_y*psi
 - df(g,p1)*p1_2y*psi;

for all psi let aa2(2,1,psi) =
 - f*td(psi,x,2) + g*td(psi,y,2)
 + df(g,p2)*p2_y*td(psi,y)
 - (df(f,p1)*p1_x+2*df(f,p2)*p2_x)*td(psi,x)
 - df(f,p2,2)*p2_x*p2_x*psi - df(f,p1,p2)*p1_x*p2_x*psi
 - df(f,p2)*p2_2x*psi;

for all psi let aa2(2,2,psi) =
 2*df(f,p2)*p1_x*td(psi,y)
 + df(f,p2)*p1_xy*psi + df(f,p1,p2)*p1_x*p1_y*psi
 + df(f,p2,2)*p1_x*p2_y*psi;

mk_cdiffop(aa3,1,{2},2)$
for all psi let aa3(1,1,psi) = aa2(1,1,psi);
for all psi let aa3(1,2,psi) = aa2(2,1,psi);
for all psi let aa3(2,1,psi) = aa2(1,2,psi);
for all psi let aa3(2,2,psi) = aa2(2,2,psi);
```

Let us check the skew-adjointness of the above bivectors:

```
conv_cdiff2superfun(aa1,sym1)$
conv_cdiff2superfun(aa2,sym2)$
conv_cdiff2superfun(aa3,sym3)$

adjoint_cdiffop(aa1,aa1_star);
adjoint_cdiffop(aa2,aa2_star);
adjoint_cdiffop(aa3,aa3_star);

for i:=1:2 do write sym1(i) + aa1_star_sf(i);
for i:=1:2 do write sym2(i) + aa2_star_sf(i);
for i:=1:2 do write sym3(i) + aa3_star_sf(i);
```

Of course the last three commands produce two zeros each.

Let us compute Schouten brackets.

```
conv_cdiff2superfun(aa1,sym1)$
conv_cdiff2superfun(aa2,sym2)$
conv_cdiff2superfun(aa3,sym3)$

conv_genfun2biv(sym1,biv1)$
conv_genfun2biv(sym2,biv2)$
conv_genfun2biv(sym3,biv3)$

schouten_bracket(biv1,biv1,sb11);
schouten_bracket(biv1,biv2,sb12);
schouten_bracket(biv1,biv3,sb13);
```

`sb11(1)` is trivially a list of zeros, while `sb12(1)` is nonzero and `sb13(1)` is again zero.

More formulae are currently being implemented in the system, like symplecticity and Nijenhuis condition for recursion operators [KKV06]. Interested readers are warmly invited to contact R. Vitolo for questions/feature requests.

20.10.13 Non-local operators

In this section we will show an experimental way to find nonlocal operators. The word ‘experimental’ comes from the lack of a comprehensive mathematical theory of nonlocal operators; in particular, it is still missing a theoretical framework for Schouten brackets of nonlocal opeartors in the odd variable language.

In any case we will achieve the results by means of a covering of the cotangent

covering. Indeed, it can be proved that there is a $1-1$ correspondence between (higher) symmetries of the initial equation and conservation laws on the cotangent covering. Such conservation laws provide new potential variables, hence a covering (see [BCD+99] for theoretical details on coverings).

In Section 20.10.15 we will also discuss a procedure for finding conservation laws from their generating functions that is of independent interest.

Non-local Hamiltonian operators for the Korteweg–de Vries equation. Here we will compute some nonlocal Hamiltonian operators for the KdV equation. The result of the computation (without the details below) has been published in [KKV04].

We have to solve equations of the type `ddx(ct)-ddt(cx)` as in 20.10.10. The main difference is that we will attempt a solution on the ℓ^*-covering (see Subsection 20.10.11). For this reason, first of all we have to determine covering variables with the usual mechanism of introducing them through conservation laws, this time on the ℓ^*-covering.

As a first step, let us compute conservation laws on the ℓ^*-covering whose components are linear in the p's. This computation can be found in the file `kdv_nlcl1` and related results and debug files.

The conservation laws that we are looking for are in $1-1$ correspondence with symmetries of the initial equation [KKV04]. We will look for conservatoin laws which correspond to Galilean boost, x-translation, t-translation at the same time. In the case of 2 independent variables and 1 dependent variable, one could prove that one component of such conservation laws can always be written as `sym*p` as follows:

```
c1x:=(t*u_x+1)*p$ % degree 1
c2x:=u_x*p$ % degree 4
c3x:=(u*u_x+u_3x)*p$ % degree 6
```

The second component must be found by solving an equation. To this aim we produce the ansatz

```
c1t:=f1*p+f2*p_x+f3*p_2x$
% degree 6
c2t:=(for each el in linodd6 sum (c(ctel:=ctel+1)*el))$
% degree 8
c3t:=(for each el in linodd8 sum (c(ctel:=ctel+1)*el))$
```

where we already introduced the sets `linodd6` and `linodd8` of 6-th and 8-th degree monomials which are linear in odd variables (see the source code). For the

first conservation law solutions of the equation

```
equ 1:=td(c1t,x) - td(c1x,t);
```

are found by hand due to the presence of 't' in the symmetry:

```
f3:=t*u_x+1$
f2:=-td(f3,x)$
f1:=u*f3+td(f3,x,2)$
```

We also have the equations

```
equ 2:=td(c2t,x)-td(c2x,t);
equ 3:=td(c3t,x)-td(c3x,t);
```

They are solved in the usual way (see the source code of the example and the results file `kdv_nlcl1_res`).

Now, we solve the equation for shadows of nonlocal symmetries in a covering of the ℓ^*-covering (source file `kdv_nlho1`). We can produce such a covering by introducing three new nonlocal (potential) variables `ra,rb,rc`. We are going to look for non-local Hamiltonian operators depending linearly on one of these variables. To this aim we modify the odd part of the equation to include the components of the above conservation laws as the derivatives of the new non-local variables `r1`, `r2`, `r3`:

```
principal_odd:={p_t,r1_x,r1_t,r2_x,r2_t,r3_x,r3_t}$
de_odd:={u*p_x+p_3x,
p*(t*u_x + 1),
p*t*u*u_x + p*t*u_3x + p*u + p_2x*t*u_x + p_2x
 - p_x*t*u_2x,
p*u_x,
p*u*u_x + p*u_3x + p_2x*u_x - p_x*u_2x,
p*(u*u_x + u_3x),
p*u**2*u_x + 2*p*u*u_3x + 3*p*u_2x*u_x + p*u_5x
 + p_2x*u*u_x + p_2x*u_3x - p_x*u*u_2x
 - p_x*u_4x - p_x*u_x**2}$
```

The scale degree analysis of the local Hamiltonian operators of the KdV equation leads to the formulation of the ansatz

```
phi:=(for each el in linodd sum (c(ctel:=ctel+1)*el))$
```

where `linext` is the list of graded mononials which are linear in odd variables and have degree 7 (see the source file). The equation for shadows of nonlocal

symmetries in ℓ^*-covering

```
equ 1:=td(phi,t)-u*td(phi,x)-u_x*phi-td(phi,x,3);
```

is solved in the usual way, obtaining (in odd variables notation):

```
phi := (c(5)*(4*p*u*u_x + 3*p*u_3x + 18*p_2x*u_x
 + 12*p_3x*u + 9*p_5x + 4*p_x*u**2
 + 12*p_x*u_2x - r2*u_x))/4$
```

Higher non-local Hamiltonian operators could also be found [KKV04]. The CRACK approach also holds for non-local computations.

20.10.14 Non-local recursion operator for the Korteweg–de Vries equation.

Following the ideas in [KKV04], a differential operator that sends symmetries into symmetries can be found as a shadow of symmetry on the ℓ-covering of the KdV equation, with the further condition that the shadows must be linear in the covering q-variables. The tangent covering of the KdV equation is

$$\begin{cases} u_t = u_{xxx} + uu_x \\ q_t = u_x q + uq_x + q_{xxx} \end{cases}$$

and we have to solve the equation $\bar{\ell}_{KdV}(\texttt{phi}) = 0$, where $\bar{\ell}_{KdV}$ means that the linearization of the KdV equation is lifted over the tangent covering.

The file containing this example is `kdv_ro1.red`. The example closely follows the computational scheme presented in [KVV12].

Usually, recursion operators are non-local: operators of the form D_x^{-1} appear in their expression. Geometrically we interpret this kind of operator as follows. We introduce a conservation law on the cotangent covering of the form

$$\omega = rt\,dx + rx\,dt$$

where $rt = uq + q_{xx}$ and $rx = q$. It has the remarkable feature of being linear with respect to q-variables. A non-local variable r can be introduced as a potential of ω, as $r_x = rx$, $r_t = rt$. A computation of shadows of symmetries on the system of PDEs

$$\begin{cases} u_t = u_{xxx} + uu_x \\ q_t = u_x q + uq_x + q_{xxx} \\ r_t = uq + q_{xx} \\ r_x = q \end{cases}$$

yields, analogously to the previous computations,

```
2*c(5)*q*u + 3*c(5)*q_2x + c(5)*r*u_x + c(2)*q.
```

The operator q stands for the identity operator, which is (and must be!) always a solution; the other solution corresponds to the Lenard–Magri operator

$$3D_{xx} + 2u + u_x D_x^{-1}.$$

20.10.15 Non-local Hamiltonian-recursion operators for Plebanski equation.

The Plebanski (or second Heavenly) equation

$$F = u_{tt}u_{xx} - u_{tx}^2 + u_{xz} + u_{ty} = 0 \tag{20.67}$$

is Lagrangian. This means that its linearization is self-adjoint: $\ell_F = \ell_F^*$, so that the tangent and cotangent covering coincide, its odd equation being

$$\ell_F(p) = p_{xz} + p_{ty} - 2u_{tx}p_{tx} + u_{2x}p_{2t} + u_{2t}p_{2x} = 0. \tag{20.68}$$

It is not difficult to realize that the above equation can be written in explicit conservative form as

$$\begin{aligned} p_{xz} + p_{ty} + u_{tt}p_{xx} + u_{xx}p_{tt} - 2u_{tx}p_{tx} \\ = D_x(p_z + u_{tt}p_x - u_{tx}p_t) + D_t(p_y + u_{xx}p_t - u_{tx}p_x) = 0, \end{aligned}$$

thus the corresponding conservation law is

$$\upsilon(1) = (p_y + u_{xx}p_t - u_{tx}p_x)\, dx\wedge dy\wedge dz + (u_{tx}p_t - p_z - u_{tt}p_x)\, dt\wedge dy\wedge dz. \tag{20.69}$$

We can introduce a potential r for the above 2-component conservation law. Namely, we can assume that

$$r_x = p_y + u_{xx}p_t - u_{tx}p_x, \quad r_t = u_{tx}p_t - p_z - u_{tt}p_x. \tag{20.70}$$

This is a new nonlocal variable for the (co)tangent covering of the Plebanski equation. We can load the Plebanski equation together with its nonlocal variable r as follows:

```
indep_var:={t,x,y,z}$
dep_var:={u}$
odd_var:={p,r}$
deg_indep_var:={-1,-1,-4,-4}$
deg_dep_var:={1}$
deg_odd_var:={1,4}$
total_order:=6$
```

```
principal_der:={u_xz}$
de:={-u_ty+u_tx**2-u_2t*u_2x}$
% rhs of the equations that define the nonlocal variable
rt:= - p_z - u_2t*p_x + u_tx*p_t$
rx:= p_y + u_2x*p_t - u_tx*p_x$
% We add conservation laws as new nonlocal odd variables
principal_odd:={p_xz,r_x,r_t}$
%
de_odd:={-p_ty+2*u_tx*p_tx-u_2x*p_2t-u_2t*p_2x,rx,rt}$
```

We can easily verify that the integrability condition for the new nonlocal variable holds:

```
td(r,t,x) - td(r,x,t);
```

the result is 0.

Now, we look for nonlocal recursion operators in the tangent covering using the new nonlocal odd variable r. We can load the equation exactly as before. We look for recursion operators which depend on r (which has scale degree 4); we produce the following ansatz for `phi`:

```
linodd:=mkalllinodd(gradmon,l_grad_odd,1,4)$
phi:=(for each el in linodd sum (c(ctel:=ctel+1)*el))$
```

then we solve the equation of shadows of symmetries:

```
equ 1:=td(phi,x,z)+td(phi,t,y)-2*u_tx*td(phi,t,x)
+u_2x*td(phi,t,2)+u_2t*td(phi,x,2)$
```

The solution is

```
phi := c(28)*r + c(1)*p
```

hence we obtain the identity operator p and the new nonlocal operator r. It can be proved that changing coordinates to the evolutionary presentation yields the local operator (which has a much more complex expression than the identity operator) and one of the nonlocal operators of [NNS05]. More details on this computation can be found in [KVV12].

20.11 CDIFF: A Package for Computations in Geometry of Differential Equations

Authors: P. Gragert, P.H.M. Kersten, G. Post and G. Roelofs

Author of this Section: R. Vitolo.

We describe CDIFF, a Reduce package for computations in geometry of Differential Equations (DEs, for short) developed by P. Gragert, P.H.M. Kersten, G. Post and G. Roelofs from the University of Twente, The Netherlands.

The package is part of the official REDUCE distribution at Sourceforge [RED], but it is also distributed on the Geometry of Differential Equations web site `http://gdeq.org` (GDEQ for short).

We start from an installation guide for Linux and Windows. Then we focus on concrete usage recipes for the computation of higher symmetries, conservation laws, Hamiltonian and recursion operators for polynomial differential equations. All programs discussed here are shipped together with this manual and can be found at the GDEQ website. The mathematical theory on which computations are based can be found in refs. [BCD+99, KKV04].

NOTE: The new REDUCE package CDE [Vit], also distributed on `http://gdeq.org`, simplifies the use of CDIFF and extends its capabilities. Interested users may read the manual of CDE where the same computations described here for CDIFF are done in a simpler way, and further capabilities allow CDE to solve a greater variety of problems.

20.11.1 Introduction

This brief guide refers to using CDIFF, a set of symbolic computation programs devoted to computations in geometry of DEs and developed by P. Gragert, P.H.M. Kersten, G. Post and G. Roelofs at the University of Twente, The Netherlands.

Initially, the development of the CDIFF packages was started by Gragert and Kersten for symmetry computations in DEs, then they have been partly rewritten and extended by Roelofs and Post. The CDIFF packages consist of 3 program files plus a utility file; only the main three files are documented [Roe92b, Roe92a, Pos96]. The CDIFF packages, as well as a copy of the documentation (including this manual) and several example programs, can be found both at Sourceforge in the sources of REDUCE [RED] and in the Geometry of Differential Equations (GDEQ for short) web site [gde]. The name of the packages, CDIFF, comes from the fact that the package is aimed at defining differential operators in total derivatives and do computations involving them. Such operators are called *$\mathcal{C}$-differential operators* (see [BCD+99]).

The main motivation for writing this manual was that REDUCE 3.8 recently became free software, and can be downloaded here [RED]. For this reason, we are able to make our computations accessible to a wider public, also thanks to the inclusion of CDIFF in the official REDUCE distribution. The readers are warmly invited to send questions, comments, etc., both on the computations and on the technical aspects of installation and configuration of REDUCE, to the author of the present manual.

Acknowledgements. My warmest thanks are for Paul H.M. Kersten, who explained to me how to use the CDIFF packages for several computations of interest in the Geometry of Differential Equations. I also would like to thank I.S. Krasil'shchik and A.M. Verbovetsky for constant support and stimulating discussions which led me to write this text.

20.11.2 Computing with CDIFF

In order to use CDIFF it is necessary to load the package by the command

```
load_package cdiff;
```

All programs that we will discuss in this manual can be found inside the subfolder `examples` in the folder which contains this manual. In order to run them just do

```
in "filename.red";
```

at the REDUCE command prompt.

There are some conventions that I adopted on writing programs which use CDIFF.

- Program files have the extension `.red`. This will load automatically the reduce-ide mode in emacs (provided you made the installation steps described in the reduce-ide guides).
- Program files have the following names:

  `equationname_typeofcomputation_version.red`

  where `equationname` stands for the shortened name of the equation (*e.g.* Korteweg–de Vries is always indicated by KdV), `typeofcomputation` stands for the type of geometric object which is computed with the given file, for example symmetries, Hamiltonian operators, etc., `version` is a version number.
- More specific information, like the date and more details on the computation done in each version, are included as comment lines at the very beginning of each file.

Now we describe some examples of computations with CDIFF. The parts of examples which are shared between all examples are described only once. We stress that all computations presented in this document are included in the official REDUCE distribution and can be also downloaded at the GDEQ website [gde]. The examples can be run with REDUCE by typing `in "program.red";` at the REDUCE prompt, as explained above.

Remark. The mathematical theories on which the computations are based can be found in [BCD+99, KKV04].

Higher symmetries

In this section we show the computation of (some) higher symmetries of Burgers' equation $B = u_t - u_{xx} + 2uu_x = 0$. The corresponding file is `Burg_hsym_1.red` and the results of the computation are in `Burg_hsym_1_res.red`.

The idea underlying this computation is that one can use the scale symmetries of Burgers' equation to assign "gradings" to each variable appearing in the equation. As a consequence, one could try different ansatz for symmetries with polynomial generating function. For example, it is possible to require that they are sum of monomials of given degrees. This ansatz yields a simplification of the equations for symmetries, because it is possible to solve them in a "graded" way, *i.e.*, it is possible to split them into several equations made by the homogeneous components of the equation for symmetries with respect to gradings.

In particular, Burgers' equation translates into the following dimensional equation:

$$[u_t] = [u_{xx}], \quad [u_{xx} = 2uu_x].$$

By the rules $[u_z] = [u] - [z]$ and $[uv] = [u] + [v]$, and choosing $[x] = -1$, we have $[u] = 1$ and $[t] = -2$. This will be used to generate the list of homogeneous monomials of given grading to be used in the ansatz about the structure of the generating function of the symmetries.

The following instructions initialize the total derivatives. The first string is the name of the vector field, the second item is the list of even variables (note that `u1`, `u2`, ... are u_x, u_{xx}, ...), the third item is the list of odd (non-commuting) variables ('ext' stands for 'external' like in external (wedge) product). Note that in this example odd variables are not strictly needed, but it is better to insert some of them for syntax reasons.

```
super_vectorfield(ddx,{x,t,u,u1,u2,u3,u4,u5,u6,u7,
u8,u9,u10,u11,u12,u13,u14,u15,u16,u17},
{ext 1,ext 2,ext 3,ext 4,ext 5,ext 6,ext 7,ext 8,ext 9,
ext 10,ext 11,ext 12,ext 13,ext 14,ext 15,ext 16,ext 17,
ext 18,ext 19,ext 20,ext 21,ext 22,ext 23,ext 24,ext 25,
```

```
ext 26,ext 27,ext 28,ext 29,ext 30,ext 31,ext 32,ext 33,
ext 34,ext 35,ext 36,ext 37,ext 38,ext 39,ext 40,ext 41,
ext 42,ext 43,ext 44,ext 45,ext 46,ext 47,ext 48,ext 49,
ext 50,ext 51,ext 52,ext 53,ext 54,ext 55,ext 56,ext 57,
ext 58,ext 59,ext 60,ext 61,ext 62,ext 63,ext 64,ext 65,
ext 66,ext 67,ext 68,ext 69,ext 70,ext 71,ext 72,ext 73,
ext 74,ext 75,ext 76,ext 77,ext 78,ext 79,ext 80
});

super_vectorfield(ddt,{x,t,u,u1,u2,u3,u4,u5,u6,u7,
u8,u9,u10,u11,u12,u13,u14,u15,u16,u17},
{ext 1,ext 2,ext 3,ext 4,ext 5,ext 6,ext 7,ext 8,ext 9,
ext 10,ext 11,ext 12,ext 13,ext 14,ext 15,ext 16,ext 17,
ext 18,ext 19,ext 20,ext 21,ext 22,ext 23,ext 24,ext 25,
ext 26,ext 27,ext 28,ext 29,ext 30,ext 31,ext 32,ext 33,
ext 34,ext 35,ext 36,ext 37,ext 38,ext 39,ext 40,ext 41,
ext 42,ext 43,ext 44,ext 45,ext 46,ext 47,ext 48,ext 49,
ext 50,ext 51,ext 52,ext 53,ext 54,ext 55,ext 56,ext 57,
ext 58,ext 59,ext 60,ext 61,ext 62,ext 63,ext 64,ext 65,
ext 66,ext 67,ext 68,ext 69,ext 70,ext 71,ext 72,ext 73,
ext 74,ext 75,ext 76,ext 77,ext 78,ext 79,ext 80
});
```

Specification of the vectorfield `ddx`. The meaning of the first index is the parity of variables. In particular here we have just even variables. The second index parametrizes the second item (list) in the `super_vectorfield` declaration. More precisely, `ddx(0,1)` stands for $\partial/\partial x$, `ddx(0,2)` stands for $\partial/\partial t$, `ddx(0,3)` stands for $\partial/\partial u$, `ddx(0,4)` stands for $\partial/\partial u_x$, ..., and all coordinates x, t, u_x, ..., are treated as even coordinates. Note that `'$'` suppresses the output.

```
ddx(0,1):=1$
ddx(0,2):=0$
ddx(0,3):=u1$
ddx(0,4):=u2$
ddx(0,5):=u3$
ddx(0,6):=u4$
ddx(0,7):=u5$
ddx(0,8):=u6$
ddx(0,9):=u7$
ddx(0,10):=u8$
ddx(0,11):=u9$
ddx(0,12):=u10$
```

```
ddx(0,13):=u11$
ddx(0,14):=u12$
ddx(0,15):=u13$
ddx(0,16):=u14$
ddx(0,17):=u15$
ddx(0,18):=u16$
ddx(0,19):=u17$
ddx(0,20):=letop$
```

The string `letop` is treated as a variable; if it appears during computations it is likely that we went too close to the highest order variables that we defined in the file. This could mean that we need to extend the operators and variable list. In case of large output, one can search in it the string `letop` to check whether errors occurred.

Specification of the vectorfield `ddt`. In the evolutionary case we never have more than one time derivative, other derivatives are $u_{txxx\ldots}$.

```
ddt(0,1):=0$
ddt(0,2):=1$
ddt(0,3):=ut$
ddt(0,4):=ut1$
ddt(0,5):=ut2$
ddt(0,6):=ut3$
ddt(0,7):=ut4$
ddt(0,8):=ut5$
ddt(0,9):=ut6$
ddt(0,10):=ut7$
ddt(0,11):=ut8$
ddt(0,12):=ut9$
ddt(0,13):=ut10$
ddt(0,14):=ut11$
ddt(0,15):=ut12$
ddt(0,16):=ut13$
ddt(0,17):=ut14$
ddt(0,18):=letop$
ddt(0,19):=letop$
sddt(0,20):=letop$
```

We now give the equation in the form one of the derivatives equated to a right-hand side expression. The left-hand side derivative is called *principal*, and the remaining derivatives are called *parametric*[25]. For scalar evolutionary equations with two independent variables internal variables are of the type $(t, x, u, u_x, u_{xx}, \ldots)$.

[25] This terminology dates back to Riquier, see [Mar09]

```
ut:=u2+2*u*u1;

ut1:=ddx ut;
ut2:=ddx ut1;
ut3:=ddx ut2;
ut4:=ddx ut3;
ut5:=ddx ut4;
ut6:=ddx ut5;
ut7:=ddx ut6;
ut8:=ddx ut7;
ut9:=ddx ut8;
ut10:=ddx ut9;
ut11:=ddx ut10;
ut12:=ddx ut11;
ut13:=ddx ut12;
ut14:=ddx ut13;
```

Test for verifying the commutation of total derivatives. Highest order defined terms may yield some `letop`.

```
operator ev;

for i:=1:17 do
    write ev(0,i):=ddt(ddx(0,i))-ddx(ddt(0,i));
```

This is the list of variables with respect to their grading, starting from degree *one*.

```
all_graded_der:={{u},{u1},{u2},{u3},{u4},{u5},
 {u6},{u7},{u8},{u9},{u10},{u11},{u12},{u13},{u14},
 {u15},{u16},{u17}};
```

This is the list of all monomials of degree $0, 1, 2, \ldots$ which can be constructed from the above list of elementary variables with their grading.

```
grd0:={1};
grd1:= mkvarlist1(1,1)$
grd2:= mkvarlist1(2,2)$
grd3:= mkvarlist1(3,3)$
grd4:= mkvarlist1(4,4)$
grd5:= mkvarlist1(5,5)$
grd6:= mkvarlist1(6,6)$
grd7:= mkvarlist1(7,7)$
grd8:= mkvarlist1(8,8)$
grd9:= mkvarlist1(9,9)$
```

```
grd10:= mkvarlist1(10,10)$
grd11:= mkvarlist1(11,11)$
grd12:= mkvarlist1(12,12)$
grd13:= mkvarlist1(13,13)$
grd14:= mkvarlist1(14,14)$
grd15:= mkvarlist1(15,15)$
grd16:= mkvarlist1(16,16)$
```

Initialize a counter `ctel` for arbitrary constants `c`; initialize equations:

```
operator c,equ;

ctel:=0;
```

We assume a generating function `sym`, *independent of* x *and* t, of degree ≤ 5.

```
sym:=
(for each el in grd0 sum (c(ctel:=ctel+1)*el))+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))+
(for each el in grd2 sum (c(ctel:=ctel+1)*el))+
(for each el in grd3 sum (c(ctel:=ctel+1)*el))+
(for each el in grd4 sum (c(ctel:=ctel+1)*el))+
(for each el in grd5 sum (c(ctel:=ctel+1)*el))$
```

This is the equation $\bar{\ell}_B(\texttt{sym}) = 0$, where $B = 0$ is Burgers' equation and `sym` is the generating function. From now on all equations are arranged in a single vector whose name is `equ`.

```
equ 1:=ddt(sym)-ddx(ddx(sym))-2*u*ddx(sym)-2*u1*sym ;
```

This is the list of variables, to be passed to the equation solver.

```
vars:={x,t,u,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,
u12,u13,u14,u15,u16,u17};
```

This is the number of initial equation(s)

```
tel:=1;
```

The following procedure uses `multi_coeff` (from the package `tools`). It gets all coefficients of monomials appearing in the initial equation(s). The coefficients are put into the vector `equ` after the initial equations.

```
procedure splitvars i;
```

```
begin;
ll:=multi_coeff(equ i,vars);
equ(tel:=tel+1):=first ll;
ll:=rest ll;
for each el in ll do equ(tel:=tel+1):=second el;
end;
```

This command initializes the equation solver. It passes

- the equation vector `equ` togeher with its length `tel` (*i.e.*, the total number of equations);
- the list of variables with respect to which the system *must not* split the equations, *i.e.*, variables with respect to which the unknowns are not polynomial. In this case this list is just {};
- the constants'vector `c`, its length `ctel`, and the number of negative indexes if any; just `0` in our example;
- the vector of free functions `f` that may appear in computations. Note that in {`f,0,0` } the second `0` stands for the length of the vector of free functions. In this example there are no free functions, but the command needs the presence of at least a dummy argument, `f` in this case. There is also a last zero which is the negative length of the vector f, just as for constants.

```
initialize_equations(equ,tel,{},{c,ctel,0},{f,0,0});
```

Run the procedure splitvars in order to obtain equations on coefficiens of each monomial.

```
splitvars 1;
```

Next command tells the solver the total number of equations obtained after running splitvars.

```
put_equations_used tel;
```

It is worth to write down the equations for the coefficients.

```
for i:=2:tel do write equ i;
```

This command solves the equations for the coefficients. Note that we have to skip the initial equations!

```
for i:=2:tel do integrate_equation i;
```

```
;end;
```

In the folder `computations/NewTests/Higher_symmetries` it is possible to find the following files:

Burg_hsym_1.red The above file, together with its results file.

KdV_hsym_1.red Higher symmetries of KdV, with the ansatz: deg(sym) ≤ 5.

KdV_hsym_2.red Higher symmetries of KdV, with the ansatz:

sym = x*(something of degree 3) + t*(something of degree 5)
+ (something of degree 2).

This yields scale symmetries.

KdV_hsym_3.red Higher symmetries of KdV, with the ansatz:

sym = x*(something of degree 1) + t*(something of degree 3)
+ (something of degree 0).

This yields Galilean boosts.

Local conservation laws

In this section we will find (some) local conservation laws for the KdV equation $F = u_t - u_{xxx} + uu_x = 0$. Concretely, we have to find non-trivial 1-forms $f = f_x dx + f_t dt$ on $F = 0$ such that $\bar{d}f = 0$ on $F = 0$. "Triviality" of conservation laws is a delicate matter, for which we invite the reader to have a look in [BCD+99].

The files for this example are `KdV_loc-cl_1.red` and `KdV_loc-cl_2.red` and the corresponding results files.

We make use of `ddx` and `ddt`, which in the even part are the same as in the previous example (subsection 20.11.2). After defining the total derivatives we prepare the list of graded variables (recall that in KdV u is of degree 2):

```
all_graded_der:={{},{u},{u1},{u2},{u3},{u4},{u5},
 {u6},{u7},{u8},{u9},{u10},{u11},{u12},{u13},{u14},
 {u15},{u16},{u17}};
```

We make the ansatz

```
fx:=
(for each el in grd0 sum (c(ctel:=ctel+1)*el))+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))+
```

```
(for each el in grd2 sum (c(ctel:=ctel+1)*el))+
(for each el in grd3 sum (c(ctel:=ctel+1)*el))$
ft:=
(for each el in grd2 sum (c(ctel:=ctel+1)*el))+
(for each el in grd3 sum (c(ctel:=ctel+1)*el))+
(for each el in grd4 sum (c(ctel:=ctel+1)*el))+
(for each el in grd5 sum (c(ctel:=ctel+1)*el))$
```

for the components of the conservation law. We have to solve the equation

```
equ 1:=ddt(fx)-ddx(ft);
```

the fact that `ddx` and `ddt` are expressed in internal coordinates on the equation means that the objects that we consider are already restricted to the equation.

We shall split the equation in its graded summands with the procedure `splitvars`, then solve it

```
initialize_equations(equ,tel,{},{c,ctel,0},{f,0,0});
splitvars 1;
pte tel;
for i:=2:tel do es i;
end;
```

As a result we get

```
fx := c(3)*u1 + c(2)*u + c(1)$
ft := (2*c(3)*u*u1 + 2*c(3)*u3
        + c(2)*u**2 + 2*c(2)*u2)/2$
```

Unfortunately it is clear that the conservation law corresponding to `c(3)` is trivial, because it is the total x-derivative of F; its restriction on the infinite prolongation of the KdV is zero. Here this fact is evident; how to get rid of less evident trivialities by an 'automatic' mechanism? We considered this problem in the file `KdV_loc-cl_2.red`, where we solved the equation

```
equ 1:=fx-ddx(f0);
equ 2:=ft-ddt(f0);
```

after having loaded the values `fx` and `ft` found by the previous program. We make the following ansatz on `f0`:

```
f0:=
(for each el in grd0 sum (cc(cctel:=cctel+1)*el))+
(for each el in grd1 sum (cc(cctel:=cctel+1)*el))+
```

```
(for each el in grd2 sum (cc(cctel:=cctel+1)*el))+
(for each el in grd3 sum (cc(cctel:=cctel+1)*el))$
```

Note that this gives a grading which is compatible with the gradings of `fx` and `ft`. After solving the system

```
initialize_equations(equ,tel,{},{cc,cctel,0},{f,0,0});
for i:=1:2 do begin splitvars i;end;
pte tel;
for i:=3:tel do es i;
end;
```

issuing the commands

```
fxnontriv := fx-ddx(f0);
ftnontriv := ft-ddt(f0);
```

we obtain

```
fxnontriv := c(2)*u + c(1)$
ftnontriv := (c(2)*(u**2 + 2*u2))/2$
```

This mechanism can be easily generalized to situations in which the conservation laws which are found by the program are difficult to treat by pen and paper.

Local Hamiltonian operators

In this section we will find local Hamiltonian operators for the KdV equation $u_t = u_{xxx} + uu_x$. Concretely, we have to solve $\bar{\ell}_{KdV}(\texttt{phi}) = 0$ over the equation

$$\begin{cases} u_t = u_{xxx} + uu_x \\ p_t = p_{xxx} + up_x \end{cases}$$

or, in geometric terminology, find the shadows of symmetries on the ℓ^*-covering of the KdV equation. The reference paper for this type of computations is [KKV04].

The file containing this example is `KdV_Ham_1.red`.

We make use of `ddx` and `ddt`, which in the even part are the same as in the previous example (subsection 20.11.2). We stress that the linearization $\bar{\ell}_{KdV}(\texttt{phi}) = 0$ is the equation

```
ddt(phi)-u*ddx(phi)-u1*phi-ddx(ddx(ddx(phi)))=0
```

but the total derivatives are lifted to the ℓ^* covering, hence they must contain also derivatives with respect to p's. This will be achieved by treating p variables as odd

and introducing the odd parts of `ddx` and `ddt`,

```
ddx(1,1):=0$
ddx(1,2):=0$
ddx(1,3):=ext 4$
ddx(1,4):=ext 5$
ddx(1,5):=ext 6$
ddx(1,6):=ext 7$
ddx(1,7):=ext 8$
ddx(1,8):=ext 9$
ddx(1,9):=ext 10$
ddx(1,10):=ext 11$
ddx(1,11):=ext 12$
ddx(1,12):=ext 13$
ddx(1,13):=ext 14$
ddx(1,14):=ext 15$
ddx(1,15):=ext 16$
ddx(1,16):=ext 17$
ddx(1,17):=ext 18$
ddx(1,18):=ext 19$
ddx(1,19):=ext 20$
ddx(1,20):=letop$
```

In the above definition the first index '1' says that we are dealing with odd variables, `ext` indicates anticommuting variables. Here, `ext 3` is p_0, `ext 4` is p_x, `ext 5` is p_{xx}, ... so `ddx(1,3):=ext 4` indicates $p_x\partial/\partial p$, etc..

Now, remembering that the additional equation is again evolutionary, we can get rid of p_t by letting it be equal to `ext 6 + u*ext 4`, as follows:

```
ddt(1,1):=0$
ddt(1,2):=0$
ddt(1,3):=ext 6 + u*ext 4$
ddt(1,4):=ddx(ddt(1,3))$
ddt(1,5):=ddx(ddt(1,4))$
ddt(1,6):=ddx(ddt(1,5))$
ddt(1,7):=ddx(ddt(1,6))$
ddt(1,8):=ddx(ddt(1,7))$
ddt(1,9):=ddx(ddt(1,8))$
ddt(1,10):=ddx(ddt(1,9))$
ddt(1,11):=ddx(ddt(1,10))$
ddt(1,12):=ddx(ddt(1,11))$
ddt(1,13):=ddx(ddt(1,12))$
ddt(1,14):=ddx(ddt(1,13))$
```

```
ddt(1,15):=ddx(ddt(1,14))$
ddt(1,16):=ddx(ddt(1,15))$
ddt(1,17):=ddx(ddt(1,16))$
ddt(1,18):=letop$
ddt(1,19):=letop$
ddt(1,20):=letop$
```

Let us make the following ansatz about the Hamiltonian operators:

```
phi:=
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd3 sum (c(ctel:=ctel+1)*el))*ext 3+

(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 4+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 4+
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 4+

(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 5+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 5+

(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 6
$
```

Note that we are looking for generating functions of shadows which are *linear* with respect to p's. Moreover, having set $[p] = -2$ we will look for solutions of maximal possible degree $+1$.

After having set

```
equ 1:=ddt(phi)-u*ddx(phi)-u1*phi-ddx(ddx(ddx(phi)));
vars:={x,t,u,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,u12,
 u13,u14,u15,u16,u17};
tel:=1;
```

we define the procedures `splitvars` as in subsection 20.11.2 and `splitext` as follows:

```
procedure splitext i;
begin;
ll:=operator_coeff(equ i,ext);
equ(tel:=tel+1):=first ll;
ll:=rest ll;
```

```
for each el in ll do equ(tel:=tel+1):=second el;
end;
```

Then we initialize the equations:

```
initialize_equations(equ,tel,{},{c,ctel,0},{f,0,0});
```

do `splitext`

```
splitext 1;
```

then `splitvars`

```
tel1:=tel;
for i:=2:tel1 do begin splitvars i;equ i:=0;end;
```

Now we are ready to solve all equations:

```
put_equations_used tel;
for i:=2:tel do write equ i:=equ i;
pause;
for i:=2:tel do integrate_equation i;
end;
```

Note that we want *all* equations to be solved!

The results are the two well-known Hamiltonian operators for the KdV:

```
phi := c(4)*ext(4) + 3*c(3)*ext(6) + 2*c(3)*ext(4)*u
 + c(3)*ext(3)*u1$
```

Of course, the results correspond to the operators

$$\texttt{ext(4)} \to D_x,$$
$$\texttt{3*c(3)*ext(6) + 2*c(3)*ext(4)*u + c(3)*ext(3)*u1} \to 3D_{xxx} + 2uD_x + u_x.$$

Note that each operator is multiplied by one arbitrary real constant, `c(4)` and `c(3)`.

Non-local Hamiltonian operators

In this section we will show an experimental way to find nonlocal Hamiltonian operators for the KdV equation. The word 'experimental' comes from the lack of a

consistent mathematical theory. The result of the computation (without the details below) has been published in [KKV04].

We have to solve equations of the type `ddx(ft)-ddt(fx)` as in 20.11.2. The main difference is that we will attempt a solution on the ℓ^*-covering (see Subsection 20.11.2). For this reason, first of all we have to determine covering variables with the usual mechanism of introducing them through conservation laws, this time on the ℓ^*-covering.

As a first step, let us compute conservation laws on the ℓ^*-covering whose components are linear in the p's. This computation can be found in the file `KdV_nloc-cl_1.red` and related results file. When specifying odd variables in `ddx` and `ddt`, we have something like

```
ddx(1,1):=0$
ddx(1,2):=0$
ddx(1,3):=ext 4$
ddx(1,4):=ext 5$
ddx(1,5):=ext 6$
ddx(1,6):=ext 7$
ddx(1,7):=ext 8$
ddx(1,8):=ext 9$
ddx(1,9):=ext 10$
ddx(1,10):=ext 11$
ddx(1,11):=ext 12$
ddx(1,12):=ext 13$
ddx(1,13):=ext 14$
ddx(1,14):=ext 15$
ddx(1,15):=ext 16$
ddx(1,16):=ext 17$
ddx(1,17):=ext 18$
ddx(1,18):=ext 19$
ddx(1,19):=ext 20$
ddx(1,20):=letop$
ddx(1,50):=(t*u1+1)*ext 3$ % degree -2
ddx(1,51):=u1*ext 3$ % degree +1
ddx(1,52):=(u*u1+u3)*ext 3$ % degree +3
```

and

```
ddt(1,1):=0$
ddt(1,2):=0$
ddt(1,3):=ext 6 + u*ext 4$
ddt(1,4):=ddx(ddt(1,3))$
ddt(1,5):=ddx(ddt(1,4))$
```

```
ddt(1,6):=ddx(ddt(1,5))$
ddt(1,7):=ddx(ddt(1,6))$
ddt(1,8):=ddx(ddt(1,7))$
ddt(1,9):=ddx(ddt(1,8))$
ddt(1,10):=ddx(ddt(1,9))$
ddt(1,11):=ddx(ddt(1,10))$
ddt(1,12):=ddx(ddt(1,11))$
ddt(1,13):=ddx(ddt(1,12))$
ddt(1,14):=ddx(ddt(1,13))$
ddt(1,15):=ddx(ddt(1,14))$
ddt(1,16):=ddx(ddt(1,15))$
ddt(1,17):=ddx(ddt(1,16))$
ddt(1,18):=letop$
ddt(1,19):=letop$
ddt(1,20):=letop$
ddt(1,50):=f1*ext 3+f2*ext 4+f3*ext 5$
ddt(1,51):=f4*ext 3+f5*ext 4+f6*ext 5$
ddt(1,52):=f7*ext 3+f8*ext 4+f9*ext 5$
```

The variables corresponding to the numbers `50,51,52` here play a dummy role, the coefficients of the corresponding vector are the unknown generating functions of conservation laws on the ℓ^*-covering. More precisely, we look for conservation laws of the form

```
fx= phi*ext 3
ft= f1*ext3+f2*ext4+f3*ext5
```

The ansatz is chosen because, first of all, `ext 4` and `ext 5` can be removed from fx by adding a suitable total divergence (trivial conservation law); moreover it can be proved that `phi` is a symmetry of KdV. We can write down the equations

```
equ 1:=ddx(ddt(1,50))-ddt(ddx(1,50));
equ 2:=ddx(ddt(1,51))-ddt(ddx(1,51));
equ 3:=ddx(ddt(1,52))-ddt(ddx(1,52));
```

However, the above choices make use of a symmetry which contains `'t'` in the generator. This would make automatic computations more tricky, but still possible. In this case the solution of `equ 1` has been found by hand and passed to the program:

```
f3:=t*u1+1$
f1:=u*f3+ddx(ddx(f3))$
f2:=-ddx(f3)$
```

together with the ansatz on the coefficients for the other equations

```
f4:=(for each el in grd5 sum (c(ctel:=ctel+1)*el))$
f5:=(for each el in grd4 sum (c(ctel:=ctel+1)*el))$
f6:=(for each el in grd3 sum (c(ctel:=ctel+1)*el))$

f7:=(for each el in grd7 sum (c(ctel:=ctel+1)*el))$
f8:=(for each el in grd6 sum (c(ctel:=ctel+1)*el))$
f9:=(for each el in grd5 sum (c(ctel:=ctel+1)*el))$
```

The previous ansatz keep into account the grading of the starting symmetry in `phi*ext 3`. The resulting equations are solved in the usual way (see the example file).

Now, we solve the equation for shadows of nonlocal symmetries in a covering of the ℓ^*-covering. We can choose between three new nonlocal variables `ra,rb,rc`. We are going to look for non-local Hamiltonian operators depending linearly on one of these variables. Higher non-local Hamiltonian operators could be found by introducing total derivatives of the r's. As usual, the new variables are specified through the components of the previously found conservation laws according with the rule

```
ra_x=fx, ra_t=ft,
```

and analogously for the others. We define

```
ddx(1,50):=(t*u1+1)*ext 3$ % degree -2
ddx(1,51):=u1*ext 3$ % degree +1
ddx(1,52):=(u*u1+u3)*ext 3$ % degree +3
```

and

```
ddt(1,50) := ext(5)*t*u1 + ext(5) - ext(4)*t*u2
 + ext(3)*t*u*u1 + ext(3)*t*u3 + ext(3)*u$
ddt(1,51) := ext(5)*u1 - ext(4)*u2 + ext(3)*u*u1
 + ext(3)*u3$
ddt(1,52) := ext(5)*u*u1 + ext(5)*u3 - ext(4)*u*u2
 - ext(4)*u1**2 - ext(4)*u4 + ext(3)*u**2*u1
 + 2*ext(3)*u*u3 + 3*ext(3)*u1*u2 + ext(3)*u5$
```

as it results from the computation of the conservation laws. The following ansatz for the nonlocal Hamiltonian operator comes from the fact that local Hamiltonian operators have gradings -1 and $+1$ when written in terms of p's. So we are looking for a nonlocal Hamiltonian operator of degree 3.

```
phi:=
(for each el in grd6 sum (c(ctel:=ctel+1)*el))*ext 50+
(for each el in grd3 sum (c(ctel:=ctel+1)*el))*ext 51+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 52+

(for each el in grd5 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd4 sum (c(ctel:=ctel+1)*el))*ext 4+
(for each el in grd3 sum (c(ctel:=ctel+1)*el))*ext 5+
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 6+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 7+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 8
$
```

As a solution, we obtain

```
phi := c(1)*(ext(51)*u1 - 9*ext(8) - 12*ext(6)*u
 - 18*ext(5)*u1 - 4*ext(4)*u**2 - 12*ext(4)*u2
 - 4*ext(3)*u*u1 - 3*ext(3)*u3)$
```

where `ext51` stands for the nonlocal variable `rb` fulfilling

```
rb_x:=u1*ext 3$
rb_t:=ext(5)*u1 - ext(4)*u2 + ext(3)*u*u1 + ext(3)*u3$
```

Remark. In the file `KdV_nloc-Ham_2.red` it is possible to find another ansatz for a non-local Hamiltonian operator of degree $+5$.

Computations for systems of PDEs

There is no conceptual difference when computing for systems of PDEs. We will look for Hamiltonian structures for the following Boussinesq equation:

$$\begin{cases} u_t - u_x v - u v_x - \sigma v_{xxx} = 0 \\ v_t - u_x - v v_x = 0 \end{cases} \tag{20.71}$$

where σ is a constant. This example also shows how to deal with jet spaces with more than one dependent variable. Here gradings can be taken as

$$[t] = -2,\ \ [x] = -1,\ \ [v] = 1,\ \ [u] = 2,\ \ [p] = [\frac{\partial}{\partial u}] = -2,\ \ [q] = [\frac{\partial}{\partial v}] = -1$$

where p, q are the two coordinates in the space of generating functions of conservation laws.

The linearization of the above system and its adjoint are, respectively

$$\ell_{\text{Bou}} = \begin{pmatrix} D_t - vD_x - v_x & -u_x - uD_x - \sigma D_{xxx} \\ -D_x & D_t - v_x - vD_x \end{pmatrix},$$
$$\ell^*_{\text{Bou}} = \begin{pmatrix} -D_t + vD_x & D_x \\ uD_x + \sigma D_{xxx} & -D_t + vD_x \end{pmatrix}$$

and lead to the ℓ^*_{Bou} covering equation

$$\begin{cases} -p_t + vp_x + q_x = 0 \\ up_x + \sigma p_{xxx} - q_t + vq_x = 0 \\ u_t - u_x v - uv_x - \sigma v_{xxx} = 0 \\ v_t - u_x - vv_x = 0 \end{cases}$$

We have to find shadows of symmetries on the above covering. Total derivatives must be defined as follows:

```
super_vectorfield(ddx,{x,t,u,v,u1,v1,u2,v2,u3,v3,u4,v4,
u5,v5,u6,v6,u7,v7,u8,v8,u9,v9,u10,v10,u11,v11,u12,v12,
u13,v13,u14,v14,u15,v15,u16,v16,u17,v17},
{ext 1,ext 2,ext 3,ext 4,ext 5,ext 6,ext 7,ext 8,ext 9,
ext 10,ext 11,ext 12,ext 13,ext 14,ext 15,ext 16,ext 17,
ext 18,ext 19,ext 20,ext 21,ext 22,ext 23,ext 24,ext 25,
ext 26,ext 27,ext 28,ext 29,ext 30,ext 31,ext 32,ext 33,
ext 34,ext 35,ext 36,ext 37,ext 38,ext 39,ext 40,ext 41,
ext 42,ext 43,ext 44,ext 45,ext 46,ext 47,ext 48,ext 49,
ext 50,ext 51,ext 52,ext 53,ext 54,ext 55,ext 56,ext 57,
ext 58,ext 59,ext 60,ext 61,ext 62,ext 63,ext 64,ext 65,
ext 66,ext 67,ext 68,ext 69,ext 70,ext 71,ext 72,ext 73,
ext 74,ext 75,ext 76,ext 77,ext 78,ext 79,ext 80
});

super_vectorfield(ddt,{x,t,u,v,u1,v1,u2,v2,u3,v3,u4,v4,
u5,v5,u6,v6,u7,v7,u8,v8,u9,v9,u10,v10,u11,v11,u12,v12,
u13,v13,u14,v14,u15,v15,u16,v16,u17,v17},
{ext 1,ext 2,ext 3,ext 4,ext 5,ext 6,ext 7,ext 8,ext 9,
ext 10,ext 11,ext 12,ext 13,ext 14,ext 15,ext 16,ext 17,
ext 18,ext 19,ext 20,ext 21,ext 22,ext 23,ext 24,ext 25,
ext 26,ext 27,ext 28,ext 29,ext 30,ext 31,ext 32,ext 33,
ext 34,ext 35,ext 36,ext 37,ext 38,ext 39,ext 40,ext 41,
ext 42,ext 43,ext 44,ext 45,ext 46,ext 47,ext 48,ext 49,
ext 50,ext 51,ext 52,ext 53,ext 54,ext 55,ext 56,ext 57,
ext 58,ext 59,ext 60,ext 61,ext 62,ext 63,ext 64,ext 65,
ext 66,ext 67,ext 68,ext 69,ext 70,ext 71,ext 72,ext 73,
ext 74,ext 75,ext 76,ext 77,ext 78,ext 79,ext 80
});
```

In the list of coordinates we alternate derivatives of u and derivatives of v. The same must be done for coefficients; for example,

```
ddx(0,1):=1$
ddx(0,2):=0$
ddx(0,3):=u1$
ddx(0,4):=v1$
ddx(0,5):=u2$
ddx(0,6):=v2$
...
```

After specifying the equation

```
ut:=u1*v+u*v1+sig*v3;
vt:=u1+v*v1;
```

we define the (already introduced) time derivatives:

```
ut1:=ddx ut;
ut2:=ddx ut1;
ut3:=ddx ut2;
...
vt1:=ddx vt;
vt2:=ddx vt1;
vt3:=ddx vt2;
...
```

up to the required order (here the order can be stopped at 15). Odd variables p and q must be specified with an appropriate length (here it is OK to stop at `ddx(1,36)`). Recall to replace p_t, q_t with the internal coordinates of the covering:

```
ddt(1,1):=0$
ddt(1,2):=0$
ddt(1,3):=+v*ext 5+ext 6$
ddt(1,4):=u*ext 5+sig*ext 9+v*ext 6$
ddt(1,5):=ddx(ddt(1,3))$
...
```

The list of graded variables:

```
all_graded_der:={{v},{u,v1},{u1,v2},{u2,v3},{u3,v4},
 {u4,v5},{u5,v6},{u6,v7},{u7,v8},{u8,v9},{u9,v10},
 {u10,v11},{u11,v12},{u12,v13},{u13,v14},{u14,v15},
 {u15,v16},{u16,v17},{u17}};
```

The ansatz for the components of the Hamiltonian operator is

```
phi1:=
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 5+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 4+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 6
$

phi2:=
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 5+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 4
$
```

and the equation for shadows of symmetries is

```
equ 1:=ddt(phi1)-v*ddx(phi1)-v1*phi1-u1*phi2-u*ddx(phi2)
-sig*ddx(ddx(ddx(phi2)));
equ 2:=-ddx(phi1)-v*ddx(phi2)-v1*phi2+ddt(phi2);
```

After the usual procedures for decomposing polynomials we obtain the following result:

```
phi1 := c(6)*ext(6)$
phi2 := c(6)*ext(5)$
```

which corresponds to the vector (D_x, D_x). Extending the ansatz to

```
phi1:=
(for each el in grd3 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 5+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 7+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 9+
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 4+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 6+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 8
$

phi2:=
(for each el in grd2 sum (c(ctel:=ctel+1)*el))*ext 3+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 5+
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 7+
(for each el in grd1 sum (c(ctel:=ctel+1)*el))*ext 4+
```

```
(for each el in grd0 sum (c(ctel:=ctel+1)*el))*ext 6
$
```

allows us to find a second (local) Hamiltonian operator

```
phi1 := (c(3)*(2*ext(9)*sig + ext(6)*v + 2*ext(5)*u
 + ext(3)*u1))/2$
phi2 := (c(3)*(2*ext(6) + ext(5)*v + ext(3)*v1))/2$
```

There is one more higher local Hamiltonian operator, and a whole hierarchy of nonlocal Hamiltonian operators [KKV04].

Explosion of denominators and how to avoid it

Here we propose the computation of the repeated total derivative of a denominator. This computation fills up the whole memory after some time, and can be used as a kind of speed test for the system. The file is `KdV_denom_1.red`.

After having defined total derivatives on the KdV equation, run the following iteration:

```
phi:=1/(u3+u*u1)$
for i:=1:100 do begin
                phi:=ddx(phi)$
                write i;
end;
```

The program shows the iteration number. At the 18th iteration the program uses about 600MB of RAM, as shown by `top` run from another shell, and 100% of one processor.

There is a simple way to avoid denominator explosion. The file for this example is `KdV_denom_2.red`.

After having defined total derivatives with respect to x (on the KdV equation, for example) consider in the same `ddx` a component with a sufficently high index **immediately after 'letop'** (otherwise `super_vectorfield` does not work!), say `ddx(0,21)`, and think of it as being the coefficient to a vector of the type

```
aa21:=1/(u3+u*u1);
```

In this case, its coefficient must be

```
ddx(0,21):=-aa21**2*(u4+u1**2+u*u2)$
```

More particularly, here follows the detailed definition of `ddx`

```
ddx(0,1):=1$
ddx(0,2):=0$
ddx(0,3):=u1$
ddx(0,4):=u2$
ddx(0,5):=u3$
ddx(0,6):=u4$
ddx(0,7):=u5$
ddx(0,8):=u6$
ddx(0,9):=u7$
ddx(0,10):=u8$
ddx(0,11):=u9$
ddx(0,12):=u10$
ddx(0,13):=u11$
ddx(0,14):=u12$
ddx(0,15):=u13$
ddx(0,16):=u14$
ddx(0,17):=u15$
ddx(0,18):=u16$
ddx(0,19):=u17$
ddx(0,20):=letop$
ddx(0,21):=-aa21**2*(u4+u1**2+u*u2)$
```

Now, suppose that we want to compute the 5th total derivative of `phi`. Write the following code:

```
phi:=aa30;
for i:=1:5 do begin
                phi:=ddx(phi)$
                write i;
end;
```

The result is then a polynomial in the additional 'denominator' variable

```
phi := aa21**2*( - 120*aa21**4*u**5*u2**5
 - 600*aa21**4*u**4*u1**2*u2**4
 - 600*aa21**4*u**4*u2**4*u4
 - 1200*aa21**4*u**3*u1**4*u2**3
 - 2400*aa21**4*u**3*u1**2*u2**3*u4
 - 1200*aa21**4*u**3*u2**3*u4**2
 - 1200*aa21**4*u**2*u1**6*u2**2
 - 3600*aa21**4*u**2*u1**4*u2**2*u4
 - 3600*aa21**4*u**2*u1**2*u2**2*u4**2
```

```
- 1200*aa21**4*u**2*u2**2*u4**3
- 600*aa21**4*u*u1**8*u2
- 2400*aa21**4*u*u1**6*u2*u4
- 3600*aa21**4*u*u1**4*u2*u4**2
- 2400*aa21**4*u*u1**2*u2*u4**3
- 600*aa21**4*u*u2*u4**4
- 120*aa21**4*u1**10 - 600*aa21**4*u1**8*u4
- 1200*aa21**4*u1**6*u4**2 - 1200*aa21**4*u1**4*u4**3
- 600*aa21**4*u1**2*u4**4 - 120*aa21**4*u4**5
+ 240*aa21**3*u**4*u2**3*u3
+ 720*aa21**3*u**3*u1**2*u2**2*u3
+ 720*aa21**3*u**3*u1*u2**4
+ 240*aa21**3*u**3*u2**3*u5
+ 720*aa21**3*u**3*u2**2*u3*u4
+ 720*aa21**3*u**2*u1**4*u2*u3
+ 2160*aa21**3*u**2*u1**3*u2**3
+ 720*aa21**3*u**2*u1**2*u2**2*u5
+ 1440*aa21**3*u**2*u1**2*u2*u3*u4
+ 2160*aa21**3*u**2*u1*u2**3*u4
+ 720*aa21**3*u**2*u2**2*u4*u5
+ 720*aa21**3*u**2*u2*u3*u4**2 + 240*aa21**3*u*u1**6*u3
+ 2160*aa21**3*u*u1**5*u2**2
+ 720*aa21**3*u*u1**4*u2*u5
+ 720*aa21**3*u*u1**4*u3*u4
+ 4320*aa21**3*u*u1**3*u2**2*u4
+ 1440*aa21**3*u*u1**2*u2*u4*u5
+ 720*aa21**3*u*u1**2*u3*u4**2
+ 2160*aa21**3*u*u1*u2**2*u4**2
+ 720*aa21**3*u*u2*u4**2*u5
+ 240*aa21**3*u*u3*u4**3 + 720*aa21**3*u1**7*u2
+ 240*aa21**3*u1**6*u5
+ 2160*aa21**3*u1**5*u2*u4 + 720*aa21**3*u1**4*u4*u5
+ 2160*aa21**3*u1**3*u2*u4**2
+ 720*aa21**3*u1**2*u4**2*u5
+ 720*aa21**3*u1*u2*u4**3 + 240*aa21**3*u4**3*u5
- 60*aa21**2*u**3*u2**2*u4 - 90*aa21**2*u**3*u2*u3**2
- 120*aa21**2*u**2*u1**2*u2*u4
- 90*aa21**2*u**2*u1**2*u3**2
- 780*aa21**2*u**2*u1*u2**2*u3 - 180*aa21**2*u**2*u2**4
- 60*aa21**2*u**2*u2**2*u6 - 180*aa21**2*u**2*u2*u3*u5
- 120*aa21**2*u**2*u2*u4**2 - 90*aa21**2*u**2*u3**2*u4
- 60*aa21**2*u*u1**4*u4 - 1020*aa21**2*u*u1**3*u2*u3
- 1170*aa21**2*u*u1**2*u2**3
- 120*aa21**2*u*u1**2*u2*u6
```

```
- 180*aa21**2*u*u1**2*u3*u5 - 120*aa21**2*u*u1**2*u4**2
- 540*aa21**2*u*u1*u2**2*u5
- 1020*aa21**2*u*u1*u2*u3*u4
- 360*aa21**2*u*u2**3*u4 - 120*aa21**2*u*u2*u4*u6
- 90*aa21**2*u*u2*u5**2 - 180*aa21**2*u*u3*u4*u5
- 60*aa21**2*u*u4**3 - 240*aa21**2*u1**5*u3
- 990*aa21**2*u1**4*u2**2 - 60*aa21**2*u1**4*u6
- 540*aa21**2*u1**3*u2*u5 - 480*aa21**2*u1**3*u3*u4
- 1170*aa21**2*u1**2*u2**2*u4 - 120*aa21**2*u1**2*u4*u6
- 90*aa21**2*u1**2*u5**2 - 540*aa21**2*u1*u2*u4*u5
- 240*aa21**2*u1*u3*u4**2 - 180*aa21**2*u2**2*u4**2
- 60*aa21**2*u4**2*u6 - 90*aa21**2*u4*u5**2
+ 10*aa21*u**2*u2*u5 + 20*aa21*u**2*u3*u4
+ 10*aa21*u*u1**2*u5 + 110*aa21*u*u1*u2*u4
+ 80*aa21*u*u1*u3**2 + 160*aa21*u*u2**2*u3
+ 10*aa21*u*u2*u7 + 20*aa21*u*u3*u6 + 30*aa21*u*u4*u5
+ 50*aa21*u1**3*u4 + 340*aa21*u1**2*u2*u3
+ 10*aa21*u1**2*u7 + 180*aa21*u1*u2**3
+ 60*aa21*u1*u2*u6 + 80*aa21*u1*u3*u5
+ 50*aa21*u1*u4**2 + 60*aa21*u2**2*u5
+ 100*aa21*u2*u3*u4 + 10*aa21*u4*u7 + 20*aa21*u5*u6
- u*u6 - 6*u1*u5 - 15*u2*u4 - 10*u3**2 - u8)$
```

where the value of `aa21` can be replaced back in the expression.

20.12 CGB: Computing Comprehensive Gröbner Bases

Authors: Andreas Dolzmann, Thomas Sturm, and Winfried Neun

20.12.1 Introduction

Consider the ideal basis $F = \{ax, x + y\}$. Treating a as a parameter, the calling sequence

```
torder({x,y},lex)$
groebner{a*x,x+y};

  {x,y}
```

yields $\{x, y\}$ as reduced Gröbner basis. This is, however, not correct under the specialization $a = 0$. The reduced Gröbner basis would then be $\{x + y\}$. Taking these results together, we obtain $C = \{x + y, ax, ay\}$, which is correct wrt. *all* specializations for a including zero specializations. We call this set C a *comprehensive Gröbner basis* (CGB).

The notion of a CGB and a corresponding algorithm has been introduced bei Weispfenning [Wei92]. This algorithm works by performing case distinctions wrt. parametric coefficient polynomials in order to find out what the head monomials are under all possible specializations. It does thus not only determine a CGB, but even classifies the contained polynomials wrt. the specializations they are relevant for. If we keep the Gröbner bases for all cases separate and associate information on the respective specializations with them, we obtain a *Gröbner system*. For our example, the Gröbner system is the following;

$$\left[\begin{array}{c|c} a \neq 0 & \{x + y, ax, ay\} \\ a = 0 & \{x + y\} \end{array}\right].$$

A CGB is obtained as the union of the single Gröbner bases in a Gröbner system. It has also been shown that, on the other hand, a Gröbner system can easily be reconstructed from a given CGB [Wei92].

The CGB package provides functions for computing both CGB's and Gröbner systems, and for turning Gröbner systems into CGB's.

20.12.2 Using the REDLOG Package

For managing the conditions occurring with the CGB computations, the CGB package uses the package REDLOG implementing first-order formulas, [DS97a, DS99], which is also part of the REDUCE distribution.

20.12.3 Term Ordering Mode

The CGB package uses the settings made with the function `torder` of the GROEBNER package. This includes in particular the choice of the main variables. All variables not mentioned in the variable list argument of `torder` are parameters. The only term ordering modes recognized by CGB are `lex` and `revgradlex`.

20.12.4 CGB: Comprehensive Gröbner Basis

The function `cgb` expects a list F of expressions. It returns a CGB of F wrt. the current `torder` setting.

Example

```
torder({x,y},lex)$
cgb{a*x+y,x+b*y};

{x + b*y,a*x + y,(a*b - 1)*y}

ws;

{b*y + x,

 a*x + y,

 y*(a*b - 1)}
```

Note that the basis returned by the `cgb` call has not undergone the standard evaluation process: The returned polynomials are ordered wrt. the chosen term order. Reevaluation changes this as can be seen with the output of `ws`.

20.12.5 GSYS: Gröbner System

The function `gsys` follows the same calling conventions as `cgb`. It returns the complete Gröbner system represented as a nested list

$$\big\{\big\{c_1, \{g_{11}, \ldots, g_{1n_1}\}\big\}, \ldots, \big\{c_m, \{g_{m1}, \ldots, g_{1n_m}\}\big\}\big\}.$$

The c_i are conditions in the parameters represented as quantifier-free REDLOG formulas. Each choice of parameters will obey at least one of the c_i. Whenever a

choice of parameters obeys some c_i, the corresponding $\{g_{i1}, \ldots, g_{in_i}\}$ is a Gröbner basis for this choice.

Example

```
torder({x,y},lex)$
gsys {a*x+y,x+b*y};

{{a*b - 1 <> 0 and a <> 0,

  {a*x + y,x + b*y,(a*b - 1)*y}},

 {a <> 0 and a*b - 1 = 0,

  {a*x + y,x + b*y}},

 {a = 0,{a*x + y,x + b*y}}}
```

As with the function `cgb`, the contained polynomials remain unevaluated.

Computing a Gröbner system is not harder than computing a CGB. In fact, `cgb` also computes a Gröbner system and then turns it into a CGB.

Switch CGBGEN: Only the Generic Case

If the switch `cgbgen` is turned on, both `gsys` and `cgb` will assume all parametric coefficients to be non-zero ignoring the other cases. For `cgb` this means that the result equals—up to auto-reduction—that of `groebner`. A call to `gsys` will return this result as a single case including the assumptions made during the computation:

Example

```
torder({x,y},lex)$
on cgbgen;
gsys{a*x+y,x+b*y};

{{a*b - 1 <> 0 and a <> 0,

  {a*x + y,x + b*y,(a*b - 1)*y}}}

off cgbgen;
```

20.12.6 GSYS2CGB: Gröbner System to CGB

The call `gsys2cgb` turns a given Gröbner system into a CGB by constructing the union of the Gröbner bases of the single cases.

Example

```
torder({x,y},lex)$
gsys{a*x+y,x+b*y}$
gsys2cgb ws;

{x + b*y,a*x + y,(a*b - 1)*y}
```

20.12.7 Switch CGBREAL: Computing over the Real Numbers

All computations considered so far have taken place over the complex numbers, more precisely, over algebraically closed fields. Over the real numbers, certain branches of the CGB computation can become inconsitent though they are not inconsistent over the complex numbers. Consider, e.g., a condition $a^2 + 1 = 0$.

When turning on the switch `cgbreal`, all simplifications of conditions are performed over the real numbers. The methods used for this are described in [DS97b].

Example

```
torder({x,y},lex)$
off cgbreal;
gsys {a*x+y,x-a*y};

   2
{{a  + 1 <> 0 and a <> 0,

                      2
  {a*x + y,x - a*y,(a  + 1)*y}},

              2
 {a <> 0 and a  + 1 = 0,{a*x + y,x - a*y}},

 {a = 0,{a*x + y,x - a*y}}}

on cgbreal;
gsys({a*x+y,x-a*y});
```

```
{{a <> 0,

                          2
  {a*x + y,x - a*y,(a  + 1)*y}},

 {a = 0,{a*x + y,x - a*y}}}
```

20.12.8 Switches

cgbreal Compute over the real numbers. See Section 20.12.7 for details.

cgbgs Gröbner simplification of the condition. The switch `cgbgs` can be turned on for applying advanced algebraic simplification techniques to the conditions. This will, in general, slow down the computation, but lead to a simpler Gröbner system.

cgbstat Statistics of the CGB run. The switch `cgbstat` toggles the creation and output of statistical information on the CGB run. The statistical information is printed at the end of the run.

cgbfullred Full reduction. By default, the CGB functions perform full reductions in contrast to pure top reductions. By turning off the switch `cgbfullred`, reduction can be restricted to top reductions.

20.13 COEFF2: A Variant of the `coeff` Operator

Authors: Fujio Kako and Masaaki Ito

In REDUCE, we can use the `coeff` operator which returns a list of coefficients of a polynomial with respect to specified variables. On the other hand, the `coeff2` operator gives a polynomial in which each coefficient is replaced by special variables `#1`, `#2`,$\cdots$. It is used with the same syntax as the `coeff` operator:

`coeff2(`⟨*exprn:polynomial*⟩, ⟨*var:kernel*⟩`)` : *algebraic*

Example:

```
off allfac;
f := (a+b)^2*x^2*y+(c+d)^2*x*y;
f2 := coeff2(f,x,y);
g := (2*c+d)*x^2+(3+a)*x*y^3;
g2 := coeff2(g,x,y);
```

would result in the output

```
      2  2             2      2  2      2
f := a *x *y + 2*a*b*x *y + b *x *y + c *x*y

                     2
      + 2*c*d*x*y + d *x*y

         2
f2 := #1*x *y + #2*x*y

          3         2      2         3
g := a*x*y  + 2*c*x  + d*x  + 3*x*y

          2          3
g2 := #3*x  + #4*x*y
```

If you want to retrieve the values of special variables `#1`, `#2`,$\cdots$, we can use the operator `nm`. The syntax for this is:

`nm(`⟨*n:integer*⟩`)` : *algebraic*

It returns the value of the variable `#n`. For example, to get the value of `#1` in the above, one could say:

```
nm(1);
```

yields the result

```
 2             2
a  + 2*a*b + b
```

It is also possible to evaluate an expression including special variables `#1,#2,···` by using the `eval2` operator. The syntax for this is:

`eval2(`⟨*exprn:rational*⟩`)` : *algebraic*

Example:

```
coeff2(f2*g2,x,y);

    4          3  4        3          2  4
#5*x *y + #6*x *y  + #7*x *y + #8*x *y

nm(8);

#2*#4

eval2(ws);

   2                 2      2               2
a*c  + 2*a*c*d + a*d  + 3*c  + 6*c*d + 3*d
```

The user may remove all values of special variables `#1,#2,···` by the operator `reset`, in the form

```
reset( );
```

20.14 CONLAW: Find Conservation Laws for Differential Equations

This package computes first integrals of ordinary differential equations (ODEs) or conservation laws (CLs) of partial differential equations (PDEs) or systems of both. Four different approaches to compute CLs have been implemented in four different procedures `CONLAW1`...`CONLAW4`. All use the package CRACK to solve the overdetermined system of conditions they generate.

Author: Thomas Wolf

20.14.1 Purpose

The procedures `CONLAW1`, `CONLAW2`, `CONLAW3`, `CONLAW4` try to find conservation laws for a given single/system of differential equation(s) (ODEs or PDEs)

$$u_J^\alpha = w^\alpha(x, u^\beta, \ldots, u_K^\beta, \ldots) \tag{20.72}$$

`CONLAW1` tries to find the conserved current P^i by solving

$$\text{Div}\, P = 0 \quad \text{modulo} \quad (20.72) \tag{20.73}$$

directly. `CONLAW3` tries to find P^i and the characteristic functions (integrating factors) Q^ν by solving

$$\text{Div}\, P = \sum_\nu Q^\nu \cdot (u_J^\nu - w^\nu) \tag{20.74}$$

identically in all u-derivatives. Applying the Euler operator (variational derivative) for each u^ν on (20.74) gives a zero left hand side and therefore conditions involving only Q^ν. `CONLAW4` tries to solve these conditions identically in all u-derivatives and to compute P^i afterwards. `CONLAW2` does substitutions based on (20.72) before solving these conditions on Q^ν and therefore computes adjoined symmetries. These are completed, if possible, to conservation laws by computing P^i from the Q^ν.

All four procedures have the same syntax. They have two parameters, both lists. The first parameter specifies the equations (20.72), the second specifies the computation to be done. One can either specify an ansatz for P^i, Q^ν or investigate a general situation, only specifying the order of the characteristic functions or the conserved current.

20.14.2 Syntax

The procedures `CONLAW`i, $i = 1, 2, 3, 4$, are called through

```
CONLAWi(problem, runmode);
```

where both parameters are lists.

The first parameter *problem* specifies the DEs to be investigated. It has the form `{equations, ulist, xlist}` where ...

equations is a list of equations, each of the form `df(ui,...)=...` where the left-hand side `df(ui,...)` is selected such that

- the right-hand side of an equation must not include the derivative on the left-hand side nor a derivative of it;
- the left-hand side of any equation should not occur in any other equation nor any derivative of the left-hand side.

If `CONLAW3` or `CONLAW4` are run where no substitutions are made then the left-hand sides of equations can be `df(ui,...)**n=...` where `n` is a number. No distinction is made between equations and constraints.

ulist is a list of function names, which can be chosen freely; the number of functions and equations need not be equal.

xlist is a list of variable names, which can be chosen freely.

The second parameter *runmode* specifies the calculation to be done. It has the form `{minord, maxord, expl, flist, inequ}` where ...

minord and *maxord* are respectively the minimum and maximum of the highest order of derivatives of u

- in `p_t` for `CONLAW1`, where `t` is the first variable in *xlist*;
- in `q_j` for `CONLAW2`, `CONLAW3` and `CONLAW4`.

expl is `t` or `nil` to indicate whether or not the characteristic functions `q_i` or conserved current may depend explicitly on the variables of *xlist*.

flist is a list of unknown functions in any ansatz for `p_i`, `q_j`, also all parameters and parametric functions in the equation that are to be calculated such that conservation laws exist; if there are no such unknown functions then *flist* is the empty list `{}`.

inequ is a list of expressions none of which may be identically zero for the conservation law to be found; if there is no such expression then *inequ* is an empty list `{}`.

The procedures CONLAW*i* return a list of conservation laws {$C_1, C_2, \ldots$}; if no non-trivial conservation law is found they return the empty list {}. Each C_i representing a conservation law has the form {{$P^1, P^2, \ldots$},{$Q^1, Q^2, \ldots$}}.

An ansatz for a conservation law can be formulated by specifying one or more of the components P^i for CONLAW1, one or more of the functions Q^μ for CONLAW2 and CONLAW4, or one or more of P^i, Q^μ for CONLAW3. The P^i are input as p_i where i stands for a variable name, and the Q^μ are input as q_i where i stands for an index, which is the number of the equation in the input list *equations* with which q_i is multiplied.

There is a restriction in the structure of all the expressions for p_i and q_j that are specified: they must be homogeneous linear in the unknown functions or constants which appear in these expressions. The reason for this restriction is not for CRACK to be able to solve the resulting overdetermined system but for CONLAW*i* to be able afterwards to extract the individual conservation laws from the general solution of the determining conditions.

All such unknown functions and constants must be listed in *flist* (see above). The dependencies of such functions must be defined before calling CONLAW*i*. This is done with the command DEPEND, e.g.

```
DEPEND f,t,x,u$
```

to specify f as a function of t, x, u. If one wants to have f as a function of derivatives of $u(t, x)$, say f depending on u_{txx}, then one *cannot* write

```
DEPEND f, df(u,t,x,2)$
```

but instead must write

```
DEPEND f, u!`1!`2!`2$
```

if *xlist* has been specified as {t,x}, because t is the first variable and x is the second variable in *xlist* and u is differentiated once wrt. t and twice wrt. x; we therefore get u!`1!`2!`2. The character ! is the escape character to allow special characters like ` to occur in an identifier name.

It is possible to add extra conditions like PDEs for P^i, Q^μ as a list cl_condi of expressions that shall vanish.

Remarks

- The input to each of CONLAW1, CONLAW2, CONLAW3 and CONLAW4 is the same apart from:

  - an ansatz for Q^ν is ignored in `CONLAW1`;
  - an ansatz for P^i is ignored in `CONLAW2` and `CONLAW4`;
  - the meaning of `mindensord`, `maxdensord` is different in `CONLAW1` on the one hand and `CONLAW2`, `CONLAW3`, and `CONLAW4` on the other (see above).

- It matters how the differential equations are input, i.e. which derivatives are eliminated. For example, the Korteweg-de Vries equation if input in the form $u_{xxx} = -uu_x - u_t$ instead of $u_t = -uu_x - u_{xxx}$ in `CONLAW1` and choosing `maxdensord`=1 then P^i will be of at most first order, Div P of second order and u_{xxx} will not be substituted and no non-trival conservation laws can be found. This does not mean that one will not find low order conservation laws at all with the substitution u_{xxx}; one only has to go to `maxdensord`=2 with longer computations as a consequence compared to the input $u_t = -uu_x - u_{xxx}$ where `maxdensord`=0 is enough to find non-trivial conservation laws.

- The drawback of using $u_t = \ldots$ compared with $u_{xxx} = \ldots$ is that when seeking all conservation laws up to some order then one has to investigate a higher-order ansatz, because with each substitution $u_t = -u_{xxx} + \ldots$ the order increases by 2. For example, if all conservation laws of order up to two in Q^ν are to be determined then in order to include a u_{tt}-dependence the dependence of Q^ν on u_x up to u_{6x} has to be considered.

- Although for any equivalence class of conserved currents with P^i differing only by a curl there exist characteristic functions Q^μ, this need not be true for any particular P^i. Therefore one cannot specify a known density P^i for `CONLAW3` and hope to calculate the remaining P^j and the corresponding Q^μ with `CONLAW3`. What one can do is to use `CONLAW1` to calculate the remaining components P^j, and from this a trivial conserved density R and characteristic functions Q^ν are computed such that

$$\text{Div}\,(P - R) = \sum_\nu Q^\nu \cdot (u^\nu_J - w^\nu).$$

- The Q^μ are uniquely determined only modulo $\Delta = 0$. If one makes an ansatz for Q^μ then this freedom should be removed by having the Q^μ independent of the left-hand sides of the equations and their derivatives. If the Q^μ were allowed to depend on anything, then (20.74) could simply be solved for one Q^ν in terms of arbitrary P^j and other arbitrary Q^ρ, giving Q^ν that are singular for solutions of the differential equations as the expression of the differential equation would appear in the denominator of Q^ν.

- Any ansatz for P^i, Q^ν should as well be independent of the left-hand sides of equations (20.72) and their derivatives.

- If in equation (20.74) the right-hand side is of order m then from the conserved current P^i a trivial conserved current can be subtracted such that the remaining conserved current is of order at most m. If the right hand side is linear in the highest derivatives of order m then subtraction of a trivial conserved current can even achieve a conserved current of order $m-1$. The relevance of this result is that if the right-hand side is known to be linear in the highest derivatives then for P^i an ansatz of order only $m-1$ is necessary. To take advantage of this relation if the right-hand side is known to be linear in the highest derivatives, a flag `quasilin_rhs` can be set to `t` (see below).

20.14.3 Flags and Parameters

```
lisp(print_ := nil/0/1/ ...)$
```

`print_:=nil` suppresses all CRACK output; for `print_:=`n (n an integer) CRACK prints only equations with at most n factors in their terms.

```
crackhelp()$
```

shows other flags controlling the solution of the overdetermined PDE-system.

```
off batch_mode$
```

solves the system of conditions with CRACK interactively.

```
lisp(quasilin_rhs := t)$
```

reduces in the ansatz for P^i the order to $m-1$ if the order of the right-hand side is m. This can be used to speed up computations if the right-hand side is known to be linear in the highest derivatives (see the note above).

20.14.4 Requirements

```
load_package crack, conlaw0, conlaw1$
```

where `conlaw1` can be replaced by `conlaw2`, `conlaw3` or `conlaw4` as appropriate.

20.14.5 Examples

Below a CRACK procedure `nodepnd` is used to clean up after each run and delete all dependencies of each function in the list of functions in the argument of `nodepnd`. More details concerning these examples are given when running the file `conlaw.tst` (in the REDUCE `packages/crack` directory).

```
lisp(print_:=nil);  % to suppress output from CRACK
```

- A single PDE:

```
depend u,x,t$
conlaw1({{df(u,t)=-u*df(u,x)-df(u,x,3)},
         {u}, {t,x}},
        {0, 1, t, {}, {}})$
nodepnd {u}$
```

- A system of equations:

```
depend u,x,t$
depend v,x,t$
conlaw1({{df(u,t)=df(u,x,3)+6*u*df(u,x)+
             2*v*df(v,x),
          df(v,t)=2*df(u,x)*v+2*u*df(v,x)},
         {u,v}, {t,x}},
        {0, 1, t, {}, {}})$
nodepnd {u,v}$
```

- A system of equations with ansatz:

```
depend u,x,t$
depend v,x,t$
depend r,t,x,u,v,u!`2,v!`2$
q_1:=r*df(u,x,2)$
conlaw2({{df(u,t)=df(v,x),
          df(v,t)=df(u,x) }, {u,v}, {t,x}},
        {2, 2, t, {r}, {r}})$
nodepnd {u,v,r}$
```

- For the determination of parameters, such that conservation laws exist:

```
depend u,x,t;
conlaw1({{df(u,t)=-df(u,x,5)-a*u**2*df(u,x)-
             b*df(u,x)*df(u,x,2)-c*u*df(u,x,3)},
```

```
          {u}, {t,x}},
         {0, 1, t, {a,b,c}, {}});
nodepnd {u};
```

- For first integrals of an ODE-system including the determination of parameter values s, b, r such that conservation laws exist:

```
depend {x,y,z},t;
depend a1,x,t;
depend a2,y,t;
depend a3,z,t;

p_t:=a1+a2+a3;
conlaw2({{df(x,t) = - s*x + s*y,
          df(y,t) = x*z + r*x - y,
          df(z,t) = x*y - b*z},
         {x,y,z},{t}
        },
        {0,0,t,{a1,a2,a3,s,r,b},{}});
nodepnd {x,y,z,a1,a2,a3};
```

20.15 CRACK: Solving Overdetermined Systems of ODEs or PDEs

CRACK is a package for solving overdetermined systems of differential equations. Examples of programs which make use of CRACK (finding symmetries of ODEs/PDEs, first integrals, an equivalent Lagrangian or a "differential factorization" of ODEs) are included. The packages APPLYSYM and CONLAW use CRACK.

Authors: Andreas Brand, Thomas Wolf

20.15.1 Introduction

Purpose

The package CRACK attempts the solution of an overdetermined system of algebraic, ordinary or partial differential equations (ODEs/PDEs) with at most polynomial nonlinearities.

Under 'normal circumstances' differential equations (DEs) which describe physical processes are not overdetermined, i.e. the number of DEs matches the number of unknown functions which are involved. Although CRACK may be successful in such cases (e.g. for characteristic ODE-systems of first order PDEs) this is not the typical application. It is rather the qualitative investigations of such differential equations, i.e. the investigation of their infinitesimal symmetries (with LIEPDE and APPLYSYM) and conservation laws (with CONLAW) which result in over-determined systems which are the main application area of CRACK.

Interactivity

The package was originally developed to run automatically and effort was made for the program to decide which computational steps are to be done next with a choice among integrations, separations, substitutions and investigation of integrability conditions. It is known from hand computations that the right sequence of operations with exactly the right equations at the right time is often crucial to avoid an explosion of the length of expressions. This statement keeps its truth for the computerized solution of systems of equations as they become more complex. As a consequence more and more interactive access has been provided to inspect data, to specify how to proceed with the computation and how to control it. This allows human intervention in critical stages of the computations (see the switch `off batch_mode` below).

General Structure

A problem consists of a system of equations and a set of inequalities. With each equation are associated a short name and numerous data, like size, which functions, derivatives and variables occur but also which investigations have already been done with this equation and which not in order to avoid unnecessary duplication of work. These data are constantly updated if the equation is modified in any way.

A set of about 30 modules is available to integrate, substitute, decouple, ... equations. A complete list can be inspected in interactive mode with the command `p2`, which lists each operation with the number by which it is called. All modules can be called interactively or automatically. Automatic computation is organized by a priority list of modules (each represented by a number) where modules are invoked in the order they appear in the priority list, each module trying to find equations in the system it can be applied to. The priority list can be inspected with the command `p1`. If a module is not successful then the next module in the list is tried; if any one is successful then execution starts again at the beginning of the priority list. See the Reference subsection below for further details.

Because each module has access to all the data, it is enough to call a module by its number. For example, inputting the number 2 in interactive mode will start the direct separation module (see below) to look for a directly separable equation and split it.

20.15.2 Syntax

The call

CRACK is called by

$$\begin{aligned}&\texttt{crack}(\{equ_1, equ_2, \ldots, equ_m\},\\ &\qquad \{ineq_1, ineq_2, \ldots, ineq_n\},\\ &\qquad \{fun_1, fun_2, \ldots, fun_p\},\\ &\qquad \{var_1, var_2, \ldots, var_q\});\end{aligned}$$

where m, n, p, q are arbitrary.

- The equ_i are identically vanishing partial differential expressions, i.e. they represent equations $0 = equ_i$, which are to be solved for the functions fun_j as far as possible, thereby drawing only necessary conclusions and not restricting the general solution.
- The $ineq_i$ are algebraic or differential expressions which must not vanish identically for any solution to be determined, i.e. only such solutions are

computed for which none of the expressions $ineq_i$ vanishes identically in all independent variables.

- The dependence of the (scalar) functions fun_j on independent variables must be defined beforehand with `depend` rather than declaring these functions as operators. Their arguments may themselves only be identifiers representing variables, not expressions. Also other unknown functions not in fun_j must not be represented as operators but only declared using `depend`.
- The functions fun_j and their derivatives may only occur polynomially.
- The var_k are further independent variables, which are not already arguments of any of the fun_j. If there are none then the fourth argument is the empty list `{}`, although it does no harm to include arguments of functions fun_j.
- The dependence of the equ_i on the independent variables and on constants and functions other than fun_j is arbitrary.
- CRACK can be run in automatic batch mode (the default) or interactively with the switch setting `off batch_mode`.

The result

The result is a list of solutions

$$\{sol_1, \ldots\},$$

where each solution is a list of 4 lists

$$\begin{aligned}&\{\{con_1, con_2, \ldots, con_q\},\\ &\{fun_a = ex_a, fun_b = ex_b, \ldots, fun_p = ex_p\},\\ &\{fun_c, fun_d, \ldots, fun_r\},\\ &\{ineq_1, ineq_2, \ldots, ineq_s\}\}.\end{aligned}$$

For example, in the case of a linear system, the input consists of at most one solution sol_1.

If CRACK finds a contradiction, e.g. $0 = 1$, then there exists no solution and it returns the empty list `{}`. If CRACK can factorize algebraically a non-linear equation then factors are set to zero individually and different sub-cases are studied by CRACK calling itself recursively. If during such a recursive call a contradiction results, then this sub-case will not have a solution but other sub-cases still may have solutions. The empty list is also returned if no solution exists which satisfies the inequalities $ineq_i \neq 0$.

The expressions con_i (if there are any), are the remaining necessary and sufficient conditions for the functions $fun_c, \ldots, fun_r$ in the third list. Those functions can be original functions from the equations to be solved (of the second argument of

the call of CRACK) or new functions or constants which arose from integrations. The dependence of new functions on variables is declared with `depend` and to visualize this dependence the algebraic mode function `fargs(`fun_i`)` can be used. If there are no con_i then all equations are solved and the functions in the third list are unconstrained. The second list contains equations $fun_i = ex_i$ where each fun_i is an original function and ex_i is the computed expression for fun_i. The elements of the fourth list are the expressions that have been assumed to be nonzero in the derivation of this solution.

Automatic versus Interactive

Under normal circumstances one will try to have problems solved automatically by CRACK. An alternative is to input `off batch_mode;` before calling CRACK and to solve problems interactively. In interactive mode it is possible to

- inspect data, like equations and their properties, unknown functions, variables, identities, and statistics;
- save, change, add or drop equations;
- add inequalities;
- inspect and change flags and parameters which govern individual modules as well as their interplay;
- pick a list of methods to be used out of about 30 different ones and specify their priorities, and in this way very easily compose an automatic solving strategy;
- or, for more interactive work, to specify how to proceed, i.e. which computational step to do and how often, like doing
  - one automatic step,
  - one specific step,
  - a number of automatic steps,
  - a specific step as often as possible or a specified number of times.

To get interactive help one enters `h` or `?`.

Flags and parameters are stored as symbolic fluid variables which means that they can be accessed by `lisp(...)`, e.g. `lisp( print_:=5 );`, before calling CRACK. `print_`, for example, is a measure of the maximal length of expressions to be printed on the screen (the number of factors in terms). A complete list of flags and parameters is given at the beginning of the file `crinit.red` (in the REDUCE `packages/crack` directory).

One more parameter shall be mentioned, which is the list of modules/procedures called `proc_list_`. In interactive mode this list can be looked at with `p` or changed with `cp`. This list defines the order in which the different modules/procedures are tried whenever CRACK has to decide what to do next. Exceptions to this rule may be specified. For example, some procedure, say P_1, requires after its execution another specific procedure, say P_2, to be executed, no matter whether P_2 is next according to `proc_list_` or not. This is managed by P_1 writing a task for procedure P_2 into a hot-list. Tasks listed in the global variable `to_do_list` are dealt with in the `to_do` step which should always come first in `proc_list_`. A way to have the convenience of running CRACK automatically and still being able to break the fixed rhythm prescribed by `proc_list_` is to have the entry `stop_batch` in `proc_list_` and have CRACK started in automatic batch mode. Then execution continues until none of the procedures which come before `stop_batch` are applicable any more so that `stop_batch` is executed next, which will stop automatic execution and go into interactive mode. This allows either to continue the computation interactively, or to change `proc_list_` with `cp` and continue in automatic mode.

The default value of `proc_list_` does not include all possible modules because not all are suitable for every kind of overdetermined system to be solved. The complete list is shown in interactive mode by `cp`. A few basic modules are described in the following subsection. The efficiency of CRACK in automatic mode is very much dependent on the content of `proc_list_` and the sequence of its elements. Optimizing `proc_list_` for a given task needs experience which cannot be formalized in a few simple rules and will therefore not be explained in more detail here. The following remarks are only guidelines.

20.15.3 Modules

The following modules are represented by numbers in the priority list. Each module can appear with modifications under different numbers. For example, integration is available under *7*, *24* and *25*. Here *7* encodes an integration of short equations $0 = \partial^n f/\partial x^n$. *7* has highest priority of the three integrations. *24* encodes the integration of an equation that leads to the substitution of a function and *25* refers to any integration and has lowest priority.

Integration and Separation

An early feature in the development of the package CRACK was the ability to integrate exact differential equations and some generalizations of them (see [Wol00]). As a consequence of integrations *7*, *24*, *25* an increasing number of functions of fewer variables is introduced which sooner or later produces equations with some independent variables occuring only explicitly and not as variables in functions.

Such equations are split by the separation module *2*. Another possibility is equations in which each independent variable occurs in at least one function in the equation but no function depends on all variables. In this case so-called indirect separations are appropriate: for linear problems *10*, *26* and for non-linear problems *48*.

Substitutions

Substitutions can have a dramatic effect on the size and complexity of systems. Therefore it is possible to have them not only done automatically but also controlled tightly, either by specifying exactly what equation should be used to substitute which unknown and where, or by picking a substitution out of a list of substitutions offered by the program *{cs}*. Substitutions to be performed automatically can be controlled with a number of filters, for example, by

- limiting the size of the equation to be used for substitution; *{length_limit}*
- limiting the size of equations in which the substitution is to be done; *{target_limit_}*
- allowing only linear equations to be used for substitutions; *{lin_subst}*
- allowing equations to increase in size only up to some factor in order for a substitution to be performed in that equation; *{cost_limit}*
- allowing a substitution for a function through an expression only if that expression involves exclusively functions of fewer variables; *{less_vars}*
- allowing substitutions only that do not lead to a case distinction coefficient = 0 or not;
- specifying whether extra effort should be spent to identify the substitution with the lowest bound on growth of the full system. *{min_growth}*

Substitution types are represented by different numbers (3–6,15–21) depending on the subset of the above filters to be used. If a substitution type is to be done automatically then from all possible substitutions passing all filters of this type that substitution is selected that leads to no sub-cases (if available) and that uses the shortest equation.

Factorization

It is very common that big algebraic systems contain equations that can be factorized. Factorizing an equation and setting the factors individually to zero simplifies the whole task because factors are simpler expressions than the whole equation and

setting them to zero may lead to substitutions and thereby further simplifications. The downside is that if problems with, say 100 unknowns, would need 40 case distinctions in order to be able to solve automatically for the remaining 60 unknowns then this would require 2^{40} cases to be investigated which is impractical. The problem is to find the right balance between delaying case distinctions in order not to generate too many cases and introducing case distinctions as early as necessary in order to simplify the system. This simplification may be necessary to solve the system but in any case it will speed up its solution (although at the price of having to solve a simplified system at least twice, depending on the number of factors).

For large systems with many factorizable equations the careful selection of the next equation to be factorized is important to gain the most from each factorization and to succeed with as few factorizations as possible. Some of the criteria which give factors and therefore equations a higher priority are

- the number of equations in which this factor occurs,
- if the factor is a single unknown function or constant, then the number of times this unknown turns up in the whole system,
- the total degree of the factor,
- the number of factors of an equation.

It also matters in which order the factors are set to zero. For example, the equation $0 = ab$ can be used to split into the 2 cases: $1.\ a = 0,\ 2.\ a \neq 0, b = 0$ or to split into the 2 cases $1.\ b = 0,\ 2.\ b \neq 0, a = 0$. If one of the 2 factors, say b, involves functions which occur only linearly then this property is to be preserved and these functions should be substituted as such substitutions preserve their linearity. But to have many such substitutions available, it is useful to know of many non-linearly occuring functions to be non-zero as they occur as coefficients of the linearly occuring functions. In the above situation it is therefore better to do the first splitting $1.\ a = 0,\ 2.\ a \neq 0, b = 0$ because $a \neq 0$ will be more useful for substitutions of linear functions than $b \neq 0$ would be.

An exception of this plausible rule occurs towards the end of all the substitutions of all the linearly occurring b_i when some b_i is an overall factor to many equations. If one would then set, say, $b_{22} = 0$ as the second case in a factorization, the first case would generate as subcases factorizations of other equations where $b_{22} = 0$ would be the second case again and so on. To avoid this one should investigate $b_{22} = 0$ as the first case in the first factorization.

The only purpose of that little thought experiment was to show that simple questions, like 'Which factored equation should be used first for case-distinctions and in which order to set factors to zero?' can already be difficult to answer in general.

CRACK currently offers two factorization steps: (8) and (47).

Elimination (Gröbner Basis) Steps

To increase safety and avoid excessive expression swell one can apart from the normal call (30) request to do Gröbner basis computation steps only if they are simplification steps replacing an equation by a shorter equation. (27)

In a different version only steps are performed in which equations are included which do not contain more than 3 unknowns. This helps to focus on steps which are more likely to solve small sub-systems with readily available simple results. (57)

Often the computationally cheapest way to obtain a consistent (involutive) system of equations is to change the ordering during the computation. This is the case when substitutions of functions are performed which are not ranked highest in a lexicographical ordering of functions. But CRACK also offers an interactive way to

- change the lexicographical ordering of variables, *{ov}*
- change the lexicographical ordering of functions, *{of}*
- give the differential order of derivatives a higher or lower priority in the total ordering than the lexicographical ordering of functions, *{og}*
- give either the total differential order of a partial derivative a higher priority than the lexicographical ordering of the derivative of that function or to take the lexicographical ordering of derivatives as the only criterion. *{of}*

Solution of an Under-Determined Differential Equation

When solving an over-determined system of linear differential equations where the general solution involves free functions, in the last computational step often a single equation for more than one function remains to be solved. Examples are the computation of symmetries and conservation laws of non-linear differential equations which are linearizable. In CRACK two procedures are available, one for under-determined linear ODEs (22) and one for linear PDEs, (23) both with non-constant coefficients.

Indirect Separation

Integrations introduce new functions of fewer variables. As equations are used to substitute functions of all variables it is only a question of time that equations are generated in which no function depends on all variables. If at least one variable occurs only explicitly then the equation can be split, which we call direct separation. But sometimes all variables appear as variables of unknown functions, e.g.

$0 = f(x) + g(y)$ although usually much more complicated with 10 or 20 independent variables and many functions that depend on different combinations of these variables. Because no variable occurs only explicitly, direct separations mentioned above are not possible. Two different algorithms, one for linear indirectly separable equations (10), (26) and one for non-linear directly separable equations (48) provide systematic ways of dealing with such equations.

Indirectly separable equations always result when an equation is integrated with respect to different variables, like $0 = f_{xy}$ to $f = g(x) + h(y)$ and a function, here $f(x, y)$, is substituted.

Function and Variable Transformations

In the interactive mode one can specify a transformation of the whole problem with `pt` in which old functions and variables are expressed as a mix of new functions and variables.

Solution of First Order Linear PDE

If a system contains a single linear first-order PDE for just one function then in an automatic step characteristic ODE-systems are generated, integrated if possible, a variable transformation for the whole system of equations is performed to have in the first-order PDE only one single derivative and to make this PDE integrable for the integration modules. (39)

Length Reduction of Equations

An algorithm designed originally to length-reduce differential equations proved to be essential in length-reducing systems of bi-linear algebraic equations or homogeneous equations which resulted from bi-linear equations during the solution process.

The aim of the method (11) is to find out whether one equation $0 = E_1$ can length-reduce another one $0 = E_2$ by replacing E_2 through an appropriate linear combination $\alpha E_1 - \beta E_2, \ \ \beta \neq 0$. To find α, β one can divide each term of E_2 through each term of E_1 and count how often each quotient occurs. If a quotient α/β occurs m times then $\alpha E_1 - \beta E_2$ will have $\leq n_1 + n_2 - 2m$ terms because $2m$ terms will cancel each other. A length reduction is found if $n_1 + n_2 - 2m \leq \max(n_1, n_2)$. The method becomes efficient after a few algorithmic refinements discussed in [Wol02b]. Length-reduced equations

- are more likely to length-reduce other equations,
- are much more likely to be factorizable,

- are more suited for substitutions as the substitution induces less growth of the whole systems and introduces fewer new occurrences of functions in equations,
- are more likely to be integrable by being exact or being an ODE if the system consists of differential equations,
- involve on average fewer unknowns and make the whole system more sparse. This sparseness can be used to plan better a sequence of eliminations.

This concludes the listing of modules. Other aspects of CRACK follow.

20.15.4 Features

Flexible Process Control

Different types of over-determined systems are more or less suited for an automatic solution. With the currrent version it is relatively safe to try solving large bilinear algebraic problems automatically. Another well-suited area concerns over-determined systems of linear PDEs. In contrast, non-linear systems of PDEs most likely require a tighter interactive control. Different modes of operation are possible. One can

- perform one *{a}* or more computational steps *{g}* automatically, where each step tries modules in the order defined by the current priority list *{p1}* until one module succeeds in its purpose;
- perform one module a specific number of times or as long as it is successful; *{l}*
- set a time limit for how long the program should run automatically; *{time_limit, limit_time}*
- interrupt an on-going automatic computation and continue the computation interactively by copying the file `_stop_crack` into the directory where the ongoing computation was started and re-naming it `_stop_` (and by deleting `_stop_` to resume automatic computation);
- arrange the priority list of module changes at a certain point in the computation when the system of equations has changed its character;
- induce a case distinction whether a user-given expression is zero or not; *{44}*
- have a module that changes the priority list *proc_list_* dynamically, depending essentially on the size and difficulty of the system but also on the success rate of previous steps. *{61, 62, 63}*

Apart from flexible control over what kind of steps to do, the steps themselves can be controlled more or less too, e.g. whether equations are selected by the module or the user.

So-called *to-do* steps have highest priority in the priority list. The list of to-do steps is usually empty but can be filled by any successful step if it requires another specific step to follow instantly. For example, if a very simple equation $0 = f_x$ is integrated then the substitution of f should follow straight away, even if substitutions would have a low priority according to the current priority list.

Total Data Control

To make wise decisions of how to continue the computation in an interactive session one needs tools to inspect large systems of equations. Helpful commands in CRACK print

- equations, inequalities, functions and variables *{e, pi, f}*;
- the occurence of all derivatives of selected functions in any equation; *{v}*
- a statistics summary of the equations of the system; *{s}*
- a matrix display of occurences of unknowns in all equations; *{pd}*
- the value of any LISP variable; *{pv}*
- the value of algebraic expressions that can be specified using equation names (e.g. `coeffn(e_5,df(f,x,y),2)`); *{pe}*
- not under-determined subsystems. *{ss}*

Inspecting a computation which already goes on for hours or a day and has performed many thousand steps is time consuming. The task is made easier with the possibility to plot graphically as a function of time: the type of steps performed, the number of unknowns, the number of remaining equations, the number of terms in these equations and the memory usage. *{ps}*

When non-linear systems are considered and many case distinctions and sub-(sub-...)case distinctions are made in a long computation, one easily loses track. With one command one can list all cases that have been considered so far with their assumption, the number of steps made until they are solved or until the next subcase distinction was made and the number of solutions contained in each completed case. *{ls}*

Safety

When working on large problems, a stage may come where computational steps are necessary, like substitution, which are risky in the sense that they may simplify the problem or complicate it by increasing its size. To avoid this risk a few safety features have been implemented.

- At any time during the computation one can save a backup of the complete current situation in a file and also load a backup. The command `sb "file_name"` saves all global variables and data into an ASCII file and the command `rb "file_name"` reads these data from a file. The format is independent of the computer used and independent of the underlying Lisp version. Apart from reading in a backup file during an interactive computation with `rb` one can also start a computation with a backup file. After loading CRACK one makes in REDUCE the call `crackshell()$` followed by the file name of the backup.

- All key strokes are automatically recorded in a list which is available after each interactive step with `ph`, or when the computation has finished through `lisp reverse history_;`. This list can be fed into CRACK at the beginning of a new computation so that the same operations are performed automatically that were performed interactively before. The purpose is to be able to do an interactive exploration first and to repeat it afterwards automatically without having to note with pen or pencil all steps that had been done.

  By assigning this list to the Lisp variable `old_history` before calling CRACK with `off batch_mode` the same steps as in the previous run are performed first as CRACK first reads input from `old_history` and then reads from the keyboard.

- During an automatic computation the program might start a computational step which turns out to take far too long. It would be better to stop this computation and try something else instead. But in computer algebra with lots of global variables involved it is not straightforward to stop a computation in the middle. If one used time as a criterion then it could happen that time is up during a garbage collection and to stop would be deadly for the session. CRACK allows to set a limit of garbage collections for any computations that have the potential to last forever, like algebraic factorizations of large expressions. With such an arrangement an automatic computation cannot get stuck due to lengthy factorizations, searches for length reductions or elimination steps. *{max_gc_elimin, max_gc_fac, max_gc_red_len, max_gc_short, max_gc_ss}*

- Due to an initiative by Winfried Neun the parallel version of REDUCE has been re-activated (and was running on the Beowulf cluster at Brock Univer-

sity [MN02]). This allows conveniently (with *{pp}*) to duplicate the current status of a CRACK computation to another computer, to try out there different operations (e.g. risky ones) until a viable way to continue the computation is found without endangering the original session.

Managing Solutions

Non-linear problems can have many solutions. The number of solutions found by CRACK can even be higher because to make progress CRACK may have factorized an equation and considered the two cases $a = 0$ and $a \neq 0$ whereas solutions in both cases could be merged to only one solution without any restriction for a. This merging of solutions can be accomplished with a separate program `merge_sol()` after the computation.

Another form of post-processing is the production of a web page for each solution, like lie.math.brocku.ca/twolf/bl/v/v1l05o35-s1.html.

If in the solution of over-determined differential equations the program performs integrations of equations before the differential Gröbner basis was computed then in the final solution there may be redundant constants or functions of integration. Redundant constants or functions in a solution are not an error but they make solutions appear unnecessarily complicated. In a postprocessing step these functions and constants can be eliminated. *{adjust_fnc, drop_const(), dropredundant()}*

Parallelization

The availability of a parallel version of CRACK was mentioned above allowing to try out different ways to continue an ongoing computation. A different possibility to make use of a cluster of computers with CRACK is to export automatically the investigation of sub-cases and sub-sub-cases to different computers to be solved in parallel.

It was explained above how factorizations may be necessary to make any progress but also their potential of exploding the time requirements. By running the computation on a cluster and being able to solve many more cases one can give factorizations a higher priority and capitalize on the benefit of factorizations, i.e. the simplification of the problem.

Relationship to Gröbner Basis Algorithms

For systems of equations in which the unknown constants or functions turn up only polynomially a well-known method is able to check the consistency of the system. For algebraic systems this is the Gröbner Basis method and for systems of differential equations this is the differential Gröbner Basis method. To guarantee

the method will terminate a total ordering of unknowns and their derivatives has to be introduced. This ordering determines which highest powers of unknowns are to be eliminated next or which highest-order derivatives have to be eliminated next using integrability conditions. Often such eliminations lead to exponential growth of the generated equations. In the package CRACK such computations are executed with only a low priority. Operations have a higher priority which reduce the length of equations, irrespective of any orderings. Violating any ordering a finite number of times still guarantees a finite algorithm. The potential gain is large as described next.

Exploiting Bi-Linearity

In bi-linear algebraic problems we have 2 sets of variables, $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$, such that all equations have the form $0 = \sum_{k=1}^{l} \gamma_k a_{i_k} b_{j_k},\ \gamma_k \in G$. Although the problem is linear in the a_i and linear in the b_j it still is a non-linear problem. A guideline which helps keep the structure of the system during computation relatively simple is to preserve the linearity of either the a_i or the b_j as long as possible. In classification problems of integrable systems the ansatz for the symmetry/first integral usually involves more terms and therefore more constants (called b_j in applications of CRACK) than the ansatz for the integrable system (with constants a_i). A good strategy therefore is to keep the system linear in the b_j during the computation, i.e. to

- substitute only a b_j in terms of a_i, b_k, or an a_i in terms of an a_k but not an a_i in terms of any b_k;
- do elimination steps for any b_j or for an a_i if the involved equations do not contain any b_k.

The proposed measures are effective not only for algebraic problems but for ODEs/PDEs too (i.e. to preserve linearity of a subset of functions as long as possible). *{flin_}*

Occurrence of sin, cos or Other Special Functions

If the equations to be solved involve special functions, like $\sin$ and $\cos$, then one is inclined to add let-rules for simplifying expressions. Before doing this the simplification rules at the end of the file `crinit.red` (in the REDUCE `packages/crack` directory) should be inspected such that new rules do not lead to cycles with existing rules. One possibility is to replace existing rules, for example to substitute the existing rule

```
trig1_ := {sin(~x)**2 => 1-cos(x)**2}$
```

by the new rule

```
trig1_ := {cos(~x)**2 => 1-sin(x)**2}$
```

These rules are switched off when integrations are performed in order not to interfere with the REDUCE integrator.

Apart from an initial customization of let-rules to be used during the whole run one can also specify and clear let-rules during a computation using the interactive commands `lr,cr`.

Exchanging Time for Memory

The optimal order of applying different methods to the equations of a system is not fixed. It does depend, for example, on the distributions of unknown functions in the equations and on what the individual method would produce in the next step. For example, it is possible that the decoupling module which applies integrability conditions through cross differentiations of equations is going well up to a stage when it suddenly produces huge equations. They not only occupy much memory, they also are slow to handle. Right *before* this explosion started other methods should have been tried (shortening of equations, any integrations, solution of underdetermined ODEs if there are any, ...). These alternative methods are normally comparatively slow or unfavourable as they introduce new functions but under the current circumstances they may be perfect to avoid any growth and to complete the calculation. How could one have known beforehand that some method will lead to an explosion? One does not know. But one can regularly make a backup with the interactive `sb` command and restart at this situation if necessary.

Customization

The addition of new modules to perform new specialized computations is easy. Only the input and output of any new module are fixed. The input consists of the system of equations, the list of inequalities and the list of unknowns to be computed. The output includes the new system of equations and new intermediate results. The module name has to be added to a list of all modules and a one line description has to be added to a list of descriptions. This makes it easy for users to add special techniques for the solution of systems with extra structure. A dummy template module *{58}* is already added and has only to be filled with content.

Debugging

A feature, useful mainly for debugging is that in the middle of an ongoing interactive computation the program can be changed by loading a different version

of CRACK procedures. Thus one could advance quickly close to the point in the execution where an error occurs, load a version of the faulty procedure that gives extensive output and watch how the fault happens before fixing it.

The possibility to interrupt REDUCE itself temporarily and to inspect the underlying LISP environment *{br}* or to execute LISP commands and to continue with the CRACK session afterwards *{pc}* led to a few improvements and fixes in REDUCE itself.

20.15.5 Technical issues

System Requirements

PSL REDUCE is faster whereas CSL REDUCE seems to be more stable under Microsoft Windows. Also it provides portable compiled code.

Memory requirements depend crucially on the application. The `crack.rlg` file can be produced by running `crack.tst` in a 4MB session running REDUCE under LINUX (the files are in the REDUCE `packages/crack` directory). On the other hand it is not difficult to formulate problems that consume any amount of memory.

Availability

The package CRACK together with LIEPDE, CONLAW and APPLYSYM are included with REDUCE. Publications related to CRACK itself and to applications based on it can be found under lie.math.brocku.ca/twolf/home/publications.html.

The files

The following files are provided with CRACK (in the REDUCE `packages/crack` directory):

`crack.red`	contains read-in statements for a number of files `cr*.red`
`crack.tst`	contains test examples
`crack.rlg`	contains the output of `crack.tst`
`crack.tex`	the original version of this manual.

20.15.6 Reference

Elements of proc_list_

The interactive command `p1` shows *proc_list_*. This list defines the order in which procedures are tried if a step is to be performed automatically. Command `p2` shows the complete list as it is shown below. To select any one procedure of the complete list interactively, one simply inputs the number shown in (). The numbering of procedures grew historically. Each number has only little or no connection with the priority of the procedure it is labelling.

`to_do (1):` hot list of steps to be taken next. Should always come first.

`subst_level_? (3-6,15-21):` substitutions of functions by expressions. Substitutions differ by their maximal allowed size and other properties. To find out which function has which properties one currently has to inspect the procedure definitions of `subst_level_?` in the file `crmain.red`.

`separation (2):` what is described as direct separation in the next subsection.

`gen_separation (26):` what is described as indirect separation in the next subsection. Only to be used for linear problems.

`quick_gen_separation (10):` generalized separation of equations with an upper size limit.

`quick_integration (7):` integration of very specific short equations.

`full_integration (24):` integration of equations which lead to a substitution.

`integration (25):` any integration.

`factorize_to_substitute (8):` splitting the computation into the investigation of different sub-cases resulting from the algebraic factorization of an equation. Only useful for non-linear problems, and applied only if each one of the factors, when individually set to zero, would enable the substitution of a function.

`factorize_any (47):` splitting into sub-cases based on a factorization even if not all factors set to zero lead to substitutions.

`change_proc_list (37):` reserved name of a procedure to be written by the user that does nothing else but changing `proc_list_` in a fixed manner. This is to be used if the computation splits naturally into different parts and if it is clear from the beginning what the computational methods (`proc_list_`) have to be.

stop_batch (38): If the first steps to simplify or partially solve a system of equations are known and should be done automatically and afterwards CRACK should switch into interactive mode then `stop_batch` is added to `proc_list` with a priority just below the steps to be done automatically.

drop_lin_dep (12): module to support solving big linear systems (still experimental).

find_1_term_eqn (13): module to support solving big linear systems (still experimental).

trian_lin_alg (14): module to support solving big linear systems (still experimental).

undetlinode (22): parametric solution of single under-determined linear ODE (with non-constant coefficients). Only applicable for linear problems. (Too many redundant functions resulting from integrations may prevent further integrations. If they are involved in single ODEs then the parametric solution of such ODEs treated as single under-determined equations is useful. Danger: new generated equations become very big if the minimal order of any function in the ODE is high.)

undetlinpde (23): parametric solution of single under-determined linear PDE (with non-constant coefficients). Only applicable for linear problems (still experimental).

alg_length_reduction (11): length reduction by algebraic combination. Only for linear problems. One has to be careful when combining it with decoupling as infinite loops may occur when shortening and lowering order reverse each other.

diff_length_reduction (27): length reduction by differential reduction.

decoupling (30): steps towards the computation of a differential Gröbner Basis.

add_differentiated_pdes (31): only useful for non-linear differential equations with leading derivative occurring non-linearly.

add_diff_ise (32): for the treatment of non-linear indirectly separable equations.

multintfac (33): to find integrating factors for a system of equations. Should have very low priority if used at all.

alg_solve_single (34): to be used for equations quadratic in the leading derivative.

alg_solve_system (35): to be used if a (sub-)system of equations shall be solved for a set of functions or their derivatives algebraically.

subst_derivative (9): substitution of a derivative of a function everywhere by a new function if such a derivative exists.

undo_subst_derivative (36): undo the above substitution.

del_redundant_fc (40): drop redundant functions and constants. For that an over-determined PDE system is formulated and solved to set redundant constants and functions of integration to zero. This may take longer if many functions occur.

find_trafo (39): finding a first-order linear PDE. By solving it the program finds a variable transformation that transforms the PDE to a single derivative and makes the PDE integrable for the integration modules. Because a variable transformation was performed the solution contains only new functions of integration which depend on single (new) variables and not on expressions of them, like sums of them. Therefore the result of the integration can be used for substitutions in other equations. If the transformation had not been made then the solution of the PDE would involve arbitrary functions of expressions and could not be used for the other equations using the current modules of CRACK. A general transformation can be done interactively with the command `cp`.

sub_problem (41): solve a subset of equations first (still experimental).

del_redundant_de (28): delete redundant equations.

idty_integration (29): integrate an identity.

gen_separation2 (48): indirect separation of a PDE. This is a second version for non-linear PDEs.

find_and_use_sub_systems12 (49): find sub-systems of equations with at least as many equations as functions. In this case find systems with at most 2 functions, none of them a function of the set `flin_`. (These are functions which occur initially only linearly in a non-linear problem; `flin_` is assigned initially by the user.)

find_and_use_sub_systems13 (50): like above only with at most 3 functions, none from `flin_`.

find_and_use_sub_systems14 (51): like above only with at most 4 functions, none from `flin_`.

find_and_use_sub_systems15 (52): like above only with at most 5 functions, none from `flin_`.

find_and_use_sub_systems22 (53): like above only with at most 2 functions. Only `flin_` are considered, all others ignored.

find_and_use_sub_systems23 (54): like above only with at most 3 functions. Only `flin_` are considered, all others ignored.

find_and_use_sub_systems24 (55): like above only with at most 4 functions. Only `flin_` are considered, all others ignored.

find_and_use_sub_systems25 (56): like above only with at most 5 functions. Only `flin_` are considered, all others ignored.

high_prio_decoupling (57): do a decoupling step with two equations that in total involve at most 3 different functions of all independent variables in these equations.

user_defined (58): This is an empty procedure which can be filled by the user with a very specific computational step that is needed in a special user application. Template:

```
symbolic procedure user_defined(arglist)$
   % arglist is a Lisp list {pdes,forg,vl_} where
   %   pdes is the list of names of all equations
   %   forg is the list of original functions +
   %     their values as far as known
   %   vl_ is the list of independent variables
begin
   ...
   return if successful then list(pdes,forg)
      % new pdes + functions and their value
      else nil
end$
```

alg_groebner (59): call of the REDUCE procedure `groebnerf` trying to solve the whole system under the assumption that it is a completely algebraic polynomial system. All resulting solutions are considered individually further.

solution_check (60): this procedure tests whether a solution that is defined in an external procedure `sol_define()` is still contained in the general solution of the system currently under investigation. This procedure is useful to find the place in a long computation where a special solution is either lost or added to the general solution of the system to be solved. Template:

```
algebraic procedure sol_define$
```

```
<< % This procedure contains the statements
   % that specify a solution
   % Example: Test whether s=h_-y**2/t**2, u=y/t
   % is a solution, where h_=h_(t)
   depend h_,t$
   % Return a list of expressions that vanish for
   % the solution to be tested, in this example:
   {s-(h_-y**2/t**2), u-y/t}
>>$
```

Online Help

The following commands and their one line descriptions appear in the same order as in the online help.

Help for Help

`hd`	Help to inspect data
`hp`	Help to proceed
`hf`	Help to change flags and parameters
`hc`	Help to change data of equations
`hi`	Help to work with identities
`hb`	Help to trace and debug

Help to Inspect Data

`e`	Print equations
`eo`	Print overview of functions in equations
`pi`	Print inequalities
`f`	Print functions and variables
`v`	Print all derivatives of all functions
`s`	Print statistics
`fc`	Print no of free cells
`pe`	Print an algebraic expression
`ph`	Print history of interactive input
`pv`	Print value of any Lisp variable
`pf`	Print no of occurences of each function
`pr`	Print active substitution rules
`pd`	Plot the occurence of functions in equations
`ps`	Plot a statistical history
`lc`	List all case distinctions

`ws` Write statistical history in file
`sn` Show name of session
`ss` Find and print sub-systems
`w` Write equations into a file

Help to Proceed

`a` Do one step automatically
`g` Go on for a number of steps automatically
`t` Toggle between automatic and user selection of equations (`expert_mode=nil/t`)
`p1` Print a list of all modules in batch mode
`p2` Print a complete list of all modules
`#` Execute the module with the number '#' once
`l` Execute a specific module repeatedly
`sb` Save complete backup to file
`rb` Read backup from file
`ep` Enable parallelism
`dp` Disable parallelism
`pp` Start an identical parallel process
`kp` Kill a parallel process
`x` Exit interactive mode for good
`q` Quit current level or crack if in level 0

Help to Change Flags and Parameters

`pl` Maximal length of an expression to be printed
`pm` Toggle to print more or less information about PDEs (`print_more`)
`pa` Toggle to print all or not all information about the PDEs (`print_all`)
`cp` Change the priorities of procedures
`og` Toggle ordering between 'lexicographical ordering of functions having a higher priority than any ordering of derivatives' and the opposite (`lex_fc=t`) resp. (`lex_fc=nil`)
`od` Toggle ordering between 'the total order of derivatives having a higher priority than lexicographical ordering' (`lex_df=nil`) or not (`lex_df=t`)
`oi` Interactive change of ordering on variables
`or` Reverse ordering on variables
`om` Mix randomly ordering on variables
`of` Interactive change of ordering on functions
`op` Print current ordering

ne	Root of the name of new generated equations (default: e_)
nf	Root of the name of new functions and constants (default: c_)
ni	Root of the name of new identities (default: id_)
na	Toggle for the nat output switch (!*nat)
as	Input of an assignment
kp	Toggle for keeping a partitioned copy of each equation (keep_parti)
fi	Toggle for allowing or not allowing integrations of equations which involve unresolved integrals (freeint_)
fa	Toggle for allowing or not allowing solutions of ODEs involving the abs function (freeabs_)
cs	Switch on/off the confirmation of intended substitutions and of the order of the investigation of subcases resulting in a factorization
fs	Enforce direct separation
ll	change of the line length
re	Toggle for allowing to re-cycle equation names (do_recycle_eqn)
rf	Toggle for allowing to re-cycle function names (do_recycle_fnc)
st	Setting a CPU time limit for un-interrupted run
cm	Adding a comment to the history_ list
lr	Adding a LET-rule
cr	Clearing a LET-rule

Help to Change Data of Equations

r	Replace or add one equation
rd	Reduce an equation modulo LET rules
n	Replace one inequality
de	Delete one equation
di	Delete one inequality
c	Change a flag or property of one PDE
pt	Perform a transformation of functions and variables

Help to Work with Identities

i	Print identities between equations
id	Delete redundand equations
iw	Write identities to a file
ir	Remove list of identities
ia	Add or replace an identity
ih	Start recording histories and identities
ip	Stop recording histories and identities
ii	Integrate an identity

`ic` Check the consistency of identity data
`iy` Print the history of equations

Help to Trace and Debug

`tm` Toggle for tracing the main procedure (`tr_main`)
`tg` Toggle for tracing the generalized separation (`tr_gensep`)
`ti` Toggle for tracing the generalized integration (`tr_genint`)
`td` Toggle for tracing the decoupling process (`tr_decouple`)
`tl` Toggle for tracing the decoupling length reduction process (`tr_redlength`)
`ts` Toggle for tracing the algebraic length reduction process (`tr_short`)
`to` Toggle for tracing the ordering procedures process (`tr_orderings`)
`tr` Trace an arbitrary procedure
`ut` Untrace a procedure
`br` Lisp break
`pc` Do a function call
`in` Reading in a REDUCE file

Global variables

The following is a complete list of identifiers used as global Lisp variables (to be precise, symbolic fluid variables) within CRACK. Some are flags and parameters, others are global variables; some of them can be accessed after the CRACK run.

```
!*allowdfint_bak !*dfprint_bak !*exp_bak !*ezgcd_bak
!*fullroots_bak !*gcd_bak !*mcd_bak !*nopowers_bak
!*ratarg_bak !*rational_bak !*batch_mode abs_
adjust_fnc allflags_ batchcount_ !*backup_
collect_sol confirm_subst cont_ contradiction_
cost_limit5 current_dir default_proc_list_
do_recycle_eqn do_recycle_fnc done_trafo eqname_
expert_mode explog_ facint_ flin_ force_sep fname_
fnew_ freeabs_ freeint_ ftem_ full_proc_list_
gcfree!* genint_ glob_var global_list_integer
global_list_ninteger global_list_number high_gensep
homogen_ history_ idname_ idnties_ independence_
ineq_ inter_divint keep_parti last_steps length_inc
level_ lex_df lex_fc limit_time lin_problem
lin_test_const logoprint_ low_gensep max_gc_counter
max_gc_elimin max_gc_fac max_gc_red_len max_gc_short
max_gc_ss max_red_len maxalgsys_ mem_eff
```

```
my_gc_counter nequ_ new_gensep nfct_ nid_ odesolve_
old_history orderings_ target_limit_0 target_limit_1
target_limit_2 target_limit_3 target_limit_4
poly_only potint_ print_ print_all print_more
proc_list_ prop_list pvm_able quick_decoup
record_hist recycle_eqns recycle_fcts recycle_ids
reducefunctions_ repeat_mode safeint_ session_
simple_orderings size_hist size_watch sol_list
solvealg_ stepcounter_ stop_ struc_dim struc_eqn
subst_0 subst_1 subst_2 subst_3 subst_4 time_
time_limit to_do_list tr_decouple tr_genint
tr_gensep tr_main tr_orderings tr_redlength tr_short
trig1_ trig2_ trig3_ trig4_ trig5_ trig6_ trig7_
trig8_ userrules_ vl_
```

Global Flags and Parameters

The list below gives a selection of flags and global parameters that are available, for example, to fine tune the performance according to specific needs of the system of equations that is studied. Usually they are not needed and very few are used regularly by the author. The interactive command that changes the flag/parameter is given in [ ], default values of the flags/parameters are given in (). All values can be changed interactively with the `as` command. The values of the flags and parameters can either be set after loading CRACK and before starting it with a Lisp assignment, for example,

```
lisp(print_ := 8)$
```

or after starting CRACK in interactive mode with specific commands, like `pl` to change specifically the print length determining parameter `print_`, or the command `as` to do an assignment. The values of parameters/flags can be inspected interactively using `pv` and changed with `as`.

`!*batch_mode [x] (t)` : running CRACK in interactive mode (`!*batch_mode=nil`) or automatically (`!*batch_mode=t`). It can also be set in algebraic mode before starting CRACK by `on/off batch_mode`. Interactive mode can be left and automatic computation be started by the interactive command `x`.

`!*iconic (nil)` : whether new processes in parallelization should appear as icons (`t`) or windows (`nil`).

`adjust_fnc (nil)` : if `t` then free constants/functions are scaled and redundant ones are dropped to simplify the result after the computation has been

completed.

collect_sol (t) : whether solutions found shall be collected and returned together at the end or not (to save memory); it matters only for non-linear problems with very many special solutions. If a computation has to be performed with any solution that is found, then these commands can be put into an `algebraic procedure crack_out(eqns, assigns, freef, ineq)` which is currently empty in file `crmain.red` but which is called for each solution.

confirm_subst [cs] (nil) : whether substitutions have to be confirmed interactively.

cont_ (nil) : interactive user control for integration or substitution of large expressions (enabled = `t`).

cost_limit5 (100) : maximal number of extra terms generated by a substitution.

do_recycle (nil) : whether function names shall be recycled or not (saves memory but computation is less clear to follow).

done_trafo (nil) : an (algebraic mode) list of back-transformations that would invert done transformations; this list is useful after CRACK completed to invert transformations if needed.

eqname_ [ne] ('e_) : name of new equations.

expert_mode [t] (nil) : For `expert_mode=t` the equations that are involved in the next computational step are selected by CRACK, for `expert_mode=nil` the user is asked to select one or two equations which are to be worked with in the next computational step.

facint_ (1000) : if `nil` then no search for integrating factors, otherwise sets the maximum of product *terms* * *kernels* when searching for an integrating factor.

flin_ (nil) : a list of functions occuring only linearly in an otherwise non-linear problem; must be assigned before calling CRACK. During execution CRACK tries to preserve the linearity of these functions as long as possible.

fname_ [nf] ('c_) : name of new functions and constants (integration).

force_sep (nil) : whether direct separation should be forced even if functions occur in the supposedly linear independent explicit expressions (for non-linear problem).

freeabs_ [fi] (t) : Do not use solutions of ODEs that involve the `abs` function.

freeint_ [fi] (t) : Do integrations only if explicit part is integrable.

genint_ (15) : if `nil` then generalized integration disabled else sets the maximal number of new functions and extra equations due to the generalized integration of one equation.

high_gensep (300) : minimum size of expressions to separate in a generalized way by `quick_gen_separation`.

homogen_ (nil) : test for homogeneity of each equation (for debugging).

idname_ [ni] ('id_) : name of new equations.

idnties_ (nil) : list of identities resulting from reductions and integrability conditions.

independence_ (nil) : interactive control of linear independence (enabled = `t`).

inter_divint (nil) : whether the integration of divergence identities with more than 2 differentiation variables shall be confirmed interactively.

keep_parti [kp] (nil) : whether for each equation a copy in partitioned form is to be stored to speed up several simplifications but which needs more memory.

last_steps (nil) : a list of the last steps generated and updated automatically in order to avoid cycles.

length_inc (1.0) : factor by which the length of an expression may grow when performing `diff_length_reduction`.

lex_df [od] (nil) : if `t` then use lexicographical instead of total degree ordering of derivatives.

lex_fc [og] (t) : if `t` then lexicographical ordering of functions has higher priority than any ordering of derivatives.

limit_time (nil) : = time() + how many more seconds allowed in batch mode.

logoprint_ (t) : print logo after CRACK call.

low_gensep (6) : maximum size of expressions to be separated in a generalized way by `quick_gen_separation`.

max_gc_counter (100000000) : maximal total number of garbage collections.

max_gc_elimin (15) : maximal number of garbage collections during elimination in decoupling.

max_gc_fac (15) : maximal number of garbage collections during factorization.

max_gc_red_len (30) : maximal number of garbage collections during length reduction.

max_gc_short (40) : maximal number of garbage collections during shortening.

max_gc_ss (10) : maximal number of garbage collections during search of sub-systems.

max_red_len (1000000) : maximal product of lengths of two equations to be combined with length-reducing decoupling.

maxalgsys_ (20) : maximum number of equations to be solved in special-sol.

mem_eff (t) : whether to be memory efficient even if slower.

my_gc_counter (0) : initial value of `my_gc_counter`.

nequ_ (1) : index of the next new equation.

new_gensep (nil) : whether or not a newer (experimental) form of `gensep` should be used.

nfct_ (1) : index of the next new function or constant.

nid_ (1) : index of the next new identity.

odesolve_ (100) : maximal length of a DE (number of terms) to be integrated as ODE.

old_history (nil) : `old_history` is interactive input to be read from this list.

poly_only (nil) : all equations are polynomials only.

potint_ (t) : allowing 'potential integration'.

print_ [pl] (12) : maximal length of an expression to be printed.

print_all [pa] (nil) : print all information about the PDEs.

print_more [pm] (t) : print more informations about the PDEs.

quick_decoup (nil) : whether decoupling should be done faster with less care for saving memory.

record_hist (nil) : whether the history of equations is to be recorded.

safeint_ (t) : use only solutions of ODEs with non-vanishing denominator.

session_ ("bu"+random number+date) : when loading CRACK or executing.

size_watch (nil) : whether before each computational step the size of the system shall be recorded in the global variable `size_hist`.

solvealg_ (nil) : Use SOLVE for algebraic equations.

struc_eqn (nil) : whether the equations have the form of structural equations (an application is Killing vector and Killing tensor computations).

subst_* : maximal length of an expression to be substituted, used with different values for different procedures `subst_level_*`.

target_limit_* (nil) : maximum of product *length(PDE) * length(substituted expression)* for a PDE which is to be used for a substitution. If `target_limit_* = nil` then no length limit, used with different values for different procedures `subst_level_*`.

time_ (nil) : print the time needed for running CRACK.

time_limit (nil) : whether a time limit is active after which batch-mode is interrupted to interactive mode.

tr_decouple [td] (nil) : trace decoupling process.

tr_genint [ti] (nil) : trace generalized integration.

tr_gensep [ts] (nil) : trace generalized separation.

tr_main [tm] (nil) : trace main procedure.

tr_orderings [to] (nil) : trace orderings stuff.

tr_redlength [tr] (nil) : trace length reduction.

20.15.7 A More Detailed Description of Some of the Modules

The package CRACK contains a number of modules. The basic ones are for computing a pseudo-differential Gröbner Basis (using integrability conditions in a systematic way), integrating exact PDEs, separating PDEs, solving DEs containing functions of only a subset of all variables and solving standard ODEs (of Bernoulli or Euler type, linear, homogeneous and separable ODEs). These facilities will be described briefly together with examples. The test file `crack.tst` (in the REDUCE `packages/crack` directory) demonstrates these and others.

Pseudo Differential Gröbner Basis

This module (called 'decoupling' in `proc_list_`) reduces derivatives in equations by using other equations and it applies integrability conditions to formulate additional equations which are subsequently reduced, and so on.

A general algorithm to bring a system of PDEs into a standard form where all integrability conditions are satisfied by applying a finite number of additions, multiplications and differentiations is based on the general theory of involutive systems [Riq10, Tho37, Jan29].

Essential to this theory is a total ordering of partial derivatives which allows assignment to each PDE of a *Leading Derivative* (LD) according to a chosen ordering of functions and derivatives. Examples for possible orderings are

- lex. order of functions > lex. order of variables,
- lex. order of functions > total differential order > lex. order of variables,
- total order > lex. order of functions > lex. order of variables,

or mixtures of them by giving weights to individual functions and variables. Above, the ">" indicate "before" in priority of criteria. The principle is then to

1. take two equations at a time and differentiate them as often as necessary to get equal LDs,
2. regard these two equations as algebraic equations in the common LD and calculate the remainder w.r.t. the LD, i.e. to generate an equation without the LD by the Euclidean algorithm, and
3. add this equation to the system.

Usually pairs of equations are taken first, such that only one of the equations must be differentiated. If in such a generation step one of the equations is not differentiated then it is called a simplification step and this equation will be replaced by the new equation.

The algorithm ends when each combination of two equations yields only equations which simplify to an identity modulo the other equations. A more detailed description is given e.g. in [BB89, Rei90].

Other programs implementing this algorithm are described e.g. in [Sch85b, BB89, FK89, Rei90, RWB96, RLW01] and [Man96].

In the interactive mode of CRACK it is possible to change the lexicographical ordering of variables, of functions, to choose between 'total differential order' ordering of variables or lexicographical ordering of variables and to choose whether lexicographical ordering of functions should have a higher priority than the ordering of the variables in a derivative, or not.

An example of the computation of a differential Gröbner Basis is given in the test file `crack.tst`.

Integrating Exact PDEs

The technical term 'exact' is adapted for PDEs from exterior calculus and is a small abuse of language but it is useful to characterize the kind of PDEs under consideration.

The purpose of the integration module in CRACK is to decide whether a given differential expression D which involves unknown functions $f^i(x^j),\ 1 \leq i \leq m$ of independent variables $x^j,\ 1 \leq j \leq n$ is a total derivative of another expression I w.r.t. some variable $x^k,\ 1 \leq k \leq n$

$$D(x^i,\ f^j,\ f^j{}_{,p},\ f^j{}_{,pq},\ldots) = \frac{dI(x^i,\ f^j,\ f^j{}_{,p},\ f^j{}_{,pq},\ldots)}{dx^k}.$$

The index k is reserved in the following for the integration variable x^k. With an appropriate function of integration c^r, which depends on all variables except x^k, it is no loss of generality to replace $0 = D$ by $0 = I + c^r$ in a system of equations.

Of course there always exists a function I with a total derivative equal to D but the question is whether for *arbitrary* f^i the integral I is functionally dependent only on the f^i and their derivatives, and *not on integrals* of f^i.

Preconditions: D is a polynomial in the f^i and their derivatives. The number of functions and variables is free. For deciding the existence of I only, the explicit occurrence of the variables x^i is arbitrary. In order to actually calculate I explicitly, D must have the property that all terms in D must either contain an unknown function of x^k or must be formally integrable w.r.t. x^k. That means if I exists then only a special explicit occurrence of x^k can prevent the calculation of I and furthermore only in those terms which do not contain any unknown function of x^k.

If such terms occur in D and I exists then I can still be expressed as a polynomial in the $f^i, f^i{}_{,j}, \ldots$ and terms containing indefinite integrals with integrands explicit in x^k.

Algorithm: Successive partial integration of the term with the highest x^k-derivative of any f^i. By that the differential order w.r.t. x^k is reduced successively. This procedure is always applicable because steps involve only differentiations and the polynomial integration ($\int h^n \frac{\partial h}{\partial x} dx = h^{n+1}/(n+1)$) where h is a partial derivative of some function f^i. For a more detailed description see [Wol00].

Stop: Iteration stops if no term with any x^k-derivative of any f^i is left. If any $f^i(x^k)$ occurs in the remaining un-integrated terms then I is not expressible with f^i and its derivatives only. In case no $f^i(x^k)$ occurs, any remaining terms can contain x^k only explicitly. Whether they can be integrated or not depends on their formal integrability. For their integration the REDUCE integrator is applied.

Speed up: The partial integration as described above preserves derivatives with respect to other variables. For example, the three terms $f_{,x}, ff_{,xxx}, f_{,xxy}$ cannot combine somehow to the same terms in the integral because if one ignores x-derivatives then it is clear that f, f^2 and $f_{,y}$ are three functionally independent expressions with respect to x-integrations. This allows the following drastic speed up for large expressions. It is possible to partition the complete sum of terms into partial sums such that each of them has to be integrable on its own. That is managed by generating a label for each term and collecting terms with the same label into partial sums. The label is produced by dropping all x-derivatives from all functions to be computed and dropping all factors which are not powers of derivatives of functions to be computed.

The partitioning into partial sums has two effects. Firstly, if the integration of one partial sum fails then the remaining sums do not have to be tried for integration. Secondly, doing partial integration for each term means doing many subtractions. It is much faster to subtract terms from small sums than from large sums.

Example: We apply the above algorithm to

$$D := 2f_{,y}\, g' + 2f_{,xy}\, g + gg'^3 + xg'^4 + 3xgg'^2 g'' = 0$$

with $f = f(x,y),\ g = g(x),\ ' \equiv d/dx$. Starting with terms containing g and at first with the highest derivative $g_{,xx}$, the steps are

$$\begin{aligned} \int 3xgg_{,x}^2\, g_{,xx}\, dx &= \int d(xgg_{,x}^3) - \int \left(\partial_x(xg) g_{,x}^3\right) dx \\ &= xgg_{,x}^3 - \int g_{,x}^3\, (g + xg_{,x})dx, \end{aligned}$$

$$I := I + xgg_{,x}^{3}$$
$$D := D - g_{,x}^{3}\,(g + xg_{,x}\,) - 3xgg_{,x}^{2}\,g_{,xx}$$

The new terms $-g_{,x}^{3}\,(g+xg_{,x}\,)$ are of lower order than $g_{,xx}$ and so in the expression D the maximal order of x-derivatives of g is lowered. The conditions that D is exact are the following.

- The leading derivative must occur linearly before each partial integration step.
- After the partial integration of the terms with first-order x-derivatives of f the remaining D must not contain f or other derivatives of f, because such terms cannot be integrated w.r.t. x without specifying f.

The result of x- and y-integration in the above example is (remember $g = g(x)$)

$$0 = 2fg + xygg_{,x}^{3} + c_1(x) + c_2(y) \;\; (= I).$$

CRACK can now eliminate f and substitute for it in all other equations.

Generalization: If after applying the above basic algorithm, terms are left which contain functions of x^k but each of these functions depends only on a subset of all $x^i,\ 1 \le i \le n$, then a generalized version of the above algorithm can still provide a formal expression for the integral I (see [Wol00]). The price consists of additional differential conditions, but they are equations in fewer variables than occur in the integrated equation. Integrating for example

$$\tilde{D} = D + g^2(y^2 + x\sin y + x^2 e^y) \tag{20.75}$$

by introducing as few new functions and additional conditions as possible gives for the integral $\tilde{I}$

$$\begin{aligned}\tilde{I} \;\; = \;\; & 2fg + xygg_{,x}^{3} + c_1(x) + c_2(y) \\ & + \frac{1}{3}y^3 c_3'' - \cos y(xc_3'' - c_3) + e^y(x^2 c_3'' - 2xc_3' + 2c_3)\end{aligned}$$

with $c_3 = c_3(x)$, $' \equiv d/dx$ and the single additional condition $g^2 = c_3'''$. The integration of the new terms of (20.75) is achieved by partial integration again, for example

$$\begin{aligned}\int g^2x^2dx \;\; &= \;\; x^2\int g^2dx - \int(2x\int g^2dx)dx \\ &= \;\; x^2\int g^2dx - 2x\iint g^2dx + 2\iiint g^2dx \\ &= \;\; x^2c_3'' - 2xc_3' + 2c_3.\end{aligned}$$

Characterization: This algorithm is a decision algorithm which does not involve any heuristic. After integration, the new equation is still a polynomial in f^i and in the new constant or function of integration. Therefore the algorithms for bringing the system into standard form can still be applied to the PDE-system after the equation $D = 0$ is replaced by $I = 0$.

The complexity of algorithms for bringing a PDE-system into a standard form depends nonlinearly on the order of these equations because of the nonlinearly increasing number of different leading derivatives and by that the number of equations generated intermediately by such an algorithm. It therefore in general pays off to integrate equations during such a standard form algorithm.

If an f^i, which depends on all variables, can be eliminated after an integration, then depending on its length it is in general helpful to substitute f^i in other equations and to reduce the number of equations and functions by one. This is especially profitable if the replaced expression is short and contains only functions of fewer variables than f^i.

Test: The corresponding test input is

```
depend f,x,y;
depend g,x;
crack({2*df(f,y)*df(g,x)+2*df(f,x,y)*g+g*df(g,x)**3
       +x*df(g,x)**4+3*x*g*df(g,x)**2*df(g,x,2)
       +g**2*(y**2+x*sin y+x**2*e**y)},
      {}, {f,g}, {});
```

The meaning of the REDUCE command `depend` is to declare that f depends in an unknown way on x and y. For more details on the algorithm see [Wol00].

Direct Separation of PDEs

As a result of repeated integrations the functions in the remaining equations have fewer and fewer variables. It therefore may happen that after a substitution an equation results where at least one variable occurs only explicitly and not as an argument of an unknown function. Consequently all coefficients of linearly independent expressions in this variable can be set to zero individually.

Example: $f = f(x, y)$, $g = g(x)$, x, y, z are independent variables. The equation is

$$0 = f_{,y} + z(f^2 + g_{,x}) + z^2(g_{,x} + yg^2) \tag{20.76}$$

x-separation? $\rightarrow$ no
y-separation? $\rightarrow$ no

z-separation? $\rightarrow$ yes: $0 = f_{,y} = f^2 + g_{,x} = g_{,x} + yg^2$
y-separation? $\rightarrow$ yes: $0 = g_{,x} = g^2$ (from the third equation from the z-separation)

If z^2 had been replaced in (20.76) by a third function $h(z)$ then direct separation would not have been possible. The situation changes if h is a parametric function which is assumed to be independently given and which should not be calculated, i.e. f and g should be calculated for any arbitrary given $h(z)$. Then the same separation could have been done with an extra treatment of the special case $h_{,zz} = 0$, i.e. h linear in z. This different treatment of unknown functions makes it necessary to input explicitly the functions to be calculated as the third argument to CRACK. The input in this case would be

```
depend f,x,y;
depend g,x;
depend h,z;
crack({df(f,y)+z*f**2+(z+h)*df(g,x)+h*y*g**2}, {},
      {f,g}, {z});
```

The fourth parameter for CRACK is necessary to make clear that in addition to the variables of f and g, z is also an independent variable.

If the flag `independence_` is not `nil` then CRACK will stop if linear independence of the explicit expressions of the separation variable (in the example $1, z, z^2$) is not clear and ask interactively whether separation should be done or not.

Indirect Separation of PDEs

For the above direct separation a precondition is that at least one variable occurs only explicitly or as an argument of parametric functions. The situation where each variable is an argument of at least one function but no function contains all independent variables of an equation needs a more elaborate treatment.

The steps are these

- A variable x_a is chosen which occurs in as few functions as possible. This variable will be separated directly later which requires that all unknown functions f_i containing x_a are to be eliminated. Therefore, as long as $F := \{f_i\}$ is not empty do the following:
  - Choose the function $f_i(y_p)$ in F with the smallest number of variables y_p and with z_{ij} as those variables on which f_i does not depend.
  - Identify all different products P_{ik} of powers of f_i-derivatives and of f_i in the equation. Determine the z_{ij}-dependent factors C_{ik} of the coefficients of P_{ik} and store them in a list.
  - For each C_{il} (i fixed, $l = 1, \ldots$) choose a z_{ij} and:

    - divide by C_{il} the equation and all following elements C_{im} with $m > l$ of this list, such that these elements are still the actual coefficients in the equation after the division,
    - differentiate the equation and the $C_{im}, m > l$ w.r.t. z_{ij}.

- The resulting equation no longer contains any unknown function of x_a and can be separated w.r.t. x_a directly in case x_a still occurs explicitly. In both cases the equation(s) is (are) free of x_a afterwards and inverting the sequence of integration and multiplication of all those equations (in case of direct separability) will also result in an equation(s) free of x_a. More exactly, the steps are
    - multiplication of the equation(s) and the C_{im} with $m < l$ by the elements of the C_{ik}-lists in exactly the inverse order,
    - integration of these exact PDEs and the C_{im} w.r.t. z_{ij}.
- The equations originating that way are used to evaluate those functions which do not depend on x_a and which survived in the above differentiations. Substituting these functions in the original equation may enable direct separability w.r.t. variables on which the f_i do not depend on.
- The whole procedure is repeated for another variable x_b if the original DE could not be separated directly and still has the property that it contains only functions of a subset of all variables in the equation.

The additional bookkeeping of coefficients C_{ik} and their updating by division, differentiation, integration and multiplication is done to use them as integrating factors for the backward integration. The following example makes this clearer. The equation is

$$0 = f(x)g(y) - \frac{1}{2}xf'(x) - g'(y) - (1+x^2)y. \tag{20.77}$$

The steps are (equal levels of indentation in the example correspond to those in the algorithm given above)

- $x_1 := x$, $F = \{f\}$
    - Identify $f_1 := f, \quad y_1 := x, \quad z_{11} := y$
    - and $P_1 = \{f', f\}, \quad C_1 = \{1, g\}$
        - Divide C_{12} and (20.77) by $C_{11} = 1$ and differentiate w.r.t. $z_{11} = y$:

$$\begin{aligned} 0 &= fg' - g'' - (1+x^2) \\ C_{12} &= g' \end{aligned} \tag{20.78}$$

∗ Divide (20.78) by $C_{12} = g'$ and differentiate w.r.t. $z_{11} = y$:

$$0 = -(g''/g')' - (1+x^2)(1/g')'$$

- Direct separation w.r.t. x and integration:

$$\begin{array}{lcllll} x^2:0 & = & (1/g')' & \Rightarrow & c_1 g' = 1 & \Rightarrow \quad g = y/c_1 + c_2 \\ x^0:0 & = & (g''/g')' & \Rightarrow & c_3 g' = g'' & \Rightarrow \quad c_3 = 0 \end{array}$$

- Substitution of g in the original DE

$$0 = (y/c_1 + c_2)f - \frac{1}{2}xf' - 1/c_1 - (1+x^2)y$$

provides a form which allows CRACK standard methods to succeed by direct separation w.r.t. y

$$\begin{array}{lcllll} y^1:0 & = & f/c_1 - 1 - x^2 & \Rightarrow & f' = 2c_1 x \\ y^0:0 & = & c_2 f - \frac{1}{2}xf' - 1/c_1 & \Rightarrow & 0 = c_2 c_1 (1+x^2) - c_1 x^2 - 1/c_1 \end{array}$$

and direct separation w.r.t. x:

$$\begin{array}{lcl} x^0:0 & = & c_2 c_1 - c_1 \\ x^2:0 & = & c_2 c_1 - 1/c_1 \\ & \Rightarrow & 0 = c_1 - 1/c_1 \\ & \Rightarrow & c_1 = \pm 1 \Rightarrow c_2 = 1. \end{array}$$

We get the two solutions $f = 1 + x^2, g = 1 + y$ and $f = -1 - x^2, g = 1 - y$. The corresponding input to CRACK would be

```
depend f,x;
depend g,y;
crack({f*g-x*df(f,x)/2-df(g,y)-(1+x**2)*y},{},{f,g},{});
```

Solving Standard ODEs

For solving standard ODEs the package ODESOLVE by Malcolm MacCallum and Francis Wright [Mac89] is applied. This package is distributed with REDUCE and can be used independently of CRACK. The syntax of ODESOLVE is quite similar to that of CRACK:

```
depend function, variable;
odesolve(ODE, function, variable);
```

The applicability of ODESOLVE is increased by a CRACK-subroutine which recognizes such PDEs in which there is only one unknown function of all variables and all occurring derivatives of this function are only derivatives w.r.t. one variable of only one partial derivative. For example the PDE for $f(x, y)$

$$0 = f_{,xxy} + f_{,xxyy}$$

can be viewed as a first order ODE in y for $f_{,xxy}$.

Acknowledgement

Andreas Brand is the author of a number of core modules of CRACK. The currently used data structure and program structure of the kernel of CRACK are due to him. He contributed to the development of CRACK until 1997.

Francis Wright contributed code to REDUCE that provides simplifications of expressions involving symbolic derivatives and integrals. Also, CRACK makes extensive use of the REDUCE program ODESOLVE written by Malcolm MacCallum and Francis Wright.

Arrigo Triulzi contributed in supporting the use of different total orderings of derivatives in doing pseudo-differential Gröbner Basis computations.

Work on this package has been supported by the Konrad Zuse Institute / Berlin through a fellowship of T.W.. Winfried Neun and Herbert Melenk are thanked for many discussions and constant support. Many of the low level control features have been provided by Winfried Neun. He ported Parallel REDUCE to a Linux PC Beowulf cluster and helped in adapting CRACK to it.

Anthony Hearn provided free copies of REDUCE to us as a REDUCE developers group which also is thankfully acknowledged.

20.16 DESIR: Differential Linear Homogeneous Equation Solutions in the Neighborhood of Irregular and Regular Singular Points

This package enables the basis of formal solutions to be computed for an ordinary homogeneous differential equation with polynomial coefficients over Q of any order, in the neighborhood of zero (regular or irregular singular point, or ordinary point).

Authors: C. Dicrescenzo, F. Richard-Jung, E. Tournier

Differential linear homogenous Equation Solutions in the
neighbourhood of Irregular and Regular singular points

Version 3.1 - Septembre 89

Groupe de Calcul Formel de Grenoble
laboratoire TIM3

(C. Dicrescenzo, F. Richard-Jung, E. Tournier)

E-mail: dicresc@afp.imag.fr

20.16.1 INTRODUCTION

This software enables the basis of formal solutions to be computed for an ordinary homogeneous differential equation with polynomial coefficients over Q of any order, in the neighbourhood of zero (regular or irregular singular point, or ordinary point).
Tools have been added to deal with equations with a polynomial right-hand side, parameters and a singular point not to be found at zero.

This software can be used in two ways :

- direct (DELIRE procedure)
- interactive (DESIR procedure)

The basic procedure is the DELIRE procedure which enables the solutions of a

linear homogeneous differential equation to be computed in the neigh- bourhood of zero.

The DESIR procedure is a procedure without argument whereby DELIRE can be called without preliminary treatment to the data, that is to say, in an interactive autonomous way. This procedure also proposes some transfor- mations on the initial equation. This allows one to start comfortably with an equation which has a non zero singular point, a polynomial right-hand side and parameters.
This document is a succint user manual. For more details on the underlying mathematics and the algorithms used, the reader can refer to :

E. Tournier : Solutions formelles d'equations differentielles - Le logiciel de calcul formel DESIR.
These d'Etat de l'Universite Joseph Fourier (Grenoble - avril 87).

He will find more precision on use of parameters in :

F. Richard-Jung : Representation graphique de solutions d'equations differentielles dans le champ complexe.
These de l'Universite Louis Pasteur (Strasbourg - septembre 88).

20.16.2 FORMS OF SOLUTIONS

We have tried to represent solutions in the simplest form possible. For that, we have had to choose different forms according to the complexity of the equation (parameters) and the later use we shall have of these solutions.

"**general solution**" = {....., { split_sol , cond },....}

cond	=	list of conditions or empty list (if there is no condition) that parameters have to verify such that split_sol is in the basis of solutions. In fact, if there are parameters, basis of solutions can have different expressions according to the values of parameters. (Note : if cond={}, the list "general solution" has one element only.)
split_sol	=	{ $q, ram, polysol, r$ } (" split solution " enables precise information on the solution to be obtained immediately)

The variable in the differential operator being x, solutions are expressed in respect to a new variable xt, which is a fractional power of x, in the following way :

q	:	polynomial in $1/xt$ with complex coefficients
ram	:	$xt = x^{ram}$ ($1/ram$ is an integer)
$polysol$	:	$polynomial$ in $log(xt)$ with formal series in xt coefficients
r	:	root of a complex coefficient polynomial ("indicial equation").

"**standard solution**" = $e^{qx} x^{r*ram} polysolx$

qx and $polysolx$ are q and $polysol$ expressions in which xt has been replaced by x^{ram}

N.B. : the form of these solutions is simplified according to the nature of the point zero.

- if 0 is a regular singular point : the series appearing in $polysol$ are convergent, ram = 1 and q = 0.
- if 0 is a regular point, we also have : $polysol$ is constant in $log(xt)$ (no logarithmic terms).

20.16.3 INTERACTIVE USE

To call the procedure : desir();
solution:=desir();

The DESIR procedure computes formal solutions of a linear homogeneous differential equation in an interactive way.
In this equation the variable *must be x*.

The procedure requires the order and the coefficients of the equation, the names of parameters if there are any, then if the user wants to transform this equation and how (for example to bring back a singular point to zero see procedures changehom, changevar, changefonc -).

This procedure DISPLAYS the solutions and RETURNS a list of general term { lcoeff, {....,{ general_solution },....}}. The number of elements in this list is linked to the number of transformations requested :

* lcoeff : list of coefficients of the differential equation
* general_solution : solution written in the general form

20.16.4 DIRECT USE

procedure delire($x, k, grille, lcoeff, param$**);**
This procedure computes formal solutions of a linear homogeneous differential equation with polynomial coefficients over Q and of any order, in the neighborhood of zero, regular or irregular singular point. In fact it initializes the call of

the NEWTON procedure that is a recursive procedure (algorithm of NEWTON-RAMIS-MALGRANGE)

x	:	variable
k	:	"number of desired terms". For each formal series in xt appearing in $polysol$, $a_0 + a_1xt + a_2xt^2 + ... + a_nxt^n + \ldots$, we compute the $k+1$ first coefficients $a_0, a_1, \ldots, a_k$.
$grille$	:	the coefficients of the differential operator are polynomial in x^{grille} (in general $grille = 1$)
$lcoeff$	:	list of coefficients of the differential operator (in increasing order of differentiation)
$param$	:	list of parameters

This procedure RETURNS the list of general solutions.

20.16.5 USEFUL FUNCTIONS

Reading of equation coefficients

procedure lectabcoef( );
This procedure is called by DESIR to read the coefficients of an equation, in *increasing order of differentiation*, but can be used independently.

reading of n : order of the equation.
reading of parameters (only if a variable other than x appears in the coefficients)
this procedure returns the list { $lcoeff, param$ } made up of the list of coefficients and the list of parameters (which can be empty).

Verification of results

procedure solvalide($solutions, solk, k$);
This procedure enables the validity of the solution number solk in the list "solutions" to be verified.
$solutions$ = {$lcoeff$,{....,{$general\_solution$},....}} is any element of the list returned by DESIR or is {$lcoeff, sol$} where sol is the list returned by DELIRE.

If we carry over the solution $e^{qx}x^{r*ram}polysolx$ in the equation, the result has the form $e^{qx}x^{r*ram}reste$, where $reste$ is a polynomial in $log(xt)$, with polynomial coefficients in xt. This procedure computes the minimal valuation V of $reste$ as polynomial in xt, using k "number of desired terms" asked for at the call of DESIR or DELIRE, and DISPLAYS the "theoretical" size order of the regular part of the result : $x^{ram*(r+v)}$.

On the other hand, this procedure carries over the solution in the equation and DISPLAYS the significative term of the result. This is of the form :

$$e^{qx}x^{a}polynomial(log(xt)), \qquad \text{with} \quad a >= ram*(r+v).$$

Finally this procedure RETURNS the complete result of the carry over of the solution in the equation.

This procedure cannot be used if the solution number solk is linked to a condition.

Writing of different forms of results

procedure standsol(solutions);

This procedure enables the simplified form of each solution to be obtained from the list "solutions", $\{lcoeff,\{...,\{general\_solution\},....\}\}$ which is one of the elements of the list returned by DESIR, or $\{lcoeff, sol\}$ where sol is the list returned by DELIRE.

This procedure RETURNS a list of 3 elements : $\{\ lcoeff, solstand, solcond\ \}$

$lcoef$	=	list of differential equation coefficients
$solstand$	=	list of solutions written in standard form
$solcond$	=	list of conditional solutions that have not been written in standard form. This solutions remain in general form.

This procedure has no meaning for "conditional" solutions. In case, a value has to be given to the parameters, that can be done either by calling the procedure SORPARAM that displays and returns these solutions in the standard form, either by calling the procedure SOLPARAM which returns these solutions in general form.

procedure sorsol(sol);

This procedure is called by DESIR to write the solution sol, given in general form, in standard form with enumeration of different conditions (if there are any).
It can be used independently.

Writing of solutions after the choice of parameters

procedure sorparam($solutions, param$**);**

This is an interactive procedure which displays the solutions evaluated : the value of parameters is requested.

$solutions$	:	$\{lcoeff,\{....,\{general\_solution\},....\}\}$
$param$	:	list of parameters.

It returns the list formed of 2 elements :

- list of evaluated coefficients of the equation
- list of standard solutions evaluated for the value of parameters.

procedure solparam($solutions, param, valparam$**);**
This procedure evaluates the general solutions for the value of parameters given by valparam and returns these solutions in general form.

$solutions$	:	$\{lcoeff$,{....,{$general\_solution$},....}}
$param$	:	list of parameters
$valparam$	:	list of parameters values

It returns the list formed of 2 elements :

- list of evaluated coefficients of the equation
- list of solutions in general form, evaluated for the value of parameters.

Transformations

procedure changehom($lcoeff, x, secmember, id$**);**
Differentiation of an equation with right-hand side.

$lcoeff$	:	list of coefficients of the equation
x	:	variable
$secmember$	:	right-hand side
id	:	order of the differentiation.

It returns the list of coefficients of the differentiated equation. It enables an equation with polynomial right-hand side to be transformed into a homogeneous equation by differentiating id times, id = degre($secmember$) + 1.
procedure changevar($lcoeff, x, v, fct$**);**
Changing of variable in the homogeneous equation defined by the list,lcoeff of its coefficients : the old variable x and the new variable v are linked by the relation $x = fct(v)$.

It returns the list of coefficients in respect to the variable v of the new equation.

examples of use :

- translation enabling a rational singularity to be brought back to zero.
- $x = 1/v$ brings the infinity to 0.

procedure changefonc($lcoeff, x, q, fct$**);**
Changing of unknown function in the homogeneous equation defined by the list lcoeff of its coefficients :

$lcoeff$	:	list of coefficients of the initial equation
x	:	variable
q	:	new unknown function
fct	:	y being the unknown function $y = fct(q)$

It returns the list of coefficients of the new equation.

Example of use :

this procedure enables the computation,in the neighbourhood of an irregular singularity, of the "reduced" equation associated to one of the slopes (the Newton polygon having a null slope of no null length). This equation gives much informations on the associated divergent series.

Optional writing of intermediary results

switch trdesir : when it is ON, at each step of the Newton algorithm, a description of the Newton polygon is displayed (it is possible to follow the break of slopes), and at each call of the FROBENIUS procedure (case of a null slope) the corresponding indicial equation is displayed.

By default, this switch is OFF.

20.16.6 LIMITATIONS

1. This DESIR version is limited to differential equations leading to indicial equations of degree $<= 3$. To pass beyond this limit, a further version written in the D5 environment of the computation with algebraic numbers has to be used.

2. The computation of a basis of solutions for an equation depending on parameters is assured only when the indicial equations are of degree $<= 2$.

20.17 DFPART: Derivatives of Generic Functions

This package supports computations with total and partial derivatives of formal function objects. Such computations can be useful in the context of differential equations or power series expansions.

Author: Herbert Melenk

The package DFPART supports computations with total and partial derivatives of formal function objects. Such computations can be useful in the context of differential equations or power series expansions.

20.17.1 Generic Functions

A generic function is a symbol which represents a mathematical function. The minimal information about a generic function function is the number of its arguments. In order to facilitate the programming and for a better readable output this package assumes that the arguments of a generic function have default names such as $f(x,y)$,$q(rho,phi)$. A generic function is declared by prototype form in a statement

`generic_function` ⟨*fname*⟩`(`⟨*arg*$_1$⟩,⟨*arg*$_2$⟩,...,⟨*arg*$_n$⟩`);`

where $fname$ is the (new) name of a function and arg_i are symbols for its formal arguments. In the following $fname$ is referred to as “generic function", $arg_1, arg_2, \ldots, arg_n$ as “generic arguments" and $fname(arg_1, arg_2, \ldots, arg_n)$ as “generic form". Examples:

```
generic_function f(x,y);
generic_function g(z);
```

After this declaration REDUCE knows that

- there are formal partial derivatives $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ $\frac{\partial g}{\partial z}$ and higher ones, while partial derivatives of f and g with respect to other variables are assumed as zero,
- expressions of the type $f()$, $g()$ are abbreviations for $f(x,y)$, $g(z)$,
- expressions of the type $f(u,v)$ are abbreviations for $sub(x=u, y=v, f(x,y))$
- a total derivative $\dfrac{df(u,v)}{dw}$ has to be computed as $\dfrac{\partial f}{\partial x}\dfrac{du}{dw} + \dfrac{\partial f}{\partial y}\dfrac{dv}{dw}$

20.17.2 Partial Derivatives

The operator `dfp` represents a partial derivative:

`dfp(`⟨*expr*⟩, ⟨*dfarg*$_1$⟩, ⟨*dfarg*$_2$⟩, ..., ⟨*dfarg*$_n$⟩`);`

where ⟨*expr*⟩ is a function expression and ⟨*dfarg*$_i$⟩ are the differentiation variables. Examples:

```
dfp(f(),{x,y});
```

means $\frac{\partial^2 f}{\partial x \partial y}$ and

```
dfp(f(u,v),{x,y});
```

stands for $\frac{\partial^2 f}{\partial x \partial y}(u,v)$. For compatibility with the DF operator the differentiation variables need not be entered in list form; instead the syntax of `DF` can be used, where the function expression is followed by the differentiation variables, eventually with repetition numbers. Such forms are interenally converted to the above form with a list as second parameter.

The expression $expr$ can be a generic function with or without arguments, or an arithmetic expression built from generic functions and other algebraic parts. In the second case the standard differentiation rules are applied in order to reduce each derivative expressions to a minimal form.

When the switch `nat` is on partial derivatives of generic functions are printed in standard index notation, that is f_{xy} for $\frac{\partial^2 f}{\partial x \partial y}$ and $f_{xy}(u,v)$ for $\frac{\partial^2 f}{\partial x \partial y}(u,v)$. Therefore single characters should be used for the arguments whenever possible. Examples:

```
generic_function f(x,y);
generic_function g(y);
dfp(f(),x,2);

f
 xx

dfp(f()*g(),x,2);

f  *g()
```

```
 xx

dfp(f()*g(),x,y);

f  *g() + f *g
 xy        x  y
```

The difference between partial and total derivatives is illustrated by the following example:

```
generic_function h(x);
dfp(f(x,h(x))*g(h(x)),x);

f (x,h(x))*g(h(x))
 x

df(f(x,h(x))*g(h(x)),x);

f (x,h(x))*g(h(x)) + f (x,h(x))*h (x)*g(h(x))
 x                    y          x

 + g (h(x))*h (x)*f(x,h(x))
    y        x
```

Cooperation of partial derivatives and Taylor series under a differential side relation $\frac{dq}{dx} = f(x,q)$:

```
load_package taylor;
operator q;
let df(q(~x),x) => f(x,q(x));
taylor(q(x0+h),h,0,3);


q(x0) + f(x0,q(x0))*h

    f (x0,q(x0)) + f (x0,q(x0))*f(x0,q(x0))
     x              y                          2
 + ----------------------------------------*h  +
                       2

(f  (x0,q(x0)) + f  (x0,q(x0))*f(x0,q(x0))
  xx             xy
```

```
+ f (x0,q(x0))*f (x0,q(x0))
   x             y

+ f  (x0,q(x0))*f(x0,q(x0))
   yx

                              2
+ f  (x0,q(x0))*f(x0,q(x0))
   yy

               2                    3       4
+ f (x0,q(x0)) *f(x0,q(x0)))/6*h  + O(h )
   y
```

Normally partial differentials are assumed as non-commutative

```
dfp(f(),x,y)-dfp(f(),y,x);

f   - f
 xy    yx
```

However, a generic function can be declared to have globally interchangeable partial derivatives using the declaration `dfp_commute` which takes the name of a generic function or a generic function form as argument. For such a function differentiation variables are rearranged corresponding to the sequence of the generic variables.

```
generic_function q(x,y);
dfp_commute q(x,y);
dfp(q(),{x,y,y}) + dfp(q(),{y,x,y}) + dfp(q(),{y,y,x});

3*q
   xyy
```

If only a part of the derivatives commute, this has to be declared using the standard REDUCE rule mechanism. Please note that then the derivative variables must be written as list.

20.17.3 Substitutions

When a generic form or a `dfp` expression takes part in a substitution the following steps are performed:

1. The substitutions are performed for the arguments. If the argument list is empty the substitution is applied to the generic arguments of the function; if these change, the resulting forms are used as new actual arguments. If the generic function itself is not affected by the substitution, the process stops here.
2. If the function name or the generic function form occurs as a left hand side in the substitution list, it is replaced by the corresponding right hand side.
3. The new form is partially differentiated according to the list of partial derivative variables.
4. The (eventually modified) actual parameters are substituted into the form for their corresponding generic variables. This substitution is done by name.

Examples:

```
generic_function f(x,y);
sub(y=10,f());

f(x,10)

sub(y=10,dfp(f(),x,2));


f  (x,10)
 xx

sub(y=10,dfp(f(y,y),x,2));

f  (10,10)
 xx

sub(f=x**3*y**3,dfp(f(),x,2));

     3
6*x*y

generic_function ff(y,z);
sub(f=ff,f(a,b));
```

```
ff(b,z)
```

The dataset `dfpart.tst` contains more examples, including a complete application for computing the coefficient equations for Runge-Kutta ODE solvers.

20.18 DUMMY: Canonical Form of Expressions with Dummy Variables

This package allows a user to find the canonical form of expressions involving dummy variables. In that way, the simplification of polynomial expressions can be fully done. The indeterminates are general operator objects endowed with as few properties as possible. In that way the package may be used in a large spectrum of applications.

Author: Alain Dresse

20.18.1 Introduction

The possibility to handle dummy variables and to manipulate dummy summations are important features in many applications. In particular, in theoretical physics, the possibility to represent complicated expressions concisely and to realize simplifications efficiently depend on both capabilities. However, when dummy variables are used, there are many more ways to express a given mathematical objects since the names of dummy variables may be chosen almost arbitrarily. Therefore, from the point of view of computer algebra the simplification problem is much more difficult. Given a definite ordering, one is, at least, to find a representation which is independent of the names chosen for the dummy variables otherwise, simplifications are impossible. The package does handle any number of dummy variables and summations present in expressions which are arbitrary multivariate polynomials and which have operator objects eventually dependent on one (or several) dummy variable(s) as some of their indeterminates. These operators have the same generality as the one existing in REDUCE. They can be noncommutative, anticommutative or commutative. They can have any kind of symmetry property. Such polynomials will be called in the following *dummy* polynomials. Any monomial of this kind will be called *dummy* monomial. For any such object, the package allows to find a well defined *normal form* in one-to-one correspondance with it.

In section 2, the convention for writing dummy summations is explained and the available declarations to introduce or suppress dummy variables are given.

In section 3, the commands allowing to give various algebraic properties to the operators are described.

In section 4, the use of the function `canonical` is explained and illustrated.

For references, see [BL85, BC82, But82, Leo80, Leo84, Leo91, Lin91, McK78, RT89, Sim71b, Sim71a, BC94, Cap97].

The use of DUMMY requires that the package ASSIST version 2.2 be available. It is loaded automatically when DUMMY is loaded.

20.18.2 Dummy variables and dummy summations

A dummy variable (let us name it `dv`) is an identifier which runs from the integer i_1 to another integer i_2. To the extent that no definite space is defined, i_1 and i_2 are assumed to be some integers which are the *same* for all dummy variables.

If `f` is any REDUCE operator, then the simplest dummy summation associated to `dv` is the sum

$$\sum_{dv=i_1}^{i_2} f(dv)$$

and is simply written as

$$f(dv).$$

No other rules govern the implicit summations. `dv` can appear as many times we want since the operator `f` may depend on an arbitrary number of variables. So, the package is potentially applicable to many contexts. For instance, it is possible to add rules of the kind one encounters in tensor calculus.

Obviously, there are as many ways we want to express the *same* quantity. If the name of another dummy variable is `dum` then the previous expression is written as

$$\sum_{dum=i_1}^{i_2} f(dum)$$

and the computer algebra system should be able to find that the expression

$$f(dv) - f(dum);$$

is equal to 0. A very special case which is *allowed* is when `f` is the identity operator. So, a generic dummy polynomial will be a sum of dummy monomials of the kind

$$\prod_i c_i * f_i(dv_1, \ldots, dv_{k_i}, fr_1, \ldots, fr_{l_i})$$

where $dv_1, \ldots,$ are dummy variables while $fr_1, \ldots,$ are ordinary or free variables.

To declare dummy variables, two commands are available:

- i.

  `dummy_base` ⟨*idp*⟩;

  where ⟨*idp*⟩ is the name of any unassigned identifier.

- ii.

  `dummy_names` ⟨*d*⟩,⟨*dp*⟩,⟨*dpp*⟩, ...;

The first one declares idp_1,...,idp_n as dummy variables i.e. all variables of the form idp_xxx where xxx is a number will be dummy variables, such as idp_1, idp_2,..., idp_23. The second one gives special names for dummy variables. All other identifiers which may appear are assumed to be *free*. However, there is a restriction: named and base dummy variables cannot be declared *simultaneously*. The above declarations are mutually *exclusive*. Here is an example showing that:

```
dummy_base dv; ==> dv

        % dummy indices are dv1, dv2, dv3, ...

dummy_names i,j,k; ==>

        ***** The created dummy base dv must be cleared
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(i)*op(dv2)$
```

means a sum over dv_1, dv_2. To clear the dummy base, and to create the dummy names i, j, k one is to do

```
clear_dummy_base; ==> t

dummy_names i,j,k; ==> t

            % dummy indices are i,j,k.
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(x)*op(i)^3*op(dv2)$
```

means a sum over i. Similarly, the command clear_dummy_names clears earlier defined named dummy variables.

One should keep in mind that every application of the above commands erases the previous ones. It is also possible to display the declared dummy names using show_dummy_names:

```
show_dummy_names(); ==> {i,j,k}
```

To suppress *all* dummy variables one can enter

```
clear_dummy_names; clear_dummy_base;
```

20.18.3 The Operators and their Properties

All dummy variables *should appear at first level* as arguments of operators. For instance, if i and j are dummy variables, the expression

```
rr:=  op(i,j)-op(j,j)
```

is allowed but the expression

```
op(i,op(j)) - op(j,op(j))
```

is *not* allowed. This is because dummy variables are not detected if they appear at a level larger than 1. Apart from that there is no restrictions. Operators may be commutative, noncommutative or even anticommutative. Therefore they may be elements of an algebra, they may be tensors, spinors, grassman variables, etc. ... By default they are assumed to be *commutative* and without symmetry properties. The REDUCE command `noncom`declaration is taken into account and, in addition, the command

```
anticom at1, at2;
```

makes the operators at_1 and at_2 anticommutative.

One can also give symmetry properties to them. The usual declarations `symmetric` and `antisymmetric` are taken into account. Moreover and most important they can be endowed with a *partial* symmetry through the command `symtree`. Here are three illustrative examples for the r operator:

```
symtree (r,{!+, 1, 2, 3, 4});
symtree (r,{!*, 1, {!-, 2, 3, 4}});
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});
```

The first one makes the operator (fully) symmetric. The second one declares it antisymmetric with respect to the three last indices. The symbols !*, !+ and !- at the beginning of each list mean that the operator has no symmetry, is symmetric or is antisymmetric with respect to the indices inside the list. Notice that the indices are not denoted by their names but merely by their natural order of appearance. 1 means the first written argument of r, 2 its second argument etc. The first command is equivalent to the declaration `symmetric` except that the number of indices of r is *restricted* to 4 i.e. to the number declared in `symtree`. In the second example r is stated to have no symmetry with respect to the first index and is declared to be antisymmetric with respect to the three last indices. In the third example, r is

made symmetric with respect to the interchange of the pairs of indices 1,2 and 3,4 respectively and is made antisymmetric separately within the pairs $(1, 2)$ and $(3, 4)$. It is the symmetry of the Riemann tensor. The anticommutation property and the various symmetry properties may be suppressed by the commands `remanticom` and `remsym`. To eliminate partial symmetry properties one can also use `symtree` itself. For example, assuming that r has the Riemann symmetry, to eliminate it do

```
symtree (r,{!*, 1, 2, 3, 4});
```

However, notice that the number of indices remains fixed and equal to 4 while with `remsym` it becomes again arbitrary.

20.18.4 The `canonical` Operator

`canonical` is the most important functionality of the package. It can be applied on any polynomial whether it is a dummy polynommial or not. It returns a normal form uniquely determined from the current ordering of the system. If the polynomial does not contain any dummy index, it is rewriten taking into account the various operator properties or symmetries described above. For instance,

```
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});

aa:=r(x3,x4,x2,x1)$

canonical aa; ==>  - r(x1,x2,x3,x4).
```

If it contains dummy indices, `canonical` takes also into account the various dummy summations, makes the relevant simplifications, eventually rename the dummy indices and returns the resulting normal form. Here is a simple example:

```
operator at1,at2;
anticom at1,at2;

dummy_names i,j,k; ==> t

show_dummy_names(); ==> {i,j,k}

rr:=at1(i)*at2(k) -at2(k)*at1(i)$


canonical rr; => 2*at1(i)*at2(j)
```

It is important to notice, in the above example, that in addition to the summa-

tions over indices i and k, and of the anticommutativity property of the operators, `canonical` has replaced the index k by the index j. This substitution is essential to get full simplification. Several other examples are given in the test file and, there, the output of `canonical` is explained.

As stated in the previous section, the dependence of operators on dummy indices is limited to *first* level. An erroneous result will be generated if it is not the case as the subsequent example illustrates:

```
operator op;

dummy_names i,j;

rr:=op(i,op(j))-op(j,op(j))$

canonical rr; ==> 0
```

Zero is obtained because, in the second term, `canonical` has replaced j by i but has left $op(j)$ unchanged because it *does not recognize* the index j which is inside. This fact has also the consequence that it is unable to simplify correctly (or at all) expressions which contain some derivatives. For instance (i and j are dummy indices):

```
aa:=df(op(x,i),x) -df(op(x,j),x)$

canonical aa; ==> df(op(x,i),x) - df(op(x,j),x)
```

instead of zero. A second limitation is that `canonical` does not add anything to the problem of simplifications when side relations (like Bianchi identities) are present.

20.19 EDS: A Package for Exterior Differential Systems

EDS is a REDUCE package for symbolic analysis of partial differential equations using the geometrical approach of exterior differential systems. The package implements much of exterior differential systems theory, including prolongation and involution analysis, and has been optimised for large, non-linear problems.

Author: David Hartley

EDS is a REDUCE package for symbolic analysis of partial differential equations using the geometrical approach of exterior differential systems. The package implements much of exterior differential systems theory, including prolongation and involution analysis, and has been optimised for large, non-linear problems.

20.19.1 Introduction

Exterior differential systems give a geometrical framework for partial differential equations and more general differential geometric problems. The geometrical formulation has several advantages stemming from its coordinate-independence, including superior treatment of nonlinear and global problems. There is not sufficient space in this manual for an introduction to exterior differential systems beyond the scant details given in section 20.19.2, but there are a number of up-to-date texts on the subject (eg [BCG+91, Spi79]).

EDS provides a number of tools for setting up and manipulating exterior differential systems and implements many features of the theory. Its main strengths are the ability to use anholonomic or moving frames and the care taken with nonlinear problems.

There has long been interest in implementing the theory of exterior differential systems in a computer algebra system (eg [ASY74, GMM+81, HT91]). The EDS package owes much to these earlier efforts, and also to related packages for PDE analysis (eg [MF93, Rei91, Sei95]), as well as to earlier versions of EDS produced at Lancaster university with R. W. Tucker and P. A. Tuckey. Finally, EDS uses the exterior calculus package EXCALC of E. Schrüfer 20.21 and the exterior ideals package XIDEAL 20.70. XIDEAL and EXCALC are loaded automatically with EDS.

This work has been supported by the Graduate College on Scientific Computing, University of Cologne and GMD St Augustin, funded by the DFG (Deutsche Forschungsgemeinschaft). I would like to express my thanks to R. W. Tucker, E. Schrüfer, P. A. Tuckey, F. W. Hehl and R. B. Gardner for helpful and encouraging discussions.

20.19.2 EDS data structures and concepts

This section presents the various structures used for expressing exterior systems quantities in EDS. In addition, some the concepts used in EDS to aid computation are described.

Coframings

Within the context of EDS, a *coframing* means a real finite-dimensional differentiable manifold with a given global cobasis. The information about a coframing required by EDS is kept in a ⟨*coframing*⟩ object. The cobasis is the identifying element of an EDS ⟨*coframing*⟩: distinct cobases for the same differentiable manifold are treated as distinct ⟨*coframing*⟩ objects in EDS. The cobasis may be either holonomic or anholonomic, allowing some manifolds with non-trivial topology (eg. group manifolds) to be treated.

In addition to the cobasis, an EDS ⟨*coframing*⟩ can be given *coordinates*, *structure equations* and *restrictions*. The coordinates may be an incomplete or overcomplete set. The structure equations express the exterior derivative of the coordinates and cobasis elements as needed. All coordinate differentials must be expressed in terms of the given cobasis, but not all cobasis differentials need be known. The restrictions are a set of inequalities (at present using just $\neq$) describing point sets not in the manifold.

The ⟨*coframing*⟩ object is, of course, by no means a full description of a differentiable manifold. For example, there is no topology and there are no charts. However, the ⟨*coframing*⟩ object carries sufficient information about the underlying manifold to allow a range of exterior systems calculations to be carried out. As such, it is convenient to accept an abuse of language and think of the ⟨*coframing*⟩ object as a manifold.

A ⟨*coframing*⟩ is constructed or selected using the `coframing` operator.

Examples:

- $\mathbf{R}^3$ with cobasis $\{\mathrm{d}x, \mathrm{d}y, \mathrm{d}z\}$ and coordinates $\{x, y, z\}$.
- $\mathbf{R}^2\backslash\{0\}$ with cobasis $\{e^1, e^2\}$, a single coordinate $\{r\}$, "structure equations" $\{\mathrm{d}r = e^1, \mathrm{d}e^1 = 0, \mathrm{d}e^2 = e^1 \wedge e^2/r\}$ and restrictions $\{r \neq 0\}$.
- $\mathbf{R}^2\backslash\{0\}$ with cobasis $\{\mathrm{d}x, \mathrm{d}y\}$, coordinates $\{x, y\}$ and restrictions $\{x^2 + y^2 \neq 0\}$.
- S^1 with cobasis $\{\omega\}$ and structure equations $\{\mathrm{d}\omega = 0\}$.

- S^2 cannot be encapsulated by an EDS ⟨*coframing*⟩ since there is no global cobasis.

20.19.3 Exterior differential systems

A simple ⟨*EDS*⟩, or exterior differential system, is a triple (S, Ω, M), where M is a ⟨*coframing*⟩ (section 20.19.2), S is a ⟨*system*⟩ (section 20.19.3) on M, and Ω is an independence condition: either a decomposable ⟨*p-form*⟩ or a ⟨*system*⟩ of 1-forms on M (exterior differential systems without independence condition are not treated by EDS).

More generally, an ⟨*EDS*⟩ is a list of simple ⟨*EDS*⟩ objects where the various coframings are all disjoint. This last requirement in not enforced within EDS unless the `edsdisjoint` switch is `on` (section 20.19.12). These more general ⟨*EDS*⟩ objects are represented as a list of simple ⟨*EDS*⟩ objects. All operators which take an ⟨*EDS*⟩ argument accept both simple and compound types.

The trivial ⟨*EDS*⟩, describing an inconsistent problem with no solutions, is defined to be ({1},{},{}).

An ⟨*EDS*⟩ is represented by the `eds` operator (section 20.19.4), and can additionally be generated using the `contact` and `pde2eds` operators (sections 20.19.4, 20.19.4).

The solutions of (S, Ω, M) are integral manifolds, or immersions (cf section 20.19.3) on which S vanishes and the rank of Ω is preserved. Solutions at a single point are described by integral elements (section 20.19.3).

Systems

In EDS, the label ⟨*system*⟩ refers to a list

{⟨*p-form expr*⟩,· · · }

of differential forms. This is distinct from an ⟨*EDS*⟩ (section 20.19.3), which has additional structure. However, many EDS operators will accept either an ⟨*EDS*⟩ or a ⟨*system*⟩ as arguments. In the latter case, any extra information which is required is taken from the background coframing (section 20.19.3).

The ⟨*system*⟩ of an ⟨*EDS*⟩ can be obtained with the `system` operator (section 20.19.5).

Background coframing

The information encapsulated in a `coframing` operator is usually inactive. However, when operations are performed on a ⟨*coframing*⟩ or an ⟨*EDS*⟩ object (sections 20.19.2, 20.19.3), this information is activated for the duration of those operations. It is possible to activate the rules and orderings of a `coframing` operator globally, by making it the *background coframing*. All subsequent EXCALC operations will be governed by those rules. Operations on ⟨*EDS*⟩ objects are unaffected, since their coframings are still activated locally. The background coframing can be set and changed with the `set_coframing` command, and inspected using `coframing`.

Integral elements

An *integral element* of an exterior system (S, Ω, M) is a subspace $P \subset T_pM$ of the tangent space at some point $p \in M$ such that all forms in S vanish when evaluated on vectors from P. In addition, no non-zero vector in P may annul every form in Ω.

Alternatively, an integral element $P \subset T_pM$ can be represented by its annihilator $P^\perp \subset T_p^*M$, comprising those 1-forms at p which annul every vector in P. This can also be understood as a maximal set of 1-forms at p such that $S \simeq 0 \pmod{P^\perp}$ and the rank of Ω is preserved modulo $P^\perp$. This is the representation used by EDS. Further, the reference to the point p is omitted, so an ⟨*integral element*⟩ in EDS is a distribution of 1-forms on M, specified as a ⟨*system*⟩ of 1-forms.

In specifying an integral element for a particular ⟨*EDS*⟩, it is possible to omit the Pfaffian component of the ⟨*EDS*⟩, since these 1-forms must be part of any integral element.

Examples:

- With $M = \mathbf{R}^3 = \{(x, y, z)\}$, $S = \{\mathrm{d}x \wedge \mathrm{d}z\}$ and $\Omega = \{\mathrm{d}x, \mathrm{d}y\}$, the integral element $P = \{\partial_x + \partial_z, \partial_y\}$ is equally determined by its annihilator $P^\perp = \{\mathrm{d}z - \mathrm{d}x\}$.
- For $S = \{\mathrm{d}z - y\mathrm{d}x\}$ and $\Omega = \{\mathrm{d}x\}$, the integral element $P = \{\partial_x + y\partial_z\}$ can be specified simply as $\{\mathrm{d}y\}$.

Properties

For large problems, it can require a great deal of computation to establish whether, for example, a system is closed or not. In order to save recomputing such properties, an ⟨*EDS*⟩ object carries a list of ⟨*properties*⟩ of the form

{ ⟨*keyword*⟩ = ⟨*value*⟩,· · · }

where ⟨*keyword*⟩ is one of the following: `closed`, `quasilinear`, `pfaffian` or `involutive`, and ⟨*value*⟩ is either `0` (false) or `1` (true). These properties are suppressed when an ⟨*EDS*⟩ is printed, unless the `nat` switch is `off`. They can be examined using the `properties` operator (section 20.19.5).

Properties are usually generated automatically by EDS as required, but may be explicitly checked using the operators in section 20.19.8. If a property is not yet present on the list, it is not yet known, and must be checked explicitly if required.

In addition to the properties just described, an ⟨*EDS*⟩ object carries a number of hidden properties which record the results of previous calculations, such as the closure or information about the prolongation of the system. These hidden properties speed up many operations which contain common sub-calculations. The hidden properties are stored using internal LISP data structures and so are not available for inspection.

Properties can be asserted when an ⟨*EDS*⟩ is constructed with the `eds` operator (section 20.19.4). Care is needed since such assertions are never checked. Properties can be erased using the `cleanup` operator (section 20.19.14).

Maps

Within EDS, a map $f : M \to N$ is given as a ⟨*map*⟩ object, a list

{ ⟨*coordinate*⟩ = ⟨*expr*⟩,· · · ,⟨*expr*⟩ neq ⟨*expr*⟩,· · · }

of substitutions and restrictions. The substitutions express coordinates on the target manifold N in terms of those on the source manifold M. The restrictions describe point sets not contained in the source manifold M. The ordering of substitutions and restrictions in the list is unimportant. It is not necessary that the restrictions and right-hand sides of the substitutions be written entirely in M coordinates, but it must be possible by repeated substitution to produce expressions on M (see the examples below). Any denominators in the substitutions are automatically added to the list of restrictions. It is not necessary to include trivial equations for coordinates which are present on both M and N. Note that projections cannot be represented in this fashion (but see the `cross` operator, section 20.19.6).

Maps are applied using the `pullback` and `restrict` operators (sections 20.19.6, 20.19.6).

Examples:

- The map $\mathbf{R}^2\backslash\{0\} \to \mathbf{R}^3$, $(x,y) \mapsto (x,y,z = x^2+y^2)$ is represented $\{z = x^2+y^2, z \neq 0\}$.
- $\{x = u+v, y = u-v\}$ might represent the coordinate change $\mathbf{R}^3 \to \mathbf{R}^3$, $(u,v,z) \mapsto (x = u+v, y = u-v, z)$.
- $\{x = u+v, y = 2u-x\}$ is the same map again.
- $\{x = 2v+y, y = 2u-x\}$ is unacceptable since x and y cannot be eliminated from the right-hand sides by repeated substitution.

Cobasis transformations

A cobasis transformation is given in EDS by a ⟨*transform*⟩, a list

{ ⟨*cobasis element*⟩ = ⟨*1-form expr*⟩,· · · }

of substitutions. When applying a transformation to a ⟨*p-form*⟩ or ⟨*system*⟩, it is necessary to specify the *forward* transformation just as for a `sub` substitution. For ⟨*EDS*⟩ and ⟨*coframing*⟩ objects, it is also possible to specify the inverse of the desired substition: EDS will automatically invert the transformation as required. For a partial change of cobasis, it is not necessary to include trivial equalities. Cobasis transformations are applied by the `transform` operator (section 20.19.6).

Examples:

- $\{\omega^1 = x\mathrm{d}y - y\mathrm{d}x, \omega^2 = x\mathrm{d}x + y\mathrm{d}y\}$ gives a transformation between Cartesian and polar cobases on $\mathbf{R}^2\backslash\{0\}$.
- On $J^1(\mathbf{R}^2,\mathbf{R})$ with cobasis $\{\mathrm{d}u,\mathrm{d}p,\mathrm{d}q,\mathrm{d}r,\mathrm{d}s,\mathrm{d}t,\mathrm{d}x,\mathrm{d}y\}$, the list $\{\theta^1 = \mathrm{d}u - p\mathrm{d}x - q\mathrm{d}y, \theta^2 = \mathrm{d}p - r\mathrm{d}x - s\mathrm{d}y, \theta^3 = \mathrm{d}q - s\mathrm{d}x - t\mathrm{d}y\}$ specifies a new cobasis in which the contact system is simply $\{\theta^1,\theta^2,\theta^3\}$.

Tableaux

For a quasilinear Pfaffian exterior differential system $(\{\theta^a\},\{\omega^i\},M)$, the tableau $A = [\pi^a_i]$ is a matrix of 1-forms such that

$$\mathrm{d}\theta^a + \pi^a_i \wedge \omega^i \simeq 0 \pmod{\{\theta^a, \omega^i \wedge \omega^j\}}$$

The π_i^a are not unique: if $\{\theta^a, \pi^\rho, \omega^i\}$ is a standard cobasis for the system (section 20.19.3), the EDS $\langle$*tableau*$\rangle$ is a matrix containing linear combinations of the π^ρ only. Zero rows are omitted.

The tableau of an $\langle$*EDS*$\rangle$ is generated by the `tableau` operator (section 20.19.7), or can be entered using the `mat` operator. The Cartan characters of a tableau are found using `characters` (section 20.19.7).

Normal form

Parts of the theory of exterior differential systems apply only at points on the underlying manifold where the system is in some sense non-singular. To ensure the theory applies, EDS automatically works all exterior systems (S, Ω, M) into a *normal form* in which

1. The Pfaffian (degree 1) component of S is in *solved* form, where each expression has a distinguished term with coefficient 1, unique to that expression.
2. The independence condition Ω is also in solved form.
3. The distinguished terms from the 1-forms in S have been eliminated from the rest of S and from Ω.
4. Any 1-forms in S which vanish modulo the independence condition are removed from the system and their coefficients are appended as 0-forms.

Conditions 1 and 2 ensure the 1-forms have constant rank, while 3 is convenient for many tests and calculations. In bringing the system into solved form, divisions will be made only by coefficients which are constants, parameters or functions which are nowhere zero on the manifold. The test for nowhere-zero functions uses the restrictions component of the $\langle$*coframing*$\rangle$ structure (cf section 20.19.2) and is still primitive: facts such as $x^2 + 1 \neq 0$ on a real manifold are overlooked. See also the switch `edssloppy` (section 20.19.11).

This "normal form" has, of course, nothing to do with the various normal forms (eg Goursat) into which some exterior systems may be brought by cobasis transformations and choices of generators.

Examples:

- On $M = \{(u, v, w) \in \mathbf{R}^3 \mid u \neq v\}$, the Pfaffian system
$$\{u\mathrm{d}u + v\mathrm{d}v + \mathrm{d}w,\ (u^2 + u - v^2)\mathrm{d}u + u\mathrm{d}v + \mathrm{d}w\}$$
has the solved form
$$\{\mathrm{d}v + (u + v)\mathrm{d}u,\ \mathrm{d}w + (-uv + u - v)\mathrm{d}u\}.$$

- Since the independence condition is defined only modulo the system, the system
$$S = \{\mathrm{d}u - \mathrm{d}x - u_y \mathrm{d}y\}, \quad \Omega = \mathrm{d}x \wedge \mathrm{d}y$$
has an equivalent normal form
$$S = \{\mathrm{d}x - \mathrm{d}u + u_y \mathrm{d}y\}, \quad \Omega = \mathrm{d}u \wedge \mathrm{d}y.$$

Standard cobasis

Given an ⟨*EDS*⟩ (S, Ω, M) in normal form (section 20.19.3), the cobasis of the ⟨*coframing*⟩ M can be decomposed into three sets: $\{\theta^a\}$, the distinguished terms from the 1-forms in S, $\{\omega^i\}$, the distinguished terms from the 1-forms in Ω, and the remainder $\{\pi^\rho\}$. Within EDS, $\{\theta^a, \pi^\rho, \omega^i\}$ is called the *standard cobasis*, and all expressions are ordered so that $\theta^a > \pi^\rho > \omega^i$. The ordering within the three sets is determined by the REDUCE ⟨*kernel*⟩ ordering.

Examples:

- For the system $S = \{\mathrm{d}u - \mathrm{d}x - u_y \mathrm{d}y\}$, $\Omega = \mathrm{d}x \wedge \mathrm{d}y$, the decomposed standard cobasis is $\{\mathrm{d}u\} \cup \{\mathrm{d}u_y\} \cup \{\mathrm{d}x, \mathrm{d}y\}$.
- For the contact system
$$S = \begin{cases} \mathrm{d}u - u_x \mathrm{d}x - u_y \mathrm{d}y \\ \mathrm{d}u_x - u_{xx} \mathrm{d}x - u_{xy} \mathrm{d}y \\ \mathrm{d}u_y - u_{xy} \mathrm{d}x - u_{yy} \mathrm{d}y \end{cases}$$
the standard cobasis is $\{\mathrm{d}u, \mathrm{d}u_x, \mathrm{d}u_y\} \cup \{\mathrm{d}u_{xx}, \mathrm{d}u_{xy}, \mathrm{d}u_{yy}\} \cup \{\mathrm{d}x, \mathrm{d}y\}$.

20.19.4 Constructing EDS objects

Before analysing an exterior system, it is necessary to enter it into EDS somehow. Several means are provided for this purpose, and are described in this section.

`coframing`

An EDS ⟨*coframing*⟩ is constructed using the `coframing` operator. There are several ways in which it can be used.

The simplest syntax

`coframing({`⟨*expr*⟩`,···})`

examines the argument for 0-form and 1-form variables and deduces a full ⟨*coframing*⟩ object capable of supporting the given expressions. This includes recursively examining the exterior derivatives of the variables appearing explicitly in the argument, taking into account prevailing `let` rules. In this form, the ordering of the final cobasis elements follows the prevailing REDUCE ordering. Free indices in indexed expressions are expanded to a list of explicit indices using `index_expand` (section 20.19.14).

A more basic syntax is

`coframing(`⟨*cobasis*⟩ [`,`⟨*coordinates*⟩] [`,`⟨*restrictions*⟩] [`,`⟨*structure equations*⟩])

where ⟨*cobasis*⟩ is a list of ⟨*kernel*⟩ 1-forms, ⟨*coordinates*⟩ is a list of ⟨*kernel*⟩ 0-forms, ⟨*restrictions*⟩ is a list of inequalities (using only $\neq$ at present), and ⟨*structure equations*⟩ is a list of rules giving the exterior derivatives of the coordinates and cobasis elements. All arguments except the cobasis are optional, and the order of arguments is unimportant. As in the first syntax, missing parts are deduced. The ordering of the final cobasis elements follows the ordering specified, rather than the prevailing REDUCE ordering.

Finally,

`coframing(`⟨*EDS*⟩`)`

returns the coframing argument of an ⟨*EDS*⟩, and

`coframing()`

returns the current background coframing (section 20.19.3).

Examples:

```
coframing {x,y,z};

   coframing({d x,d y,d z},{x,y,z},{},{})

coframing({e 1,e 2},{r},{r neq 0},
          {d r=>e 1,d e 1=>0,d e 2=>e 1^e 2/r});
```

```
            1  2
coframing({e ,e },{r},

                                 1  2
              1          2     e ^e            1
          {d e  => 0,d e  => -------,d r => e },
                                  r

          {r neq 0})

coframing({e 2})
     where {d r=e 1,d e 1=0,d e 2=e 1^e 2/r};

            1  2
coframing({e ,e },{r},

                                 1  2
              1          2     e ^e            1
          {d e  => 0,d e  => -------,d r => e },
                                  r

          {r neq 0})
```

eds

A simple ⟨*EDS*⟩ is constructed using the `eds` operator.

`eds`(⟨*system*⟩,⟨*indep. condition*⟩ [,⟨*coframing*⟩] [,⟨*properties*⟩])

(cf sections 20.19.3, 20.19.2, 20.19.3). The ⟨*indep. condition*⟩ can be either a decomposable ⟨*p-form*⟩ or a ⟨*system*⟩ of 1-forms. Free indices in indexed expressions are expanded to a list of explicit indices using `index_expand` (section 20.19.14).

The ⟨*coframing*⟩ argument can be omitted, in which case the expressions from the ⟨*system*⟩ and ⟨*indep. condition*⟩ are fed to the `coframing` operator (section 20.19.4) to construct a suitable working space.

The ⟨*properties*⟩ argument is optional, allowing the given properties to be asserted. This can save considerable time for large systems, but care is needed since the assertions are never checked.

The ⟨*EDS*⟩ is put into normal form (section 20.19.3) before being returned.

On output, only the ⟨*system*⟩ and ⟨*indep. condition*⟩ are displayed, unless the `nat` switch is off, in which case the ⟨*coframing*⟩ and ⟨*properties*⟩ are shown too. This is so that an ⟨*EDS*⟩ can be written out to a file and read back in.

The parts of an ⟨*EDS*⟩ are obtained with the operators `system`, `cobasis`, `independence` and `properties` (sections 20.19.5, 20.19.5, 20.19.5 and 20.19.5).

Examples:

```
pform {x,y,z,p,q}=0,{e(i),w(i,j)}=1;

indexrange {i,j,k}={1,2},{a,b,c}={3};

eds({d z - p*d x - q*d y, d p^d q},{d x,d y});

   EDS({d z - p*d x - q*d y,d p^d q},{d x,d y})

OMrules :=
   index_expand {d e(i)=>-w(i,-j)^e(j),
                 w(i,-j)+w(j,-i)=>0}$

eds({e(a)},{e(i)}) where OMrules;

         3    1  2
   EDS({e },{e ,e })

coframing ws;
                3  2    1  2          1         2  2
   coframing({e ,w   ,e ,e },{},{d e  =>  - e ^w   ,
                    1                                1
                  2     1  2
                d e  => e ^w   },{})
                              1
```

`contact`

Many PDE problems are formulated as exterior systems using a jet bundle contact system. To facilitate construction of these systems, the `contact` operator is provided. The syntax is

contact(⟨*order*⟩,⟨*source manifold*⟩,⟨*target manifold*⟩)

where ⟨*order*⟩ is a non-negative integer, and the two remaining arguments are either ⟨*coframing*⟩ objects or lists of ⟨*p-form*⟩ expressions. In the latter case, the expressions are fed to the `coframing` operator (section 20.19.4). The contact system for the bundle $J^r(M,N)$ of r-jets of maps $M \to N$ is thus returned by

`contact(r,M,N)`. Source and target spaces may have anholonomic cobases. Indexed names for the jet bundle fibre coordinates are constructed using the identifiers in the source and target cobases.

Examples:

```
pform {x,y,z,u,v}=0,{e i,w a}=1;
indexrange {i}={1,2},{a}=1;
contact(1,{x,y,z},{u,v});

   EDS({d u - u *d x - u *d y - u *d z,
               x         y         z
        d v - v *d x - v *d y - v *d z},{d x,d y,d z})
               x         y         z

OMrules := index_expand{d e(1)=>e(1)^e(2),
                        d e(2)=>0,d w(a)=>0}$

contact(2,{e(i)},{w(a)}) where OMrules;

          1     1     1     1     2
   EDS({w   - w    *e   - w    *e ,
                1           2
             1       1       1     1       2
        d w     - w      *e   - w      *e ,
              1       1 1           1 2
             1           1         1     1     1       2
        d w     + ( - w       + w    )*e   - w      *e },
              2           1 2       1           2 2

          1  2
        {e ,e })
```

pde2eds

A PDE system can be encoded into an ⟨*EDS*⟩ using `pde2eds`. The syntax is

`pde2eds(`⟨*pde*⟩ [,⟨*dependent*⟩,⟨*independent*⟩])

where ⟨*pde*⟩ is a list of equations or expressions (implicitly assumed to vanish) specifying the PDE system using either the standard REDUCE `df` operator, or the EXCALC `@` operator. If the optional variable lists ⟨*dependent*⟩ and ⟨*independent*⟩

are not given, pde2eds infers them from the expressions in ⟨*pde*⟩. The order of each dependent variable is determined automatically.

The result returned by pde2eds is an ⟨*EDS*⟩ based on the contact system of the relevant mixed-order jet bundle. Any of the ⟨*pde*⟩ members which is in solved form is used to pull back this contact system. Any remaining expressions or unresolved equations are simply appended as 0-forms: before many of the analysis tools (section 20.19.7) can be applied, it is necessary to convert this to a system generated in positive degree using the lift operator (section 20.19.6).

The automatic inference of dependent and independent variables is governed by the following rules. The independent variables are all those with respect to which derivatives appear. The dependent variables are those for which explicit derivatives appear, as well as any which have dependencies (as declared by depend or fdomain) or which are 0-forms. To exclude a variable from the dependent variable list (for example, because it is regarded as given) or to include extra independent variables, use the optional arguments to pde2eds.

One of the awkward points about pde2eds is that implicit dependence is changed globally. In order for the df and @ operators to be used to express the PDE, the ⟨*dependent*⟩ variables must depend (via depend or fdomain) on the ⟨*independent*⟩ variables. On the other hand, in the ⟨*EDS*⟩, these variables are all completely independent coordinates. The pde2eds operator thus removes the implicit dependence so that the ⟨*EDS*⟩ is correct upon return. This means that the ⟨*pde*⟩ will no longer evaluate properly until such time as the dependence is manually restored, whereupon the ⟨*EDS*⟩ will no longer be correct, and so on.

To assist with this difficulty, pde2eds saves a record of the dependencies it has removed in the shared variable dependencies. The operator mkdepend can be used to restore the initial state.

See also the operators pde2jet (section 20.19.14) and mkdepend (section 20.19.14).

Example:

```
depend u,x,y; depend v,x,y;
pde2eds({df(u,y,y)=df(v,x),df(v,y)=y*df(v,x)});

   EDS({d u - u *d x - u *d y,
               x         y

        d u  - u   *d x - u   *d y,
           x    x x         y x

        d u  - u   *d x - v *d y,
```

```
            y     y x          x

         d v - v *d x - v *y*d y},d x^d y)
                x          x
```

```
dependencies;

   {{u,y,x},{v,y,x}}
```

set_coframing

The background coframing (section 20.19.3) is set with `set_coframing`. The syntax is

`set_coframing` ⟨*arg*⟩

where ⟨*arg*⟩ is a ⟨*coframing*⟩ or an ⟨*EDS*⟩ and the previous background coframing is returned. All rules, orderings etc pertaining to the previous background coframing are removed and replaced by those for the new ⟨*coframing*⟩. The special form

`set_coframing()`

clears the background coframing entirely and returns the previous one.

20.19.5 Inspecting EDS objects

Given an ⟨*EDS*⟩ or some other EDS structure, it is often desirable to inspect or extract some part of it. The operators described in this section do just that. Many of them accept various types of arguments and return the relevant information in each case.

cobasis

`cobasis` ⟨*arg*⟩

returns the cobasis for ⟨*arg*⟩, which may be either a ⟨*coframing*⟩ or an ⟨*EDS*⟩ (sections 20.19.2, 20.19.3). The order of the items in the list gives the ⟨*kernel*⟩ ordering which applies when the ⟨*coframing*⟩ in ⟨*arg*⟩ is active.

coordinates

`coordinates` ⟨*arg*⟩

returns the coordinates for ⟨*arg*⟩, which may be either a ⟨*coframing*⟩, an ⟨*EDS*⟩, or a list of ⟨*expr*⟩ (sections 20.19.2, 20.19.3). The coordinates in a list of ⟨*expr*⟩ are defined to be those 0-form ⟨*kernels*⟩ with no implicit dependencies.

Examples:

```
coordinates contact(3,{x},{u});

   {x,u,u ,u   ,u     }
         x  x x  x x x

fdomain u=u(x);
coordinates {d u+d y};

   {x,y}
```

structure_equations

`structure_equations` ⟨*arg*⟩

returns the structure equations (cf section 20.19.2) for ⟨*arg*⟩, which may be either a ⟨*coframing*⟩, an ⟨*EDS*⟩, or a ⟨*transform*⟩ (sections 20.19.2, 20.19.3, 20.19.3). In the case of a ⟨*transform*⟩, it is assumed the exterior derivatives of the right-hand sides are known, and a list giving the exterior derivatives of the left-hand sides is returned. This requires inverting the transformation. In case this has already been done, and was time consuming, an alternative syntax

`structure_equations`(⟨*transform*⟩,⟨*inverse transform*⟩)

avoids recomputing the inverse.

Example:

```
structure_equations{e 1=d x/x,e 2=x*d y};

       1          2      1  2
   {d e  => 0,d e  => e ^e }
```

restrictions

`restrictions` ⟨*arg*⟩

returns the restrictions for ⟨*arg*⟩, which may be either a ⟨*coframing*⟩ or an ⟨*EDS*⟩ (sections 20.19.2, 20.19.3). The result is a list of inequalities.

system

`system` ⟨*EDS*⟩

returns the system component of an ⟨*EDS*⟩ (sections 20.19.3, 20.19.3) as a list of ⟨*p-form*⟩ expressions. (The REDUCE command `system` operates as before: the syntax

`system` "⟨*command*⟩"

executes an operating system command.)

independence

`independence` ⟨*EDS*⟩

returns the independence condition of an ⟨*EDS*⟩ (section 20.19.3) as a list of ⟨*1-form*⟩ expressions.

properties

`properties` ⟨*EDS*⟩

returns the currently known properties of an ⟨*EDS*⟩ (sections 20.19.3, 20.19.3) as a list of equations of the form ⟨*keyword*⟩ `=` ⟨*value*⟩.

Example:

```
properties closure contact(1,{x},{u});

   {closed=1,pfaffian=1,quasilinear=1}
```

one_forms

`one_forms` ⟨*arg*⟩

returns the 1-forms in ⟨*arg*⟩, which may be either an ⟨*EDS*⟩ or a list of ⟨*expr*⟩ (sections 20.19.3, 20.19.3).

Example:

```
one_forms {5,x*y - u,d u - x*d y,d u^d x- x*d y^d x};

   {d u - d y*x}
```

zero_forms, nought_forms

`zero_forms` ⟨*arg*⟩

returns the 0-forms in ⟨*arg*⟩, which may be either an ⟨*EDS*⟩ or a list of ⟨*expr*⟩ (sections 20.19.3, 20.19.3). The alternative syntax `nought_forms` does the same thing.

Example:

```
zero_forms {5,x*y - u,d u - x*d y,d u^d x- x*d y^d x};

   {5, - u + x*y}
```

20.19.6 Manipulating EDS objects

The abililty to change coordinates or cobasis, or to modify the system or coframing can make the difference between an intractible problem and a solvable one. The facilities described in this section form the low-level core of EDS functions.

Most of the operators in this section can be applied to both ⟨*EDS*⟩ and ⟨*coframing*⟩ objects. Where it makes sense (eg `pullback`, `restrict` and `transform`), they can be applied to a ⟨*system*⟩, or list of differential forms as well.

augment

`augment`(⟨*EDS*⟩,⟨*system*⟩)

appends the extra forms in the second argument to the system part of the first. If the forms in the ⟨*system*⟩ do not live on the coframing of the ⟨*EDS*⟩, an error results. The original ⟨*EDS*⟩ is unchanged.

Example:

```
% Non-Pfaffian system for a Monge-Ampere equation
S := contact(1,{x,y},{z})$
S := augment(S,{d z(-x)^d z(-y)});

   s := EDS({d z - z *d x - z *d y,
                  x        y

             d z ^d z },{d x,d y})
                x    y
```

`cross`

The infix operator `cross` gives the direct product of ⟨*coframing*⟩ objects. The syntax is

⟨*arg1*⟩ `cross` ⟨*arg2*⟩ [`cross` ···]

The first argument may be either a ⟨*coframing*⟩ (section 20.19.2) or an ⟨*EDS*⟩ (section 20.19.3). The remaining arguments may be either ⟨*coframing*⟩ objects or any valid argument to the `coframing` operator (section 20.19.4), in which case the corresponding ⟨*coframing*⟩ is automatically inferred. The arguments may not contain any common coordinates or cobasis elements.

If the first argument is an ⟨*EDS*⟩, the result is the ⟨*EDS*⟩ lifted to the direct product space. In this way, it is possible to execute a pullback under a projection.

Example:

```
coordinates(contact(1,{x,y},{u}) cross {v});

   {x,y,u,u ,u ,v}
            x  y
```

pullback

Pullbacks with respect to an EDS ⟨*map*⟩ (section 20.19.3) have the syntax

```
pullback(⟨arg⟩,⟨map⟩)
```

where ⟨*arg*⟩ can be any one of ⟨*EDS*⟩, ⟨*coframing*⟩, ⟨*system*⟩ or ⟨*p-form*⟩ expression (sections 20.19.3, 20.19.2, 20.19.3). The result is of the same type as ⟨*arg*⟩.

For an ⟨*EDS*⟩ or ⟨*coframing*⟩ with anholonomic cobasis, `pullback` calculates the pullbacks of the cobasis elements and chooses a cobasis for the source coframing itself. For a ⟨*system*⟩, any zeroes in the result are dropped from the list.

Examples:

```
pullback(contact(1,{x,y},{u}),{u(-y) = u*u(-x)});

   EDS({d u - u *d x - u *u*d y},{d x,d y})
               x         x

M := coframing({e 1,e 2},{r},{r neq 0},
               {d r=>e 1,d e 1=>0,d e 2=>e 1^e 2/r})$

pullback(M,{r=1/x});
                                      2
               2                2     e ^d x
   coframing({e ,d x},{x},{d e   => --------},{x neq 0})
                                       x

pullback(ws,{x=0});

 ***** Map image not within target coframing in pullback

pullback({y*d y,d y - d x},{y=x});

   {d x*x}
```

restrict

Restrictions with respect to an EDS ⟨*map*⟩ (section 20.19.3) have the syntax

```
restrict(⟨arg⟩,⟨map⟩)
```

where ⟨*arg*⟩ can be any one of ⟨*EDS*⟩, ⟨*coframing*⟩, ⟨*system*⟩ or ⟨*p-form*⟩ expres-

sion (sections 20.19.3, 20.19.2, 20.19.3). The result is of the same type as ⟨*arg*⟩. The action of `restrict` is similar to that of `pullback`, except that only scalar coefficients are affected: 1-form variables are unchanged.

Examples:

```
% Bring a system into normal form
%  by restricting the coframing

S := eds({u*d v - v*d u},{d x});

   s := EDS({v*d u - u*d v},{d x})

restrict(S,{u neq 0});

                 v
   EDS({d v - ---*d u},{d x})
                 u

% Difference between restrict and pullback

pullback({x*d x - y*d y},{x=y,y=1});

   {}

restrict({x*d x - y*d y},{x=y,y=1});

   {d x - d y}
```

`transform`

A change of cobasis is made using the `transform` operator

`transform`(⟨*arg*⟩,⟨*transform*⟩)

where ⟨*arg*⟩ can be any one of ⟨*EDS*⟩, ⟨*coframing*⟩, ⟨*system*⟩ or ⟨*p-form*⟩ expression (sections 20.19.3, 20.19.2, 20.19.3) and ⟨*transform*⟩ is a list of substitutions (cf section 20.19.3). The result is of the same type as ⟨*arg*⟩.

For an ⟨*EDS*⟩ or ⟨*coframing*⟩, `transform` can detect whether the tranformation is given in the forward or reverse direction and invert accordingly. Structure equations are updated correctly. If an exact cobasis element is eliminated, its expression in terms of the new cobasis is added to the list of structure equations, since the corresponding coordinate may still be present elsewhere in the structure.

Example:

```
S := contact(1,{x},{u});

   s := EDS({d u - u *d x},{d x})
                     x

new := {e(1) = first system S,w(1) = d x};

             1                  1
   new := {e =d u - d x*u ,w =d x}
                           x

S := transform(S,new);

                1    1
   s := EDS({e },{w })

structure_equations s;

       1              1
   {d e  =>  - d u ^w ,
                  x
       1
    d w  => 0,

             1        1
    d u => e  + u *w ,
                 x
             1
    d x => w }
```

`lift`

Many of the analysis tools (section 20.19.7) cannot treat systems containing 0-forms. The `lift` operator

`lift` ⟨*EDS*⟩

solves the 0-forms in the system and uses the solution to pull back to a smaller manifold. This may generate new 0-form conditions (in the course of bringing the pulled-back system into normal form), in which case the process is repeated until the system is generated in positive degree. In non-linear problems, the solution

space of the 0-forms may be a variety, in which case a compound ⟨*EDS*⟩ (section 20.19.3) will result. If `edsverbose` is on (section 20.19.9), the solutions are displayed as they are generated.

Example:

```
S := augment(contact(2,{x,y},{u}),{u(-y,-y)-u(-x,-x)})$
on edsverbose;
lift S;
   Solving 0-forms
   New equations:
   u   =u
    y y  x x

   EDS({d u - u *d x - u *d y,
               x        y

        d u  - u   *d x - u   *d y,
           x    x x         x y

        d u  - u   *d x - u   *d y},{d x,d y})
           y    x y         x x
```

20.19.7 Analysing exterior systems

This section describes higher level operators for extracting information about exterior systems. Many of them require a ⟨*EDS*⟩ in normal form (section 20.19.3) generated in positive degree as input, but some can also analyse a ⟨*system*⟩ (section 20.19.3) or a single ⟨*p-form*⟩. Only trivial examples are provided in this section, but many of these operators are used in the longer examples in the test file which accompanies this package.

`cartan_system`

The *Cartan system* of a form or system S is the smallest Pfaffian system C such that $\Lambda(C)$ contains a set I of forms algebraically equivalent to S. The Cartan system is also known as the *associated Pfaff system* or *retracting space*. An alternative characterisation is to note that the annihilator $C^{\perp}$ comprises all vectors V satisfying $i_V S \simeq 0 \pmod{S}$. Note this is a purely algebraic concept: S need not be closed under differentiation. See also `cauchy_system` (section 20.19.7).

The operator

`cartan_system` ⟨*arg*⟩

returns the Cartan system of ⟨*arg*⟩, which may be an ⟨*EDS*⟩, a ⟨*system*⟩ or a single ⟨*p-form*⟩ expression (sections 20.19.3, 20.19.3). For an ⟨*EDS*⟩, the result is a Pfaffian ⟨*EDS*⟩ on the same manifold, otherwise it is a ⟨*system*⟩. The argument must be generated in positive degree.

Example:

```
cartan_system{d u^d v + d v^d w + d x^d y};

   {d u - d w,d v,d x,d y}
```

`cauchy_system`

The *Cauchy system* C of a form or system S is the Cartan system or retracting space of its closure under exterior differentiation (section 20.19.7). The annihilator $C^{\perp}$ consists of the Cauchy vectors for the S.

The operator

`cauchy_system` ⟨*arg*⟩

returns the Cauchy system of ⟨*arg*⟩, which may be an ⟨*EDS*⟩, a ⟨*system*⟩ or a single ⟨*p-form*⟩ expression (sections 20.19.3, 20.19.3). For an ⟨*EDS*⟩, the result is a Pfaffian ⟨*EDS*⟩ on the same manifold, otherwise it is a ⟨*system*⟩. The argument must be generated in positive degree.

Example:

```
cauchy_system{u*d v + v*d w + x*d y};

   {d u,d v,d w,d x,d y}
```

characters

The Cartan characters $\{s_1, ..., s_n\}$ of an ⟨*EDS*⟩ or ⟨*tableau*⟩ (sections 20.19.3, 20.19.3) are obtained with

`characters` ⟨*EDS*⟩
or
`characters` ⟨*tableau*⟩

The zeroth character s_0 is not returned, it is simply the number of 1-forms in the ⟨*EDS*⟩ (cf `one_forms`, section 20.19.5). The definition used for the last character: $s_n = (d - n) - (s_0 + s_1 + ... + s_{n-1})$, where d is the manifold dimension, allows Cartan's test to be used even when Cauchy characteristics are present.

For a nonlinear ⟨*EDS*⟩, the Cartan characters can vary from stratum to stratum of the Grassmann bundle variety of ordinary integral elements (cf. the operator `grassmann_variety` in section 20.19.7). Nonetheless, they are constant on each stratum, so it suffices to calculate them at one point (ie at one integral element). This is done using the syntax

`characters`(⟨*EDS*⟩,⟨*integral element*⟩)

where ⟨*integral element*⟩ is a list of 1-forms (cf section 20.19.3).

The Cartan characters are calculated from the reduced characters for a fixed flag of integral elements based on the 1-forms in the independence condition of an ⟨*EDS*⟩. This can lead to incorrect results if the flag is somehow singular, so two switches are provided to overcome this (section 20.19.13). First, `genpos` looks at a generic flag by using a general linear transformation to put the system in *general position*. This guarantees correct results, but can be too slow for practical purposes. Secondly, `ranpos` performs a linear transformation using a sparse matrix of random integers. In most cases, this is much faster than using general position, and a few repetitions give some confidence in the results.

Example:

```
S := pullback(contact(2,{x,y},{u}),{u(-x,-y)=0});

   s := EDS({d u - u *d x - u *d y,
                    x         y

             d u   - u    *d x,
                x     x x
```

```
              d u  - u   *d y},{d x,d y})
                 y    y y

characters S;

   {1,1}

on ranpos; characters S;

   {2,0}
```

fclosure

`closure` ⟨*EDS*⟩

returns the closure of the ⟨*EDS*⟩ under exterior differentiation.

Owing to conflicts with the requirements of a normal form (section 20.19.3), `closure` cannot guarantee that the resulting system is closed if the ⟨*EDS*⟩ contains 0-forms.

derived_system

`derived_system` ⟨*arg*⟩

returns the first derived system of ⟨*arg*⟩, which must be a Pfaffian ⟨*EDS*⟩ or ⟨*system*⟩. Repeated use gives the derived flag leading to the maximal integrable subsystem.

Example:

```
pform {p,r,x,y,z}=0; korder z;
derived_system eds({d z - q*d y,d p - e**z*d y,
                    d r - e**z*p*d y,d x},{d y});

                z            z
   EDS({d p - e *d y,d r - e *p*d y,d x},{d y})

derived_system ws;

                1
   EDS({d p - ---*d r,d x},{d y})
```

```
                p

derived_system ws;

                1
   EDS({d p - ---*d r,d x},{d y})
                p
```

fdim_grassmann_variety

`dim_grassmann_variety` ⟨*EDS*⟩

returns the dimension of the Grassmann bundle variety of ordinary integral elements for an ⟨*EDS*⟩ (cf `grassmann_variety`, section 20.19.7). This number is useful, for example, in Cartan's test. For a nonlinear ⟨*EDS*⟩, this can vary from stratum to stratum of the variety, so

dim_grassmann_variety(⟨*EDS*⟩,⟨*integral element*⟩)

returns the dimension of the stratum containing the ⟨*integral element*⟩ (cf section 20.19.3).

dim

`dim` ⟨*arg*⟩

returns the dimension of the manifold underlying ⟨*arg*⟩, which can be either an ⟨*EDS*⟩ or a ⟨*coframing*⟩ (sections 20.19.3, 20.19.2).

involution

`involution` ⟨*EDS*⟩

repeatedly prolongs an ⟨*EDS*⟩ until it reaches involution or inconsistency (cf `prolong`, section 20.19.7). The system must be in normal form (section 20.19.3) and generated in positive degree. For nonlinear problems, all branches of the prolongation tree are followed. The result is an ⟨*EDS*⟩ (usually a compound one for nonlinear problems, see section 20.19.3) giving the involutive prolongation. In case some variety couldn't be resolved during the process, the relevant branch is truncated at that point and represented by a system with 0-forms, as with

`grassmann_variety` (section 20.19.7). The result of `involution` might then *not* be in involution.

If the `edsverbose` switch is on (section 20.19.9), a trace of the prolongations is produced. See the Janet problem in the test file for an example.

`linearise`, `linearize`

A nonlinear exterior system can be linearised at some point on the manifold with respect to any integral element, yielding a constant coefficient exterior system with the same Cartan characters. In EDS, reference to the point is omitted, so the result is an exterior system linearised with respect to a distribution of integral elements. The syntax is

`linearise`(⟨*EDS*⟩,⟨*integral element*⟩)

but `linearize` will work just as well. See the isometric embeddings example in the test file.

For a quasilinear ⟨*EDS*⟩ (cf. section 20.19.8),

`linearise` ⟨*EDS*⟩

returns an equivalent exterior system containing only linear generators.

Example:

```
f := d u^d x + d v^d y$
S := eds({f,d v^f},{d x,d y});

   s := EDS({d u^d x + d v^d y,d u^d v^d x},{d x,d y})

linearise S;

   EDS({d u^d x + d v^d y},{d x,d y})
```

`integral_element`

`integral_element` ⟨*EDS*⟩

returns a random ⟨*integral element*⟩ of the ⟨*EDS*⟩ (section 20.19.3). The system must be in normal form (section 20.19.3) and generated in positive degree. This

integral element is found using the method described by Wahlquist [Wah93] (essentially the Cartan-Kähler construction filling in the free variables from each polar system with random integer values). This method can fail on non-involutive systems, or ⟨*EDS*⟩ objects whose independence conditions indicate a singular flag of integral elements (cf the discussion about Cartan characters, section 20.19.7).

See the isometric embedding problem in the test file for an example.

prolong

`prolong` ⟨*EDS*⟩

calculates the prolongation of the ⟨*EDS*⟩ to the Grassmann bundle variety of integral elements. The system must be in normal form (section 20.19.3) and generated in positive degree. The variety is decomposed using essentially the REDUCE `solve` operator. If no solutions can be found, the variety is empty, and the prolongation is the inconsistent ⟨*EDS*⟩ (section 20.19.3). Otherwise, the result is a list of variety components, which fall into three classes:

1. a submanifold of the Grassmann bundle which surjects onto the base manifold. The result in this case is the pullback of the Grassmann bundle contact ⟨*EDS*⟩ to this submanifold.

2. a submanifold of the Grassmann bundle which does not surject onto the base manifold (ie cannot be presented by solving for Grassmann bundle fibre coordinates). The result in this case is the pullback of the original ⟨*EDS*⟩ to the projection onto the base manifold. If 0-forms arise in bringing the pullback to normal form, these are solved recursively and the system pulled back again until the result is generated in positive degree (cf `lift`, section 20.19.6).

3. a component of the variety which `solve` was not able to resolve explicitly. The result in this case is the Grassmann bundle contact ⟨*EDS*⟩ augmented with the 0-forms which `solve` couldn't treat. This can be extracted from the result of `prolong` and manipulated further "by hand",

The result returned by `prolong` will, in general, be a compound ⟨*EDS*⟩ (section 20.19.3). If the switch `edsverbose` (section 20.19.9) is on, a trace of the prolongation is printed.

The ⟨*map*⟩s which are generated in a `prolong` call are available subsequently in the global variable `pullback_maps`. This facility is still very primitive and unstructured. It should be extended to the `involution` operator as well...

Example:

```
pde := {u(-y,-y)=u(-x,-x)**2/2,u(-x,-y)=u(-x,-x)};

                        2
                  (u   )
                    x x
   pde := {u   =---------,u   =u   }
           y y     2      x y  x x

S := pullback(contact(2,{x,y},{u}),pde)$
on edsverbose;
prolong S;
   Reduction using new equations:
   u   =1
    x x
   Prolongation using new equations:
   u     =0
    x x x
   u     =0
    x x y

   {EDS({d u - u *d x - u *d y,
               x        y

         d u  - d x - d y,
            x
                        1
         d u  - d x - ---*d y},{d x,d y}),
            y           2

    EDS({d u - u *d x - u *d y,
               x        y

         d u  - u   *d x - u   *d y,
            x    x x        x x
                                  2
                           (u   )
                             x x
         d u  - u   *d x - ---------*d y,
            y    x x           2

         d u   },{d x,d y})}
            x x
```

tableau

`tableau` ⟨*EDS*⟩

returns the ⟨*tableau*⟩ (section 20.19.3) of a quasilinear Pfaffian ⟨*EDS*⟩, which must be in normal form and generated in positive degree.

Example:

```
tableau contact(2,{x,y},{u});

   [d u     d u   ]
   [   x x     x y]
   [               ]
   [d u     d u   ]
   [   x y     y y]
```

torsion

For a semilinear Pfaffian exterior differential system, the torsion corresponds to first-order integrability conditions for the system. Specifically,

`torsion` ⟨*EDS*⟩

returns a list of 0-forms describing the projection of the Grassmann bundle variety of integral elements onto the base manifold. If the switch `edssloppy` (section 20.19.11) is on, quasilinear systems are treated as semilinear. A semilinear system is involutive if both the torsion is empty, and Cartan's test for the reduced characters is satisfied.

Example:

```
S := pullback(contact(2,{x,y},{u}),
              {u(-y,-y)=u(-x),u(-x,-y)=u});

   s := EDS({d u - u *d x - u *d y,
                  x        y

             d u  - u   *d x - u*d y,
                x    x x

             d u  - u*d x - u *d y},{d x,d y})
```

```
                y             x
torsion s;

   {u     - u }
     x x     y
```

grassmann_variety

Given an exterior system (S, Ω, M) with independence condition of rank n, the Grassmann bundle of n-planes over M contains a submanifold characterised by those n-planes compatible with the independence condition. All integral elements must lie in this submanifold. The operator

`grassmann_variety` ⟨*EDS*⟩

returns the contact system for this part of the Grassmann bundle augmented by the 0-forms specifying the variety of integral elements of S. In cases where `prolong` (section 20.19.7) is unable to decompose the variety automatically, `grassmann_variety` can be used in combination with `zero_forms` (section 20.19.5) to calculate the variety conditions. Any solutions found "by hand" can be incorporated using `pullback` (section 20.19.6).

Example: Using the system from the example in section 20.19.7:

```
zero_forms grassmann_variety S;

   { - u      *u     + u      ,
        x x x  x x     x x y

     - u       + u      }
        x x x     x x y

solve ws;

   Unknowns: {u      ,u      ,u    }
               x x x  x x y  x x

   {{u      =0,u      =0},
      x x y     x x x

    {u    =1,u      =u      }}
      x x     x x x  x x y
```

The second solution contains an integrability condition.

20.19.8 Testing exterior systems

The operators in this section allow various properties of an ⟨*EDS*⟩ to be checked. These checks are done automatically when required on entry to the routines in sections 20.19.6 and 20.19.7, but sometimes it is useful to know explicitly. The result is either a `1` (true) or a `0` (false), so the operators can be used in boolean expressions within `if` statements etc. Since checking these properties can be very time-consuming, the result of the first test is stored on the ⟨*properties*⟩ record of an ⟨*EDS*⟩ to avoid re-checking. This memory can be cleared using the `cleanup` operator.

`closed`

`closed` ⟨*arg*⟩

checks whether ⟨*arg*⟩, which may be an ⟨*EDS*⟩, a ⟨*system*⟩ or a single ⟨*p-form*⟩ is closed under exterior differentiation.

Examples:

```
closed(x*d x);

   1

closed {d u - p*d x,d p - p/y*d x};

   0
```

`involutive`

`involutive` ⟨*EDS*⟩

checks whether ⟨*EDS*⟩ is involutive, using Cartan's test. See the test file for examples.

`pfaffian`

`pfaffian` ⟨*EDS*⟩

checks whether ⟨*EDS*⟩ is a Pfaffian system: generated by a set of 1-forms and their exterior derivatives. The ⟨*EDS*⟩ must be in normal form (section 20.19.3) for this to succeed. Systems with 0-forms are non-Pfaffian by definition in EDS.

Examples:

```
pfaffian eds({d u - p*d x - q*d y,d p^d x+d q^d y},
             {d x,d y});

   1

pfaffian eds({d u - p*d x - q*d y,d p^d q},{d x,d y});

   0
```

quasilinear

An exterior system (S, Ω, M) is said to be *quasilinear* if, when written in the standard cobasis $\{\theta^a, \pi^\rho, \omega^i\}$ (section 20.19.3), its *closure* can be generated by a set of forms which are of combined total degree 1 in $\{\theta^a, \pi^\rho\}$. The operation

`quasilinear` ⟨*EDS*⟩

checks whether the *closure* of ⟨*EDS*⟩ is a quasilinear system. The ⟨*EDS*⟩ must be in normal form (section 20.19.3) for this to succeed. Systems with 0-forms are not quasilinear by definition in EDS.

Examples:

```
% A system where pi(rho)={d p,d q},
%  and which looks non-linear

S := eds({d u - p*d x - q*d y,d p^d q^d y},{d x,d y})$

quasilinear S;

   1

linearise closure S;

   EDS({d u - p*d x - q*d y,
         - d p^d x - d q^d y},{d x,d y})
```

```
% One which is really non-linear

quasilinear
     eds({d u - p*d x - q*d y,d p^d q},{d x,d y});

   0
```

semilinear

Let (S, Ω, M) be such that, written in the standard cobasis $\{\theta^a, \pi^\rho, \omega^i\}$ (section 20.19.3), its closure is explicitly quasilinear. If the coefficients of $\{\pi^\rho\}$ depend only on the independent variables, then the system is said to be *semilinear*. The operation

`semilinear` ⟨*EDS*⟩

checks whether *closure* of ⟨*EDS*⟩ is a semilinear system. The ⟨*EDS*⟩ must be in normal form (section 20.19.3) for this to succeed. Systems with 0-forms are not semilinear by definition in EDS.

For semilinear systems, the expressions determining the Grassmann bundle variety of integral elements will be linear in the Grassmann bundle fibre coordinates, with coefficients which depend only upon the independent variables. This allows alternative, faster algorithms to be used in analysis.

If the switch `edssloppy` is on (section 20.19.11), all quasilinear systems are treated as if they are semilinear.

Examples:

```
% A semilinear system: @(u,y) = y*@(u,x)
S := eds({d u - p*d x - p*y*d y},{d x,d y})$
semilinear S;

   1
% A quasilinear system: @(u,y) = u*@(u,x)
S := eds({d u - p*d x - p*u*d y},{d x,d y})$
quasilinear S;

   1
semilinear S;

   0
```

```
on edssloppy;
semilinear S;

   1
```

frobenius

`frobenius` ⟨*arg*⟩

checks whether ⟨*arg*⟩, which may be an ⟨*EDS*⟩ or a ⟨*system*⟩, is a completely integrable Pfaffian system.

Examples:

```
if frobenius eds({d u -p*(d x+d y)},d x^d y)
  then yes else no;

   no

if frobenius eds({d u -u*(d x+d y)},d x^d y)
  then yes else no;

   yes
```

equiv

⟨*EDS1*⟩ `equiv` ⟨*EDS2*⟩

checks whether ⟨*EDS1*⟩ and ⟨*EDS2*⟩ are algebraically equivalent as exterior systems (ie generate the same algebraic ideal).

Examples:

```
S1 := contact(2,{x,y},{u})$
S2 := augment(S1,foreach f in system S1
                        join {d f,d x^d f})$
if S1 equiv S2 then yes else no;

   no

if closure S1 equiv S2 then yes else no;
```

```
yes
```

20.19.9 Switches

EDS provides several switches to govern the display of information and speed or reliability of the results.

edsverbose

If `edsverbose` is `on`, a number of operators (eg `prolong`, `involution`) will display additional information as the calculation progresses. For large problems, this can produce too much output to be useful, so `edsverbose` is `off` by default. This allows only warning (`***`) and error (`*****`) messages to be printed.

20.19.10 edsdebug

If `edsdebug` is `on`, EDS produces copious quantities of information, in addition to that produced with `edsverbose` on. This information is for debugging purposes, and may not make much sense without knowledge of the inner workings of EDS. `edsdebug` is `off` by default.

20.19.11 edssloppy

Normally, EDS will not divide by any expressions it does not know to be nowhere zero. If an ⟨*EDS*⟩ can be brought into normal form only by restricting away from the zeroes of some coefficients, then these restrictions should be made using the `restrict` operator (section 20.19.6). However, if `edssloppy` is `on`, then EDS will, as a last resort, divide by whatever is necessary to bring an ⟨*EDS*⟩ into normal form, invert a transformation, and so on. The relevant restrictions will be made automatically, so no inconsistency should arise. In addition, with `edssloppy` `on`, all quasilinear systems are treated as if they were semilinear (cf section 20.19.8). `edssloppy` is `off` by default.

20.19.12 edsdisjoint

When decomposing a variety into (something like) smooth components, EDS normally pays no attention to whether the components are disjoint. Turning `on` the switch `edsdisjoint` forces EDS to ensure the decomposition is a disjoint union (cf `disjoin`, section 20.19.14). For large problems this can lead to a proliferation of singular pieces. If some of these turn out to be uninteresting, EDS cannot

re-join the remaining pieces into a smaller decomposition. `edsdisjoint` is `off` by default.

20.19.13 `ranpos`, `genpos`

When calculating Cartan characters (eg to check involution), EDS uses the independence condition of an ⟨*EDS*⟩ *as presented* to define a flag of integral elements. Depending on the cobasis and ordering, this flag may be singular, leading to incorrect Cartan characters. To overcome this problem, the switches `ranpos` and `genpos` provide a means to select other flags. With `ranpos on`, a flag defined by taking a random linear transformation of the 1-forms in the independence condition will be used. The results may still be incorrect, but the likelihood is much lower. With `genpos` on, a generic (upper triangular) transformation is used. this guarantees the correct Cartan characters, but reduces performance too much to be useful for large problems. Both switches are `off` by default, and switching one `on` automatically switches the other `off`. See section 20.19.7 for an example.

20.19.14 Auxiliary functions

This section describes various operators designed to ease working with exterior forms and exterior systems in REDUCE.

`invert`

`invert` ⟨*transform*⟩

returns a ⟨*transform*⟩ which is inverse to the given one (section 20.19.6). If the ⟨*transform*⟩ given is only partial, the 1-form ⟨*kernel*⟩s to eliminate are chosen based on the prevailing kernel ordering. If a background coframing (section 20.19.3) is active, and `edssloppy` (section 20.19.11) is `off`, `invert` will divide by nowhere-zero expressions only.

Examples:

```
set_coframing coframing{u,v,w,x,y,z}$
invert {d u = 3*d x - d y + 5*d z, d v = d x + 2*d z};

   {d x=d v - 2*d z,d y= - d u + 3*d v - d z}

% A y coefficient forces a different choice of inverse
```

```
invert {d u = 3*d x - y*d y + 5*d z, d v = d x + 2*d z};

   {d x=2*d u - 5*d v + 2*d y*y,

    d z= - d u + 3*d v - d y*y}
```

linear_divisors

linear_divisors ⟨*pform*⟩

returns a basis for the space of linear divisors (1-form factors) of a ⟨*p-form*⟩.

Example:

```
f := d p^d q^d u - d p^d q^d x*x + d p^d q^d z*y
     - d u^d v^d x*x + d u^d v^d z*y + d u^d x^d y
     + d x^d y^d z*y$
linear_divisors f;

   {d u - d x*x + d z*y}
```

exfactors

exfactors ⟨*pform*⟩

returns a list of factors for a ⟨*p-form*⟩, consisting of the linear divisors plus one more factor. The list is ordered such that the original expression is a product of the factors in this order.

Example:

```
f := d p^d q^d u - d p^d q^d x*x + d p^d q^d z*y
     - d u^d v^d x*x + d u^d v^d z*y + d u^d x^d y
     + d x^d y^d z*y$
exfactors f;

   {d p^d q - d v^d x*x + d v^d z*y + d x^d y,
    d u - d x*x + d z*y}

f - (part(ws,0) := ^);

   0
```

index_expand

EXCALC caters for indexed variables in which various index names have been assigned a specific set of values. Any expression with *paired* indices is expanded automatically to an explicit sum over the index set (provided the EXCALC command `nosum` has not been applied). The EDS operator `index_expand` is designed to expand an expression with *free* indices to an explicit list over the index set, taking some limited account of the possible index symmetries.

The syntax is

`index_expand` ⟨*arg*⟩

where ⟨*arg*⟩ can be an expression, a rule or equation or a boolean expression, or an arbitrarily nested list of these items. The result is a flattened list.

Examples:

```
indexrange {i,j,k}={1,2,3},{a,b}={x,y};
pform {e(i),o(a,b)}=1;
index_expand(e(i)^e(j));

     1  2  1  3  2  3
   {e ^e ,e ^e ,e ^e }

index_expand{o(-a,-b)+o(-b,-a) => 0};

   {2*o    => 0,o    + o    => 0, 2*o    => 0}
       x x       x y    y x          y y
```

pde2jet

A PDE system can be encoded into EDS jet variable notation using `pde2jet`. The syntax is as for `pde2eds`:

`pde2jet`(⟨*pde*⟩ [,⟨*dependent*⟩,⟨*independent*⟩])

where ⟨*pde*⟩ is a list of equations or expressions (implicitly assumed to vanish) specifying the PDE system using either the standard REDUCE `df` operator, or the EXCALC `@` operator. If the optional variable lists ⟨*dependent*⟩ and ⟨*independent*⟩ are not given, `pde2jet` infers them from the expressions in ⟨*pde*⟩, using the same rules as `pde2eds` (section 20.19.4).

The result of `pde2jet` is the input ⟨*pde*⟩, with all derivatives of dependent variables replaced by indexed 0-form variables from the appropriate jet bundle. Unlike `pde2eds`, `pde2jet` does not disturb the variable dependencies.

Example:

```
depend u,x,y; depend v,x,y;
pde2jet({df(u,y,y)=df(v,x),df(v,y)=y*df(v,x)});

   {u   =v ,
     y y  x

    v =v *y}
     y  x
```

`mkdepend`

The `mkdepend` operator is intended for restoring the dependencies destroyed by a call to `pde2eds` (section 20.19.4). The syntax is

`mkdepend {`⟨*list of variables*⟩`,⋯ }`

where the first variable in each list is declared to depend on the remaining ones.

`disjoin`

The `disjoin` operator takes a list of ⟨*maps*⟩ (section 20.19.3) describing a decomposition of a variety, and returns an equivalent list of ⟨*maps*⟩ such that the components are all disjoint. The background coframing (section 20.19.3) should be set appropriately before calling `disjoin`. The syntax is

`disjoin {`⟨*map*⟩`,⋯ }`

Example:

```
set_coframing coframing {x,y};
disjoin {{x=0},{y=0}};

   {{y=0,x neq 0},{x=0,y neq 0},{y=0,x=0}}
```

cleanup

To avoid lengthy recomputations, EDS stores various properties (section 20.19.3) and also many intermediate results in a hidden list attached to each ⟨*EDS*⟩. When EDS detects a change in circumstances which could make the information innacurate, it is discarded and recomputed. Unfortunately, this mechanism is not perfect, and occasionally misses something which renders the results incorrect. In such a case, it is possible to discard all the properties and hidden information using the `cleanup` operator. The call

`cleanup` ⟨*arg*⟩

returns a copy of ⟨*arg*⟩, which may be a ⟨*coframing*⟩ or an ⟨*EDS*⟩ which has been stripped of this auxilliary information. Note that the original input (with possible innacuracies) is left undisturbed by this operation: the result of `cleanup` must be used.

Example:

```
% An erroneous property assertion
S := eds({d u - p*d x},{d x,d y},{closed = 1})$
closure S;

   EDS({d u - p*d x},{d x,d y});

S := cleanup S$
properties S;

   {}

closure S;

   EDS({d u - p*d x, - d p^d x},{d x,d y});
```

reorder

All operations with a ⟨*coframing*⟩ or ⟨*EDS*⟩ temporarily override the prevailing kernel order with their own. Thus the ordering of the cobasis elements in a ⟨*coframing*⟩ operator remains fixed, even when a REDUCE `korder` statement is issued. To enforce conformity to the prevailing kernel order, the `reorder` operator is available. The call

`reorder` ⟨*arg*⟩

returns a copy of ⟨*arg*⟩, which may be a ⟨*coframing*⟩ or an ⟨*EDS*⟩ which has been reordered. Note that the original input is left undisturbed by this operation: the result of `reorder` must be used.

Example:

```
M := coframing {x,y,z};

   m := coframing({d x,d y,d z},{x,y,z},{},{})

korder z,y,x;
reorder m;

   coframing({d z,d y,d x},{z,y,x},{},{})
```

20.19.15 Experimental facilities

This section describes various operators in EDS which either not algorithmically well-founded, or whose implementation is very unstable, or which have known bugs.

poincare

The `poincare` operator implements the homotopy integral found in the proof of Poincaré's lemma. The expansion point is the origin of the coordinates found in the input. The syntax is

`poincare` ⟨*p-form*⟩

If `f` is a p-form, then `poincare f` is a $(p-1)$-form, and `f - poincare d f` is an exact p-form.

Examples:

```
poincare(3*d x^d y^d z);

   d x^d y*z - d x^d z*y + d y^d z*x

d ws;

   3*d x^d y^d z

2*x*d y - poincare d(2*x*d y);

   d x*y + d y*x
```

invariants

The `invariants` operator implements the algorithm implicit in the inductive proof of the Frobenius theorem. The syntax is

`invariants`(⟨*system*⟩ [,⟨*list of coordinate*⟩])

where ⟨*system*⟩ is a set of 1-forms satisfying the Frobenius condition. The optional second argument specifies the order in which the coordinates are projected away to get a trivially integrable system. The CRACK and ODESOLVE packages are used to solve the ODE systems which arise, so the limitations of these packages constrain the scope of this operator as well.

Examples:

```
invariants {d x*y + d y*x*z + d z*log(y)*x*y};

         z
   { - y *x}

invariants {d y*z**2 - d y*z + d z*y,
            d x*(1 - z) + d z*x};

        x      y*(z - 1)
   {-------,-----------}
     z - 1        z
```

symbol_relations

The `symbol_relations` operator finds the linear relations between the entries of the tableau matrix for a quasilinear system. The syntax is

`symbol_relations(`⟨*EDS*⟩`,` ⟨*identifier*⟩`)`

where ⟨*EDS*⟩ is a quasilinear Pfaffian system and ⟨*identifier*⟩ is used to create a 2-indexed 1-form which will label the tableau entries.

Example:

```
S := pde2eds {df(u,y,y) = df(u,x,x)};

   s := EDS({d u - u *d x - u *d y,
                  x        y

             d u  - u   *d x - u   *d y,
                x    x x        x y

             d u  - u   *d x - u   *d y},d x^d y)
                y    x y        x x

symbol_relations(S,pi);

      1       2
   {pi    - pi   ,
        x       y
      1       2
    pi    - pi   }
        y       x
```

symbol_matrix

The `symbol_matrix` operator returns the symbol matrix for a quasilinear system in terms of a given variable. The syntax is

`symbol_matrix(`⟨*EDS*⟩`,` ⟨*identifier*⟩`)`

where ⟨*EDS*⟩ is a quasilinear Pfaffian system and ⟨*identifier*⟩ is used to create an indexed 0-form which will parameterise the matrix.

Example:

```
% With the same system as for symbol_relations:

symbol_matrix(S,xi);

   [xi    - xi ]
   [  x       y]
   [            ]
   [xi    - xi ]
   [  y       x]
```

`characteristic_variety`

The `characteristic_variety` operator returns the equations specifying the characteristic variety for a quasilinear system in terms of a given variable. The syntax is

`characteristic_variety`(⟨*EDS*⟩, ⟨*identifier*⟩)

where ⟨*EDS*⟩ is a quasilinear Pfaffian system and ⟨*identifier*⟩ is used to create an indexed 0-form variable. The result is a list of two lists: the first being the variety equations and the second the variables involved.

Example:

```
% With the same system as for symbol_relations:

characteristic_variety(S,xi);

          2         2
   {{(xi )   - (xi ) },
        x         y
    {xi ,xi }}
       x   y
```

20.19.16 Command tables

The tables in this appendix summarise the commands available in EDS. More detailed descriptions of the syntax and function of each command are to be found in the earlier sections. In each case, examples of the command are given, whereby the argument variables have the following types (see section 20.19.2):

E, E'	⟨*EDS*⟩
S	⟨*system*⟩
M, N	⟨*coframing*⟩, or a ⟨*system*⟩ specifying a ⟨*coframing*⟩
r	⟨*integer*⟩
Ω	⟨*p-form*⟩
f	⟨*map*⟩
rsx	⟨*list of inequalities*⟩
cob	⟨*list of 1-form variables*⟩
crd, dep, ind	⟨*list of 0-form variables*⟩
drv	⟨*list of rules for exterior derivatives*⟩
pde	⟨*list of expressions or equations*⟩
X	⟨*transform*⟩
T	⟨*tableau*⟩
P	⟨*integral element*⟩

Command	**Function**
`coframing(`cob`,`crd`,`rsx`,`drv`)`	constructs a ⟨*coframing*⟩ with the given cobasis cob, coordinates crd, restrictions rsx and structure equations drv: crd, rsx and drv are optional
`coframing(`S`)`	constructs a ⟨*coframing*⟩ capable of supporting the given ⟨*system*⟩
`eds(`S`,`Ω`,`M`)`	constructs a simple ⟨*EDS*⟩ object with given system and independence condition: if M is not supplied, it is deduced from the rest
`contact(`r`,`M`,`N`)`	constructs the ⟨*EDS*⟩ for the contact system of the jet bundle $J^r(M,N)$
`pde2eds(`pde`,`dep`,`ind`)`	converts a PDE system to an EDS: dependent and independent variables are deduced if they are not specified (variable dependencies are removed)
`set_coframing(`M`)` `set_coframing(`E`)`	sets background coframing and returns previous one
`set_coframing()`	clears background coframing and returns previous one

Table 20.8: Commands for constructing EDS objects

Command	**Function**
`coframing(`E`)`	extracts the underlying ⟨*coframing*⟩
`coframing()`	returns the current background coframing
`cobasis(`M`)` `cobasis(`E`)`	extracts the underlying cobasis
`coordinates(`M`)` `coordinates(`E`)`	extracts the coordinates
`structure_equations(`M`)` `structure_equations(`E`)`	extracts the rules for exterior derivatives for cobasis and coordinates
`restrictions(`M`)` `restrictions(`E`)`	extracts the inequalities describing the restrictions in the ⟨*coframing*⟩
`system(`E`)`	extracts the ⟨*system*⟩ part of E
`independence(`E`)`	extracts the independence condition from E as a Pfaffian ⟨*system*⟩
`properties(`E`)`	returns the currently known properties of the ⟨*EDS*⟩ E as a list of equations ⟨*keyword*⟩=⟨*value*⟩
`one_forms(`E`)` `one_forms(`S`)`	selects the 1-forms from a system
`zero_forms(`E`)` `zero_forms(`S`)`	selects the 0-forms from a system

Table 20.9: Commands for inspecting EDS objects

Command	**Function**
`augment(`E`,`S`)`	appends the extra forms in S to the system in E
M `cross` N E `cross` N	forms the direct product of two coframings: an ⟨*EDS*⟩ E is lifted to the extended space

`pullback(`E`,`f`)`
`pullback(`S`,`f`)`
`pullback(`Ω`,`f`)`
	pulls back the first argument using the ⟨*map*⟩ f

`pullback(`M`,`f`)`	returns a ⟨*coframing*⟩ N suitable as the source for $f : N \to M$

`restrict(`E`,`f`)`
`restrict(`S`,`f`)`
`restrict(`Ω`,`f`)`
	restricts the first argument to the points specified by the ⟨*map*⟩ f

`restrict(`M`,`f`)`	adds the restrictions in f to M

`transform(`M`,`X`)`
`transform(`E`,`X`)`
`transform(`S`,`X`)`
`transform(`Ω`,`X`)`
	applies the change of cobasis X to the first argument: for a ⟨*coframing*⟩ M or an ⟨*EDS*⟩ E, X may be specified in either the forward or reverse direction

`lift(`E`)`	eliminates any 0-forms in E by solving and pulling back

Table 20.10: Commands for manipulating EDS objects

Command	**Function**

`cartan_system(`E`)`
`cartan_system(`S`)`
`cartan_system(`Ω`)`
	calculates the Cartan system (associated Pfaff system, retracting space): no differentiations are performed

`cauchy_system(`E`)`
`cauchy_system(`S`)`
`cauchy_system(`Ω`)`
	calculates the Cauchy system: the Cartan system of the closure under exterior differentiation

`characters(`E`)` `characters(`T`)`	calculates the (reduced) Cartan characters $\{s_1, ..., s_n\}$ (E quasilinear)
`characters(`E`,`P`)`	Cartan characters for a non-linear E at integral element P
`closure(`E`)`	calculates the closure of E under exterior differentiation
`derived_system(`E`)` `derived_system(`S`)`	calculates the first derived system of the Pfaffian system E or S
`dim_grassmann_variety(`E`)` `dim_grassmann_variety(`E`,`P`)`	dimension of the Grassman bundle variety of integral elements: for non-linear E, the base element P must be given
`dim(`M`)` `dim(`E`)`	returns the manifold dimension
`involution(`E`)`	repeatedly prolongs E to involution (or inconsistency)
`linearise(`E`,`P`)`	linearise the (non-linear) EDS E with respect to the integral element P
`integral_element(`E`)`	find a random $\langle$*integral element*$\rangle$ of E
`prolong(`E`)`	prolongs E, and projects back down to a subvariety of the original manifold if integrability conditions arise
`tableau(`E`)`	calculates the $\langle$*tableau*$\rangle$ of the quasilinear Pfaffian $\langle$*EDS*$\rangle$ E
`torsion(`E`)`	returns a $\langle$*system*$\rangle$ of 0-forms specifying the integrability conditions for the semilinear or quasilinear Pfaffian $\langle$*EDS*$\rangle$ E
`grassmann_variety(`E`)`	returns the contact $\langle$*EDS*$\rangle$ for the Grassmann bundle of n-planes over the manifold of E, augmented by the 0-forms specifying the variety of integral elements of E

Table 20.11: Commands for analysing exterior systems

Command	**Function**

`closed(`E`)`
`closed(`S`)`
`closed(`Ω`)`
	checks for closure under exterior differentiation

`involutive(`E`)`	applies Cartan's test for involution
`pfaffian(`E`)`	checks if E is generated by 1-forms and their exterior derivatives
`quasilinear(`E`)`	tests if the *closure* of E can be generated by forms at most linear in the complement of the independence condition
`semilinear(`E`)`	tests if the *closure* of E is quasilinear and, in addition, the coefficients of the linear terms contain only independent variables or constants
E `equiv` E'	checks whether E and E' are algebraically equivalent

Table 20.12: Commands for testing exterior systems

Switch	**Function**
`edsverbose`	if `on`, displays additional information as calculations progress
`edsdebug`	if `on`, produces copious quantities of internal information, in addition to that produced by `edsverbose`
`edssloppy`	if `on`, allows EDS to divide by expressions not known to be non-zero and treats quasilinear systems as semilinear
`edsdisjoint`	if `on`, forces varieties to be decomposed into disjoint components
`ranpos` `genpos`	if `on`, uses a random or generic flag of integral elements when calculating Cartan characters: otherwise the independence condition as presented guides the choice of flag

Table 20.13: Switches (all `off` by default)

Command	**Function**
`coordinates(`S`)`	scans the expressions in S for coordinates
`invert(`X`)`	returns the inverse $\langle$*transform*$\rangle$ X^{-1}
`structure_equations(`X`)` `structure_equations(`X`,`X^{-1}`)`	returns exterior derivatives of lhs(X)
`linear_divisors(`Ω`)`	calculates a basis for the space of 1-form factors of Ω
`exfactors(`Ω`)`	as for `linear_divisors`, but with the additional (non-linear) factor
`index_expand(`any`)`	returns a list of copies of its argument, with free EXCALC indices replaced by successive values from the relevant index range
`pde2jet(`pde`,`dep`,`ind`)`	converts a PDE system into jet bundle notation, replacing derivatives by jet bundle coordinates (variable dependencies are not affected)
`mkdepend(`$list$`)`	restores variable dependencies destroyed by `pde2eds`
`disjoin(`$\{f, g, ...\}$`)`	decomposes the variety specified by the given $\langle$*map*$\rangle$ variables into a disjoint union
`cleanup(`E`)` `cleanup(`M`)`	returns a fresh copy of E or M with all properties and stored results removed
`reorder(`E`)` `reorder(`M`)`	returns a fresh copy of E or M, conforming to the prevailing REDUCE kernel order

Table 20.14: Auxilliary functions

Command	**Function**
`poincare(`Ω`)`	calculates the homotopy integral from the proof of Poincaré's lemma: if Ω is exact, then the result is a form whose exterior derivative gives back Ω
`invariants(`E`,`crd`)` `invariants(`S`,`crd`)`	calculates the invariants (first integrals) of a completely integrable Pfaffian system using the inductive proof of the Frobenius theorem: the optional second argument specifies the order in which the coordinates are to be projected away
`symbol_relations(`E`,`π`)`	returns relations between the entries of the tableau matrix, represented by 2-indexed $\langle$*1-form*$\rangle$ variables ${\pi^a}_i$
`symbol_matrix(`E`,`ξ`)`	returns the symbol matrix for a quasilinear $\langle$*EDS*$\rangle$ E as a function of $\langle$*0-form*$\rangle$ variables ξ_i
`characteristic_variety(`E`,`ξ`)`	returns equations describing the characteristic variety of E in terms of $\langle$*0-form*$\rangle$ variables ξ_i

Table 20.15: Experimental functions (unstable)

20.20 ELLIPFN: A Package for Elliptic Functions and Integrals

Additional documentation and examples can be found in the files `efellip.tex`, `eftheta.tst` and `efweier.tst` in the `packages/ellipfn` directory.

Author: Lisa Temme and Alan Barnes, with contributions from Winfried Neun and several others

20.20.1 Elliptic Functions: Introduction

The package ELLIPFN is designed to provide algebraic and numeric manipulations of many elliptic functions, namely:

- Jacobi's Elliptic Functions;
- Elliptic Integrals;
- Nome and Related Functions;
- Jacobi's Theta Functions and their derivatives;
- Weierstrass Elliptic Functions and the Sigma Function;
- Other Sigma Functions;
- Period Lattice and Related Functions;
- Inverse Elliptic Functions.

The implementation of the functions in this and the next two subsections have been substantially revised by Alan Barnes in 2019. This is to bring the notation more into line with standard (British) texts such as Whittaker & Watson [WW69] and Lawden [Law89] and also to correct a number of errors and omissions. These changes and additions will be itemised in the relevant sections below. New subsections has been added starting in 2021 to support Weierstrassian Elliptic functions, Sigma functions, inverse Jacobi elliptic functions and finally symmetric elliptic integrals.

The functions in this subsection are for the most part autoloading; the exceptions being the subsidiary utility functions such as the AGM function, the quasi-period factors, lattice functions, derivatives of the theta functions and symmetric elliptic integrals.

20.20.2 Jacobi Elliptic Functions

The following functions have been implemented:

- The 12 Jacobi Elliptic Functions
- The Jacobi Amplitude Function
- Arithmetic Geometric Mean
- Descending Landen Transformation

The following Jacobi elliptic functions are available:-

- jacobisn(u,k)
- jacobidn(u,k)
- jacobicn(u,k)
- jacobicd(u,k)
- jacobisd(u,k)
- jacobind(u,k)
- jacobidc(u,k)
- jacobinc(u,k)
- jacobisc(u,k)
- jacobins(u,k)
- jacobids(u,k)
- jacobics(u,k)

These differ somewhat from the originals implemented by Lisa Temme in that the second argument is now the modulus (usually denoted by k in most texts rather than its square m). The notation for the most part follows Lawden [Law89]. The last nine Jacobi functions are related to the three basic ones: `jacobisn(u,k)`, `jacobicn(u,k)` and `jacobidn(u,k)` and use Glaisher's notation. For example

$$\mathrm{ns}(x,k) = \frac{1}{\mathrm{sn}(u,k)}, \qquad \mathrm{cs}(x,k) = \frac{\mathrm{cn}(u,k)}{\mathrm{sn}(u,k)}, \qquad \mathrm{cd}(x,k) = \frac{\mathrm{cn}(u,k)}{\mathrm{dn}(u,k)}.$$

All twelve functions are doubly periodic in the complex plane. The primitive periods and the positions of the zeros and poles of the functions are conveniently

expressed in terms of the so-called quarter periods $\mathrm{K}(k)$ and $i\mathrm{K}'(k)$ which are complete elliptic integrals of the first kind. The details are displayed in the table below; in all cases the pole is single and the corresponding residue is given in the last column of the table. As the functions are doubly-periodic, there are, of course, infinitely many other zeros and poles. These occur at points 'congruent' to the points given in the table obtained by translating by $2m\mathrm{K} + 2in\mathrm{K}'$ where m and n are arbitrary integers.

Function	Period1	Period2	Zero	Pole	Residue
sn	$4\mathrm{K}$	$2i\mathrm{K}'$	0	$i\mathrm{K}'$	$1/k$
ns	$4\mathrm{K}$	$2i\mathrm{K}'$	$i\mathrm{K}'$	0	1
cd	$4\mathrm{K}$	$2i\mathrm{K}'$	K	$\mathrm{K} + i\mathrm{K}'$	$-1/k$
dc	$4\mathrm{K}$	$2i\mathrm{K}'$	$\mathrm{K} + i\mathrm{K}'$	K	-1
cn	$4\mathrm{K}$	$2(\mathrm{K} + i\mathrm{K}')$	K	$i\mathrm{K}'$	$-i/k$
nc	$4\mathrm{K}$	$2(\mathrm{K} + i\mathrm{K}')$	$i\mathrm{K}'$	K	$-1/k'$
sd	$4\mathrm{K}$	$2(\mathrm{K} + i\mathrm{K}')$	0	$\mathrm{K} + i\mathrm{K}'$	$-i/(kk')$
ds	$4\mathrm{K}$	$2(\mathrm{K} + i\mathrm{K}')$	$\mathrm{K} + i\mathrm{K}'$	0	1
dn	$4i\mathrm{K}'$	$2\mathrm{K}$	$\mathrm{K} + i\mathrm{K}'$	$i\mathrm{K}'$	$-i$
nd	$4i\mathrm{K}'$	$2\mathrm{K}$	$i\mathrm{K}'$	$\mathrm{K} + i\mathrm{K}'$	$-i/k'$
sc	$4i\mathrm{K}'$	$2\mathrm{K}$	0	K	$-1/k'$
cs	$4i\mathrm{K}'$	$2\mathrm{K}$	K	0	1

All other periods of the Jacobi functions can be expressed as linear combinations of the two primitive periods with integer coefficients. Thus, for example, $4i\mathrm{K}'$ is a period of cn as

$$4i\mathrm{K}' = 2(\mathrm{K} + \mathrm{K}') - 4\mathrm{K}$$

Extensive rule lists are provided for differentiation of these functions with respect to either argument, for argument shifts by integer multiples of the two quarter periods $\mathrm{K}(k)$ and $i\mathrm{K}'(k)$ and finally Jacobi's transformations for a purely imaginary first argument.

Rules are also provided for the values of the twelve Jacobi functions at the 'eighth' period values: $\mathrm{K}/2$, $i\mathrm{K}'/2$ and $(\mathrm{K} + i\mathrm{K}')/2$. For these rules to yield correct values it is essential that the switch `precise_complex` is ON, otherwise Reduce will often incorrectly simplify the results if they involve complex values of k or k'. It should also be noted that the rule for $\mathrm{dn}((\mathrm{K}(k) + i\mathrm{K}'(k))/2, k)$ differs from that on DLMF:NIST which appears to give incorrect values for some values of k. The rule used in REDUCE is not only simpler, but appears to give correct values in all cases as do the derived rules for cd, sd, nd, ds and dc. It is derived from the third of the identities (2.2.19) of Lawden [Law89] with $u = -(\mathrm{K}(k) + i\mathrm{K}'(k))/2$ and the corresponding identities for sn and cn on the DLMF NIST site.

Some care must be exercised when applying the 'eighth' period rules when the modulus k belongs to the negative real or imaginary axes as $\mathrm{K}(k)$ and all twelve Jacobi functions $\mathrm{sn}(x, k)$ etc. are even functions of the modulus k. The rule should

be applied when the modulus has no assigned value and then resimplifying the result after assigning the modulus its required value.

Complex split functions have recently been implemented for the 12 Jacobi functions for the cases when the modulus k is either real or imaginary. Thus `REPART` and `IMPART` will now return rational expressions involving the twelve Jacobi functions with real arguments and a real modulus $|k| < 1$. As far as I (AB) am aware there are no known methods of implementing split functions for general complex values of the modulus.

Four useful rule lists `TO_SN`, `TO_CN`, `TO_DN` and `NO_GLAISHER` are provided for user convenience. The first `TO_SN` replaces occurences of the squares of cn and dn in favour of the square of sn using the 'Pythagorean' identities. `TO_CN` and `TO_DN` apply these identities to replace squares in favour of cn and dn respectively. At most one of these rule sets should be active at any one time to avoid a potential infinite recursion in the simplification. The fourth rule list `NO_GLAISHER` replaces all occurences of the nine subsidiary Jacobi elliptic functions ns, sc, sd, ... by reciprocals and quotients of the 'basic' Jacobi elliptic functions sn, cn and dn.

A fifth rule list `JACOBIADDITIONRULES` applies addition rules when the first argument of any Jacobi function is a sum. Note that this rule list is NO LONGER active by default. There are many equivalent ways of expressing the right-hand sides of these rules. Previously all the rules were expressed using only the 'basic' Jacobi functions: sn, cn and dn. Currently, however, the right-hand sides of the rules for the reciprocal and quotient Glaisher functions ns, cs, sd etc are expressed using the three Glaisher functions with the same 'denominator' as on the left-hand side. Thus the rule for ns is expressed in terms of ns, cs and ds whilst that for cd is expressed in terms of nd, cd and sd whereas the rules for the three 'basic' functions are unchanged and use only sn, cn and dn. Note, however, that the previous form of these rules will be obtained if the rule-list `NO_GLAISHER` is also active.

If the switch `rounded` is ON and both arguments of a Jacobi function are purely numerical, it will be evaluated numerically (previously it was necessary for both the switches `rounded` and `complex` to be ON). Of course, if either argument is a complex number, the switch `complex` must also be ON for numerical evaluation to be triggered.

The numerical evaluation of the Jacobi functions uses their definitions in terms of theta functions which are valid for all complex values of the argument and modulus. Since theta functions are inherently complex-valued, it is necessary to turn the switch `complex` ON during the evaluation even if both the argument and the modulus of the Jacobi function are real (for example when the modulus $|k| > 1$). The switch `complex` is, however, returned to its old value as the computation completes.

The traditional AGM and ϕ-function based algorithms which were used when the

argument and modulus are both real and $|k| < 1$ have been corrected. There was a problem with the numerical evaluation of `dn, nd, sd, ds, cd, dc` resulting in largish rounding errors for some values of the arguments. There was no problem with previous evaluation of `sn, ns, cn, nc, sc, cs` and these produced the same results (modulo acceptable rounding errors) as the theta function based methods. In fact in all cases when both the arguments are real and $|k| < 1$, the traditional algorithms have been reinstated as they are somewhat faster than the theta function based methods.

Jacobi Amplitude Function

This function is defined as

$$\mathrm{am}(u,k) = \int_0^u \mathrm{dn}(z,k)\mathrm{d}z.$$

As $\mathrm{dn}(z,k)$ has poles with residue $-i$ at points $z = (4m+1)i\mathrm{K}'(k) + 2n\mathrm{K}(k)$ and others with residue $+i$ when $z = (4m+3)i\mathrm{K}'(k) + 2n\mathrm{K}(k)$ where m and n are arbitrary integers, $\mathrm{am}(z,k)$ has logarithmic singularities at these points. In fact the amplitude function is multivalued with its principal value v, say, being given by choosing the contour in the defining integral to be the straight line segment between 0 and u. Other values are given by $v + 2n\pi$ where n is an arbitrary integer and depend on how the chosen contour winds around the logarithmic branch points. To obtain a single valued function it is necessary to introduce branch cuts between the points $i\mathrm{K}'(k)$ and $3i\mathrm{K}'(k)$ and between corresponding congruent branch points.

A rule list is provided for to simplify this function for special values of the arguments and for differentiation with respect to either argument. When the switch `rounded` is ON and both arguments are numeric, REDUCE will attempt to evaluate `jacobiam(z,k)` numerically. Several possible methods are available to evalaute the function, for example summation of its Fourier series and an AGM-based method due to Sala (see DLMF:NIST for more details. Currently the AGM-based method is used and this certainly produces reliable results if the modulus k is real and $|\Im(z)| < \mathrm{K}'(k)$. These agree with those produced by the Fourier series method although rounding errors become significant as $|\Im(z)|$ approaches $\mathrm{K}'(k)$. If k is complex both methods produce the same results when the imaginary parts of k and z are not too large, but it is difficult to give precise bounds on the size of these.

For general values of z and k the AGM-based method always produces a value except perhaps at the logarithmic branch points but these, I believe, are not correct as they fail to satisfy the identity

$$\mathrm{sn}(z,k) = \sin(\mathrm{am}(z,k))$$

where sn is calculated using the theta function method. The Fourier series method fails to converge in these cases and so is not used except for comparison purposes. Currently an alternative method of evaluation due to Sala is being investigated; it uses the Poisson summation formula and is claimed to be valid for all z and k except at branch points.

20.20.3 Some Numerical Procedures

This section briefly describes several procedures which are primarily intended for use in the numerical evaluation of the various elliptic functions and integrals rather than for direct use by users.

Arithmetic Geometric Mean (AGM)

A procedure to evaluate the AGM of initial values a_0, b_0, c_0 exists as `AGM_function(`a_0, b_0, c_0`)` and will return $\{N, AGM, \{a_N, \ldots, a_0\}, \{b_N, \ldots, b_0\}, \{c_N, \ldots, c_0\}\}$, where N is the number of steps to compute the AGM to the desired accuracy.

To determine the Elliptic Integrals K(m), E(m) we use initial values $a_0 = 1$; $b_0 = \sqrt{1 - k^2}$; $c_0 = k$.

Descending Landen Transformation

The procedure to evaluate the Descending Landen Transformation of ϕ and α uses the following equations:

$$(1 + \sin\alpha_{n+1})(1 + \cos\alpha_n) = 2 \text{ where } \alpha_{n+1} < \alpha_n,$$
$$\tan(\phi_{n+1} - \phi_n) = \cos\alpha_n \tan\phi_n \text{ where } \phi_{n+1} > \phi_n.$$

It can be called using `landentrans(`ϕ_0, α_0`)` and will return $\{\{\phi_0, \ldots, \phi_n\}, \{\alpha_0, \ldots, \alpha_n\}\}$.

20.20.4 Legendre's Elliptic Integrals

The following functions have been implemented:

- Complete & Incomplete Elliptic Integrals of the First Kind
- Complete & Incomplete Elliptic Integrals of the Second Kind
- Jacobi's Epsilon & Zeta Functions

- Complete & Incomplete Ellpitic Integrals of the Third Kind
- Some Utility Functions

These again differ somewhat from the originals implemented by Lisa Temme as the second argument is now the modulus k rather that its square. Also in the original implementation there was some confusion between Legendre's form and Jacobi's form of the incomplete elliptic integrals of the second kind; $E(u,k)$ denoted the first in numerical evaluations and the second in the derivative formulae for the Jacobi elliptic functions with respect to their second argument. This confusion was perhaps understandable as in the literature some authors use the notation $\mathrm{E}(u,k)$ for the Legendre form and others for Jacobi's form.

To bring the notation more into line with that in the NIST Digital Library of Mathematical Functions and avoid any possible confusion, $\mathrm{E}(u,k)$ is used for the Legendre form and $\mathcal{E}(u,k)$ for Jacobi's form. This differs from the 2019 version of this section which followed Lawden [Law89] chapter 3, where the notation $\mathrm{D}(\phi,k)$ and $\mathrm{E}(u,k)$ were used for the Legendre and Jacobi forms respectively.

A number of rule lists have been provided to implement, where appropriate, derivatives of these functions, addition rules and periodicity and quasi-periodicity properties and to provide simplifications for special values of the arguments.

Elliptic F

The Elliptic F function can be used as `EllipticF(phi,k)` and will return the value of the *Incomplete Elliptic Integral of the First Kind*:

$$\mathrm{F}(\phi,k) = \int_0^{\phi} (1-k^2\sin^2\theta)^{-1/2}\mathrm{d}\theta.$$

This is actually closely related to the inverse Jacobi function arcsn; in fact for $-\pi/2 <= \Re\phi <= \pi/2$:

$$\mathrm{F}(\phi,k) = \mathrm{arcsn}(\sin\phi,k)$$

.

Elliptic K

The Elliptic K function can be used as `EllipticK(k)` and will return the value of the *Complete Elliptic Integral of the First Kind*:

$$\mathrm{K}(k) = \mathrm{F}(\pi/2,k) = \int_0^{\pi/2} (1-k^2\sin^2\theta)^{-1/2}\mathrm{d}\theta.$$

This is one of the quarter periods of the Jacobi elliptic functions and is often used in the calculation of other elliptic functions. The complementary Elliptic K′ function can be used as `EllipticK!'(k)`. Note that $i\mathrm{K}'(k)$ is the other quarter period of the Jacobi elliptic functions.

Elliptic E

The Elliptic E function comes with either one or two arguments; used with two arguments as `EllipticE(u,k)` it will return the value of Legendre's form of the *Incomplete Elliptic Integral of the Second Kind*:

$$\mathrm{E}(\phi, k) = \int_0^{\phi} \sqrt{1 - k^2 \sin^2\theta}\, \mathrm{d}\theta.$$

When called with one argument `EllipticE(k)` will return the value of the *Complete Elliptic Integral of the Second Kind*:

$$\mathrm{E}(k) = \mathrm{E}(\pi/2, k) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2\theta}\, \mathrm{d}\theta.$$

The complementary Elliptic E′ function can be used as `EllipticE!'(k)`.

The complete integrals are actually multi-valued; to obtain single valued functions it is necessary to introduce branch cuts. For $\mathrm{K}(k)$ and $\mathrm{E}(k)$ these are $(-\infty, -1] \cup [1, +\infty)$. The functions are continuous if the two components of the branch cut are approached from the second and fourth quadrants respectively. For $\mathrm{K}'(k)$ and $\mathrm{E}'(k)$ the branch cut is $(-\infty, 0]$ with continuity if the cut is approached from the second quadrant. For more details, see Lawden [Law89] sections 8.12 to 8.14. The principal values of $\mathrm{K}(k)$ and $\mathrm{E}(k)$ are even functions of k.

The numerical evaluation of the complete integrals is more robust and uses symmetric elliptic integrals. They should now work for all complex values of the modulus and return the principal value of the integral concerned. Note that for all complex values of k, the well-known identities:

$$\mathrm{K}'(k) = \mathrm{K}(\sqrt{1 - k^2}) \qquad \mathrm{E}'(k) = \mathrm{E}(\sqrt{1 - k^2}).$$

are not actually valid for the principal values of the functions concerned. It is necessary that $\Re k >= 0$ with in addition if $\Re k = 0$, $\Im k >= 0$. One of the consequences of this is that the principal values of $\mathrm{K}'(k)$ and $\mathrm{E}'(k)$ are not even functions.

General values for the four complete integrals together with the principal value when $\Re k < 0$ are given in the table below, where n is an arbitrary integer, $k' = \sqrt{1 - k^2}$ and the expressions in the second and third columns refer always to principal values:

Function	General Value	Principal Value when $\Re(k) < 0$ and when $\Re(k) = 0$ and $\Im(k) < 0$
$\mathrm{K}(k)$	$\mathrm{K}(k) - 2in\mathrm{K}(k')$	$\mathrm{K}(-k)$
$\mathrm{K}'(k)$	$\mathrm{K}(k') - 4in\mathrm{K}(k)$	$\mathrm{K}(k') \mp 2i\mathrm{K}(-k)$
$\mathrm{E}(k)$	$\mathrm{E}(k) - 2in(\mathrm{K}(k') - \mathrm{E}(k'))$	$\mathrm{E}(-k)$
$\mathrm{E}'(k)$	$\mathrm{E}(k') - 4in(\mathrm{K}(k) - \mathrm{E}(k))$	$\mathrm{E}(k') \mp 2i(\mathrm{K}(-k) - \mathrm{E}(-k))$

In the third column the upper and lower alternative signs are taken when k lies in the second and third quadrants respectively.

One quite subtle point arises: although the twelve Jacobi elliptic functions and $\mathrm{K}(k)$ are even functions of the modulus k, the quarter period $i\mathrm{K}'(k)$ is not (see the second row of the table above). If $\Re(k) > 0$, then for example $4\mathrm{K}(k)$ and $2i\mathrm{K}'(k) = 2i\mathrm{K}(k')$ are primitive periods of $\mathrm{sn}(x, k)$. However, if we use $-k$ as the modulus, the first primitive period obtained is the same (since $\mathrm{K}(k)$ is an even function) but the second is different namely: $2i\mathrm{K}'(-k) = 2i\mathrm{K}(k') \pm 4\mathrm{K}(k)$. This explains why some of the values returned by the 'eighth' period rules for sn etc. are not even functions of k.

Elliptic D

The Elliptic D function also comes with either one or two arguments; used with two arguments as `EllipticD(u,k)` it will return the value of an alternative form of Legendre's *Incomplete Elliptic Integral of the Second Kind*:

$$\mathrm{D}(\phi, k) = \int_0^{\phi} \frac{\sin^2\theta}{\sqrt{1 - k^2\sin^2\theta}}\,\mathrm{d}\theta.$$

When called with one argument `EllipticD(k)` will return the value of the *Complete Elliptic Integral of the Second Kind*:

$$\mathrm{D}(k) = \mathrm{D}(\pi/2, k) = \int_0^{\pi/2} \frac{\sin^2\theta}{\sqrt{1 - k^2\sin^2\theta}}\,\mathrm{d}\theta.$$

The integrals of the first and second kind are related:

$$\mathrm{E}(k) = \mathrm{K}(k) - k^2\mathrm{D}(k), \qquad \mathrm{E}(\phi, k) = \mathrm{F}(\phi, k) - k^2\mathrm{D}(\phi, k).$$

For all the elliptic integrals, rule lists are provided for simplification for special values of the argument(s), differentiation with respect to either argument and shift rules for the incomplete integrals so that their first argument lies in the range $0 <= \phi <= \pi/2$.

If the arguments of the elliptic integral function are numeric and the switch `rounded` is ON, the numerical value will be returned. As with Jacobi elliptic functions, if any argument is complex then numerical evaluation will only occur if the switch `complex` is also ON.

Numerical evaluation should now succeed for all complex values of the arguments provided, of course, the corresponding integral exists. The incomplete integrals $\mathrm{F}(\phi, k)$, $\mathrm{E}(\phi, k)$, $\mathrm{D}(\phi, k)$ and $\Pi(\phi, a, k)$ are all multi-valued as there are branch points in the defining integrands when $\sin(\phi) = \pm 1$ and $k \sin(\phi) = \pm 1$.

For numerical evaluation when the first argument ϕ is complex there are issues that arise which are absent when ϕ is real. One might use the the straight line contour from $\theta = 0$ to $\theta = \phi$, but this is leads to integrals that are difficult to evaluate. When $|\Re\phi| \leq \pi/2$, the change of variable $x = \sin\theta$ is applied and the value returned uses the straight line contour between $x = 0$ and $x = \sin(\phi)$. This will produce a different value if the loop formed by the two contours encloses a branch point. When $|\Re\phi| > \pi/2$ the contour used is the straight line segment along the real axis from 0 to the multiple of π nearest to ϕ followed by the straight line segment from between $x = 0$ and $x = \sin(\phi)$. When ϕ is real the contours in the θ and x planes coincide. Other contours between the two endpoints will yield different values depending on how the contour winds around the two branch points.

Note also that if the contour contains a branch point, there is sign ambiguity as the branch point is crossed depending on which branch of the square root in the integrand is chosen. It seems logical to choose the principal branch.

For example when $k > 1$ & $1/k < y < 1$

$$\int_0^y \frac{\mathrm{d}x}{\sqrt{1-x^2}\sqrt{1-k^2x^2}}$$

is taken to be

$$\int_0^{1/k} \frac{\mathrm{d}x}{\sqrt{1-x^2}\sqrt{1-k^2x^2}} - i\int_{1/k}^y \frac{\mathrm{d}x}{\sqrt{1-x^2}\sqrt{k^2x^2-1}}.$$

Elliptic Π

The Elliptic Π function can be used as `EllipticPi( )` and will return the value of the *Elliptic Integral of the Third Kind*.

The Elliptic Π function comes with either two or three arguments; when called with three arguments `EllipticPi(phi,a,k)` will return the value of the *Incomplete Elliptic Integral of the Third Kind*:

$$\Pi(\phi, a, k) = \int_0^{\phi} \left((1 - a^2\sin^2\theta)\sqrt{1 - k^2\sin^2\theta}\right)^{-1} \mathrm{d}\theta.$$

For $-\pi/2 \leq \Re\phi \leq \pi/2$, the incomplete integral may be written in the form:

$$\Pi(\phi, a, k) = \int_0^{sin\phi} \left((1 - a^2t^2)\sqrt{(1-t^2)(1-k^2t^2)}\right)^{-1} \mathrm{d}t.$$

When called with two arguments as `EllipticPi(a,k)` it will return the value of the *Complete Elliptic Integral of the Third Kind*:

$$\begin{aligned}\Pi(a,k) = \Pi(\pi/2,a,k) &= \int_0^{\pi/2} \left((1-a^2\sin^2\theta)\sqrt{1-k^2\sin^2\theta}\right)^{-1}\,\mathrm{d}\theta \\ &= \int_0^1 \left((1-a^2t^2)\sqrt{(1-t^2)(1-k^2t^2)}\right)^{-1}\,\mathrm{d}t.\end{aligned}$$

For certain values of a namely 0, ± 1 and $\pm k$ the integrals reduce to elliptic integrals of the first and second kinds.

20.20.5 Jacobi's Elliptic Integrals

Jacobi E

The Jacobi E function can be used as `jacobiE(u,k)`; it will return the value of Jacobi's form of the *Incomplete Elliptic Integral of the Second Kind*:

$$\mathcal{E}(u,k) = \int_0^u \mathrm{dn}^2(v,k)\,\mathrm{d}v.$$

The relationship between the two forms of incomplete elliptic integrals can be expressed as

$$\mathcal{E}(u,k) = \mathrm{E}(\mathrm{am}(u),k).$$

Note that

$$\mathrm{E}(k) = \mathcal{E}(\mathrm{K}(k),k) = \int_0^{\mathrm{K}(k)} \mathrm{dn}^2(v,k)\,\mathrm{d}v.$$

On a GUI that supports calligraphic characters (NB. this is now the case with the CSL GUI), there is no problem and it is rendered as $\mathcal{E}(u,k)$ in accordance with NIST usage. On non-GUI interfaces the Jacobi E function is rendered as E_j.

Jacobi's Zeta Function

This can be called as `jacobiZeta(u,k)` and refers to Jacobi's (elliptic) Zeta function $\mathrm{Z}(u,k)$ whereas the operator `Zeta` will invoke Riemann's ζ function. It is closely related to Jacobi's Epsilon function `jacobie`; in fact

$$\mathrm{Z}(u,k) = \mathcal{E}(u,k) - u\mathrm{E}(k)/\mathrm{K}(k).$$

20.20.6 Symmetric Elliptic Integrals

The functions in this section are currently only intended for use for in the numerical evaluation of the 'classical' elliptic integrals discussed above and in certain other 'basic' elliptic integrals. The switch `rounded` should be ON.

Symmetric elliptic integrals are a relatively new development in the theory of elliptic functions mainly due to Carlson and his collaborators; an extensive bibliography is available on the NIST Digital Library of Mathematical Functions starting at the link DLMF:NIST, Bibliography. They have a number of advantages over more traditional approaches; see for example Advantages of Symmetry.

The fundamental symmetric integrals are all integrals over the positive real line and involve the function $s(t) = \sqrt{t+x}\sqrt{t+y}\sqrt{t+z}$ where $x, y, z \in \mathbb{C}\backslash(-\infty, 0]$ but, except where otherwise stated, at most one of x, y, z may be zero.

$$\begin{aligned}
\mathrm{RF}(x,y,z) &= \frac{1}{2}\int_0^\infty \frac{\mathrm{d}t}{s(t)} \\
\mathrm{RG}(x,y,z) &= \frac{1}{4}\int_0^\infty \frac{1}{s(t)}\left(\frac{x}{t+x}+\frac{y}{t+y}+\frac{z}{t+z}\right)t\mathrm{d}t \\
\mathrm{RJ}(x,y,z,p) &= \frac{3}{2}\int_0^\infty \frac{\mathrm{d}t}{s(t)(t+p)} \\
\mathrm{RD}(x,y,z) &= \frac{3}{2}\int_0^\infty \frac{\mathrm{d}t}{s(t)(t+z)} \qquad z \neq 0, \quad x+y \neq 0 \\
\mathrm{RC}(x,y) &= \frac{1}{2}\int_0^\infty \frac{\mathrm{d}t}{\sqrt{t+x}(t+y)} \qquad y \neq 0
\end{aligned}$$

The first three integrals defined above are symmetric in x, y, z and are termed *symmetric elliptic integrals of the first, second and third kinds* respectively. RD and RC are degnerate versions of RJ and RF respectively as

$$\mathrm{RD}(x,y,z) = \mathrm{RJ}(x,y,z,z) \qquad \mathrm{RC}(x,y) = \mathrm{RF}(x,y,y)$$

RD is an elliptic integral of the second kind and is only symmetric in x, y whilst RC is an elementary integral, but can be numerically evaluated efficiently without needing to distinguish between the trigonometric and hyperbolic cases. For certain special values of p the integral RJ of the third kind degenerates into integrals of the first and second kinds. For numeric evaluations it turns out that the use of RD is more convenient than that of RG (which is not currently implemented in REDUCE). For more information see §19.15-19.29 of the DLMF:NIST.

The Legendre elliptic integrals may all be expressed in terms of the symmetric

integrals; the complete integrals satisfy

$$\begin{aligned}
\mathrm{K}(k) &= \mathrm{RF}(0, k'^2, 1) \\
\mathrm{D}(k) &= \mathrm{RD}(0, k'^2, 1)/3 \\
\mathrm{E}(k) &= \mathrm{RF}(0, k'^2, 1) - k^2\mathrm{RD}(0, k'^2, 1)/3 \\
\Pi(a, k) &= \mathrm{RF}(0, k'^2, 1) + a^2\mathrm{RJ}(0, k'^2, 1, 1 - a^2)/3
\end{aligned}$$

where here and below $k' = \sqrt{1 - k^2}$. For the incomplete integrals, defining $s = \sin\phi$ with $-\pi/2 \leq \Re\phi \leq \pi/2$, we have

$$\begin{aligned}
\mathrm{F}(\phi, k) &= s\mathrm{RF}(1 - s^2, 1 - k^2s^2, 1) \\
\mathrm{D}(\phi, k) &= s^3\mathrm{RD}(1 - s^2, 1 - k^2s^2, 1)/3 \\
\mathrm{E}(\phi, k) &= \mathrm{F}(\phi, k) - k^2s^3\mathrm{RD}(1 - s^2, 1 - k^2s^2, 1)/3 \\
\Pi(\phi, a, k) &= \mathrm{F}(\phi, k) + a^2s^3\mathrm{RJ}(1 - s^2, 1 - k^2s^2, 1, 1 - a^2s^2)/3
\end{aligned}$$

The above formulae are only valid when the range of integration does not include one or more branch points of the integrand (when s is real and $s^2 > 1$ or when ks is real and $(ks)^2 > 1$). In these cases it is necessary to split the range of integration at the branch points resulting in two or three elliptic integrals.

Currently the Carlson's duplication method is used to evaluate the symmetric elliptic integrals of the first, second and third kinds RF, RD and RJ (and also the related elementary integral RC). For more details see the DLMF website: Duplication Method. In particular the REDUCE code is essentially a translation from Fortran of the code of Carlson & Notis [CN81] generalised for complex arguments and structured to avoid GO TO.

Alternatively the symmetric integral RF may be evaluated using a sequence of quadratic transformations which converge rapidly to the elementary hyperbolic integral:

$$\mathrm{RC}(X^2 + Y^2, X^2) = \mathrm{RF}(X^2, X^2, X^2 + Y^2) = \operatorname{arctanh}(Y/(X^2 + Y^2))/Y.$$

More information on this method due to Carlson in the 1990's may be found on the DLMF website: Quadratic Transformations.

Basic Elliptic Integrals

Elliptic integrals involving the square root $s(t)$ of a cubic or quartic polynomial in which the range of integration is an arbitrary interval $[y, x]$ of the real line are considered. Again there are three basic kinds: first, second and third. Carlson [Car88] & [Car99] showed how each of these can be expressed as a fundamental

symmetric elliptic integral of the corresponding kind over the range $[0,\infty)$. Let

$$\begin{array}{rll} X_\alpha = & \sqrt{a_\alpha + b_\alpha x} & 1 \le \alpha \le 5 \\ Y_\beta = & \sqrt{a_\beta + b_\beta y} & 1 \le \beta \le 5 \\ d_{\alpha\beta} = & a_\alpha b_\beta - a_\beta b_\alpha & d_{\alpha\beta} \ne 0 \text{ if } \alpha \ne \beta \\ s(t) = & \prod_{\alpha=1}^{4} \sqrt{a_\alpha + b_\alpha t} & \end{array}$$

and where the four line segments with end points $a_\alpha + b_\alpha y$ and $a_\alpha + b_\alpha x$ for $1 \le \alpha \le 4$ lie in $\mathbb{C} \setminus (-\infty, 0)$. Note that if $s(t)$ is a cubic then one simply chooses supplies unity i.e. $a_\alpha = 1, b_\alpha = 0$ as a fourth factor of $s(t)$.

Then integrals of the first and second kinds take the form:

$$\begin{aligned} \int_y^x \frac{\mathrm{d}t}{s(t)} &= 2\mathrm{RF}(U_{12}^2, U_{13}^2, U_{23}^2) \\ \int_y^x \frac{(a_1 + b_1 t)\mathrm{d}t}{(a_4 + b_4 t)s(t)} &= \frac{2}{3} d_{12} d_{14} \mathrm{RD}(U_{12}^2, U_{13}^2, U_{23}^2) + \frac{2X_1 Y_1}{X_4 Y_4 U_{23}} \qquad \text{if } U_{23} \ne 0 \end{aligned}$$

where in the second equation we have chosen (wlog) the distinguished factors to have indices 1 & 4.

For any permutation $\alpha, \beta, \gamma, \delta$ of 1,2,3,4, let

$$\begin{array}{rll} U_{\alpha\beta} = & (X_\alpha X_\beta Y_\gamma Y_\delta + X_\gamma X_\delta Y_\alpha Y_\beta)/(x-y) & \text{for } x, y \text{ finite} \\ = & \sqrt{b_\alpha}\sqrt{b_\beta} Y_\gamma Y_\delta + Y_\alpha Y_\beta \sqrt{b_\gamma}\sqrt{b_\delta} & \text{for } x = \infty \\ = & X_\alpha X_\beta \sqrt{-b_\gamma}\sqrt{-b_\delta} + X_\gamma X_\delta \sqrt{-b_\alpha}\sqrt{-b_\beta} X_\gamma & \text{for } y = -\infty \end{array}$$

Clearly $U_{\alpha\beta} = U_{\beta\alpha} = U_{\gamma\delta} = U_{\delta\gamma}$ and at most one of U_{12}, U_{13}, U_{23} is zero since $U_{\alpha\beta}^2 - U_{\alpha\gamma}^2 = d_{\alpha\delta} d_{\beta\gamma} \ne 0$, thus at most one of the parameters of RF and RD is zero. If X_4 or Y_4 is zero,that is if $a_4 + b_4 t$ vanishes at one end of the range of integration, the integral of the second kind diverges.

In the 'awkward' case when $U_{23} = 0$ or $U_{12}^2 + U_{13}^2 = 0$, the above method breaks down. However it is still possible to calculate the required integral; choose $\beta = 2$ or $\beta = 3$ so that $b_\beta \ne 0$ and note that

$$\begin{aligned} b_\beta \int \frac{(a_1 + b_1 t)\mathrm{d}t}{(a_4 + b_4 t)s(t)} - b_1 \int \frac{(a_\beta + b_\beta t)\mathrm{d}t}{(a_4 + b_4 t)s(t)} &= d_{1\beta} \int \frac{\mathrm{d}t}{(a_4 + b_4 t)s(t)} \\ b_4 \int \frac{(a_\beta + b_\beta t)\mathrm{d}t}{(a_4 + b_4 t)s(t)} - b_\beta \int \frac{(a_4 + b_4 t)\mathrm{d}t}{(a_4 + b_4 t)s(t)} &= d_{\beta 4} \int \frac{\mathrm{d}t}{(a_4 + b_4 t)s(t)}. \end{aligned}$$

The second term on the lhs of the second equation reduces to $b_\beta \int 1/s(t)\mathrm{d}t$ and so is an integral of the first kind. Thus the required integral can be expressed in terms of an integral of the second kind and one of the first kind.

Integrals of the third kind take the form

$$\int_y^x \frac{(a_1 + b_1 t)\mathrm{d}t}{(a_5 + b_5 t)s(t)} = \frac{2 d_{12} d_{13} d_{14}}{3 d_{15}} \mathrm{RJ}(U_{12}^2, U_{13}^2, U_{23}^2, U_{15}^2) + \mathrm{RC}(S_{15}^2, Q_{15}^2)$$

where

$$\begin{aligned}
S_{15} &= \left(\frac{X_2X_3X_4}{X_1}Y_5^2 + \frac{Y_2Y_3Y_4}{Y_1}X_5^2\right)/(x-y) \qquad \text{for } x, y \text{ finite} \\
&= \frac{X_2X_3X_4}{X_1}Y_5^2 + \frac{Y_2Y_3Y_4}{Y_1}X_5^2 \qquad \text{for } x = \infty \text{ or } y = -\infty \\
U_{15}^2 &= U_{1\beta}^2 - \frac{d_{1\gamma}d_{1\delta}d_{\beta 5}}{d_{15}} = U_{\beta\gamma}^2 - \frac{d_{1\beta}d_{1\gamma}d_{\delta 5}}{d_{15}} \neq 0 \\
Q_{15}^2 &= \frac{(X_5Y_5)^2}{(X_1Y_1)^2}U_{15}^2 \qquad \text{where } S_{15}^2 - Q_{15}^2 = \frac{d_{25}d_{35}d_{45}}{d_{15}} \neq 0
\end{aligned}$$

where β, γ, δ is a permutation of 2,3,4. Again at most one of the parameters of RJ is zero. This method will break down when calculating S_{15} if $a_1 + b_1 t$ vanishes at either the upper or lower limit of integration as either X_1 or Y_1 is zero. However the situation can be remedied by choosing β so that $X_\beta \neq 0$, $Y_\beta \neq 0$ and $b_\beta \neq 0$ and using similar methods to that used for the awkward case for integrals of the second kind.

Note if the integration range is $(-\infty, +\infty)$, the above formulae for integrals of all three kinds are not valid and the integral needs to be split into two at, say, zero. Similarly if the integration range is such that the integrand has branch points, the integration range will need to be split into two at each of these branch points.

In REDUCE elliptic integrals of the three kinds may be evaluated numerically when the switch `rounded` is ON by calling the functions `ellint_1st`, `ellint_2nd` and `ellint_3rd`. The first two parameters are the lower and upper limits of the integration range; these are followed by 4 (or 5 for `ellint_3rd`) two-element lists $\{a_1, b_1\}, \{a_2, b_2\} \ldots$.

Known bug: As pointed out by Carlson & FitzSimmons [CF00], the algorithms for RF & RJ may break down when $s(t)$ is the product of two quadratic factors each with complex conjugate zeros $x_1 \pm iy_1$ & $x_2 \pm iy_2$. One of the three quantities U_{12}, U_{13}, U_{23} may be negative and this causes the algorithm to return an incorrect result. This error only arises when the crossing point (neccessarily real) of the diagonals of the parallellogram in the complex plane formed by the four zeros of $s(t)$ lies inside the range of integration. The algorithm also appears to produce the correct result when the crossing point is inside the range of integration but is 'sufficiently close' to either end of the range – the precise condition appears to be unknown. In the same paper Carlson & FitzSimmons propose a modified algorithm of the same ilk to overcome these difficulties, however it is yet to be implemented in REDUCE.

Some Examples

In this subsection some of the advantages of symmetry are illustrated; the formula for the integral of the first kind replaces the 28 formulae in Gradshteyn and Ryzhik.

For the cubic case one of the factors in $s(t)$ is simply taken to be unity whilst for cases of the form

$$\int_\alpha^\beta \frac{\mathrm{d}x}{\sqrt{(a+bx^2)(c+dx^2)}}$$

one must use the substitution $t = x^2$ to obtain

$$\frac{1}{2}\int_{\alpha^2}^{\beta^2} \frac{\mathrm{d}t}{\sqrt{t(a+bt)(c+dt)}}$$

Moreover the restriction that one limit of integration be a branch point of the integrand is eliminated without doubling the number of standard integrals in the result. Similarly the formula for the integral of the second kind replaces no less than 144 cases in Gradshteyn and Ryzhik (72 each for the quartic and cubic cases). For cases where the numerator in the integrand is unity one simply takes $a_1 = 0, b_1 = 1$ and if there is to be no apparent term of the form $(a_4 + b_4 t)^{3/2}$ in its denominator one takes $a_4 = 0, b_4 = 1$.

When $0 \le \phi \le \pi/2$, the fundamental integrals of the first, second and third kinds $\mathrm{F}(\phi, k)$, $\mathrm{E}(\phi, k)$, $\mathrm{D}(\phi, k)$ and $\Pi(\phi, a, k)$ may be written as

$$\begin{aligned}
\mathrm{F}(\phi,k) &= \frac{1}{2}\int_0^{s^2} \frac{\mathrm{d}t}{\sqrt{t(1-t)(1-k^2t)}} \\
\mathrm{E}(\phi,k) &= \frac{1}{2}\int_0^{s^2} \frac{\sqrt{1-k^2t}\mathrm{d}t}{\sqrt{t(1-t)}} \\
\mathrm{D}(\phi,k) &= \frac{1}{2}\int_0^{s^2} \frac{\sqrt{t}\mathrm{d}t}{\sqrt{(1-t)(1-k^2t)}} \\
\Pi(\phi,a,k) &= \frac{1}{2}\int_0^{s^2} \frac{\mathrm{d}t}{\sqrt{t(1-t)(1-k^2t)}(1-a^2t)}
\end{aligned}$$

where in all four cases use has been made of the substitution $t = \sin^2\theta$ and $s = \sin\phi$. In the last three equations one takes $a_4 = 1, b_4 = 0$ so that the fourth factor in $s(t)$ is unity and in the fourth equation one also takes $a_1 = 1, b_1 = 0$.

20.20.7 Some Numerical Utility Functions

Five utility functions are provided:

- `nome2mod(q)`
- `nome2mod!'(q)`
- `nome2!K(q)`
- `nome2!K!'(q)`

- `nome(k)`

These are only operative when the switch `rounded` is on and their argument is numerical. The first pair relate the nome q of the theta functions with the moduli k and $k' = \sqrt{1-k^2}$ of the associated Jacobi elliptic functions.

The second pair return the quarter periods K and K$'$ respectively of the Jacobi elliptic functions associated with the nome q.

Finally, `nome(k)` returns the nome q associated with the modulus k of a Jacobi elliptic function and is essentially the inverse of `nome2mod`.

20.20.8 Jacobi Theta Functions

These theta functions differ from those originally defined by Lisa Temme in a number of respects. Firstly four separate functions of two arguments are defined:

- `ellipticTheta1(u,tau)` $\vartheta_1(u,\tau)$
- `ellipticTheta2(u,tau)` $\vartheta_2(u,\tau)$
- `ellipticTheta3(u,tau)` $\vartheta_3(u,\tau)$
- `ellipticThetaS(u,tau)` $\vartheta_4(u,\tau)$

rather than a single function with three arguments (with the first argument taking integer values in the range 1 to 4). Secondly the periods are $2\pi, 2\pi, \pi$ and π respectively (NOT 4K, 4K, 2K and 2K). Thirdly the second argument is the modulus $\tau = a + ib$ where $b = \Im\tau > 0$ and hence the quasi-period of the theta functions is $\pi\tau$.

The second parameter was previously the nome $q = e^{i\pi\tau}$ where $|q| < 1$ since $\Im\tau > 0$. As a consequence `ellipticTheta1` and `ellipticTheta2` were multi-valued owing to the appearance of $q^{1/4}$ in their defining expansions. `ellipticTheta3` and `ellipticTheta4` were, however, single-valued functions of q.

Regarded as functions of τ, `ellipticTheta1` and `ellipticTheta2` are single-valued functions. The nome is given by $q = \exp(i\pi\tau)$ so that the condition $\Im(\tau) > 0$ ensures that $|q| < 1$. Note also in this case $q^{1/4}$ is interpreted as $\exp(i\pi\tau/4)$ rather than the principal value of $q^{1/4}$. Thus, τ, $2+\tau$, $4+\tau$ and $6+\tau$ produce four different values of both `ellipticTheta1` and `ellipticTheta2` although they all correspond to the same nome q.

The four theta functions are defined by their Fourier series:

$$\vartheta_1(z,\tau) = 2e^{i\pi\tau/4}\sum_{n=0}^{\infty}(-1)^n q^{n^2+n}\sin(2n+1)z$$
$$\vartheta_2(z,\tau) = 2e^{i\pi\tau/4}\sum_{n=0}^{\infty} q^{n^2+n}\cos(2n+1)z$$
$$\vartheta_3(z,\tau) = 1+2\sum_{n=1}^{\infty} q^{n^2}\cos 2nz$$
$$\vartheta_4(z,\tau) = 1+2\sum_{n=1}^{\infty}(-1)^n q^{n^2}\cos 2nz.$$

Utilising the periodicity and quasi-periodicity of the theta functions some generalised shift rules are implemented to shift their first argument into the base period parallelogram with vertices

$$(\pi/2,\pi\tau/2),\quad (-\pi/2,\pi\tau/2),\quad (-\pi/2,-\pi\tau/2),\quad (\pi/2,-\pi\tau/2).$$

Together with the relation $\vartheta_1(0,\tau)=0$, these shift rules serve to simplify all four theta functions to zero when appropriate.

Numerical Evaluation

When the switch `rounded` is ON and the arguments are purely numerical, the theta functions are evaluated numerically. Note that as the modulus τ is necessarily complex with a positive imaginary part, the switch `complex` should normally be ON.

However, there is *one exception* if the second parameter is purely real and lies between 0 and 1, it is interpreted as the nome q rather than the modulus. This exception facilitates the evaluation and graphing of theta functions in the case when both the argument and the nome are real which occurs frequently in practical applications. Theta functions are not defined if the second parameter has a negative imaginary part or if it is real and $<= 0$ or $>= 1$ and the numerical evaluation fails.

In what follows a and b will denote the real and imaginary parts of τ respectively and so $|q| = \exp(-\pi b)$ and $\arg q = \pi a$. The series for the theta functions are fairly rapidly convergent due to the quadratic growth of the exponents of the nome q – except for values of q for which $|q|$ is near to 1 (i.e. $b = \Im\tau$ close to zero). In such cases the direct algorithm would suffer from slow convergence and rounding errors. For such values of $|q|$, Jacobi's transformation $\tau' = -1/\tau$ can be used to produce a smaller value of the nome and so increase the rate of convergence. This works very well for real values of q, or equivalently for τ purely imaginary since

$q' = q^{1/b^2}$, but for complex values the gains are somewhat smaller. The Jacobi transformation produces a nome q' for which $|q'| = |q|^{1/(a^2+b^2)}$.

When $\Re q < 0$, the Jacobi transformation is preceded by either the modular transformation $\tau' = \tau + 1$ when $\Im q < 0$, or $\tau' = \tau - 1$ when $\Im q > 0$, which both have the effect of multiplying q by -1, so that the new nome has a non-negative real part and $|a| \leq 1/2$. Thus the worst case occurs for values of the nome q near to $\pm i$ where $|q'| \approx |q|^4$.

By using a series of Jacobi transformations preceded, if necessary by τ-shifts to ensure $|a| <= 1/2$, $|q|$ may be reduced to an acceptable level. Somewhat arbitrarily these Jacobi's transformations are used until $b > 0.6$ (i.e. $|q| < 0.15$). This seems to produce reasonable behaviour. In practice more than two applications of Jacobi transformations are rarely necessary.

The previous version of the numerical code returned the principal values of ϑ_1 and ϑ_2, that is the ones obtained by taking the principal value of $q^{1/4}$ in their series expansions. The current version replaces $q^{1/4}$ by $\exp(i\pi\tau/4)$. If the principal value is required, it is easily obtained by multiplying by the 'correcting' factor $q^{1/4}\exp(-i\pi\tau/4)$.

Derivatives of Theta Functions

Four functions are provided:

- `theta1d(u,ord,tau)`
- `theta2d(u,ord,tau)`
- `theta3d(u,ord,tau)`
- `theta4d(u,ord,tau)`

These return the dth derivatives of the respective theta functions with respect to their first argument u; the third parameter τ is as for theta functions; either the modulus if $\Im\tau > 0$ or the nome if τ is real and lies between 0 and 1. These functions are only triggered when the switch `rounded` is ON and their arguments are numeric with d being a positive integer. They are provided mainly to support the implementation the Weierstrassian and Sigma functions discussed in the following subsection.

The numeric code simply sums the Fourier series for the required derivatives. Unlike the theta functions themselves the code does not use the quasi-periodicity nor modular transformations to speed up the convergence of the series by reducing the sizes of $\Im u$ and $|q|$. In the numerical evaluation of the Weierstrassian and Sigma functions these functions are only called after the necessary shifts of the argument

u and modular transformations of τ have been performed. These are much simpler in this context.

Nevertheless they may be used from top level and numerical experiments reveal that the rounding errors are not significant provided $|q|$ is not near one (say $|q| < 0.9$) and u is real or at least has a relatively small imaginary part.

20.20.9 Weierstrass Elliptic & Sigma Functions

Three main functions of three arguments are defined:

- $\wp(u, \omega_1, \omega_3)$ — `weierstrass(u,omega1,omega3)`
- $\zeta_w(u, \omega_1, \omega_3)$ — `weierstrassZeta(u,omega1,omega3)`
- $\sigma(u, \omega_1, \omega_3)$ — `weierstrass_sigma(u,omega1,omega3)`

The notation used is broadly similar used by Lawden [Law89] which is also used in the NIST Digital Library of Mathematical Functions DLMF:NIST. However, ζ_w is used for the Weierstrassian Zeta function to distinguish it from the Riemann Zeta function and the usual symbol $\wp$ is used for the Weierstrassian elliptic function itself.

The two primitive periods of the Weierstrass function are $2\omega_1$ and $2\omega_3$ and these must satisfy $\Im(\omega_3/\omega_1) \neq 0$. The two periods are normally numbered so that $\tau = \omega_3/\omega_1$ has a positive imaginary part and hence the nome $q = exp(i\pi\tau)$ satisfies $|q| < 1$.

Any linear combination $\Omega_{m,n} = 2m\omega_1 + 2n\omega_3$ where m and n are integers (not both zero) is also a period. The set of all such periods plus the origin form a lattice. In the literature $-(\omega_1 + \omega_3)$ is often denoted by ω_2 and $2\omega_2$ is clearly also a half-period; this accounts for the gap in the numbering of primitive periods. The period ω_2 is little used in the REDUCE rule sets for the Weierstrassian elliptic and related functions.

The primitive periods are not unique; indeed any periods $2\Omega_1$ and $2\Omega_3$ defined by the unimodular integer bilinear transformation:

$$\Omega_1 = a\omega_1 + b\omega_3, \qquad \Omega_3 = c\omega_1 + d\omega_3, \qquad \text{where } ad - bc = 1$$

are also primitive. This fact is very useful in the numerical evaluation of the Weierstrassian and Sigma functions as a sequence of such transformations may be used to increase the size $\Im\tau$ and so reduce the size of $|q|$. Thus the Fourier series for the theta functions and their derivatives will converge rapidly. In theory these transformations may be used to reduce the size of $|q|$ until $\Im\tau \geq \sqrt{3}/2$ when $|q| < 0.06$. However, in numerical evaluations in REDUCE it is sufficient to use these transformations only until $\Im\tau > 0.7$, i.e. until $|q| < 0.11$. In practice only two or three

iterations are required and usually very much smaller values of $|q|$ are achieved particularly when τ is purely imaginary i.e. q is real.

The Weierstrassian function is even and has a pole of order 2 at all lattice points. The Zeta and Sigma functions are only quasi-periodic on the lattice. Zeta is odd and has simple poles of residue 1 at all lattice points. The basic Sigma function $\sigma(u, \omega_1, \omega_3)$ is odd and regular everywhere as is the function $\vartheta_1(u, \tau)$ to which it is closely related. It has zeros at all lattice points. All three functions $\wp$, ζ_w and σ are homogenous of degrees -2, -1 and $+1$ respectively. The functions are related by

$$\wp(u) = -\zeta'_w(u), \qquad \zeta_w(u) = \sigma'(u)/\sigma(u),$$

where the lattice parameters have been omitted for conciseness.

Rule sets are provided which implement all the properties such as double periodicity discussed above. The rule for the derivative of the Weierstrass function has recently been corrected; it involves the square root of $4\wp(u)^3 - g_2\wp(u) - g_3$ and to allow for the ambiguity of the sign of the square root it includes an operator `epsilon_w` whose value is either $+1$ or -1. Its value changes at the poles of the Weierstrass function, at points where the value of the Weierstrass function is one of the three lattice roots e_1, e_2 or e_3 and also where $4\wp(u)^3 - g_2\wp(u) - g_3$ becomes negative as its square root has a branch cut there.

For numerical evaluation the switches `rounded` and `complex` must both be ON and all three parameters of the Weierstrassian functions must be numeric. It is not, however, necessary to ensure $\Im(\omega_3/\omega_1) > 0$ as the second and third parameters will be swapped if required. Numerical evaluation of ϵ_w has now been implemented for all complex values of its three parameters, but currently it should be regarded as somewhat experimental; it uses fairly crude numerical approximations to the derivative in order to choose the sign of thecorrect branch of the square root. By and large the literature does not discuss how the sign may be determined for general complex values of the arguments. One exception is the Dover Handbook of Mathematical Functions[AS72] which gives a prescription (without proof) only for the special cases where the lattice associated with the Weierstrass function is either rectangular or rhombic, i.e. when the invariants g_2 and g_3 are real.

Alternative forms of the Weierstrass Functions

Three commonly used alternative forms of the Weierstrassian functions in which they are regarded as functions of the lattice invariants g_2 and g_3 rather than the primitive half-periods ω_1 and ω_3 are provided:

- $\wp(u \mid g_2, g_3)$ — `weierstrass1(u,g2,g3)`
- $\zeta_w(u \mid g_2, g_3)$ — `weierstrassZeta1(u,g2,g3)`

- $\sigma(u \mid g_2, g_3)$ — `weierstrass_sigma0(u,g2,g3).`

Note the trailing zero rather than 1 in the third of these functions to avoid a clash with the notation for the other sigma functions below. Note also that for output they are distinguished by separating the first and second arguments by a vertical bar rather than a comma.

The rule for differentiation of this variant of the Weierstrass function involves the operator `epsilon_w1` whose value is either $+1$ or -1 and is analogous to the operator `epsilon_w`.

If the arguments of the Weierstrassian and sigma are all numeric, the functions will only be evaluated if both `rounded` and `complex` are ON. The only exceptions are the functions weierstrass1, weierstrassZeta1 and weierstrass_sigma0; these are real-valued for real arguments and if all three arguments are real it suffices that the switch `rounded` is ON. Similar remarks apply to those lattice functions (see below) which are functions of the lattice invariants g_2 and g_3. Numerical evaluation of ϵ_{w1} has now been implemented for all complex values of its three parameters as for ϵ_w although for real values of its three parameters it suffices that the switch ROUNDED is on.

Numerical Evaluation

If the discriminant $g_2^3 - 27g_3^2$ vanishes, the functions Weierstrass1, WeierstrassZeta and Weierstrass_sigma0 become elementary:

$$\begin{aligned}
\wp(u|4b^4/3, 8b^6/27) &= b^2(\mathrm{cosec}^2(bu) - 1/3) \\
\wp(u|4b^4/3, -8b^6/27) &= b^2(\mathrm{cosech}^2(bu) + 1/3) \\
\zeta_w(u|4b^4/3, 8b^6/27) &= b(\cot(bu) + bu/3) \\
\zeta_w(u|4b^4/3, -8b^6/27) &= b(\coth(bu) - bu/3) \\
\sigma(u|4b^4/3, 8b^6/27) &= exp(b^2u^2/6)\sin(bu) \\
\sigma(u|4b^4/3, -8b^6/27) &= exp(-b^2u^2/6)\sinh(bu)
\end{aligned}$$

where b is an arbitrary complex constant chosen for typographical simplicity. In these cases the Weierstrass function becomes singly periodic, the other period becoming infinite.

Laurent Series Expansions

Currently the Weierstrass and Weierstrass Zeta functions may be expanded as power series using the extendible power series package tps. The expansions about zero and the points $m\omega_1 + n\omega_3$ for integer m and n are informative whilst those about other points whilst correct are not. Power series expansion of the Weierstrass function about points where it vanishes (there are two such points in each

fundamental parallelogram) is also of interest. Currently the Weierstrass Sigma functions (described below) cannot be expanded using tps and result in rather inelegant errors.

Currently using the taylor package the expansions of the Weierstrass, Weierstrass Zeta and Weierstrass Sigma about zero and $2m\omega_1 + 2n\omega_3$ are incorrect whilst those about the mid-lattice points $m\omega_1 + n\omega_3$ are formally correct, but not fully simplified.

Other Sigma Functions

Three further Sigma functions are also provided:

- $\sigma_1(u, \omega_1, \omega_3)$ — `weierstrass_sigma1(u,omega1,omega3)`
- $\sigma_2(u, \omega_1, \omega_3)$ — `weierstrass_sigma2(u,omega1,omega3)`
- $\sigma_3(u, \omega_1, \omega_3)$ — `weierstrass_sigma3(u,omega1,omega3)`

These are all even functions, regular everywhere, homogenous of degree zero and doubly quasi-periodic. They are closely related to the theta functions ϑ_2, ϑ_3 and ϑ_4 respectively; but *note the difference in numbering*. For more information on the properties these sigma functions, see Lawden [Law89]; they do not appear in the NIST Digital Library of Mathematical Functions, but are included here for completeness. These three functions have no corresponding versions where the second and third arguments are the lattice invariants g_2 and g_3.

Quasi-Period Factors & Lattice Functions

Ten functions are provided:

- $e_1(\omega_1, \omega_3)$ — `lattice_e1(omega1, omega3);`
- $e_2(\omega_1, \omega_3)$ — `lattice_e2(omega1, omega3);`
- $e_3(\omega_1, \omega_3)$ — `lattice_e3(omega1, omega3);`
- $g_2(\omega_1, \omega_3)$ — `lattice_g2(omega1, omega3);`
- $g_3(\omega_1, \omega_3)$ — `lattice_g3(omega1, omega3);`
- $\Delta(\omega_1, \omega_3)$ — `lattice_delta(omega1, omega3);`
- $\mathrm{G}(\omega_1, \omega_3)$ — `lattice_g(omega1, omega3);`
- $\eta_1(\omega_1, \omega_3)$ — `eta1(omega1, omega3);`

- $\eta_2(\omega_1, \omega_3)$ — `eta2(omega1, omega3);`
- $\eta_3(\omega_1, \omega_3)$ — `eta3(omega1, omega3).`

These are operative when the switches `rounded` and `complex` are ON and their arguments are numerical. The first three are referred to as lattice roots and are related to the invariants g_2, g_3, the discriminant $\Delta = g_2^3 - 27g_3^2$ and a closely related invariant $\mathrm{G} = g_2^3/(27g_3^2)$ of the Weierstrassian elliptic function $\wp$. The lattice roots also appear in the numerical evaluation of the Weierstrass function. These lattice roots satisfy:

$$e_1 + e_2 + e_3 = 0, \qquad g_2 = 2(e_1^2 + e_2^2 + e_3^2), \qquad g_3 = 4e_1e_2e_3.$$

If the discriminant Δ vanishes or equivalently if $\mathrm{G} = 1$, there are at most two distinct lattice roots and the elliptic function degenerates to an elementary one. The advantage of the invariant G is that it is a function of $\tau = \omega_3/\omega_1$ only.

The remaining three functions `eta1`, `eta2` & `eta3` appear in the rules for the quasi-periodicity of the four sigma functions and of the Weierstrassian Zeta function. They are also used in the numerical evaluation of these functions when the switches `rounded` and `complex` are ON. The quasi-period relations are:

$$\begin{aligned}
\zeta_w(u + 2\omega_j) &= \zeta_w(u) + 2\eta_j \\
\sigma(u + 2\omega_j) &= -exp(2\eta_j(u + \omega_j))\sigma(u) \\
\sigma_k(u + 2\omega_j) &= exp(2\eta_j(u + \omega_j))\sigma_k(u) \quad \text{if } j \neq k \\
\sigma_j(u + 2\omega_j) &= -exp(2\eta_j(u + \omega_j))\sigma_j(u) \\
\zeta_w(\omega_j) &= \eta_j \\
\sigma_j(\omega_j) &= 0,
\end{aligned}$$

where the lattice parameters have been omitted for conciseness and $j, k = 1 \ldots 3$. The quasi-period factors satisfy

$$\eta_1 + \eta_2 + \eta_3 = 0, \qquad \eta_1\omega_3 - \eta_3\omega_1 = \eta_2\omega_1 - \eta_1\omega_2 = \eta_3\omega_2 - \eta_2\omega_3 = i\pi/2.$$

Nine functions are provided which depend on the lattice invariants g_2 and g_3 rather than the lattice half-periods:

- $e_1(|g_2, g_3)$ — `lattice1_e1(g2, g3);`
- $e_2(|g_2, g_3)$ — `lattice1_e2(g2, g3);`
- $e_3(|g_2, g_3)$ — `lattice_e3(g2, g3);`
- $\omega_1(|g_2, g_3)$ — `lattice_omega1(g2, g3);`
- $\omega_2(|g_2, g_3)$ — `lattice_omega2(g2, g3);`

- $\omega_3(|g_2, g_3)$ — `lattice_omega3(g2, g3);`
- $\eta_1(|g_2, g_3)$ — `eta_1(g2,g3);`
- $\eta_2(|g_2, g_3)$ — `eta_2(g2,g3);`
- $\eta_3(|g_2, g_3)$ — `eta_3(g2,g3);`

The first three return the lattice roots, the second triple return the three lattice half-periods and the final three return the quasi-period factors. As with the alternative Weierstrass functions which depend on the invariants, these nine functions are distinguished from the similar functions depending on the periods by preceding their arguments with a vertical bar.

As well as the scalar-valued functions discussed above in this section, there are six functions which return a list as their value:

- `lattice_roots(omega1,omega3)` — returns $\{e_1,\ e_2,\ e_3\}$
- `lattice_invariants(omega1,omega3)` — returns $\{g_2,\ g_3,\ \Delta,\ \mathrm{G}\}$;
- `quasi_period_factors(omega1,omega3)` — returns $\{\eta_1,\ \eta_2,\ \eta_3\}$;
- `lattice_generators(g2,g3)` — returns $\{\omega_1,\ \omega_3\}$.
- `lattice1_roots(g2,g3)` — returns $\{e_1,\ e_2,\ e_3\}$
- `quasi_periods(g2,g3)` — returns $\{\eta_1,\ \eta_2,\ \eta_3\}$.

The first three depend on the lattice half-periods ω_1 and ω_3 whilst the fourth, fifth and sixth are functions of the invariants g_2 and g_3. These list-valued functions are actually more efficient than calling the corresponding scalar-valued functions individually. These functions are only useful when the switches `rounded` and `complex` are ON and their arguments are all numerical. Note that the call sequence:

```
lattice_generators(g2,g3);
lattice_invariants(first ws, second ws);
{first ws, second ws};
```

should reproduce the list {g2, g3}, perhaps with small rounding errors. The corresponding sequence with the function `lattice_generators` being called after `lattice_invariants` (and g2 & g3 replaced by w1 & w3), in general, will not produce the same pair of lattice generators since the generators are only defined up to a unimodular bilinear integer transformation.

For details of the algorithm used to calculate the lattice generators from the invariants see the DLMF:NIST chapter on Lattice Calculations.

20.20.10 Inverse Jacobi Elliptic Functions

The following inverses of the 12 Jacobi elliptic functions are available:-

- arcsn(u,k)
- arcdn(u,k)
- arccn(u,k)
- arccd(u,k)
- arcsd(u,k)
- arcnd(u,k)
- arcdc(u,k)
- arcnc(u,k)
- arcsc(u,k)
- arcns(u,k)
- arcds(u,k)
- arccs(u,k)

Thus, for example,

```
jacobisn(arcsn(x, k), k)   --> x
jacobisc(arcsc(x, k), k)   --> x
```

A rule list is provided to simplify these functions for special values of their arguments such $x = 0$, $k = 0$ and $k = 1$, to implement the inverse function simplification formulae illustrated immediately above and for differentiation of these functions with respect to their two arguments.

Note that arccs is not defined to be an odd function of its first argument unlike cs. Instead it is taken to satisfy:

$$\mathrm{arccs}(-x, k) = 2\mathrm{K}(k) - \mathrm{arccs}(x, k).$$

This is analogous to the situation in Reduce for acot where

$$\arctan(-x) = -\arctan(x), \qquad \mathrm{arccot}(-x) = \pi - \mathrm{arccot}(x).$$

This choice means that the range of (real) principal values of arccs is *connected* – it is the open set $(0, 2K(k))$.

When their arguments are *numerical*, these functions will be evaluated numerically if the `rounded` switch is ON. Note that in some cases the result may have an imaginary part even if both arguments are real, hence the switch `complex` is always turned ON temporarily during numerical evaluation and afterwards restored to its original value.

Note also that for arcdn and arcnd a zero value of the modulus k is excluded (since $\mathrm{dn}(x,0) = \mathrm{nd}(x,0) = 1 \quad \forall x$).

As the Jacobi elliptic functions are doubly periodic, their inverse functions are multi-valued. The numerical value returned is the principal value v which lies in the parallelogram in the complex plane whose vertices are given in the table below. Other values of the inverse functions are indicated in the fifth column of the table below where m and n are arbitrary integers.

Function	Quarter Periods		Principal	Other
	p	q	Parallelogram	Values
arcsn	K	$i\mathrm{K}'$	$-(p+q)$, $-p+q$, $p+q$, $p-q$	$2mp + 2nq + (-1)^m v$
arccn	K	$\mathrm{K} + i\mathrm{K}'$	$-q$, q, $2p+q$, $2p-q$	$4mp + 2nq \pm v$
arcdn	$i\mathrm{K}'$	K	0, $2p$, $2(p+q)$, $2q$	$2mq + 4np \pm v$
arcns	K	$i\mathrm{K}'$	$-(p+q)$, $-p+q$, $p+q$, $p-q$	$2mp + 2nq + (-1)^m v$
arcnc	K	$\mathrm{K} + i\mathrm{K}'$	$-q$, q, $2p+q$, $2p-q$	$4mp + 2nq \pm v$
arcnd	$i\mathrm{K}'$	K	0, $2p$, $2(p+q)$, $2q$	$2mq + 4np \pm v$
arccd	K	$i\mathrm{K}'$	$-q$, q, $2p+q$, $2p-q$	$4mp + 2nq \pm v$
arcdc	K	$i\mathrm{K}'$	$-q$, q, $2p+q$, $2p-q$	$4mp + 2nq \pm v$
arcsd	K	$\mathrm{K} + i\mathrm{K}'$	$-(p+q)$, $-p+q$, $p+q$, $p-q$	$2mp + 2nq + (-1)^m v$
arcds	K	$\mathrm{K} + i\mathrm{K}'$	$-(p+q)$, $-p+q$, $p+q$, $p-q$	$2mp + 2nq + (-1)^m v$
arcsc	$i\mathrm{K}'$	K	$-(p+q)$, $-p+q$, $p+q$, $p-q$	$2mq + 2nq + (-1)^n v$
arccs	$i\mathrm{K}'$	K	$-p$, p, $p+2q$, $-p+2q$	$2mq + 2nq + (-1)^n v$

When both arguments are real and $|k| <= 1$ and when there are certain restrictions on the range of the first parameter x (see the table below), then the principal value of the inverse function is real. It lies in the range given in the third column of the table below. (c.f. the inverse trigonometric functions). In these cases the integrand in the defining integral has no branch points and a faster algorithm which uses only real arithmetic is adopted. Other *real* values of the inverse functions are indicated in the fourth column of the table below where n is an arbitrary integer.

Fn	Domain	Principal Value v	Other real values
arcsn:	$\|x\| \le 1$	$-\mathrm{K}(k) \le v \le \mathrm{K}(k)$	$2n\mathrm{K}(k) + (-1)^n v$
arccn:	$\|x\| \le 1$	$0 \le v \le 2\mathrm{K}(k)$	$4n\mathrm{K}(k) \pm v$
arccd:	$\|x\| \le 1$	$0 \le v \le 2\mathrm{K}(k)$	$4n\mathrm{K}(k) \pm v$
arcns:	$\|x\| \ge 1$	$-\mathrm{K}(k) \le v \le \mathrm{K}(k)$ & $v \ne 0$	$2n\mathrm{K}(k) + (-1)^n v$
arcnc:	$\|x\| \ge 1$	$0 \le v \le 2\mathrm{K}(k)$ & $v \ne \mathrm{K}(k)$	$4n\mathrm{K}(k) \pm v$
arcdc:	$\|x\| \ge 1$	$0 \le v \le 2\mathrm{K}(k)$ & $v \ne \mathrm{K}(k)$	$4n\mathrm{K}(k) \pm v$
arcdn:	$k' \le x \le 1$	$0 \le v \le \mathrm{K}(k)$	$2n\mathrm{K}(k) \pm v$
arcnd:	$1 \le x \le 1/k'$	$0 \le v \le \mathrm{K}(k)$	$2n\mathrm{K}(k) \pm v$
arcds:	$\|x\| \ge k'$	$-\mathrm{K}(k) \le v \le \mathrm{K}(k)$ & $v \ne 0$	$2n\mathrm{K}(k) + (-1)^n v$
arcsd:	$\|x\| \le 1/k'$	$-\mathrm{K}(k) \le v \le \mathrm{K}(k)$	$2n\mathrm{K}(k) + (-1)^n v$
arcsc:	$x \in \mathbb{R}$	$-\mathrm{K}(k) < v < \mathrm{K}(k)$	$2n\mathrm{K}(k) + v$
arccs:	$x \in \mathbb{R}$	$0 < v < 2\mathrm{K}(k)$	$2n\mathrm{K}(k) + v$ $(v \ne 0)$

The numerical values of the inverse functions are calculated by expressing them in terms of the symmetric elliptic integral:

$$R_F(x, y, z) = \int_0^\infty 1/\sqrt{(t-x)(t-y)(t-z)}\,\mathrm{d}t.$$

For more details see the DLMF website: Inverse Jacobian Elliptic Functions.

20.20.11 Table of Elliptic Functions and Integrals

Function	Operator
$\mathrm{am}(u,k)$	`jacobiam(u,k)`
$\mathrm{sn}(u,k)$	`jacobisn(u,k)`
$\mathrm{dn}(u,k)$	`jacobidn(u,k)`
$\mathrm{cn}(u,k)$	`jacobicn(u,k)`
$\mathrm{cd}(u,k)$	`jacobicd(u,k)`
$\mathrm{sd}(u,k)$	`jacobisd(u,k)`
$\mathrm{nd}(u,k)$	`jacobind(u,k)`
$\mathrm{dc}(u,k)$	`jacobidc(u,k)`
$\mathrm{nc}(u,k)$	`jacobinc(u,k)`
$\mathrm{sc}(u,k)$	`jacobisc(u,k)`
$\mathrm{ns}(u,k)$	`jacobins(u,k)`
$\mathrm{ds}(u,k)$	`jacobids(u,k)`
$\mathrm{cs}(u,k)$	`jacobics(u,k)`
Inverse Functions of the above:	
$\mathrm{arcsn}(u,k)$	`arcsn(u,k)`
$\mathrm{arccn}(u,k)$	`arccn(u,k)`
$\vdots$	
$\mathrm{arccs}(u,k)$	`arccs(u,k)`
Complete Integral	
(1st kind) $\mathrm{K}(k)$	`ellipticK(k)`
$\mathrm{K}'(k)$	`ellipticK!'(k)`
Incomplete Integral	
(1st kind) $\mathrm{F}(\phi,k)$	`ellipticF(phi,k)`
Complete Integral	
(2nd kind) $\mathrm{E}(k)$	`ellipticE(k)`
$\mathrm{E}'(k)$	`ellipticE!'(k)`
Legendre Incomplete	
Integral (2nd kind) $\mathrm{E}(u,k)$	`ellipticE(u,k)`
Alternative Incomplete	
Integral (2nd kind) $\mathrm{D}(u,k)$	`ellipticD(u,k)`
Jacobi Incomplete	
Integral (2nd kind) $\mathcal{E}(u,k)$	`jacobiE(u,k)`
Jacobi's Zeta $\mathrm{Z}(u,k)$	`jacobiZeta(u,k)`
$\vartheta_1(u,\tau)$	`elliptictheta1(u,tau)`
$\vartheta_2(u,\tau)$	`elliptictheta2(u,tau)`
$\vartheta_3(u,\tau)$	`elliptictheta3(u,tau)`
$\vartheta_4(u,\tau)$	`elliptictheta4(u,tau)`
$\wp(u,\omega_1,\omega_3)$	`weierstrass(u,omega1,omega3)`
$\zeta_w(u,\omega_1,\omega_3)$	`weierstrassZeta(u,omega1,omega3)`
$\sigma(u,\omega_1,\omega_3)$	`weierstrass_sigma(u,omega1,omega3)`
$\sigma_1(u,\omega_1,\omega_3)$	`weierstrass_sigma1(u,omega1,omega3)`
$\sigma_2(u,\omega_1,\omega_3)$	`weierstrass_sigma2(u,omega1,omega3)`
$\sigma_3(u,\omega_1,\omega_3)$	`weierstrass_sigma3(u,omega1,omega3)`
$\wp(u \mid g_2,g_3)$	`weierstrass1(u,g2,g3)`
$\zeta_w(u \mid g_2,g_3)$	`weierstrassZeta1(u,g2,g3)`
$\sigma(u \mid g_2,g_3)$	`weierstrass_sigma0(u,g2,g3)`

20.21 EXCALC: A Differential Geometry Package

EXCALC is designed for easy use by all who are familiar with the calculus of Modern Differential Geometry. The program is currently able to handle scalar-valued exterior forms, vectors and operations between them, as well as non-scalar valued forms (indexed forms). It is thus an ideal tool for studying differential equations, doing calculations in general relativity and field theories, or doing simple things such as calculating the Laplacian of a tensor field for an arbitrary given frame.

Author: Eberhard Schrüfer

Acknowledgments

This program was developed over several years. I would like to express my deep gratitude to Dr. Anthony Hearn for his continuous interest in this work, and especially for his hospitality and support during a visit in 1984/85 at the RAND Corporation, where substantial progress on this package could be achieved. The Heinrich Hertz-Stiftung supported this visit. Many thanks are also due to Drs. F.W. Hehl, University of Cologne, and J.D. McCrea, University College Dublin, for their suggestions and work on testing this program.

20.21.1 Introduction

EXCALC is designed for easy use by all who are familiar with the calculus of Modern Differential Geometry. Its syntax is kept as close as possible to standard textbook notations. Therefore, no great experience in writing computer algebra programs is required. It is almost possible to input to the computer the same as what would have been written down for a hand-calculation. For example, the statement

```
f*x^y + u _| (y^z^x)
```

would be recognized by the program as a formula involving exterior products and an inner product. The program is currently able to handle scalar-valued exterior forms, vectors and operations between them, as well as non-scalar valued forms (indexed forms). With this, it should be an ideal tool for studying differential equations, doing calculations in general relativity and field theories, or doing such simple things as calculating the Laplacian of a tensor field for an arbitrary given frame. With the increasing popularity of this calculus, this program should have an application in almost any field of physics and mathematics.

Since the program is completely embedded in REDUCE, all features and facilities of REDUCE are available in a calculation. Even for those who are not quite comfortable in this calculus, there is a good chance of learning it by just playing with

the program.

This is the last release of version 2. A much extended differential geometry package (which includes complete symbolic index simplification, tensors, mappings, bundles and others) is under development.

Complaints and comments are appreciated and should be send to the author. If the use of this program leads to a publication, this document should be cited, and a copy of the article to the above address would be welcome.

20.21.2 Declarations

Geometrical objects like exterior forms or vectors are introduced to the system by declaration commands. The declarations can appear anywhere in a program, but must, of course, be made prior to the use of the object. Everything that has no declaration is treated as a constant; therefore zero-forms must also be declared.

An exterior form is introduced by

pform ⟨ *declaration*$_1$ ⟩, ⟨ *declaration*$_2$ ⟩, ... ;

where

⟨ *declaration* ⟩ ::= ⟨ *name* ⟩ | ⟨ *list of names* ⟩=⟨ *number* ⟩ | ⟨ *identifier* ⟩ |
⟨ *expression* ⟩
⟨ *name* ⟩ ::= ⟨ *identifier* ⟩ | ⟨ *identifier* ⟩(⟨ *arguments* ⟩)

For example

```
pform u=k,v=4,f=0,w=dim-1;
```

declares `u` to be an exterior form of degree `k`, `v` to be a form of degree 4, `f` to be a form of degree 0 (a function), and `w` to be a form of degree `dim`-1.

If the exterior form should have indices, the declaration would be

```
pform curv(a,b)=2,chris(a,b)=1;
```

The names of the indices are arbitrary.

Exterior forms of the same degree can be grouped into lists to save typing.

```
pform {x,y,z}=0,{rho(k,l),u,v(k)}=1;
```

The declaration of vectors is similar. The command `tvector` takes a list of names.

tvector ⟨ *name*$_1$ ⟩, ⟨ *name*$_2$ ⟩, ... ;

For example, to declare x as a vector and comm as a vector with two indices, one would say

```
tvector x,comm(a,b);
```

If a declaration of an already existing name is made, the old declaration is removed, and the new one is taken.

The exterior degree of a symbol or a general expression can be obtained with the function

exdegree ⟨ *expression* ⟩;

Example:

```
exdegree(u + 3*chris(k,-k));

1
```

20.21.3 Exterior Multiplication

Exterior multiplication between exterior forms is carried out with the nary infix operator ^ (wedge). Factors are ordered according to the usual ordering in REDUCE using the commutation rule for exterior products.

Example 1

```
pform u=1,v=1,w=k;

u^v;

u^v

v^u;

- u^v

u^u;

0

w^u^v;

       k
( - 1)  *u^v^w
```

```
(3*u-a*w)^(w+5*v)^u;

a*(5*u^v^w - u^w^w)
```

It is possible to declare the dimension of the underlying space by

spacedim ⟨ *number* ⟩ | ⟨ *identifier* ⟩;

If an exterior product has a degree higher than the dimension of the space, it is replaced by 0:

```
spacedim 4;

pform u=2,v=3;

u^v;

0
```

20.21.4 Partial Differentiation

Partial differentiation is denoted by the operator `@`. Its capability is the same as the REDUCE `df` operator.

Example 2

```
@(sin x,x);

cos(x)

@(f,x);

0
```

An identifier can be declared to be a function of certain variables. This is done with the command `fdomain`. The following would tell the partial differentiation operator that `f` is a function of the variables `x` and `y` and that `h` is a function of `x`.

```
fdomain f=f(x,y),h=h(x);
```

Applying `@` to `f` and `h` would result in

```
@(x*f,x);
```

```
f + x*@  f
       x

@(h,y);

0
```

The partial derivative symbol can also be an operator with a single argument. It then represents a natural base element of a tangent vector.

Example 3

```
a*@ x + b*@ y;

a*@  + b*@
   x      y
```

20.21.5 Exterior Differentiation

Exterior differentiation of exterior forms is carried out by the operator `d`. Products are normally differentiated out, *i.e.*

```
pform x=0,y=k,z=m;

d(x * y);

x*d y + d x^y

d(r*y);

r*d y

d(x*y^z);

      k
( - 1) *x*y^d z  + x*d y^z + d x^y^z
```

This expansion can be suppressed by the command `noxpnd d`.

```
noxpnd d;

d(y^z);
```

```
d(y^z)
```

To obtain a canonical form for an exterior product when the expansion is switched off, the operator `d` is shifted to the right if it appears in the leftmost place.

```
d y ^ z;

          k
- ( - 1) *y^d z + d(y^z)
```

Expansion is performed again when the command `xpnd d` is executed.

Functions which are implicitly defined by the `fdomain` command are expanded into partial derivatives:

```
pform x=0,y=0,z=0,f=0;

fdomain f=f(x,y);

d f;

@  f*d x + @  f*d y
 x          y
```

If an argument of an implicitly defined function has further dependencies the chain rule will be applied *e.g.*

```
fdomain y=y(z);

d f;

@  f*d x + @  f*@  y*d z
 x          y    z
```

Expansion into partial derivatives can be inhibited by `noxpnd @` and enabled again by `xpnd @`.

The operator is of course aware of the rules that a repeated application always leads to zero and that there is no exterior form of higher degree than the dimension of the space.

```
d d x;
```

```
0

pform u=k;
spacedim k;

d u;

0
```

20.21.6 Inner Product

The inner product between a vector and an exterior form is represented by the diphthong _| (underscore or-bar), which is the notation of many textbooks. If the exterior form is an exterior product, the inner product is carried through any factor.

Example 4

```
pform x=0,y=k,z=m;

tvector u,v;

u _| (x*y^z);

          k
x*(( - 1) *y^u _| z + u _| y^z)
```

In repeated applications of the inner product to the same exterior form the vector arguments are ordered *e.g.*

```
(u+x*v) _| (u _| (3*z));

- 3*u _| v _| z
```

The duality of natural base elements is also known by the system, *i.e.*

```
pform {x,y}=0;

(a*@ x+b*@(y)) _| (3*d x-d y);

3*a - b
```

20.21.7 Lie Derivative

The Lie derivative can be taken between a vector and an exterior form or between two vectors. It is represented by the infix operator `|_` . In the case of Lie differentiating, an exterior form by a vector, the Lie derivative is expressed through inner products and exterior differentiations, *i.e.*

```
pform z=k;

tvector u;

u |_ z;

u _| d z + d(u _| z)
```

If the arguments of the Lie derivative are vectors, the vectors are ordered using the anticommutivity property, and functions (zero forms) are differentiated out.

Example 5

```
tvector u,v;

v |_ u;

- u |_ v

pform x=0,y=0;

(x*u) |_ (y*v);

- u*y*v _| d x + v*x*u _| d y + x*y*u |_ v
```

20.21.8 Hodge-* Duality Operator

The Hodge-* duality operator maps an exterior form of degree `k` to an exterior form of degree `n-k`, where `n` is the dimension of the space. The double application of the operator must lead back to the original exterior form up to a factor. The following example shows how the factor is chosen here

```
spacedim n;
pform x=k;

# # x;
```

```
          2
        (k  + k*n)
( - 1)           *x*sgn
```

The indeterminate sgn in the above example denotes the sign of the determinant of the metric. It can be assigned a value or will be automatically set if more of the metric structure is specified (via **coframe**), *i.e.* it is then set to $g/|g|$, where g is the determinant of the metric. If the Hodge-* operator appears in an exterior product of maximal degree as the leftmost factor, the Hodge-* is shifted to the right according to

```
pform {x,y}=k;

# x ^ y;

          2
        (k  + k*n)
( - 1)           *x^# y
```

More simplifications are performed if a coframe is defined.

20.21.9 Variational Derivative

The function `vardf` returns as its value the variation of a given Lagrangian n-form with respect to a specified exterior form (a field of the Lagrangian). In the shared variable `bndeq!*`, the expression is stored that has to yield zero if integrated over the boundary.

Syntax:

vardf(⟨ *Lagrangian n-form* ⟩,⟨ *exterior form* ⟩)

Example 6

```
spacedim 4;

pform l=4,a=1,j=3;

%Lagrangian of the e.m. field
l:=-1/2*d a ^ # d a - a^# j$

vardf(l,a);
```

```
- (# j + d # d a)          %Maxwell's equations

bndeq!*;

- 'a^# d a                 %Equation at the boundary
```

Restrictions:

In the current implementation, the Lagrangian must be built up by the fields and the operations `d`, `#`, and `@`. Variation with respect to indexed quantities is currently not allowed.

For the calculation of the conserved currents induced by symmetry operators (vector fields), the function `noether` is provided. It has the syntax:

noether(⟨ *Lagrangian n-form* ⟩,⟨ *field* ⟩,⟨ *symmetry generator* ⟩)

Example 7

```
pform l=4,a=1,f=2;

spacedim 4;

l := -1/2*d a^#d a;   %Free Maxwell field;

tvector x;         %An unspecified generator;

noether(l,a,x);

 ( - 2*d(x _| a)^# d a + d a^x _| # d a

   - x _| d a^# d a)/2
```

The above expression would be the canonical energy momentum 3-forms of the Maxwell field, if X is interpreted as a translation;

20.21.10 Handling of Indices

Exterior forms and vectors may have indices. On input, the indices are given as arguments of the object. A positive argument denotes a superscript and a negative argument a subscript. On output, the indexed quantity is displayed two dimensionally if `nat` is on. Indices may be identifiers or numbers.

Example 8

```
pform om(k,l)=m,e(k)=1;

e(k)^e(-l);

 k
e ^e
    l

om(4,-2);

  4
om
   2
```

In the current release, full simplification is performed only if an index range is specified. It is hoped that this restriction can be removed soon. If the index range (the values that the indices can obtain) is specified, the given expression is evaluated for all possible index values, and the summation convention is understood.

Example 9

```
indexrange t,r,ph,z;

pform e(k)=1,s(k,l)=2;

w := e(k)*e(-k);

          t       r        ph       z
w := e *e  + e *e  + e  *e   + e *e
      t       r       ph        z


s(k,l):=e(k)^e(l);

 t t
s    := 0

 r t       t  r
s    := - e ^e

 ph t       t  ph
s     := - e ^e

  .
```

```
.
.
```

If the expression to be evaluated is not an assignment, the values of the expression are displayed as an assignment to an indexed variable with name `ns`. This is done only on output, *i.e.* no actual binding to the variable ns occurs.

```
e(k)^e(l);

  t t
ns    := 0

  r t       t  r
ns    := - e ^e
  .
  .
  .
```

It should be noted, however, that the index positions on the variable ns can sometimes not be uniquely determined by the system (because of possible reorderings in the expression). Generally it is advisable to use assignments to display complicated expressions.

A range can also be assigned to individual index-names. For example, the declaration

```
indexrange {k,l}={x,y,z},{u,v,w}={1,2};
```

would assign to the index identifiers k,l the range values x,y,z and to the index identifiers u,v,w the range values 1,2. The use of an index identifier not listed in previous indexrange statements has the range of the union of all given index ranges.

With the above example of an indexrange statement, the following index evaluations would take place

```
pform w n=0;

w(k)*w(-k);

     x        y        z
w *w   + w *w   + w *w
 x        y        z
```

```
w(u)*w(-u);

     1        2
w *w   + w *w
 1         2

w(r)*w(-r);

     1        2         x         y         z
w *w   + w *w   + w *w   + w *w   + w *w
 1         2         x         y         z
```

In certain cases, one would like to inhibit the summation over specified index names, or at all. For this the command

nosum ⟨ *indexname*$_1$ ⟩, ...;

and the switch `nosum` are available. The command `nosum` has the effect that summation is not performed over those indices which had been listed. The command `renosum` enables summation again. The switch `nosum`, if on, inhibits any summation.

It is possible to declare symmetry properties for an indexed quantity by the command `index_symmetries`. A prototypical example is as follows

```
index_symmetries u(k,l,m,n):
                      symmetric     in {k,l},{m,n}
                      antisymmetric in {{k,l},{m,n}},
                 g(k,l),h(k,l): symmetric;
```

It declares the object `u` symmetric in the first two and last two indices and antisymmetric with respect to commutation of the given index pairs. If an object is completely symmetric or antisymmetric, the indices need not to be given after the corresponding keyword as shown above for `g` and `h`.

If applicable, this command should be issued, since great savings in memory and execution time result. Only strict components are printed.

The commands symmetric and antisymmetric of earlier releases have no effect.

20.21.11 Metric Structures

A metric structure is defined in EXCALC by specifying a set of basis one-forms (the coframe) together with the metric.

Syntax:

coframe ⟨ *identifier* ⟩⟨ *(index*$_1$*)* ⟩=⟨ *expression*$_1$ ⟩,
⟨ *identifier* ⟩⟨ *(index*$_2$*)* ⟩=⟨ *expression*$_2$ ⟩,
.
.
.
⟨ *identifier* ⟩⟨ *(index*$_n$*)* ⟩=⟨ *expression*$_n$ ⟩
with metric ⟨ *name* ⟩=⟨ *expression* ⟩;

This statement automatically sets the dimension of the space and the index range. The clause `with metric` can be omitted if the metric is Euclidean and the shorthand `with signature` ⟨ *diagonal elements* ⟩ can be used in the case of a pseudo-Euclidean metric. The splitting of a metric structure in its metric tensor coefficients and basis one-forms is completely arbitrary including the extremes of an orthonormal frame and a coordinate frame.

Example 10

```
%Polar coframe
coframe e r=d r, e(ph)=r*d ph
  with metric g=e(r)*e(r)+e(ph)*e(ph);

%Same as before
coframe e(r)=d r,e(ph)=r*d(ph);

%A Lorentz coframe
coframe o(t)=d t, o x=d x
  with signature -1,1;

%A lightcone coframe
coframe b(xi)=d xi, b(eta)=d eta
  with metric w=-1/2*(b(xi)*b(eta)+b(eta)*b(xi));

%Polar coordinate basis
coframe e r=d r, e ph=d ph
  with metric g=e r*e r+r**2*e ph*e ph;
```

Individual elements of the metric can be accessed just by calling them with the desired indices. The value of the determinant of the covariant metric is stored in the variable `detm!*`. The metric is not needed for lowering or raising of indices as the system performs this automatically, *i.e.* no matter in what index position values were assigned to an indexed quantity, the values can be retrieved for any index position just by writing the indexed quantity with the desired indices.

Example 11

```
coframe e t=d t,e x=d x,e y=d y
 with signature -1,1,1;

pform f(k,l)=0;

index_symmetries f(k,l): antisymmetric;

f(k,l) := 0$
f(-t,-x):=ex$ f(-x,-y):=b$
on nero;

f(k,-l);

  x
ns     := - ex
    t

  t
ns     := - ex
    x

  y
ns     := - b
    x

  x
ns    := b
    y
```

Any expression containing differentials of the coordinate functions will be transformed into an expression of the basis one-forms.The system also knows how to take the exterior derivative of the basis one-forms.

Example 12 (Spherical coordinates)

```
coframe e(r)=d(r),
        e(th)=r*d(th),
        e(ph)=r*sin(th)*d(ph);

d r^d th;

  r  th
(e ^e  )/r

d(e(th));

  r  th
(e ^e  )/r


pform f=0;

fdomain f=f(r,th,ph);

factor e;

on rat;

%the "gradient" of f in spherical coordinates;
d f;

 r           th               ph
e *@  f + (e  *@   f)/r + (e  *@   f)/(r*sin(th))
    r           th              ph
```

The frame dual to the frame defined by the `coframe` command can be introduced by **frame** command.

frame ⟨ *identifier* ⟩;

This command causes the dual property to be recognized, and the tangent vectors of the coordinate functions are replaced by the frame basis vectors.

Example 13

```
%Cylindrical coframe;
coframe b r=d r,b ph=r*d ph,e z=d z;

frame x;
```

```
on nero;

x(-k) _| b(l);

    r
ns    := 1
  r

     ph
ns       := 1
  ph

    z
ns    := 1
  z

%The commutator of the dual frame;
x(-k) |_ x(-l);


ns     := x  /r
  ph r     ph

ns     := ( - x  )/r
  r ph          ph

%i.e. it is not a coordinate base;
```

As a convenience, the frames can be displayed at any point in a program by the command `displayframe;`.

The Hodge-* duality operator returns the explicitly constructed dual element if applied to coframe base elements. The metric is properly taken into account.

The total antisymmetric Levi-Cevita tensor `eps` is also available. The value of `eps` with an even permutation of the indices in a covariant position is taken to be +1.

20.21.12 Riemannian Connections

The command `riemannconx` is provided for calculating the connection 1 forms. The values are stored on the name given to `riemannconx`. This command is far more efficient than calculating the connection from the differential of the basis one-forms and using inner products.

Example 14(Calculate the connection 1-form and curvature 2-form on S(2))

```
coframe e th=r*d th,e ph=r*sin(th)*d ph;

riemannconx om;

om(k,-l);           %Display the connection forms;

  th
ns      := 0
     th

  ph         ph
ns      := (e  *cos(th))/(sin(th)*r)
     th

  th            ph
ns      := ( - e  *cos(th))/(sin(th)*r)
     ph

  ph
ns      := 0
     ph

pform curv(k,l)=2;


curv(k,-l):=d om(k,-l) + om(k,-m)^om(m-l);
              %The curvature forms

    th
curv      := 0
       th

    ph            th  ph   2
curv      := ( - e  ^e  )/r
       th
            %Of course it was a sphere with
            %radius R.

    th          th  ph   2
curv      := (e  ^e  )/r
       ph
```

```
     ph
curv      := 0
       ph
```

20.21.13 Killing Vectors

The command `killing_vector` is provided for calculating the determining system of partial differential equations of Killing vectors for a given metric structure provided by the coframe statement. The result is a list where the first entry is a vector constructed from the identifier given to the command and the second entry consists of a list of partial differential equations for the coefficients of this vector.

Example 15 (Calculate the determining pde's for a Killing vector of S(2))

```
coframe e th = d th,e ph = sin th*d ph;

killing_vector u;
```

```
      ph         th
{@   *u    + @   *u   ,
  ph          th

       th
 {@   (u   ),
   th

       ph          2         th
  @   (u   )*sin(th)   + @   (u   ),
   th                     ph

           th         ph
  cos(th)*u    + @   (u   )*sin(th)}}
                   ph
```

20.21.14 Ordering and Structuring

The ordering of an exterior form or vector can be changed by the command `forder`. In an expression, the first identifier or kernel in the arguments of `forder` is ordered ahead of the second, and so on, and ordered ahead of all not

appearing as arguments. This ordering is done on the internal level and not only on output. The execution of this statement can therefore have tremendous effects on computation time and memory requirements. `remforder` brings back standard ordering for those elements that are listed as arguments.

An expression can be put in a more structured form by renaming a subexpression. This is done with the command `KEEP` which has the syntax

keep ⟨ *name*$_1$ ⟩=⟨ *expression*$_1$ ⟩,⟨ *name*$_2$ ⟩=⟨ *expression*$_2$ ⟩, ...

The effect is that rules are set up for simplifying ⟨ *name* ⟩ without introducing its definition in an expression. In an expression the system also tries by reordering to generate as many instances of ⟨ *name* ⟩ as possible.

Example 16

```
pform x=0,y=0,z=0,f=0,j=3;

keep j=d x^d y^d z;

j;

j

d j;

0

j^d x;

0

fdomain f=f(x);

d f^d y^d z;

@  f*j
 x
```

The capabilities of `keep` are currently very limited. Only exterior products should occur as righthand sides in `keep`.

20.21.15 Summary of Operators and Commands

Table 20.16 summarizes EXCALC commands and the page number they are defined on.

^	Exterior Multiplication	731
@	Partial Differentiation	732
@	Tangent Vector	733
#	Hodge-* Operator	736
_|	Inner Product	735
|_	Lie Derivative	736
coframe	Declaration of a coframe	742
d	Exterior differentiation	733
displayframe	Displays the frame	745
eps	Levi-Civita tensor	745
exdegree	Calculates the exterior degree of an expression	731
fdomain	Declaration of implicit dependencies	732
forder	Ordering command	747
frame	Declares the frame dual to the coframe	744
indexrange	Declaration of indices	739
index_symmetries	Declares arbitrary index symmetry properties	741
keep	Structuring command	748
killing_vector	Structuring command	747
metric	Clause of COFRAME to specify a metric	742
noether	Calculates the Noether current	738
nosum	Inhibits summation convention	741
noxpnd d	Inhibits the use of product rule for d	733
noxpnd @	Inhibits expansion into partial derivatives	734
pform	Declaration of exterior forms	730
remforder	Clears ordering	748
renosum	Enables summation convention	741
riemannconx	Calculation of a Riemannian Connection	745
signature	Clause of coframe to specify a pseudo-Euclidean metric	742
spacedim	Command to set the dimension of a space	732
tvector	declaration of vectors	730
vardf	Variational derivative	737
xpnd d	Enables the use of product rule for d (default)	734
xpnd @	Enables expansion into partial derivatives (default)	734

Table 20.16: EXCALC Command Summary

20.21.16 Examples

The following examples should illustrate the use of EXCALC. It is not intended to show the most efficient or most elegant way of stating the problems; rather the variety of syntactic constructs are exemplified. The examples are on a test file distributed with EXCALC.

```
% Problem:
% --------
% Calculate the PDE's for the isovector of the heat equation.
%         (c.f. B.K. Harrison, f.B. Estabrook,
%          "Geometric Approach...",
%          J. Math. Phys. 12, 653, 1971)

% The heat equation @   psi = @  psi is equivalent to the set
%                    xx        t

%  of exterior equations (with u=@ psi, y=@ psi):
%                                  T        x


pform {psi,u,x,y,t}=0,a=1,{da,b}=2;

a := d psi - u*d t - y*d x;

da := - d u^d t - d y^d x;

b := u*d x^d t - d y^d t;


% Now calculate the PDE's for the isovector.

tvector v;

pform {vpsi,vt,vu,vx,vy}=0;
fdomain vpsi=vpsi(psi,t,u,x,y),
        vt=vt(psi,t,u,x,y),vu=vu(psi,t,u,x,y),
        vx=vx(psi,t,u,x,y),vy=vy(psi,t,u,x,y);

v := vpsi*@ psi + vt*@ t + vu*@ u + vx*@ x + vy*@ y;


factor d;
on rat;

i1 := v |_ a - l*a;

pform o=1;
```

```
o := ot*d t + ox*d x + ou*d u + oy*d y;

fdomain f=f(psi,t,u,x,y);

i11 := v _| d a - l*a + d f;

let vx=-@(f,y),vt=-@(f,u),vu=@(f,t)+u*@(f,psi),
    vy=@(f,x)+y*@(f,psi),vpsi=f-u*@(f,u)-y*@(f,y);

factor ^;

i2 := v |_ b - xi*b - o^a + zeta*da;

let ou=0,oy=@(f,u,psi),ox=-u*@(f,u,psi),
    ot=@(f,x,psi)+u*@(f,y,psi)+y*@(f,psi,psi);

i2;

let zeta=-@(f,u,x)-@(f,u,y)*u-@(f,u,psi)*y;

i2;

let xi=-@(f,t,u)-u*@(f,u,psi)+@(f,x,y)
       +u*@(f,y,y)+y*@(f,y,psi)+@(f,psi);

i2;

let @(f,u,u)=0;

i2;      % These PDE's have to be solved.


clear a,da,b,v,i1,i11,o,i2,xi,t;
remfdomain f,vpsi,vt,vu,vx,vy;
clear @(f,u,u);


% Problem:
% --------
% Calculate the integrability conditions for the
% system of PDE's:
% (c.f. B.F. Schutz,
% "Geometrical Methods of Mathematical Physics"
% Cambridge University Press, 1984, p. 156)


% @ z /@ x + a1*z  + b1*z  = c1
%    1           1       2
```

```
% @ z /@ y + a2*z  + b2*z  = c2
%    1           1       2

% @ z /@ x + f1*z  + g1*z  = h1
%    2           1       2

% @ z /@ y + f2*z  + g2*z  = h2
%    2           1       2      ;


pform w(k)=1,integ(k)=4,{z(k),x,y}=0,{a,b,c,f,g,h}=1,
      {a1,a2,b1,b2,c1,c2,f1,f2,g1,g2,h1,h2}=0;

fdomain  a1=a1(x,y),a2=a2(x,y),b1=b1(x,y),b2=b2(x,y),
         c1=c1(x,y),c2=c2(x,y),f1=f1(x,y),f2=f2(x,y),
         g1=g1(x,y),g2=g2(x,y),h1=h1(x,y),h2=h2(x,y);


a:=a1*d x+a2*d y$
b:=b1*d x+b2*d y$
c:=c1*d x+c2*d y$
f:=f1*d x+f2*d y$
g:=g1*d x+g2*d y$
h:=h1*d x+h2*d y$

% The equivalent exterior system:
factor d;
w(1) := d z(-1) + z(-1)*a + z(-2)*b - c;
w(2) := d z(-2) + z(-1)*f + z(-2)*g - h;
indexrange 1,2;
factor z;
% The integrability conditions:

integ(k) := d w(k) ^ w(1) ^ w(2);

clear a,b,c,f,g,h,x,y,w(k),integ(k),z(k);
remfdomain a1,a2,b1,c1,c2,f1,f2,g1,g2,h1,h2;

% Problem:
% --------
% Calculate the PDE's for the generators of the
% d-theta symmetries of the Lagrangian system of the
% planar Kepler problem.
% c.f. W.Sarlet, F.Cantrijn, Siam Review 23, 467, 1981
% Verify that time translation is a d-theta symmetry
% and calculate the corresponding integral.

pform {t,q(k),v(k),lam(k),tau,xi(k),eta(k)}=0,theta=1,f=0,
      {l,glq(k),glv(k),glt}=0;
```

```
tvector gam,y;

indexrange 1,2;

fdomain tau=tau(t,q(k),v(k)),xi=xi(t,q(k),v(k)),
        f=f(t,q(k),v(k));

l := 1/2*(v(1)**2 + v(2)**2) + m/r$      % The Lagrangian.

pform r=0;
fdomain r=r(q(k));
let @(r,q 1)=q(1)/r,@(r,q 2)=q(2)/r,q(1)**2+q(2)**2=r**2;

lam(k) := -m*q(k)/r;                            % The force.

gam := @ t + v(k)*@(q(k)) + lam(k)*@(v(k))$

eta(k) := gam _| d xi(k) - v(k)*gam _| d tau$

% Symmetry generator.
y  := tau*@ t + xi(k)*@(q(k)) + eta(k)*@(v(k))$

theta := l*d t + @(l,v(k))*(d q(k) - v(k)*d t)$

factor @;

s := y |_ theta - d f$

glq(k) := @(q k) _| s;
glv(k) := @(v k) _| s;
glt := @(t) _| s;

% Translation in time must generate a symmetry.
xi(k) := 0;
tau := 1;

glq k := glq k;
glv k := glv k;
glt;

% The corresponding integral is of course the energy.
integ := - y _| theta;


clear l,lam k,gam,eta k,y,theta,s,glq k,glv k,glt,t,
      q k,v k,tau,xi k;
remfdomain r,f,tau,xi;
```

```
% Problem:
% --------
% Calculate the "gradient" and "Laplacian" of a function
% and the "curl" and "divergence" of a one-form in
% elliptic coordinates.


coframe e u = sqrt(cosh(v)**2 - sin(u)**2)*d u,
        e v = sqrt(cosh(v)**2 - sin(u)**2)*d v,
        e phi = cos u*sinh v*d phi;

pform f=0;

fdomain f=f(u,v,phi);

factor e,^;
on rat,gcd;
order cosh v, sin u;
% The gradient:
d f;

factor @;
% The Laplacian:
# d # d f;

% Another way of calculating the Laplacian:
-#vardf(1/2*d f^#d f,f);

remfac @;

% Now calculate the "curl" and the "divergence"
% of a one-form.

pform w=1,a(k)=0;

fdomain a=a(u,v,phi);

w := a(-k)*e k;
% The curl:
x := # d w;

factor @;
% The divergence:
y := # d # w;


remfac @;
clear x,y,w,u,v,phi,e k,a k;
remfdomain a,f;
```

```
% Problem:
% --------
% Calculate in a spherical coordinate system
% the Navier Stokes equations.

coframe e r=d r, e theta =r*d theta,
        e phi = r*sin theta *d phi;
frame x;

fdomain v=v(t,r,theta,phi),p=p(r,theta,phi);

pform v(k)=0,p=0,w=1;

% We first calculate the convective derivative.

w := v(-k)*e(k)$

factor e; on rat;

cdv := @(w,t) + (v(k)*x(-k)) |_ w - 1/2*d(v(k)*v(-k));

%next we calculate the viscous terms;

visc := nu*(d#d# w - #d#d w) + mu*d#d# w;

% Finally we add the pressure term and print the components
% of the whole equation.

pform nasteq=1,nast(k)=0;

nasteq := cdv - visc + 1/rho*d p$

factor @;

nast(-k) := x(-k) _| nasteq;

remfac @,e;

clear v k,x k,nast k,cdv,visc,p,w,nasteq,e k;
remfdomain p,v;


% Problem:
% --------
% Calculate from the Lagrangian of a vibrating rod
% the equation of motion and show that the invariance
% under time translation leads to a conserved current.
```

```
pform {y,x,t,q,j}=0,lagr=2;

fdomain y=y(x,t),q=q(x),j=j(x);

factor ^;

lagr := 1/2*(rho*q*@(y,t)**2 - e*j*@(y,x,x)**2)*d x^d t;

vardf(lagr,y);

% The Lagrangian does not explicitly depend on time;
% therefore the vector field @ t generates a symmetry.
% The conserved current is

pform c=1;
factor d;

c := noether(lagr,y,@ t);

% The exterior derivative of this must be zero or a multiple
% of the equation of motion (weak conservation law)
% to be a conserved current.

remfac d;

d c;

% i.e. it is a multiple of the equation of motion.

clear lagr,c,j,y,q;
remfdomain y,q,j;

% Problem:
% --------
% Show that the metric structure given by Eguchi and Hanson
% induces a self-dual curvature.
% c.f. T. Eguchi, P.B. Gilkey, A.J. Hanson,
% "Gravitation, Gauge Theories and Differential Geometry",
% Physics Reports 66, 213, 1980

for all x let cos(x)**2=1-sin(x)**2;

pform f=0,g=0;
fdomain f=f(r), g=g(r);

coframe   o(r) = f*d r,
      o(theta) = (r/2)*(sin(psi)*d theta
                  - sin(theta)*cos(psi)*d phi),
```

```
        o(phi) = (r/2)*(-cos(psi)*d theta
                  - sin(theta)*sin(psi)*d phi),
        o(psi) = (r/2)*g*(d psi + cos(theta)*d phi);

frame e;


pform gamma(a,b)=1,curv2(a,b)=2;
index_symmetries gamma(a,b),curv2(a,b): antisymmetric;

factor o;

gamma(-a,-b) := -(1/2)*( e(-a) _| (e(-c) _| (d o(-b)))
                        -e(-b) _| (e(-a) _| (d o(-c)))
                        +e(-c) _| (e(-b) _| (d o(-a))) )
                      *o(c)$


curv2(-a,b) := d gamma(-a,b) + gamma(-c,b)^gamma(-a,c)$

let f=1/g,g=sqrt(1-(a/r)**4);

pform chck(k,l)=2;
index_symmetries chck(k,l): antisymmetric;

% The following has to be zero for a self-dual curvature.

chck(k,l) := 1/2*eps(k,l,m,n)*curv2(-m,-n) + curv2(k,l);

clear gamma(a,b),curv2(a,b),f,g,chck(a,b),o(k),e(k),
      r,phi,psi;
remfdomain f,g;

% Example:
% -------
% 6-dimensional FRW model with quadratic curvature terms in
% the Lagrangian (Lanczos and Gauss-Bonnet terms).
% cf. Henriques, Nuclear Physics, B277, 621 (1986)

for all x let cos(x)**2+sin(x)**2=1;

pform {r,s}=0;
fdomain r=r(t),s=s(t);

coframe o(t) = d t,
        o(1) = r*d u/(1 + k*(u**2)/4),
        o(2) = r*u*d theta/(1 + k*(u**2)/4),
        o(3) = r*u*sin(theta)*d phi/(1 + k*(u**2)/4),
        o(4) = s*d v1,
```

```
        o(5) = s*sin(v1)*d v2
 with metric g =-o(t)*o(t)+o(1)*o(1)+o(2)*o(2)+o(3)*o(3)
               +o(4)*o(4)+o(5)*o(5);

frame e;

on nero; factor o,^;

riemannconx om;

pform curv(k,l)=2,{riemann(a,b,c,d),ricci(a,b),riccisc}=0;

index_symmetries
       curv(k,l): antisymmetric,
       riemann(k,l,m,n): antisymmetric in {k,l},{m,n}
                         symmetric in {{k,l},{m,n}},
       ricci(k,l): symmetric;

curv(k,l) := d om(k,l) + om(k,-m)^om(m,l);

riemann(a,b,c,d) := e(d) _| (e (c) _| curv(a,b));

% The rest is done in the Ricci calculus language,

ricci(-a,-b) := riemann(c,-a,-d,-b)*g(-c,d);

riccisc := ricci(-a,-b)*g(a,b);

pform {laglanc,inv1,inv2} = 0;

index_symmetries riemc3(k,l),riemri(k,l),
                 hlang(k,l),einst(k,l): symmetric;

pform {riemc3(i,j),riemri(i,j)}=0;

riemc3(-i,-j) := riemann(-i,-k,-l,-m)*riemann(-j,k,l,m)$
inv1 := riemc3(-i,-j)*g(i,j);
riemri(-i,-j) := 2*riemann(-i,-k,-j,-l)*ricci(k,l)$
inv2 := ricci(-a,-b)*ricci(a,b);
laglanc := (1/2)*(inv1 - 4*inv2 + riccisc**2);


pform {einst(a,b),hlang(a,b)}=0;

hlang(-i,-j) := 2*(riemc3(-i,-j) - riemri(-i,-j) -
                   2*ricci(-i,-k)*ricci(-j,K) +
                   riccisc*ricci(-i,-j) -
                   (1/2)*laglanc*g(-i,-j));
```

```
% The complete Einstein tensor:

einst(-i,-j) := (ricci(-i,-j) - (1/2)*riccisc*g(-i,-j))*alp1
                 + hlang(-i,-j)*alp2$

alp1 := 1$
factor alp2;

einst(-i,-j) := einst(-i,-j);

clear o(k),e(k),riemc3(i,j),riemri(i,j),curv(k,l),
      riemann(a,b,c,d),ricci(a,b),riccisc,t,u,v1,v2,
      theta,phi,r,om(k,l),einst(a,b),hlang(a,b);

remfdomain r,s;

% Problem:
% --------
% Calculate for a given coframe and given torsion the
% Riemannian part and the torsion induced part of the
% connection.  Calculate the curvature.

% For a more elaborate example see:
% E.Schruefer, F.W. Hehl, J.D. McCrea,
% "Application of the REDUCE package EXCALC to the Poincare
%  gauge field theory of gravity",
% GRG Journal, vol. 19, (1988)  197--218

pform {ff, gg}=0;

fdomain ff=ff(r), gg=gg(r);

coframe o(4) = d u + 2*b0*cos(theta)*d phi,
        o(1) = ff*(d u + 2*b0*cos(theta)*d phi) + d r,
        o(2) = gg*d theta,
        o(3) = gg*sin(theta)*d phi
 with metric g = -o(4)*o(1)-o(4)*o(1)+o(2)*o(2)+o(3)*o(3);

frame e;

pform {tor(a),gwt(a)}=2,gamma(a,b)=1,
      {u1,u3,u5}=0;

index_symmetries gamma(a,b): antisymmetric;

fdomain u1=u1(r),u3=u3(r),u5=u5(r);

tor(4) := 0$
```

```
tor(1) := -u5*o(4)^o(1) - 2*u3*o(2)^o(3)$

tor(2) := u1*o(4)^o(2) + u3*o(4)^o(3)$

tor(3) := u1*o(4)^o(3) - u3*o(4)^o(2)$

gwt(-a) := d o(-a) - tor(-a)$

% The following is the combined connection.
% The Riemannian part could have equally well been
% calculated by the RIEMANNCONX statement.

gamma(-a,-b) := (1/2)*( e(-b) _| (e(-c) _| gwt(-a))
                      +e(-c) _| (e(-a) _| gwt(-b))
                      -e(-a) _| (e(-b) _| gwt(-c)) )*o(c);

pform curv(a,b)=2;
index_symmetries curv(a,b): antisymmetric;
factor ^;

curv(-a,b) := d gamma(-a,b) + gamma(-c,b)^gamma(-a,c);

clear o(k),e(k),curv(a,b),gamma(a,b),theta,phi,
      x,y,z,r,s,t,u,v,p,q,c,cs;
remfdomain u1,u3,u5,ff,gg;

showtime;
end;
```

20.22 FIDE: Finite Difference Method for Partial Differential Equations

This package performs automation of the process of numerically solving partial differential equations systems (PDES) by means of computer algebra. For PDES solving, the finite difference method is applied. The computer algebra system REDUCE and the numerical programming language FORTRAN are used in the presented methodology. The main aim of this methodology is to speed up the process of preparing numerical programs for solving PDES. This process is quite often, especially for complicated systems, a tedious and time consuming task.

Author: Richard Liska

20.22.1 Abstract

The FIDE package performs automation of the process of numerical solving partial differential equations systems (PDES) by means of computer algebra. For PDES solving finite difference method is applied. The computer algebra system REDUCE and the numerical programming language FORTRAN are used in the presented methodology. The main aim of this methodology is to speed up the process of preparing numerical programs for solving PDES. This process is quite often, especially for complicated systems, a tedious and time consuming task. In the process one can find several stages in which computer algebra can be used for performing routine analytical calculations, namely: transforming differential equations into different coordinate systems, discretization of differential equations, analysis of difference schemes and generation of numerical programs. The FIDE package consists of the following modules:

EXPRES for transforming PDES into any orthogonal coordinate system.

IIMET for discretization of PDES by integro-interpolation method.

APPROX for determining the order of approximation of difference scheme.

CHARPOL for calculation of amplification matrix and characteristic polynomial of difference scheme, which are needed in Fourier stability analysis.

HURWP for polynomial roots locating necessary in verifying the von Neumann stability condition.

LINBAND for generating the block of FORTRAN code, which solves a system of linear algebraic equations with band matrix appearing quite often in difference schemes.

Version 1.1 of the FIDE package is the result of porting FIDE package to REDUCE 3.4. In comparison with Version 1.0 some features has been changed in the LINBAND module (possibility to interface several numerical libraries).

For reference, see [LD90].

20.22.2 EXPRES

A Module for Transforming Differential Operators and Equations into an Arbitrary Orthogonal Coordinate System

This module makes it possible to express various scalar, vector, and tensor differential equations in any orthogonal coordinate system. All transformations needed are executed automatically according to the coordinate system given by the user. The module was implemented according to the similar MACSYMA module from [1].

The specification of the coordinate system

The coordinate system is specified using the following statement:

```
SCALEFACTORS <d>,<tr 1>,...,<tr d>,<cor 1>,...,<cor d>;
<d> ::= 2 | 3                coordinate system dimension
<tr i> ::= "algebraic expression" the expression of the
                             i-th Cartesian coordinate
                             in new coordinates
<cor i> ::= "identifier"     the i-th new coordinate
```

All evaluated quantities are transformed into the coordinate system set by the last SCALEFACTORS statement. By default, if this statement is not applied, the three-dimensional Cartesian coordinate system is employed. During the evaluation of SCALEFACTORS statement the metric coefficients, i.e. scale factors SF(i), of a defined coordinate system are computed and printed. If the WRCHRI switch is ON, then the nonzero Christoffel symbols of the coordinate system are printed too. By default the WRCHRI switch is OFF.

The declaration of tensor quantities

Tensor quantities are represented by identifiers. The VECTORS declaration declares the identifiers as vectors, the DYADS declaration declares the identifiers as dyads. i.e. two-dimensional tensors, and the TENSOR declaration declares the identifiers as tensor variables. The declarations have the following syntax:

```
<declaration> <id 1>,<id 2>,...,<id n>;
<declaration> ::= VECTORS | DYADS | TENSOR
<id i> ::= "identifier"
```

The value of the identifier V declared as vector in the two-dimensional coordinate system is (V(1), V(2)), where V(i) are the components of vector V. The value of the identifier T declared as a dyad is ((T(1,1), T(1,2)), (T(2,1), T(2,2))). The value of the tensor variable can be any tensor (see below). Tensor variables can be used only for a single coordinate system, after the coordinate system redefining by a new SCALEFACTORS statement, the tensor variables have to be re-defined using the assigning statement.

New infix operators

For four different products between the tensor quantities, new infix operators have been introduced (in the explaining examples, a two-dimensional coordinate system, vectors U, V, and dyads T, W are considered):

```
. - scalar product         U.V = U(1)*V(1)+U(2)*V(2)
? - vector product         U?V = U(1)*V(2)-U(2)*V(1)
& - outer product          U&V = ((U(1)*V(1),U(1)*V(2)),
                                  (U(2)*V(1),U(2)*V(2)))
# - double scalar product T#W = T(1,1)*W(1,1)+
              T(1,2)*W(1,2)+T(2,1)*W(2,1)+T(2,2)*W(2,2)
```

The other usual arithmetic infix operators +, -, *, ** can be used in all situations that have sense (e.g. vector addition, a multiplication of a tensor by a scalar, etc.).

New prefix operators

New prefix operators have been introduced to express tensor quantities in its components and the differential operators over the tensor quantities:

VECT - the explicit expression of a vector in its components

DYAD - the explicit expression of a dyad in its components

GRAD - differential operator of gradient

DIV - differential operator of divergence

LAPL - Laplace's differential operator

CURL - differential operator of curl

DIRDF - differential operator of the derivative in direction (1st argument is the directional vector)

The results of the differential operators are written using the DIFF operator. DIFF(<scalar>,<cor i>) expresses the derivative of <scalar> with respect to the coordinate <cor i>. This operator is not further simplified. If the user wants to make it simpler as common derivatives, he performs the following declaration:

```
FOR ALL X,Y LET DIFF(X,Y) = DF(X,Y);   .
```

Then, however, we must realize that if the scalars or tensor quantities do not directly explicitly depend on the coordinates, their dependencies have to be declared using the DEPEND statements, otherwise the derivative will be evaluated to zero. The dependence of all vector or dyadic components (as dependence of the name of vector or dyad) has to appear before VECTORS or DYADS declarations, otherwise after these declarations one has to declare the dependencies of all components. For formulating the explicit derivatives of tensor expressions, the differentiation operator DF can be used (e.g. the differentiation of a vector in its components).

Tensor expressions

Tensor expressions are the input into the EXPRES module and can have a variety of forms. The output is then the formulation of the given tensor expression in the specified coordinate system. The most general form of a tensor expression <tensor> is described as follows (the conditions (d=i) represent the limitation on the dimension of the coordinate system equalling i):

```
<tensor> ::= <scalar> | <vector> | <dyad>
<scalar> ::= "algebraic expression, can contain <scalars>" |
             "tensor variable with scalar value" |
             <vector 1>.<vector 2> | <dyad 1>#<dyad 2> |
             (d=2)<vector 1>?<vector 2> | DIV <vector> |
             LAPL <scalar> | (d=2) ROT <vector> |
             DIRDF(<vector>,<scalar>)
<vector> ::= "identifier declared by VECTORS statement" |
             "tensor variable with vector value" |
             VECT(<scalar 1>,...,<scalar d>) | -<vector> |
             <vector 1>+<vector 2> | <vector 1>-<vector 2> |
             <scalar>*<vector> | <vector>/<scalar> |
             <dyad>.<vector> | <vector>.<dyad> | (d=3)
             <vector 1>?<vector 2> | (d=2) <vector>?<dyad> |
             (d=2) <dyad>?<vector> | GRAD <scalar> |
             DIV <dyad> | LAPL <vector> | (d=3) ROT
             <vector> | DIRDF(<vector 1>,<vector 2>) |
             DF(<vector>,"usual further arguments")
<dyad>   ::= "identifier declared by DYADS statement" |
```

```
"tensor variable with dyadic value" |
DYAD((<scalar 11>,...,<scalar 1d>),...,
(<scalar d1>,...,<scalar dd>)) | -<dyad> |
<dyad 1>+<dyad 2> | <dyad 1>-<dyad 2> |
<scalar>*<dyad> | <dyad>/<scalar> |
<dyad 1>.<dyad 2> | <vector 1>&<vector 2> |
(d=3) <vector>?<dyad> | (d=3) <dyad>?<vector> |
GRAD <vector> |
DF(<dyad>,"usual further arguments")
```

Assigning statement

The assigning statement for tensor variables has a usual syntax, namely:

```
<tensor variable> := <tensor>
<tensor variable> ::= "identifier declared TENSOR"  .
```

The assigning statement assigns the tensor variable the value of the given tensor expression, formulated in the given coordinate system. After a change of the coordinate system, the tensor variables have to be redefined.

For reference, see [Wir80].

20.22.3 IIMET

A Module for Discretizing the Systems of Partial Differential Equations

This program module makes it possible to discretize the specified system of partial differential equations using the integro-interpolation method, minimizing the number of the used interpolations in each independent variable. It can be used for non-linear systems and vector or tensor variables as well. The user specifies the way of discretizing individual terms of differential equations, controls the discretization and obtains various difference schemes according to his own wish.

Specification of the coordinates and the indices corresponding to them

The independent variables of differential equations will be called coordinates. The names of the coordinates and the indices that will correspond to the particular coordinates in the difference scheme are defined using the COORDINATES statement:

```
COORDINATES  <coordinate 1>{,<coordinate i>} [ INTO
          <index 1>{,<index i>}];
<coordinate i> ::= "identifier" - name of coordinate
<index i> ::= "identifier"      - name of index
```

This statement specifies that the <coordinate i> will correspond to the <index i>. A new COORDINATES statement cancels the definitions given by the preceding COORDINATES statement. If the part [INTO ...] is not included in the statement, the statement assigns the coordinates the indices I, J, K, L, M, N, respectively. If it is included, the number of coordinates and the number of indices should be the same.

2.2 Difference grids

In the discretization, orthogonal difference grids are employed. In addition to the basic grid, called the integer one, there is another, the half-integer grid in each coordinate, whose cellular boundary points lie in the centers of the cells of the integer grid. The designation of the cellular separating points and centers is determined by the CENTERGRID switch: if it is ON and the index in the given coordinate is I, the centers of the grid cells are designated by indices I, I + 1,..., and the boundary points of the cells by indices I + 1/2,..., if, on the contrary, the switch is OFF, the cellular centers are designated by indices I + 1/2,..., and the boundary points by indices I, I + 1,... (see Fig. 2.1).

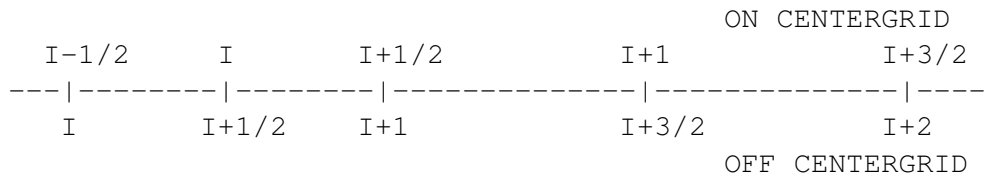


Figure 2.1 Types of grid

In the case of ON CENTERGRID, the indices i,i+1,i-1... thus designate the centers of the cells of the integer grid and the boundary points of the cells of the half-integer grid, and, similarly, in the case of OFF CENTERGRID, the boundaries of the cells of the integer grid and the central points of the half-integer grid. The meaning of the integer and half-integer grids depends on the CENTERGRID switch in the described way. After the package is loaded, the CENTERGRID is ON. Obviously, this switch is significant only for non-uniform grids with a variable size of each cell. The grids can be uniform, i.e. with a constant cell size - the step of the grid. The following statement:

```
GRID UNIFORM,<coordinate>{,<coordinate>};
```

defines uniform grids in all coordinates occurring in it. Those coordinates that do not occur in the GRID UNIFORM statement are supposed to have non-uniform grids. In the outputs, the grid step is designated by the identifier that is made by putting the character H before the name of the coordinate. For a uniform grid,

this identifier (e.g. for the coordinate X the grid step HX) has the meaning of a step of an integer or half-integer grids that are identical. For a non-uniform grid, this identifier is an operator and has the meaning of a step of an integer grid, i.e. the length of a cell whose center (in the case of ON CENTERGRID) or beginning (in the case of OFF CENTERGRID) is designated by a single argument of this operator. For each coordinate s designated by the identifier i, this step of the integer non-uniform grid is defined as follows:

```
Hs(i+j) = s(i+j+1/2) - s(i+j-1/2)   at ON CENTERGRID
Hs(i+j) = s(i+j+1) - s(i+j)         at OFF CENTERGRID
```

for all integers j (s(k) designates the value of the coordinate s in the cellular boundary point subscripted with the index k). The steps of the half-integer non-uniform grid are not applied in outputs.

Declaring the dependence of functions on coordinates

In the system of partial differential equations, two types of functions, in other words dependent variables can occur: namely, the given functions, whose values are known before the given system is solved, and the sought functions, whose values are not available until the system of equations is solved. The functions can be scalar, vector, or tensor, for vector or tensor functions the EXPRES module has to be applied at the same time. The names of the functions employed in the given system and their dependence on the coordinates are specified using the DEPENDENCE statement.

```
DEPENDENCE <dependence>{,<dependence>};
<dependence> ::= <function>([<order>],<coordinate>{,
                 <coordinate>})
<function> ::= "identifier"  - the name of the function
<order> ::= 1|2 tensor order of the function (the value
            of the function is 1 - vector, 2 - dyad
            (two-dimensional tensor))
```

Every <dependence> in the statement determines on which <coordinates> the <function> depends. If the tensor <order> of the function occurs in the <dependence>, the <function> is declared as a vector or a dyad. If, however, the <function> has been declared by the VECTORS and DYADS statements of the EXPRES module, the user need not present the tensor <order>. By default, a function without any declaration is regarded as scalar. In the discretization, all scalar components of tensor functions are replaced by identifiers that arise by putting successively the function name and the individual indices of the given component (e.g. the tensor component T(1,2), written in the EXPRES module as T(1,2), is repre-

sented by the identifier T12). Before the DEPENDENCE statement is executed, the coordinates have to be defined using the COORDINATES statement. There may be several DEPENDENCE statements. The DEPENDENCE statement cancels all preceding determinations of which grids are to be used for differentiating the function or the equation for this function. These determinations can be either defined by the ISGRID or GRIDEQ statements, or computed in the evaluation of the IIM statement. The GIVEN statement:

```
GIVEN <function>{,<function>};
```

declares all functions included in it as given functions whose values are known to the user or can be computed. The CLEARGIVEN statement:

```
CLEARGIVEN;
```

cancels all preceding GIVEN declarations. If the TWOGRID switch is ON, the given functions can be differentiated both on the integer and the half-integer grids. If the TWOGRID switch is OFF, any given function can be differentiated only on one grid. After the package is loaded, the TWOGRID is ON.

Functions and difference grids

Every scalar function or scalar component of a vector or a dyadic function occurring in the discretized system can be discretized in any of the coordinates either on the integer or half-integer grid. One of the tasks of the IIMET module is to find the optimum distribution of each of these dependent variables of the system on the integer and half-integer grids in all variables so that the number of the performed interpolations in the integro-interpolation method will be minimal. Using the statement

```
SAME <function>{,<function>};
```

all functions given in one of these declarations will be discretized on the same grids in all coordinates. In each SAME statement, at least one of these functions in one SAME statement must be the sought one. If the given function occurs in the SAME statement, it will be discretized only on one grid, regardless of the state of the TWOGRID switch. If a vector or a dyadic function occurs in the SAME statement, what has been said above relates to all its scalar components. There are several SAME statements that can be presented. All SAME statements can be canceled by the following statement:

```
CLEARSAME;
```

The SAME statement can be successfully used, for example, when the given func-

tion depends on the function sought in a complicated manner that cannot be included either in the differential equation or in the difference scheme explicitly, and when both the functions are desired to be discretized in the same points so that the user will not be forced to execute the interpolation during the evaluation of the given function. In some cases, it is convenient too to specify directly which variable on which grid is to be discretized, for which case the ISGRID statement is applied:

```
ISGRID <s-function>{,<s-function>};
<s-function> ::= <function>([<component>,]<s-grid>
                            {,<s-grid>})
<s-grid> ::= <coordinate> .. <grid>,
<grid> ::= ONE | HALF            designation of the integer
                                 (ONE) and half-integer
                                 (HALF) grids
<component> ::= <i-dim> |        for the vector <function>
                <i-dim>,<i-dim>  for the dyadic <function>
                                 it is not presented for the
                                 scalar <function>
<i-dim> ::= *| "natural number from 1 to the space dimension
              the space dimension is specified in the EXPRES
              module by the SCALEFACTORS statement, * means
              all components
```

The statement defines that the given functions or their components will be discretized in the specified coordinates on the specified grids, so that, for example, the statement ISGRID U (X..ONE,Y..HALF), V(1,Z..ONE), T(*,1,X..HALF); defines that scalar U will be discretized on the integer grid in the coordinate X, and on the half-integer one in the coordinate Y, the first component of vector V will be on the integer grid in the coordinate Z, and the first column of tensor T will be on the half-integer grid in the coordinate X. The ISGRID statement can be applied more times. The functions used in this statement have to be declared before by the DEPENDENCE statement.

Equations and difference grids

Every equation of the system of partial differential equations is an equation for some sought function (specified in the IIM statement). The correspondence between the sought functions and the equations is mutually unambiguous. The GRIDEQ statement makes it possible to determine on which grid an individual equation will be discretized in some or all coordinates

```
GRIDEQ <g-function>{,<g-function>};
<g-function> ::= <function>(<s-grid>{,<s-grid>})
```

Every equation can be discretized in any coordinate either on the integer or half-

integer grid. This statement determines the discretization of the equations given by the functions included in it in given coordinates, on given grids. The meaning of the fact that an equation is discretized on a certain grid is as follows: index I used in the DIFMATCH statements (discussed in the following section), specifying the discretization of the basic terms, will be located in the center of the cell of this grid, and indices I+1/2, I-1/2 from the DIFMATCH statement on the boundaries of the cell of this grid. The actual name of the index in the given coordinate is determined using the COORDINATES statement, and its location on the grid is set by the CENTERGRID switch.

Discretization of basic terms

The discretization of a system of partial differential equations is executed successively in individual coordinates. In the discretization of an equation in one coordinate, the equation is linearized into its basic terms first that will be discretized independently then. If D is the designation for the discretization operator in the coordinate x, this linearization obeys the following rules:

```
1. D(a+b) = D(a)+D(b)
2. D(-a)  = -D(a)
3. D(p.a) = p.D(a)  (p does not depend on coordinate x)
4. D(a/p) = D(a)/p
```

The linearization lasts as long as some of these rules can be applied. The basic terms that must be discretized after the linearization have then the forms of the following quantities:

1. The actual coordinate in which the discretization is performed.
2. The sought function.
3. The given function.
4. The product of the quantities 1 - 7.
5. The quotient of the quantities 1 - 7.
6. The natural power of the quantities 1 - 7.
7. The derivative of the quantities 1 - 7 with respect to the actual coordinate.

The way of discretizing these basic terms, while the functions are on integer and half-integer grids, is determined using the DIFMATCH statement:

```
DIFMATCH <coordinate>,<pattern term>,{{<grid specification>,}
         <number of interpolations>, <discretized term>};
```

```
<coordinate> ::= ALL | "identifier" - coordinate name from
                                     COORDINATES statement
<pattern term> ::= <pattern coordinate>|
     <pattern sought function>|
     <pattern given function>|<pattern term> *
     <pattern term>|<pattern term> / <pattern term>|
     <pattern term> ** <exponent>|
     DIFF(<pattern term>,<pattern coordinate>[,<order
     of derivative>])|
     <declared operator>(<pattern term>{,<pattern term>})
<pattern coordinate> ::= X
<pattern sought function> ::= U | V | W
<pattern given function> ::= F | G
<exponent> ::= N | "integer greater than 1"
<order of derivative> ::= "integer greater than 2"
<grid specification> ::= <pattern function>=<grid>
<pattern function> ::= <pattern sought function>|
                       <pattern given function>
<number of interpolations> ::= "non-negative integer"
<discretized term> ::=
              <pattern operator>(<index expression>)|
              "natural number"|DI|DIM1|DIP1|DIM2|DIP2|
              <declared term> | - <discretized term> |
              <discretized term> + <discretized term> |
              <discretized term> * <discretized term> |
              <discretized term> / <discretized term> |
              (<discretized term>) |
              <discretized term> **<exponent>
<pattern operator> ::= X | U | V | W | F | G
<index expression> ::= <pattern index> |
                       <pattern index> + <increment> |
                       <pattern index> - <increment>
<pattern index> ::= I
<increment> = "rational number"
DIFCONST <declared term>{,<declared term>};
<declared term> ::= "identifier" - the constant parameter of
                                   the difference scheme.
DIFFUNC <declared operator>{,<declared operator>};
<declared operator> ::= "identifier" - prefix operator, that
             can appear in discretized equations (e.g. SIN).
```

The first parameter of the DIFMATCH statement determines the coordinate for which the discretization defined in it is valid. If ALL is used, the discretization will be valid for all coordinates, and this discretization is accepted when it has been checked whether there has been no other discretization defined for the given coordinate and the given pattern term. Each pattern sought function, occurring in the pattern term, must be included in the specification of the grids. The pattern given functions from the pattern term can occur in the grid specification, but in some cases (see below) need not. In the grid specification the maximum number

of 3 pattern functions may occur. The discretization of each pattern term has to be specified in all combinations of the pattern functions occurring in the grid specification, on the integer and half-integer grids, that is 2**n variants for the grid specification with n pattern functions (n=0,1,2,3). The discretized term is the discretization of the pattern term in the pattern coordinate X in the point X(I) on the pattern grid (see Fig. 2.2), and the pattern functions occurring in the grid specification are in the discretized term on the respective grids from this specification (to the discretized term corresponds the grid specification preceding it).

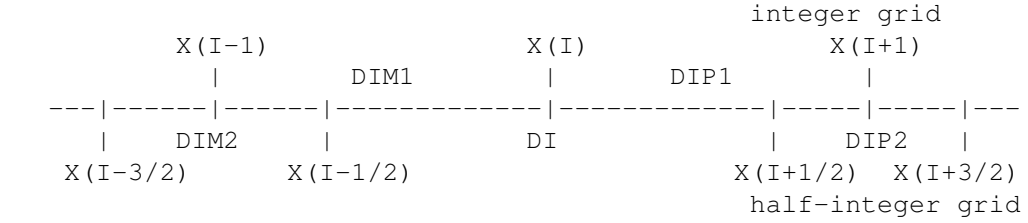


Figure 2.2 Pattern grid

The pattern grid steps defined as

```
DIM2 = X(I - 1/2) - X(I - 3/2)
DIM1 = X(I) - X(I - 1)
DI   = X(I + 1/2) - X(I - 1/2)
DIP1 = X(I + 1) - X(I)
DIP2 = X(I + 3/2) - X(I + 1/2)
```

can occur in the discretized term. In the integro-interpolation method, the discretized term is specified by the integral

```
<discretized term>=1/(X(I+1/2)-X(I-1/2))*
                   DINT(X(I-1/2),X(I+1/2),
                        <pattern term>,X),
```

where DINT is operator of definite integration DINT(from, to, function, variable). The number of interpolations determines how many interpolations were needed for calculating this integral in the given discrete form (the function on the integer or half-integer grid). If the integro-interpolation method is not used, the more convenient is the distribution of the functions on the half-integer and integer grids, the smaller number is chosen by the user. The parameters of the difference scheme defined by the DIFCONST statement can occur in the discretized expression too (for example, the implicit-explicit scheme on the implicit layer multiplied by the constant C and on the explicit one by (1-C)). As a matter of fact, all DIFMATCH statements create a base of pattern terms with the rules of how to discretize these terms in individual coordinates under the assumption that the functions occurring in the pattern terms are on the grids determined in the grid specification (all combinations must be included). The DIFMATCH statement does not check whether the

discretized term is actually the discretization of the pattern term or whether in the discretized term occur the functions from the grid specification on the grids given by this specification. An example can be the following definition of the discretization of the first and second derivatives of the sought function in the coordinate R on a uniform grid:

```
DIFMATCH R,DIFF(U,X),U=ONE,2,(U(I+1)-U(I-1))/(2*DI);
                        U=HALF,0,(U(I+1/2)-U(I-1/2))/DI;
DIFMATCH R,DIFF(U,X,2),U=ONE,0,(U(I+1)-2*U(I)+U(I-1))/DI**2,
  U=HALF,2,(U(I+3/2)-U(I+1/2)-U(I-1/2)+U(I-3/2))/(2*DI**2);

   All DIFMATCH statements can be cleared by the statement

CLEARDIFMATCH;
```

After this statement user has to supply its own DIFMATCH statements. But now back to the discretizing of the basic terms obtained by the linearization of the partial differential equation, as mentioned at the beginning of this section. Using the method of pattern matching, for each basic term a term representing its pattern is found in the base of pattern terms (specified by the DIFMATCH statements). The pattern matching obeys the following rules:

1. The pattern for the coordinate in which the discretization is executed is the pattern coordinate X.

2. The pattern for the sought function is some pattern sought function, and this correspondence is mutually unambiguous.

3. The pattern for the given function is some pattern given function, or, in case the EQFU switch is ON, some pattern sought function, and, again, the correspondence of the pattern with the given function is mutually unambiguous (after loading the EQFU switch is ON).

4. The pattern for the products of quantities is the product of the patterns of these quantities, irrespective of their sequence.

5. The pattern for the quotient of quantities is the quotient of the patterns of these quantities.

6. The pattern for the natural power of a quantity is the same power of the pattern of this quantity or the power of this quantity with the pattern exponent N.

7. The pattern for the derivative of a quantity with respect to the coordinate in which the discretization is executed is the derivative of the pattern of this quantity with respect to the pattern coordinate X of the same order of differentiation.

8. The pattern for the sum of the quantities that have the same pattern with the identical correspondence of functions and pattern functions is this common pattern (so that it will not be necessary to multiply the parentheses during discretizing the products in the second and further coordinates).

When matching the pattern of one basic term, the program finds the pattern term and the functions corresponding to the pattern functions, maybe also the exponent corresponding to the pattern exponent N. After determining on which grids the individual functions and the individual equations will be discretized, which will be discussed in the next section, the program finds in the pattern term base the discretized term either with pattern functions on the same grids as are the functions from the basic term corresponding to them in case that the given equation is differentiated on the integer grid, or with pattern functions on inverse grids (an inverse integer grid is a half-integer grid, and vice versa) compared with those used for the functions from the basic term corresponding to them in case the given equation is differentiated on the half-integer grid (the discretized term in the DIFMATCH statement is expressed in the point X(I), i.e. on the integer grid, and holds for the discretizing of the equation on the integer grid; with regard to the substitutions for the pattern index I mentioned later, it is possible to proceed in this way and not necessary to define the discretization in the points X(I+1/2) too, i.e. on the half-integer grid). The program replaces in the thus obtained discretized term:

1. The pattern coordinate X with the particular coordinate s in which the discretization is actually performed.
2. The pattern index I and the grid steps DIM2, DIM1, DI, DIP1, DIP2 with the expression given in table 2.1 according to the state of the CENTERGRID switch and to the fact whether the given equation is discretized on the integer or half-integer grid (i is the index corresponding to the coordinate s according to the COORDINATES statement, the grid steps were defined in section 2.2)
3. The pattern functions with the corresponding functions from the basic term and, possibly, the pattern exponent with the corresponding exponent from the basic term.

	the equation discretized on			
	the integer grid		the half-integer grid	
	CENTERGRID OFF	CENTERGRID ON	CENTERGRID OFF	CENTERGRID ON
I	i		i+1/2	
DIM2	(Hs(i-2)+Hs(i-1))/2	Hs(i-1)	(Hs(i-1)+Hs(i))/2	
DIM1	Hs(i-1)	(Hs(i-1)+Hs(i))/2	Hs(i)	
DI	(Hs(i-1)+Hs(i))/2	Hs(i)	(Hs(i)+Hs(i+1))/2	
DIP1	Hs(i)	(Hs(i)+Hs(i+1))/2	Hs(i+1)	

```
|DIP2|(Hs(i)+Hs(i+1))/2  |     Hs(i+1)          |(Hs(i+1)+Hs(i+2))/2|
-------------------------------------------------------------------
```

Table 2.1 Values of the pattern index and the pattern grid steps.

More details will be given now to the discretization of the given functions and its specification. The given function may occur in the SAME statement, which makes it bound with some sought function, in other words it can be discretized only on one grid. This means that all basic terms, in which this function occurs, must have their pattern terms in whose discretization definitions by the DIFMATCH statement the pattern function corresponding to the mentioned given function has to occur in the grid specification. If the given function does not occur in the SAME statement and the TWOGRID switch is OFF, i.e. it can be discretized only on one grid again, the same holds true. If, however, the given function does not occur in the SAME statement and the TWOGRID switch is ON, i.e. it can be discretized simultaneously on the integer and the half-integer grids, then the basic terms of the equations including this function have their pattern terms in whose discretization definitions the pattern function corresponding to the mentioned given function need not occur in the grid specification. If, however, in spite of all, this pattern function in the discretization definition does occur in the grid specification, it is the alternative with a smaller number of interpolations occurring in the DIFMATCH statement that is selected for each particular basic term with a corresponding pattern (the given function can be on the integer or half-integer grid). Before the discretization is executed, it is necessary to define using the DIFMATCH statements the discretization of all pattern terms that are the patterns of all basic terms of all equations appearing in the discretized system in all coordinates. The fact that the pattern terms of the basic terms of partial equations occur repeatedly in individual systems has made it possible to create a library of the discretizations of the basic types of pattern terms using the integro-interpolation method. This library is a component part of the IIMET module (in its end) and makes work easier for those users who find the pattern matching mechanism described here too difficult. New DIFMATCH statements have to be created by those whose equations will contain a basic term having no pattern in this library, or those who need another method to perform the discretization. The described implemented algorithm of discretizing the basic terms is sufficiently general to enable the use of a nearly arbitrary discretization on orthogonal grids.

Discretization of a system of equations

All statements influencing the run of the discretization that one want use in this run have to be executed before the discretization is initiated. The COORDINATES, DEPENDENCE, and DIFMATCH statements have to occur in all applications. Further, if necessary, the GRID UNIFORM, GIVEN, ISGRID, GRIDEQ,

SAME, and DIFCONST statements can be used, or some of the CENTREGRID, TWOGRID, EQFU, and FULLEQ switches can be set. Only then the discretization of a system of partial differential equations can be started using the IIM statement:

```
IIM <array>{,<sought function>,<equation>};
<array> ::= "identifier" - the name of the array for
                           storing the result
<sought function> ::= "identifier" - the name of the
                      function whose behavior is
                      described by the equation
<equation> ::= <left side> = <right side>
<left side> ::= "algebraic expression", the derivatives
                are designated by the DIFF operator
<right side> ::= "algebraic expression"
```

Hence, in the IIM statement the name of the array in which the resulting difference schemes will be stored, and the pair sought function - equation, which describes this function, are specified. The meaning of the relation between the sought function and its equation during the discretization lies in the fact that the sought function is preferred in its equation so that the interpolation is not, if possible, used in discretizing the terms of this equation that contain it. In the equations, the functions and the coordinates appear as identifiers. The identifiers that have not been declared as functions by the DEPENDENCE statement or as coordinates by the COORDINATES statement are considered constants independent of the coordinates. The partial derivatives are expressed by the DIFF operator that has the same syntax as the standard differentiation operator DF. The functions and the equations can also have the vector or tensor character. If these non-scalar quantities are applied, the EXPRES module has to be used together with the IIMET module, and also non-scalar differential operators such as GRAD, DIV, etc. can be employed. The sequence performed by the program in the discretization can be briefly summed up in the following items:

1. If there are non-scalar functions or equations in a system of equations, they are automatically converted into scalar quantities by means of the EXPRES module.

2. In each equation, the terms containing derivatives are transferred to the left side, and the other terms to the right side of the equation.

3. For each coordinate, with respect to the sequence in which they occur in the COORDINATES statement, the following is executed:

   a) It is determined on which grids all functions and all equations in the actual coordinate will be discretized, and simultaneously the limits are kept resulting from the ISGRID, GRIDEQ, and SAME statements if they were used.

Such a distribution of functions and equations on the grids is selected among all possible variants that ensures the minimum sum of all numbers of the interpolations of the basic terms (specified by the DIFMATCH statement) of all equations if the FULLEQ switch is ON, or of all left sides of the equations if the FULLEQ switch is OFF (after the loading the FULLEQ switch is ON).

b) The discretization itself is executed, as specified by the DIFMATCH statements.

4. If the array name is A, then if there is only one scalar equation in the IIM statement, the discretized left side of this equation is stored in A(0) and the discretized right side in A(1) (after the transfer mentioned in item 2), if there are more scalar equations than one in the IIM statement, the discretization of the left side of the i-th scalar equation is stored in A(i,0) and the discretization of the right side in A(i,1).

The IIM statement can be used more times during one program run, and between its calls, the discretizing process can be altered using other statements of this module.

Error messages

The IIMET module provides error messages in the case of the user's errors. Similarly as in the REDUCE system, the error reporting is marked with five stars : "*****" on the line start. Some error messages are identical with those of the REDUCE system. Here are given some other error messages that require a more detailed explanation:

```
***** Matching of X term not found
 - the discretization of the pattern term that is the
   pattern of the basic term printed on the place X
   has not been defined (using the DIFMATCH statement)
***** Variable of type F not defined on grids in DIFMATCH
 - in the definition of the discretizing of the pattern
   term the given functions were not used in the grid
   specification and are needed now
***** X Free vars not yet implemented
 - in the grid specification in the DIFMATCH statement
   more than 3 pattern functions were used
***** All grids not given for term X
 - in the definition of the discretization of the
   pattern of the basic term printed on the place X not
   all necessary combinations of the grid specification
   of the pattern functions were presented
```

20.22.4 APPROX

A Module for Determining the Precision Order of the Difference Scheme

This module makes it possible to determine the differential equation that is solved by the given difference scheme, and to determine the order of accuracy of the solution of this scheme in the grid steps in individual coordinates. The discrete function values are expanded into the Taylor series in the specified point.

Specification of the coordinates and the indices corresponding to them

The COORDINATES statement, described in the IIMET module manual, specifying the coordinates and the indices corresponding to them is the same for this program module as well. It has the same meaning and syntax. The present module version assumes a uniform grid in all coordinates. The grid step in the input difference schemes has to be designated by an identifier consisting of the character H and the name of the coordinate, e.g. the step of the coordinate X is HX.

Specification of the Taylor expansion

In the determining of the approximation order, all discrete values of the functions are expanded into the Taylor series in all coordinates. In order to determine the Taylor expansion, the program needs to know the point in which it performs this expansion, and the number of terms in the Taylor series in individual coordinates. The center of the Taylor expansion is specified by the CENTER statement and the number of terms in the Taylor series in individual coordinates by the MAXORDER statement:

```
CENTER <center>{,<center>};
<center> ::= <coordinate> = <increment>
<increment> ::= "rational number"
MAXORDER <order>{,<order>};
<order> ::= <coordinate> = <number of terms>
<number of terms> ::= "natural number"
```

The increment in the CENTER statement determines that the center of the Taylor expansion in the given coordinate will be in the point specified by the index I + <increment>, where I is the index corresponding to this coordinate, defined using the COORDINATES statement, e.g. the following example

```
COORDINATE T,X INTO N,J;
CENTER T = 1/2, X = 1;
MAXORDER T = 2, X = 3;
```

specifies that the center of the Taylor expansion will be in the point (t(n+1/2),x(j+1)) and that until the second derivatives with respect to t (second powers of ht) and until the third derivatives with respect to x (third powers of hx) the expansion will be performed. The CENTER and MAXORDER statements can be placed only after the COORDINATES statement. If the center of the Taylor expansion is not defined in some coordinate, it is supposed to be in the point given by the index of this coordinate (i.e. zero increment). If the number of the terms of the Taylor expansion is not defined in some coordinate, the expansion is performed until the third derivatives with respect to this coordinate.

Function declaration

All functions whose discrete values are to be expanded into the Taylor series must be declared using the FUNCTIONS statement:

```
FUNCTIONS <name of function>{,<name of function>};
<name of function> ::= "identifier"
```

In the specification of the difference scheme, the functions are used as operators with one or more arguments, designating the discrete values of the functions. Each argument is the sum of the coordinate index (from the COORDINATES statement) and a rational number. If some index is omitted in the arguments of a function, this functional value is supposed to lie in the point in which the Taylor expansion is performed, as specified by the CENTER statement. In other words, if the COORDINATES and CENTER statements, shown in the example in the previous section, are valid, then it holds that U(N+1) = U(N+1,J+1) and U(J-1) = U(N+1/2,J-1). The FUNCTIONS statement can declare both the sought and the known functions for the expansion.

Order of accuracy determination

The order of accuracy of the difference scheme is determined by the APPROX statement:

```
APPROX (<diff. scheme>);
<diff. scheme> ::=  <l. side> = <r. side>
<l. (r.) side> ::= "algebraic expression"
```

In the difference scheme occur the functions in the form described in the preceding section, the coordinate indices and the grid steps described in section 3.1, and the other symbolic parameters of the difference scheme. The APPROX statement expands all discrete values of the functions declared in the FUNCTIONS statement into the Taylor series in all coordinates (the point in which the Taylor expansion

is performed is specified by the CENTER statement, and the number of the expansion terms by the MAXORDER statement), substitutes the expansions into the difference scheme, which gives a modified differential equation. The modified differential equation, containing the grid steps too, is an equation that is really solved by the difference scheme (into the given orders in the grid steps). The partial differential equation, whose solution is approximated by the difference scheme, is determined by replacing the grid steps by zeros and is displayed after the following message:

"Difference scheme approximates differential equation"

Then the following message is displayed:

"with orders of approximation:"

and the lowest powers (except for zero) of the grid steps in all coordinates, occurring in the modified differential equation are written. If the PRAPPROX switch is ON, then the rest of the modified differential equation is printed. If this rest is added to the left hand side of the approximated differential equation, one obtain modified equation. By default the PRAPPROX switch is OFF. If the grid steps are found in some denominator in the modified equation, i.e. with a negative exponent, the following message is written, preceding the approximated differential equation:

"Reformulate difference scheme, grid steps remain in denominator"

and the approximated differential equation is not correctly determined (one of its sides is zero). Generally, this message means that there is a term in the difference scheme that is not a difference replacement of the derivative, i.e. the ratio of the differences of the discrete function values and the discrete values of the coordinates (the steps of the difference grid). The user, however, must realize that in some cases such a term occurs purposefully in the difference scheme (e.g. on the grid boundary to keep the scheme conservative).

20.22.5 CHARPOL

A Module for Calculating the Amplification Matrix and the Characteristic Polynomial of the Difference Scheme

This program module is used for the first step of the stability analysis of the difference scheme using the Fourier method. It substitutes the Fourier components into the difference scheme, calculates the amplification matrix of the scheme for transition from one time layer to another, and computes the characteristic polynomial of this matrix.

Commands common with the IIMET module

The COORDINATES and GRID UNIFORM statements, described in the IIMET module manual, are applied in this module as well, having the same meaning and syntax. The time coordinate is assumed to be designated by the identifier T. The present module version requires all coordinates to have uniform grids, i.e. to be declared in the GRID UNIFORM statement. The grid step in the input difference schemes has to be designated by the identifier consisting of the character H and the name of the coordinate, e.g. the step of the time coordinate T is HT.

Function declaration

The UNFUNC statement declares the names of the sought functions used in the difference scheme:

```
UNFUNC <function>{,<function>}
<function> ::= "identifier" - name of sought function
```

The functions are used in the difference schemes as operators with one or more arguments for designating the discrete function values. Each argument is the sum of the index (from the COORDINATES statement) and a rational number. If some index is omitted in the function arguments, this function value is supposed to lie in the point specified only by this index, which means that, with the indices N and J and the function U, it holds that U(N+1) = U(N+1,J) and U(J-1) = U(N,J-1). As two-step (in time) difference schemes may be used only, the time index may occur either completely alone in the arguments, or in the sum with a one.

Amplification matrix

The AMPMAT matrix operator computes the amplification matrix of a two-step difference scheme. Its argument is an one column matrix of the dimension (1,k), where k is the number of the equations of the difference scheme, that contains the difference equations of this scheme as algebraic expressions equal to the difference of the right and left sides of the difference equations. The value of the AMPMAT matrix operator is the square amplification matrix of the dimension (k,k). During the computation of the amplification matrix, two new identifiers are created for each spatial coordinate. The identifier made up of the character K and the name of the coordinate represents the wave number in this coordinate, and the identifier made up of the character A and the name of the coordinate represents the product of this wave number and the grid step in this coordinate divided by the least common multiple of all denominators occurring in the scheme in the function argument containing the index of this coordinate. On the output an equation is displayed defining the latter identifier. For example, if in the case of function U and

index J in the coordinate X the expression U(J+1/2) has been used in the scheme (and, simultaneously, no denominator higher than 2 has occurred in the arguments with J), the following equation is displayed: AX: = (KX*HX)/2. The definition of these quantities As allows to express every sum occurring in the argument of the exponentials as the sum of these quantities multiplied by integers, so that after a transformation, the amplification matrix will contain only sin(As) and cos(As) (for all spatial coordinates s). The AMPMAT operator performs these transformations automatically. If the PRFOURMAT switch is ON (after the loading it is ON), the matrices H0 and H1 (the amplification matrix is equal to -H1**(-1)*H0) are displayed during the evaluation of the AMPMAT operator. These matrices can be used for finding a suitable substitution for the goniometric functions in the next run for a greater simplification. The TCON matrix operator transforms the square matrix into a Hermit-conjugate matrix, i.e. a transposed and complex conjugate one. Its argument is the square matrix and its value is Hermit-conjugate matrix of the argument. The Hermit-conjugate matrix is used for testing the normality and unitarity of the amplification matrix in the determining of the sufficient stability condition.

Characteristic polynomial

The CHARPOL operator calculates the characteristic polynomial of the given square matrix. The variable of the characteristic polynomial is designated by the LAM identifier. The operator has one argument, the square matrix, and its value is its characteristic polynomial in LAM.

Automatic denotation

Several statements and procedures are designed for automatic denotation of some parts of algebraic expressions by identifiers. This denotation is namely useful when we obtain very large expressions, which cannot fit into the available memory. We can denote subparts of an expression from the previous step of calculation by identifiers, replace these subparts by these identifiers and continue the analytic calculation only with these identifiers. Every time we use this technique we have to explicitly survive in processed expressions those algebraic quantities which will be necessary in the following steps of calculation. The process of denotation and replacement is performed automatically and the algebraic values which are denoted by these new identifiers can be written out at any time. We describe how this automatic denotation can be used. The statement DENOTID defines the beginning letters of newly created identifiers. Its syntax is

```
DENOTID <id>;
<id> ::= "identifier"
```

After this statement the new identifiers created by the operators DENOTEPOL and DENOTEMAT will begin with the letters of the identifier <id> used in this statement. Without using any DENOTID statement all new identifiers will begin with one letter A. We suggest to use this statement every time before using operators DENOTEPOL or DENOTEMAT with some new identifier and to choose identifiers used in this statement in such a way that the newly created identifiers are not equal to any identifiers used in the expressions you are working with. The operator DENOTEPOL has one argument, a polynomial in LAM, and denotes the real and imaginary part of its coefficients by new identifiers. The real part of the j-th LAM power coefficient is denoted by the identifier <id>R0j and the imaginary part by <id>I0j, where <id> is the identifier used in the last DENOTID statement. The denotation is done only for non-numeric coefficients. The value of this operator is the polynomial in LAM with coefficients constructed from the new identifiers. The algebraic expressions which are denoted by these identifiers are stored as LISP data structure standard quotient in the LISP variable DENOTATION!* (assoc. list). The operator DENOTEMAT has one argument, a matrix, and denotes the real and imaginary parts of its elements. The real part of the (j,k) matrix element is denoted by the identifier <id>Rjk and the imaginary part by <id>Ijk. The returned value of the operator is the original matrix with non-numeric elements replaced by <id>Rjk + I*<id>Ijk. Other matters are the same as for the DENOTEPOL operator. The statement PRDENOT has the syntax

```
PRDENOT;
```

and writes from the variable DENOTATION!* the definitions of all new identifiers introduced by the DENOTEPOL and DENOTEMAT operators since the last call of CLEARDENOT statement (or program start) in the format defined by the present setting of output control declarations and switches. The definitions are written in the same order as they have been entered, so that the definitions of the first DENOTEPOL or DENOTEMAT operators are written first. This order guarantees that this statement can be utilized directly to generate a semantically correct numerical program (the identifiers from the first denotation can appear in the second one, etc.). The statement CLEARDENOT with the syntax

```
CLEARDENOT;
```

clears the variable DENOTATION!*, so that all denotations saved earlier by the DENOTEPOL and DENOTEMAT operators in this variable are lost. The PRDENOT statement succeeding this statement writes nothing.

20.22.6 HURWP

A Module for Polynomial Roots Locating

This module is used for verifying the stability of a polynomial, i.e. for verifying if all roots of a polynomial lie in a unit circle with its center in the origin. By investigating the characteristic polynomial of the difference scheme, the user can determine the conditions of the stability of this scheme.

Conformal mapping

The HURW operator transforms a polynomial using the conformal mapping LAM=(z+1)/(z-1). Its argument is a polynomial in LAM and its value is a transformed polynomial in LAM (LAM=z). If P is a polynomial in LAM, then it holds: all roots LAM1i of the polynomial P are in their absolute values smaller than one, i.e. |LAM1i|<1, iff the real parts of all roots LAM2i of the HURW(P) polynomial are negative, i.e. Re (LAM2i)<0. The elimination of the unit polynomial roots (LAM=1), which has to occur before the conformal transformation is performed, is made by the TROOT1 operator. The argument of this operator is a polynomial in LAM and its value is a polynomial in LAM not having its root equal to one any more. Mostly, the investigated polynomial has some more parameters. For some special values of those parameters, the polynomial may have a unit root. During the evaluation of the TROOT1 operator, the condition concerning the polynomial parameters is displayed, and if it is fulfilled, the resulting polynomial has a unit root.

Investigation of polynomial roots

The HURWITZP operator checks whether a polynomial is the Hurwitz polynomial, i.e. whether all its roots have negative real parts. The argument of the HURWITZP operator is a polynomial in LAM with real or complex coefficients, and its value is YES if the argument is the Hurwitz polynomial. It is NO if the argument is not the Hurwitz polynomial, and COND if it is the Hurwitz polynomial when the conditions displayed by the HURWITZP operator during its analysis are fulfilled. These conditions have the form of inequalities and contain algebraic expressions made up of the polynomial coefficients. The conditions have to be valid either simultaneously, or they are designated and a proposition is created from them by the AND and OR logic operators that has to be fulfilled (it is the condition concerning the parameters occurring in the polynomial coefficient) by a polynomial to be the Hurwitz one. This proposition is the sufficient condition, the necessary condition is the fulfillment of all the inequalities displayed. If the HURWITZP operator is called interactively, the user is directly asked if the inequalities are or are not valid. The user responds "Y" if the displayed inequality is valid, "N" if it is not, and "?" if he does not know whether the inequality is true or not.

20.22.7 LINBAND

A Module for Generating the Numeric Program for Solving a System of Linear Algebraic Equations with Band Matrix

The LINBAND module generates the numeric program in the FORTRAN language, which solves a system of linear algebraic equations with band matrix using the routine from the LINPACK, NAG ,IMSL or ESSL program library. As input data only the system of equations is given to the program. Automatically, the statements of the FORTRAN language are generated that fill the band matrix of the system in the corresponding memory mode of chosen library, call the solving routine, and assign the chosen variables to the solution of the system. The module can be used for solving linear difference schemes often having the band matrix.

Program generation

The program in the FORTRAN language is generated by the GENLINBANDSOL statement (the braces in this syntax definition occur directly in the program and do not have the usual meaning of the possibility of repetition, they designate REDUCE lists):

```
GENLINBANDSOL (<n-lower>,<n-upper>,{<system>});
<n-lower> ::= "natural number"
<n-upper> ::= "natural number"
<system> ::= <part of system> | <part of system>,<system>
<part of system>::= {<variable>,<equation>} | <loop>
<variable> ::= "kernel"
<equation> ::= <left side> = <right side>
<left side> ::= "algebraic expression"
<right side> ::= "algebraic expression"
<loop> ::= {DO,{<parameter>,<from>,<to>,<step>},<c-system>}
<parameter> ::= "identifier"
<from> ::= <i-expression>
<to> ::= <i-expression>
<step> ::= <i-expression>
<i-expression> ::= "algebraic expression" with natural value
                                        (evaluated in FORTRAN)
<c-system> ::= <part of c-system> | <part of c-system>,<c-
     system>
<part of c-system> ::= {<variable>,<equation>}
```

The first and second argument of the GENLINBANDSOL statement specifies the number of the lower (below the main diagonal) and the upper diagonals of the band matrix of the system. The system of linear algebraic equations is specified by means of lists expressed by braces in the REDUCE system. The variables of the equation system can be identifiers, but most probably they are operators with an

argument or with arguments that are analogous to array in FORTRAN. The left side of each equation has to be a linear combination of the system variables, the right side, on the contrary, is not allowed to contain any variables of the system. The sequence of the band matrix lines is given by the sequence of the equations, and the sequence of the columns by the sequence of the variables in the list describing the equation system. The meaning of the loop in the system list is similar to that of the DO loop of the FORTRAN language. The individual variables and equations described by the loop are obtained as follows:

1. <parameter> = <from>. 2. The <parameter> value is substituted into the variables and equations of the <c-system> loop, by which further variables and equations of the system are obtained. 3. <parameter> is increased by <step>. 4. If <parameter> is less or equal <to>, then go to step 2, else all variables and equations described by the loop have already been obtained.

The variables and equations of the system included in the loop usually contain the loop parameter, which mostly occur in the operator arguments in the REDUCE language, or in the array indices in the FORTRAN language. If NL = <n-lower>, NU = <n-upper>, and for some loop F = <from>, T = <to>, S = <step> and N is the number of the equations in the loop <c-system>, it has to be true that

```
UP(NL/N) + UP(NU/N) < DOWN((T-F)/S)
```

where UP represents the rounding-off to a higher natural number, and DOWN the rounding-off to a lower natural number. With regard to the fact that, for example, the last variable before the loop is not required to equal the last variable from the loop system, into which the loop parameter equal to F-S is substituted, when the band matrix is being constructed, from the FORTRAN loop that corresponds to the loop from the specification of the equation system, at least the first NL variables-equations have to be moved to precede the FORTRAN loop, and at least the last NU variables-equations have to be moved to follow this loop in order that the correspondence of the system variables in this loop with the system variables before and after this loop will be secured. And this move requires the above mentioned condition to be fulfilled. As, in most cases, NL/N and NU/N are small with respect to (T-F)/S, this condition does not represent any considerable constrain. The loop parameters <from>, <to>, and <step> can be natural numbers or expressions that must have natural values in the run of the FORTRAN program.

Choosing the numerical library

The user can choose the routines of which numerical library will be used in the generated FORTRAN code. The supported numerical libraries are: LINPACK, NAG, IMSL and ESSL (IBM Engineering and Scientific Subroutine Library) . The routines DGBFA, DGBSL (band solver) and DGTSL (tridiagonal solver) are

used from the LINPACK library, the routines F01LBF, F04LDF (band solver) and F01LEF, F04LEF (tridiagonal solver) are used from the NAG library, the routine LEQT1B is used from the IMSL library and the routines DGBF, DGBS (band solver) and DGTF, DGTS (tridiagonal solver) are used from the ESSL library. By default the LINPACK library routines are used. The using of other libraries is controlled by the switches NAG,IMSL and ESSL. All these switches are by default OFF. If the switch IMSL is ON then the IMSL library routine is used. If the switch IMSL is OFF and the switch NAG is ON then NAG library routines are used. If the switches IMSL and NAG are OFF and the switch ESSL is ON then the ESSL library is used. During generating the code using LINPACK, NAG or ESSL libraries the special routines are use for systems with tridiagonal matrices, because tridiagonal solvers are faster than the band matrix solvers.

Completion of the generated code

The GENLINBANDSOL statement generates a block of FORTRAN code (a block of statements of the FORTRAN language) that performs the solution of the given system of linear algebraic equations. In order to be used, this block of code has to be completed with some declarations and statements, thus getting a certain envelope that enables it to be integrated into the main program. In order to be able to work, the generated block of code has to be preceded by:

1. The declaration of arrays as described by the comments generated into the FORTRAN code (near the calling of library routines)
2. The assigning the values to the integer variables describing the real dimensions of used arrays (again as described in generated FORTRAN comments)
3. The filling of the variables that can occur in the loop parameters.
4. The filling or declaration of all variables and arrays occurring in the system equations, except for the variables of the system of linear equations.
5. The definition of subroutine ERROUT the call to which is generated after some routines found that the matrix is algorithmically singular

The mentioned envelope for the generated block can be created manually, or directly using the GENTRAN program package for generating numeric programs. The LINBAND module itself uses the GENTRAN package, and the GENLINBANDSOL statement can be applied directly in the input files of the GENTRAN package (template processing). The GENTRAN package has to be loaded prior to loading of the LINBAND module. The generated block of FORTRAN code has to be linked with the routines from chosen numerical library.

For reference, see [Lis91].

20.23 GCREF: A Graph Cross Referencer

This package reuses the code of the RCREF package to create a graph displaying the interdependency of procedures in a Reduce source code file.

Authors: A. Dolzmann, T. Sturm

20.23.1 Basic Usage

Similarly to the Reduce cross referencer, it is used via switches as follows:

```
load_package gcref;
on gcref;
in "<filename>.red";
off gcref;
```

At `off gcref;` the graph is printed to the screen in TGF format. To redirect this output to a file, use the following:

```
load_package gcref;
on gcref;
in "<filename>.red";
out "<filename>.tgf";
off gcref;
shut "<filename>.tgf";
```

20.23.2 Shell Script "gcref"

There is a shell script "gcref" in this directory automizing this like

```
./gcref filename.red
```

"gcref" is configured to use CSL Reduce. To use PSL Reduce instead, set $REDUCE in the environment. To use PSL by default, define

```
REDUCE=redpsl
```

in line 3 of "gcref".

20.23.3 Rendering with yED

The obtained TGF file can be viewed with a graph editor. I recommend using the free software yED, which is written in Java and available for many platforms.

`http://www.yworks.com/en/products_yed_about.html`

Note that TGF is not suitable for storing rendering information. After opening the TGF file with yED, the graph has to be rendered explicitly as follows:

* From menu "Layout" choose "Hierarchical Layout".

To resize the nodes to the procedure names

* from menu "Tools" choose "Fit Node to Label".

Feel free to experiment with yED and use other layout and layout options, which might be suitable for your particular software.

For saving your particular layout at the end, use the GRAPHML format instead of TGF.

20.24 GENTRAN: A Code Generation Package

GENTRAN is an automatic code GENerator and TRANslator. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR, PASCAL or C code can be generated through a series of interactive commands or under the control of a template processing routine. Large expressions can be automatically segmented into subexpressions of manageable size, and a special file-handling mechanism maintains stacks of open I/O channels to allow output to be sent to any number of files simultaneously and to facilitate recursive invocation of the whole code generation process.

Author: Barbara L. Gates

Further documentation is available at https://reduce-algebra.sourceforge.io/extra-docs/gentran.pdf.

20.25 GRINDER: Calculation of three-loop diagrams in Heavy Quark Effective Theory

Author: Andrey G. Grozin

A description of the algorithm can be found on the arXiv page.

20.26 GROEBNER: A Gröbner Basis Package

GROEBNER is a package for the computation of Gröbner Bases using the Buchberger algorithm and related methods for polynomial ideals and modules. It can be used over a variety of different coefficient domains, and for different variable and term orderings.

Gröbner Bases can be used for various purposes in commutative algebra, e.g. for elimination of variables, converting surd expressions to implicit polynomial form, computation of dimensions, solution of polynomial equation systems etc. The package is also used internally by the `SOLVE` operator.

Authors: Herbert Melenk, H.M. Möller and Winfried Neun

Gröbner bases are a valuable tool for solving problems in connection with multivariate polynomials, such as solving systems of algebraic equations and analyzing polynomial ideals. For a definition of Gröbner bases, a survey of possible applications and further references, see [Buc85]. Examples are given in [BGK86], in [Buc88] and also in the test file for this package.

The GROEBNER package calculates Gröbner bases using the Buchberger algorithm. It can be used over a variety of different coefficient domains, and for different variable and term orderings.

The current version of the package uses parts of a previous version, written by R. Gebauer, A.C. Hearn, H. Kredel and H. M. Möller. The algorithms implemented in the current version are documented in [FGLM93], [GM88], [KW88] and [GMN+91]. The operator `saturation` has been implemented in July 2000 (Herbert Melenk).

20.26.1 Background

Variables, Domains and Polynomials

The various functions of the GROEBNER package manipulate equations and/or polynomials; equations are internally transformed into polynomials by forming the difference of left-hand side and right-hand side, if equations are given.

All manipulations take place in a ring of polynomials in some variables $x1, \ldots, xn$ over a coefficient domain d:

$$d[x1, \ldots, xn],$$

where d is a field or at least a ring without zero divisors. The set of variables $x1, \ldots, xn$ can be given explicitly by the user or it is extracted automatically from the input expressions.

All REDUCE kernels can play the role of "variables" in this context; examples are

```
x y z22 sin(alpha) cos(alpha) c(1,2,3) c(1,3,2) farina4711
```

The domain d is the current REDUCE domain with those kernels adjoined that are not members of the list of variables. So the elements of d may be complicated polynomials themselves over kernels not in the list of variables; if, however, the variables are extracted automatically from the input expressions, d is identical with the current REDUCE domain. It is useful to regard kernels not being members of the list of variables as "parameters", e.g.

```
a * x + (a - b) * y**2
```
with "variables" $\{x, y\}$
and "parameters" a and b .

The exponents of GROEBNER variables must be positive integers.

A GROEBNER variable may not occur as a parameter (or part of a parameter) of a coefficient function. This condition is tested in the beginning of the GROEBNER calculation; if it is violated, an error message occurs (with the variable name), and the calculation is aborted. When the GROEBNER package is called by `solve`, the test is switched off internally.

The current version of the Buchberger algorithm has two internal modes, a field mode and a ring mode. In the starting phase the algorithm analyzes the domain type; if it recognizes d as being a ring it uses the ring mode, otherwise the field mode is needed. Normally field calculations occur only if all coefficients are numbers and if the current REDUCE domain is a field (e.g. rational numbers, modular numbers modulo a prime). In general, the ring mode is faster. When no specific REDUCE domain is selected, the ring mode is used, even if the input formulas contain fractional coefficients: they are multiplied by their common denominators so that they become integer polynomials. Zeroes of the denominators are included in the result list.

Term Ordering

In the theory of Gröbner bases, the terms of polynomials are considered as ordered. Several order modes are available in the current package, including the basic modes:

$$lex, gradlex, revgradlex$$

All orderings are based on an ordering among the variables. For each pair of variables (a, b) an order relation must be defined, e.g. "$a \gg b$". The greater sign $\gg$ does not represent a numerical relation among the variables; it can be interpreted only in terms of formula representation: "a" will be placed in front of "b" or "a" is

more complicated than "b".

The sequence of variables constitutes this order base. So the notion of

$$\{x1, x2, x3\}$$

as a list of variables at the same time means

$$x1 \gg x2 \gg x3$$

with respect to the term order.

If terms (products of powers of variables) are compared with *lex*, that term is chosen which has a greater variable or a higher degree if the greatest variable is the first in both. With *gradlex* the sum of all exponents (the total degree) is compared first, and if that does not lead to a decision, the *lex* method is taken for the final decision. The *revgradlex* method also compares the total degree first, but afterward it uses the *lex* method in the reverse direction; this is the method originally used by Buchberger.

Example 1 with $\{x, y, z\}$:

lex:

$x * y * *3$	$\gg$	$y * *48$	(heavier variable)
$x * *4 * y * *2$	$\gg$	$x * *3 * y * *10$	(higher degree in 1st variable)

gradlex:

$y * *3 * z * *4$	$\gg$	$x * *3 * y * *3$	(higher total degree)
$x * z$	$\gg$	$y * *2$	(equal total degree)

revgradlex:

$y * *3 * z * *4$	$\gg$	$x * *3 * y * *3$	(higher total degree)
$x * z$	$\ll$	$y * *2$	(equal total degree, so reverse order of lex)

The formal description of the term order modes is similar to [Kre88]; this description regards only the exponents of a term, which are written as vectors of integers with 0 for exponents of a variable which does not occur:

$(e) = (e1, \ldots, en)$ representing $x1 * *e1\ x2 * *e2 \cdots xn * *en$.
$\deg(e)$ is the sum over all elements of (e)
$(e) \gg (l) \Longleftrightarrow (e) - (l) \gg (0) = (0, \ldots, 0)$

lex:
$(e) > lex > (0) \quad \Longrightarrow \quad e_k > 0$ and $e_j = 0$ for $j = 1, \ldots, k-1$

gradlex:
$(e) > gl > (0) \quad \Longrightarrow \quad \deg(e) > 0$ or $(e) > lex > (0)$

revgradlex:
$(e) > rgl > (0) \quad \Longrightarrow \quad \deg(e) > 0$ or $(e) < lex < (0)$

Note that the *lex* ordering is identical to the standard REDUCE kernel ordering, when `korder` is set explicitly to the sequence of variables.

lex is the default term order mode in the GROEBNER package.

It is beyond the scope of this manual to discuss the functionality of the term order modes. See [Buc88].

The list of variables is declared as an optional parameter of the `torder` statement (see below). If this declaration is missing or if the empty list has been used, the variables are extracted from the expressions automatically and the REDUCE system order defines their sequence; this can be influenced by setting an explicit order via the `korder` statement.

The result of a Gröbner calculation is algebraically correct only with respect to the term order mode and the variable sequence which was in effect during the calculation. This is important if several calls to the GROEBNER package are done with the result of the first being the input of the second call. Therefore we recommend that you declare the variable list and the order mode explicitly. Once declared it remains valid until you enter a new *torder* statement. The operator *gvars* helps you extract the variables from a given set of polynomials, if an automatic reordering has been selected.

The Buchberger Algorithm

The Buchberger algorithm of the package is based on GEBAUER/MÖLLER [GM88]. Extensions are documented in [MMN88] and [GMN+91].

20.26.2 Loading of the Package

The following command loads the package into REDUCE (this syntax may vary according to the implementation):

```
load_package groebner;
```

The package contains various operators, and switches for control over the reduction process. These are discussed in the following.

20.26.3 The Basic Operators

Term Ordering Mode

torder($vl, m, [p_1, p_2, \ldots]$)**;**

where vl is a variable list (or the empty list if no variables are declared explicitly), m is the name of a term ordering mode *lex*, *gradlex*, *revgradlex* (or another implemented mode) and $[p_1, p_2, \ldots]$ are additional parameters for the term ordering mode (not needed for the basic modes).

`torder` sets variable set and the term ordering mode. The default mode is *lex*. The previous description is returned as a list with corresponding elements. Such a list can alternatively be passed as sole argument to `torder`.

If the variable list is empty or if the `torder` declaration is omitted, the automatic variable extraction is activated.

gvars(*{exp*1*, exp*2*, . . ., exp*n*}*);**

where $\{exp1, exp2, \ldots, expn\}$ is a list of expressions or equations.

`gvars` extracts from the expressions $\{exp1, exp2, \ldots, expn\}$ the kernels, which can play the role of variables for a Gröbner calculation. This can be used e.g. in a `torder` declaration.

GROEBNER: Calculation of a Gröbner Basis

groebner $\{exp1, exp2, \ldots, expm\}$;

where $\{exp1, exp2, \ldots, expm\}$ is a list of expressions or equations.

GROEBNER calculates the Gröbner basis of the given set of expressions with respect to the current `torder` setting.

The Gröbner basis $\{1\}$ means that the ideal generated by the input polynomials is the whole polynomial ring, or equivalently, that the input polynomials have no zeroes in common.

As a side effect, the sequence of variables is stored as a REDUCE list in the shared variable

`gvarslast`.

This is important if the variables are reordered because of optimization: you must set them afterwards explicitly as the current variable sequence if you want to use the Gröbner basis in the sequel, e.g. for a `preduce` call. A basis has the property "Gröbner" only with respect to the variable sequences which had been active during its computation.

Example 2

```
torder({},lex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };

            2
  {8*x - 2*y  + 5*y + 3,

      3      2
   2*y  - 3*y  - 16*y + 21}
```

This example used the default system variable ordering, which was $\{x, y\}$. With the other variable ordering, a different basis results:

```
torder({y,x},lex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };

            2
  {2*y + 2*x  - 3*x - 6,

      3      2
   2*x  - 5*x  - 5*x}
```

Another basis yet again results with a different term ordering:

```
torder({x,y},revgradlex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };

   2
{2*y  - 5*y - 8*x - 3,

 y*x - y + x + 3,

   2
 2*x  + 2*y - 3*x - 6}
```

The operation of GROEBNER can be controlled by the following switches:

groebopt – If set `on`, the sequence of variables is optimized with respect to execution speed; the algorithm involved is described in [BGK86]; note that

the final list of variables is available in `gvarslast`.

An explicitly declared dependency supersedes the variable optimization. For example

```
depend a, x, y;
```

guarantees that `a` will be placed in front of `x` and `y`. So `groebopt` can be used even in cases where elimination of variables is desired.

By default `groebopt` is `off`, conserving the original variable sequence.

groebfullreduction – If set `off`, the reduction steps during the GROEBNER operation are limited to the pure head term reduction; subsequent terms are reduced otherwise.

By default `groebfullreduction` is on.

gltbasis – If set on, the leading terms of the result basis are extracted. They are collected in a basis of monomials, which is available as value of the global variable with the name `gltb`.

glterms – If $\{exp_1, \ldots, exp_m\}$ contain parameters (symbols which are not member of the variable list), the shared variable `glterms` contains a list of expression which during the calculation were assumed to be nonzero. A Gröbner basis is valid only under the assumption that all these expressions do not vanish.

The following switches control the print output of GROEBNER; by default all these switches are set `off` and nothing is printed.

groebstat – A summary of the computation is printed including the computing time, the number of intermediate h–polynomials and the counters for the hits of the criteria.

trgroeb – Includes `groebstat` and the printing of the intermediate h-polynomials.

trgroebs – Includes `trgroeb` and the printing of intermediate s–polynomials.

trgroeb1 – The internal pairlist is printed when modified.

Gzerodim!?: Test of $\dim = 0$

gzerodim!? ***bas***

where *bas* is a Gröbner basis in the current setting. The result is `nil` if *bas* is the basis of an ideal of polynomials with more than finitely many common zeros. If the ideal is zero dimensional, i. e. the polynomials of the ideal have only finitely many zeros in common, the result is an integer k which is the number of these common zeros (counted with multiplicities).

`gdimension`, `gindependent_sets`: compute dimension and independent variables

The following operators can be used to compute the dimension and the independent variable sets of an ideal which has the Gröbner basis *bas* with arbitrary term order:

`gdimension bas`

`gindependent_sets bas` `gindependent_sets` computes the maximal left independent variable sets of the ideal, that are the variable sets which play the role of free parameters in the current ideal basis. Each set is a list which is a subset of the variable list. The result is a list of these sets. For an ideal with dimension zero the list is empty. `gdimension` computes the dimension of the ideal, which is the maximum length of the independent sets.

The switch `groebopt` plays no role in the algorithms `gdimension` and `gindependent_sets`. It is set `off` during the processing even if it is set `on` before. Its state is saved during the processing.

The "Kredel-Weispfenning" algorithm is used (see [KW88], extended to general ordering in [BWK93].

Conversion of a Gröbner Basis

`glexconvert`: Conversion of an Arbitrary Gröbner Basis of a Zero Dimensional Ideal into a Lexical One

`glexconvert`$(\{exp,\ldots,expm\}\,[,\{var1\ldots,varn\}]\,[,maxdeg=mx]$
$[,newvars=\{nv1,\ldots,nvk\}])$
where $\{exp1,\ldots,expm\}$ is a Gröbner basis with $\{var1,\ldots,varn\}$ as variables in the current term order mode, mx is an integer, and $\{nv1,\ldots,nvk\}$ is a subset of the basis variables. For this operator the source and target variable sets must be specified explicitly.

`glexconvert` converts a basis of a zero-dimensional ideal (finite number of isolated solutions) from arbitrary ordering into a basis under *lex* ordering. During the call of `glexconvert` the original ordering of the input basis must be still active!

newvars defines the new variable sequence. If omitted, the original variable sequence is used. If only a subset of variables is specified here, the partial ideal basis is evaluated. For the calculation of a univariate polynomial, *new-vars* should be a list with one element.

maxdeg is an upper limit for the degrees. The algorithm stops with an error message, if this limit is reached.

A warning occurs if the ideal is not zero dimensional.

`glexconvert` is an implementation of the FLGM algorithm by FAUGÈRE, GIANNI, LAZARD and MORA [FGLM93]. Often, the calculation of a Gröbner basis with a graded ordering and subsequent conversion to *lex* is faster than a direct *lex* calculation. Additionally, `glexconvert` can be used to transform a *lex* basis into one with different variable sequence, and it supports the calculation of a univariate polynomial. If the latter exists, the algorithm is even applicable in the non zero-dimensional case, if such a polynomial exists. If the polynomial does not exist, the algorithm computes until `maxdeg` has been reached.

```
torder({{w,p,z,t,s,b},gradlex)

g  :=  groebner  { f1 := 45*p + 35*s -165*b -36,
       35*p + 40*z + 25*t - 27*s, 15*w + 25*p*s +30*z
      -18*t -165*b**2, -9*w + 15*p*t  + 20*z*s,
      w*p + 2*z*t - 11*b**3, 99*w - 11*s*b +3*b**2,
      b**2 + 33/50*b + 2673/10000};

g := {60000*w + 9500*b + 3969,

    1800*p - 3100*b - 1377,

    18000*z + 24500*b + 10287,

    750*t - 1850*b + 81,

    200*s - 500*b - 9,
           2
    10000*b  + 6600*b + 2673}

 glexconvert(g,{w,p,z,t,s,b},maxdeg=5,newvars={w});

              2
  100000000*w  + 2780000*w + 416421

 glexconvert(g,{w,p,z,t,s,b},maxdeg=5,newvars={p});

        2
  6000*p  - 2360*p + 3051
```

`groebner_walk`: Conversion of a (General) Total Degree Basis into a Lex One

The algorithm `groebner_walk` convertes from an arbitrary polynomial system a *graduated* basis of the given variable sequence to a *lex* one of the same sequence. The job is done by computing a sequence of Gröbner bases of correspondig monomial ideals, lifting the original system each time. The algorithm has been described (more generally) by [AGK96a],[AGK96b],[AG98] and [CKM97]. `groebner_walk` should be only called if the direct calculation of a *lex* Gröbner base does not work. The computation of `groebner_walk` includes some overhead (e. g. the computation divides polynomials). Normally `torder` must be called before to define the variables and the variable sorting. The reordering of variables makes no sense with `groebner_walk`; so do not call `groebner_walk` with `groebopt` on!

`groebner_walk` g

where g is a polynomial ideal basis computed under *gradlex* or under *weighted* with a one-element, non zero weight vector with only one element, repeated for each variable. The result is a corresponding *lex* basis (if that is computable), independent of the degree of the ideal (even for non zero degree ideals). The variabe `gvarslast` is not set.variable

`groebnerf`: Factorizing Gröbner Bases

Background

If Gröbner bases are computed in order to solve systems of equations or to find the common roots of systems of polynomials, the factorizing version of the Buchberger algorithm can be used. The theoretical background is simple: if a polynomial p can be represented as a product of two (or more) polynomials, e.g. $h = f * g$, then h vanishes if and only if one of the factors vanishes. So if during the calculation of a Gröbner basis h of the above form is detected, the whole problem can be split into two (or more) disjoint branches. Each of the branches is simpler than the complete problem; this saves computing time and space. The result of this type of computation is a list of (partial) Gröbner bases; the solution set of the original problem is the union of the solutions of the partial problems, ignoring the multiplicity of an individual solution. If a branch results in a basis $\{1\}$, then there is no common zero, i.e. no additional solution for the original problem, contributed by this branch.

`groebnerf` Call

The syntax of `groebnerf` is the same as for `groebner`.

$$\texttt{groebnerf}(\{exp1, exp2, \ldots, expm\}[, \{\}, \{nz1, \ldots nzk\}]);$$

where $\{exp1, exp2, \ldots, expm\}$ is a given list of expressions or equations, and $\{nz1, \ldots nzk\}$ is an optional list of polynomials known to be non-zero.

`groebnerf` tries to separate polynomials into individual factors and to branch the computation in a recursive manner (factorization tree). The result is a list of partial Gröbner bases. If no factorization can be found or if all branches but one lead to the trivial basis $\{1\}$, the result has only one basis; nevertheless it is a list of lists of polynomials. If no solution is found, the result will be $\{\{1\}\}$. Multiplicities (one factor with a higher power, the same partial basis twice) are deleted as early as possible in order to speed up the calculation. The factorizing is controlled by some switches.

As a side effect, the sequence of variables is stored as a REDUCE list in the shared variable

`gvarslast`.

If `gltbasis` is on, a corresponding list of leading term bases is also produced and is available in the variable `gltb`.

The third parameter of `groebnerf` allows one to declare some polynomials nonzero. If any of these is found in a branch of the calculation the branch is cancelled. This can be used to save a substantial amount of computing time. The second parameter must be included as an empty list if the third parameter is to be used.

```
torder({x,y},lex)$
groebnerf { 3*x**2*y + 2*x*y + y + 9*x**2 + 5*x = 3,
            2*x**3*y - x*y - y + 6*x**3 - 2*x**2
              - 3*x = -3,
             x**3*y + x**2*y + 3*x**3 + 2*x**2 };
```

```
    {{y - 3,x},

                      2
 {2*y + 2*x - 1,2*x  - 5*x - 5}}
```

It is obvious here that the solutions of the equations can be read off immediately.

All switches from `groebner` are valid for `groebnerf` as well:

```
groebopt
gltbasis
groebfullreduction
groebstat
trgroeb
trgroebs
rgroeb1
```

Additional switches for `groebnerf`:

`trgroebr` – All intermediate partial basis are printed when detected.

By default `trgroebr` is off.

`groebmonfac` `groebresmax` `groebrestriction`
These variables are described in the following paragraphs.

Suppression of Monomial Factors

The factorization in `groebnerf` is controlled by the following switches and variables. The variable `groebmonfac` is connected to the handling of "monomial factors". A monomial factor is a product of variable powers occurring as a factor, e.g. $x**2*y$ in $x**3*y-2*x**2*y**2$. A monomial factor represents a solution of the type "$x = 0$ or $y = 0$" with a certain multiplicity. With `groebnerf` the multiplicity of monomial factors is lowered to the value of the shared variable

`groebmonfac`

which by default is 1 (= monomial factors remain present, but their multiplicity is brought down). With

`groebmonfac` := 0

the monomial factors are suppressed completely.

Limitation on the Number of Results

The shared variable

`groebresmax`

controls the number of partial results. Its default value is 300. If `groebresmax` partial results are calculated, the calculation is terminated. `groebresmax` counts

all branches, including those which are terminated (have been computed already), give no contribution to the result (partial basis 1), or which are unified in the result with other (partial) bases. So the resulting number may be much smaller. When the limit of `groeresmax` is reached, a warning

```
warning:  GROEBRESMAX limit reached
```

is issued; this warning in any case has to be taken as a serious one. For "normal" calculations the `groebresmax` limit is not reached. *var* is a shared variable (with an integer value); it can be set in the algebraic mode to a different (positive integer) value.

Restriction of the Solution Space

In some applications only a subset of the complete solution set of a given set of equations is relevant, e.g. only nonnegative values or positive definite values for the variables. A significant amount of computing time can be saved if nonrelevant computation branches can be terminated early.

Positivity: If a polynomial has no (strictly) positive zero, then every system containing it has no nonnegative or strictly positive solution. Therefore, the Buchberger algorithm tests the coefficients of the polynomials for equal sign if requested. For example, in $13 * x + 15 * y * z$ can be zero with real nonnegative values for x, y and z only if $x = 0$ and $y = 0$ or $z = 0$; this is a sort of "factorization by restriction". A polynomial $13 * x + 15 * y * z + 20$ never can vanish with nonnegative real variable values.

Zero point: If any polynomial in an ideal has an absolute term, the ideal cannot have the origin point as a common solution.

By setting the shared variable

```
groebrestriction
```

`groebnerf` is informed of the type of restriction the user wants to impose on the solutions:

`groebrestiction:=nonnegative;`
only nonnegative real solutions are of interest

`groebrestriction:=positive;`
only nonnegative and nonzero solutions are of interest

`groebrestriction:=zeropoint;`
only solution sets which contain the point $\{0, 0, \ldots, 0\}$ are or interest.

If `groebnerf` detects a polynomial which formally conflicts with the restriction,

it either splits the calculation into separate branches, or, if a violation of the restriction is determined, it cancels the actual calculation branch.

`greduce`, `preduce`: Reduction of Polynomials

Background

Reduction of a polynomial "p" modulo a given sets of polynomials "b" is done by the reduction algorithm incorporated in the Buchberger algorithm. Informally it can be described for polynomials over a field as follows:

loop1: % head term elimination
if there is one polynomial b in B such that the leading
term of p is a multiple of the leading term of P do
$p := p - lt(p)/lt(b) * b$ (the leading term vanishes)
do this loop as long as possible;
loop2: % elimination of subsequent terms
for each term s in p do
if there is one polynomial b in B such that s is a
multiple of the leading term of p do
$p := p - s/lt(b) * b$ (the term s vanishes)
do this loop as long as possible;

If the coefficients are taken from a ring without zero divisors we cannot divide by each possible number like in the field case. But using that in the field case, $c * p$ is reduced to $c * q$, if p is reduced to q, for arbitrary numbers c, the reduction for the ring case uses the least c which makes the (field) reduction for $c * p$ integer. The result of this reduction is returned as (ring) reduction of p eventually after removing the content, i.e. the greatest common divisor of the coefficients. The result of this type of reduction is also called a pseudo reduction of p.

Reduction via Gröbner Basis Calculation

`greduce`$(exp, \{exp1, exp2, \ldots, expm\}]);$

where *exp* is an expression, and $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations.

`greduce` first converts the list of expressions $\{exp1, \ldots, expn\}$ to a Gröbner basis, and then reduces the given expression modulo that basis. An error results if the list of expressions is inconsistent. The returned value is an expression representing the reduced polynomial. As a side effect, `greduce` sets the variable `gvarslast` in the same manner as `groebner` does.

Reduction with Respect to Arbitrary Polynomials

$$\texttt{preduce}(exp, \{exp1, exp2, \ldots, expm\});$$

where *expm* is an expression, and $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations.

`preduce` reduces the given expression modulo the set $\{exp1, \ldots, expm\}$. If this set is a Gröbner basis, the obtained reduced expression is uniquely determined. If not, then it depends on the subsequence of the single reduction steps (see 2). `preduce` does not check whether $\{exp1, exp2, \ldots, expm\}$ is a Gröbner basis in the actual order. Therefore, if the expressions are a Gröbner basis calculated earlier with a variable sequence given explicitly or modified by optimization, the proper variable sequence and term order must be activated first.

Example 3(`preduce` called with a Gröbner basis):

```
torder({x,y},lex);
gb:=groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
             2*x**3*y - x*y - y + 6*x**3 - 2*x**2
               - 3*x + 3,
             x**3*y + x**2*y + 3*x**3 + 2*x**2}$
preduce (5*y**2 + 2*x**2*y + 5/2*x*y + 3/2*y
           + 8*x**2 + 3/2*x - 9/2, gb);

    2
   y
```

`greduce_orders`: Reduction with several term orders

The shortest polynomial with different polynomial term orders is computed with the operator `greduce_orders`:

`greduce_orders`(exp**, {**$exp1$**, $exp2$**, ..., **$expm$**} [,{**$v_1, v_2, \ldots, v_n$**}]);**
where *exp* is an expression and $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations. The list of variables $v_1, v_2 \ldots v_n$ may be omitted; if set, the variables must be a list.

The expression *exp* is reduced by `greduce` with the orders in the shared variable `gorders`, which must be a list of term orders (if set). By default it is set to

{*revgradlex*, *gradlex*, *lex*}

The shortest polynomial is the result. The order with the shortest polynomial is set to the shared variable `gorder`. A Gröbner basis of the system $\{exp1, exp2, \ldots,$

$expm\}$ is computed for each element of *orders*. With the default setting `gorders` in most cases will be set to *revgradlex*. If the variable set is given, these variables are taken; otherwise all variables of the system $\{exp1, exp2, \ldots, expm\}$ are extracted.

The Gröbner basis computations can take some time; if interrupted, the intermediate result of the reduction is set to the shared variable `greduce_result`, if one is done already. However, this is not nesessarily the minimal form.

If the variable `gorders` should be set to orders with a parameter, the term oder has to be replaced by a list; the first element is the term oder selected, followed by its parameter(s), e.g.

$$\mathit{orders} := \{\{\mathit{gradlexgradlex}, 2\}, \{\mathit{lexgradlex}, 2\}\}$$

Reduction Tree

In some case not only are the results produced by `greduce` and `preduce` of interest, but the reduction process is of some value too. If the switch

`groebprot`

is set on, `groebner`, `greduce` and `preduce` produce as a side effect a trace of their work as a REDUCE list of equations in the shared variable

`groebprotfile`.

Its value is a list of equations with a variable "candidate" playing the role of the object to be reduced. The polynomials are cited as "*poly1*", "*poly2*", ... If read as assignments, these equations form a program which leads from the reduction input to its result. Note that, due to the pseudo reduction with a ring as the coefficient domain, the input coefficients may be changed by global factors.

Example 4

```
on groebprot$

preduce(5*y**2 + 2*x**2*y + 5/2*x*y + 3/2*y + 8*x**2

    + 3/2*x - 9/2, gb);



     2
    y
```

```
groebprotfile;

    candidate
         2          2                          2
     =4*x *y + 16*x  + 5*x*y + 3*x + 10*y  + 3*y - 9,

              2
     poly1=8*x - 2*y  + 5*y + 3,

              3      2
     poly2=2*y  - 3*y  - 16*y + 21,
     candidate=2*candidate,
     candidate= - x*y*poly1 + candidate,
     candidate= - 4*x*poly1 + candidate,
     candidate=4*candidate,

                  3
     candidate= - y *poly1 + candidate,
     candidate=2*candidate,

                    2
     candidate= - 3*y *poly1 + candidate,
     candidate=13*y*poly1 + candidate,
     candidate=candidate + 6*poly1,

                    2
     candidate= - 2*y *poly2 + candidate,
     candidate= - y*poly2 + candidate,
     candidate=candidate + 6*poly2
```

This means

$$16(5y^2 + 2x^2y + \frac{5}{2}xy + \frac{3}{2}y + 8x^2 + \frac{3}{2}x - \frac{9}{2}) =$$

$$(-8xy - 32x - 2y^3 - 3y^2 + 13y + 6)\,\text{poly1}$$

$$+ (-2y^2 - 2y + 6)\,\text{poly2} + y^2.$$

Tracing with `groebnert` and `preducet`

Given a set of polynomials $\{f_1, \ldots, f_k\}$ and their Gröbner basis $\{g_1, \ldots, g_l\}$, it is well known that there are matrices of polynomials C_{ij} and D_{ji} such that

$$f_i = \sum_j C_{ij} g_j \quad \text{and} \quad g_j = \sum_i D_{ji} f_i$$

and these relations are needed explicitly sometimes. In BUCHBERGER [Buc85], such cases are described in the context of linear polynomial equations. The standard technique for computing the above formulae is to perform Gröbner reductions, keeping track of the computation in terms of the input data. In the current package such calculations are performed with (an internally hidden) cofactor technique: the user has to assign unique names to the input expressions and the arithmetic combinations are done with the expressions and with their names simultaneously. So the result is accompanied by an expression which relates it algebraically to the input values.

There are two complementary operators with this feature: `groebnert` and `preducet`; functionally they correspond to `groebner` and `preduce`. However, the sets of expressions here ***must be*** equations with unique single identifiers on their left side and the *lhs* are interpreted as names of the expressions. Their results are sets of equations (`groebnert`) or equations (`preducet`), where a *lhs* is the computed value, while the *rhs* is its equivalent in terms of the input names.

Example 5

We calculate the Gröbner basis for an ellipse (named "$p1$") and a line (named "$p2$"); $p2$ is member of the basis immediately and so the corresponding first result element is of a very simple form; the second member is a combination of $p1$ and $p2$ as shown on the *rhs* of this equation:

```
gb1:=groebnert {p1=2*x**2+4*y**2-100,p2=2*x-y+1};

gb1 := {2*x - y + 1=p2,
            2
         9*y  - 2*y - 199= - 2*x*p2 - y*p2 + 2*p1 + p2}
```

Example 6

We want to reduce the polynomial `x**2` *wrt* the above Gröbner basis and need knowledge about the reduction formula. We therefore extract the basis polynomials from $gb1$, assign unique names to them (here $g1$, $g2$) and call `preducet`. The polynomial to be reduced here is introduced with the name Q, which then appears on the *rhs* of the result. If the name for the polynomial is omitted, its formal value is used on the right side too.

```
gb2 := for k := 1:length gb1 collect
      mkid(g,k) = lhs part(gb1,k)$
preducet (q=x**2,gb2);

- 16*y + 208= - 18*x*g1 - 9*y*g1 + 36*q + 9*g1 - g2
```

This output means

$$x^2 = (\frac{1}{2}x + \frac{1}{4}y - \frac{1}{4})g1 + \frac{1}{36}g2 + (-\frac{4}{9}y + \frac{52}{9}).$$

Example 7

If we reduce a polynomial which is member of the ideal, we consequently get a result with *lhs* zero:

```
preducet(q=2*x**2+4*y**2-100,gb2);

0= - 2*x*g1 - y*g1 + 2*q + g1 - g2
```

This means

$$q = (x + \frac{1}{2}y - \frac{1}{2})g1 + \frac{1}{2}g2.$$

With these operators the matrices C_{ij} and D_{ji} are available implicitly, D_{ji} as side effect of `groebnert`, c_{ij} by *calls* of `preducet` of f_i *wrt* $\{g_j\}$. The latter by definition will have the *lhs* zero and a *rhs* with linear f_i.

If $\{1\}$ is the Gröbner basis, the `groebnert` calculation gives a "proof", showing, how 1 can be computed as combination of the input polynomials.

Remark: Compared to the non-tracing algorithms, these operators are much more time consuming. So they are applicable only on small sized problems.

Gröbner Bases for Modules

Given a polynomial ring, e.g. $r = z[x_1 \cdots x_k]$ and an integer $n > 1$: the vectors with n elements of r form a *module* under vector addition (= componentwise addition) and multiplication with elements of r. For a submodule given by a finite basis a Gröbner basis can be computed, and the facilities of the GROEBNER package can be used except the operators `groebnerf` and `groesolve`.

The vectors are encoded using auxiliary variables which represent the unit vectors in the module. E.g. using v_1, v_2, v_3 the module element $[x_1^2, 0, x_1 - x_2]$ is represented as $x_1^2 v_1 + x_1 v_3 - x_2 v_3$. The use of v_1, v_2, v_3 as unit vectors is set up by assigning the set of auxiliary variables to the shared variable `gmodule`, e.g.

```
   gmodule := {v1,v2,v3};
```

After this declaration all monomials built from these variables are considered as an algebraically independent basis of a vector space. However, you had best use them only linearly. Once *gmodule* has been set, the auxiliary variables automatically will be added to the end of each variable list (if they are not yet member there). Example:

```
   torder({x,y,v1,v2,v3},lex)$
   gmodule := {v1,v2,v3}$
   g:=groebner{x^2*v1 + y*v2,x*y*v1 - v3,2y*v1 + y*v3};

        2
g := {x *v1 + y*v2,

                2
       x*v3 + y *v2,

        3
       y *v2 - 2*v3,

       2*y*v1 + y*v3}

   preduce((x+y)^3*v1,g);

              1   3         2
 - x*y*v2 - ---*y *v3 - 3*y *v2 + 3*y*v3
              2
```

In many cases a total degree oriented term order will be adequate for computations in modules, e.g. for all cases where the submodule membership is investigated. However, arranging the auxiliary variables in an elimination oriented term order can give interesting results. E.g.

```
   p1:=(x-1)*(x^2-x+3)$  p2:=(x-1)*(x^2+x-5)$
   gmodule := {v1,v2,v3};
   torder({v1,x,v2,v3},lex)$
   gb:=groebner {p1*v1+v2,p2*v1+v3};

gb := {30*v1*x - 30*v1 + x*v2 - x*v3 + 5*v2 - 3*v3,

         2       2
        x *v2 - x *v3 + x*v2 + x*v3 - 5*v2 - 3*v3}
```

```
   g:=coeffn(first gb,v1,1);

g := 30*(x - 1)

   c1:=coeffn(first gb,v2,1);

c1 := x + 5

   c2:=coeffn(first gb,v3,1);

c2 :=  - x - 3

   c1*p1 + c2*p2;

30*(x - 1)
```

Here two polynomials are entered as vectors $[p_1, 1, 0]$ and $[p_2, 0, 1]$. Using a term ordering such that the first dimension ranges highest and the other components lowest, a classical cofactor computation is executed just as in the extended Euclidean algorithm. Consequently the leading polynomial in the resulting basis shows the greatest common divisor of p_1 and p_2, found as a coefficient of v_1 while the coefficients of v_2 and v_3 are the cofactors c_1 and c_2 of the polynomials p_1 and p_2 with the relation $gcd(p_1, p_2) = c_1p_1 + c_2p_2$.

Additional Orderings

Besides the basic orderings, there are ordering options that are used for special purposes.

Separating the Variables into Groups

It is often desirable to separate variables and formal parameters in a system of polynomials. This can be done with a *lex* Gröbner basis. That however may be hard to compute as it does more separation than necessary. The following orderings group the variables into two (or more) sets, where inside each set a classical ordering acts, while the sets are handled via their total degrees, which are compared in elimination style. So the Gröbner basis will eliminate the members of the first set, if algebraically possible. *torder* here gets an additional parameter which describe the grouping

torder (*vl*, *gradlexgradlex*, n)
torder (*vl*, *gradlexrevgradlex*,n)
torder (*vl*, *lexgradlex*, n)
torder (*vl*, *lexrevgradlex*, n)

Here the integer n is the number of variables in the first group and the names combine the local ordering for the first and second group, e.g.

lexgradlex, 3 for $\{x_1, x_2, x_3, x_4, x_5\}$:
$x_1^{i_1} \dots x_5^{i_5} \gg x_1^{j_1} \dots x_5^{j_5}$
if $(i_1, i_2, i_3) \gg_{lex} (j_1, j_2, j_3)$
or $(i_1, i_2, i_3) = (j_1, j_2, j_3)$
and $(i_4, i_5) \gg_{gradlex} (j_4, j_5)$

Note that in the second place there is no *lex* ordering available; that would not make sense.

Weighted Ordering

The statement

`torder`(*vl*, weighted, $\{n_1, n_2, n_3 \dots\}$);

establishes a graduated ordering, where the exponents are first multiplied by the given weights. If there are less weight values than variables, the weight 1 is added automatically. If the weighted degree calculation is not decidable, a *lex* comparison follows.

Graded Ordering

The statement

`torder`(*vl*, graded, $\{n_1, n_2, n_3 \dots\}$, $order_2$);

establishes a graduated ordering, where the exponents are first multiplied by the given weights. If there are less weight values than variables, the weight 1 is added automatically. If the weighted degree calculation is not decidable, the term order $order_2$ specified in the following argument(s) is used. The ordering *graded* is designed primarily for use with the operator `dd_groebner`.

Matrix Ordering

The statement

`torder`(*vl*, matrix, m);

where m is a matrix with integer elements and row length which corresponds to the variable number. The exponents of each monomial form a vector; two monomials are compared by multiplying their exponent vectors first with m and comparing the resulting vector lexicographically. E.g. the unit matrix establishes the classical *lex* term order mode, a matrix with a first row of ones followed by the rows of a unit matrix corresponds to the *gradlex* ordering.

The matrix m must have at least as many rows as columns; a non–square matrix contains redundant rows. The matrix must have full rank, and the top non–zero element of each column must be positive.

The generality of the matrix based term order has its price: the computing time spent in the term sorting is significantly higher than with the specialized term orders. To overcome this problem, you can compile a matrix term order ; the compilation reduces the computing time overhead significantly. If you set the switch `comp` on, any new order matrix is compiled when any operator of the GROEBNER package accesses it for the first time. Alternatively you can compile a matrix explicitly

```
torder_compile(<n>,<m>);
```

where $< n >$ is a name (an identifier) and $< m >$ is a term order matrix. `torder_compile` transforms the matrix into a LISP program, which is compiled by the LISP compiler when `comp` is on or when you generate a fast loadable module. Later you can activate the new term order by using the name $< n >$ in a `torder` statement as term ordering mode.

Gröbner Bases for Graded Homogeneous Systems

For a homogeneous system of polynomials under a term order *graded*, *gradlex*, *revgradlex* or *weighted* a Gröbner Base can be computed with limiting the grade of the intermediate s–polynomials:

`dd_groebner`($d1, d2, \{p_1, p_2, \ldots\}$)**;**

where $d1$ is a non-negative integer and $d2$ is an integer $> d1$ or "infinity". A pair of polynomials is considered only if the grade of the lcm of their head terms is between $d1$ and $d2$. See [BWK93] for the mathematical background. For the term

orders *graded* or *weighted* the (first) weight vector is used for the grade computation. Otherwise the total degree of a term is used.

20.26.4 Ideal Decomposition & Equation System Solving

Based on the elementary Gröbner operations, the GROEBNER package offers additional operators, which allow the decomposition of an ideal or of a system of equations down to the individual solutions.

Solutions Based on Lex Type Gröbner Bases

groesolve: Solution of a Set of Polynomial Equations

The `groesolve` operator incorporates a macro algorithm; lexical Gröbner bases are computed by `groebnerf` and decomposed into simpler ones by ideal decomposition techniques; if algebraically possible, the problem is reduced to univariate polynomials which are solved by `solve`; if `rounded` is on, numerical approximations are computed for the roots of the univariate polynomials.

$$groesolve(\{exp1, exp2, \ldots, expm\}[, \{var1, var2, \ldots, varn\}]);$$

where $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations, $\{var1, var2, \ldots, varn\}$ is an optional list of variables.

The result is a set of subsets. The subsets contain the solutions of the polynomial equations. If there are only finitely many solutions, then each subset is a set of expressions of triangular type $\{exp1, exp2, \ldots, expn\}$, where $exp1$ depends only on $var1$, $exp2$ depends only on $var1$ and $var2$ etc. until $expn$ which depends on $var1, \ldots, varn$. This allows a successive determination of the solution components. If there are infinitely many solutions, some subsets consist in less than n expressions. By considering some of the variables as "free parameters", these subsets are usually again of triangular type.

Example 8(Intersubsections of a line with a circle):

$$groesolve(\{x**2 - y**2 - a, p*x + q*y + s\}, \{x, y\});$$

```
                  2      2    2              2    2
{{x=(sqrt( - a*p  + a*q  + s )*q - p*s)/(p  - q ),
                    2      2    2              2    2
  y= - (sqrt( - a*p  + a*q  + s )*p - q*s)/(p  - q )},
                   2      2    2              2    2
 {x= - (sqrt( - a*p  + a*q  + s )*q + p*s)/(p  - q ),
                  2      2    2              2    2
```

```
  y=(sqrt( - a*p  + a*q  + s )*p + q*s)/(p  - q )}}
```

If the system is zero–dimensional (has a number of isolated solutions), the algorithm described in [Hil99] is used, if the decomposition leaves a polynomial with mixed leading term. Hillebrand has written the article and Möller was the tutor of this job.

The reordering of the `groesolve` variables is controlled by the REDUCE switch `varopt`. If `varopt` is `on` (which is the default of `varopt`), the variable sequence is optimized (the variables are reordered). If `varopt` is `off`, the given variable sequence is taken (if no variables are given, the order of the REDUCE system is taken instead). In general, the reordering of the variables makes the Gröbner basis computation significantly faster. A variable dependency, declare by one (or several) `depend` statements, is regarded (if `varopt` is `on`). The switch `groebopt` has no meaning for `groesolve`; it is stored during its processing.

groepostproc: Postprocessing of a Gröbner Basis

In many cases, it is difficult to do the general Gröbner processing. If a Gröbner basis with a *lex* ordering is calculated already (e.g., by very individual parameter settings), the solutions can be derived from it by a call to `groepostproc`. `groesolve` is functionally equivalent to a call to `groebnerf` and subsequent calls to `groepostproc` for each partial basis.

$$groepostproc(\{exp1, exp2, \ldots, expm\}[, \{var1, var2, \ldots, varn\}]);$$

where $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions, $\{var1, var2, \ldots, varn\}$ is an optional list of variables. The expressions must be a *lex* Gröbner basis with the given variables; the ordering must be still active.

The result is the same as with `groesolve`.

```
groepostproc({x3**2 + x3 + x2 - 1,
              x2*x3 + x1*x3 + x3 + x1*x2 + x1 + 2,
              x2**2 + 2*x2 - 1,
              x1**2 - 2},{x3,x2,x1});

{{x3= - sqrt(2),

  x2=sqrt(2) - 1,

  x1=sqrt(2)},

 {x3=sqrt(2),
```

```
 x2= - (sqrt(2) + 1),

 x1= - sqrt(2)},

     sqrt(4*sqrt(2) + 9) - 1
{x3=-----------------------,
                2

 x2= - (sqrt(2) + 1),

 x1=sqrt(2)},

      - (sqrt(4*sqrt(2) + 9) + 1)
{x3=----------------------------,
                  2

 x2= - (sqrt(2) + 1),

 x1=sqrt(2)},

     sqrt( - 4*sqrt(2) + 9) - 1
{x3=---------------------------,
                 2

 x2=sqrt(2) - 1,

 x1= - sqrt(2)},

      - (sqrt( - 4*sqrt(2) + 9) + 1)
{x3=-------------------------------,
                   2

 x2=sqrt(2) - 1,

 x1= - sqrt(2)}}
```

Idealquotient: Quotient of an Ideal and an Expression

Let i be an ideal and f be a polynomial in the same variables. Then the algebraic quotient is defined by

$$i : f = \{p \mid p * f \text{ member of } i\} .$$

The ideal quotient $i : f$ contains i and is obviously part of the whole polynomial ring, i.e. contained in $\{1\}$. The case $i : f = \{1\}$ is equivalent to f being a member of i. The other extremal case, $i : f = i$, occurs, when f does not vanish at any general zero of i. The explanation of the notion "general zero" introduced by van der Waerden, however, is beyond the aim of this manual. The operation of `groesolve`/`groepostproc` is based on nested ideal quotient calculations.

If i is given by a basis and f is given as an expression, the quotient can be calculated by

`idealquotient` ({⟨*exp1*⟩, ⟨*exp2*⟩,..., ⟨*expm*⟩}, ⟨*exp*⟩);

where {⟨*exp1*⟩, ⟨*exp2*⟩, ..., ⟨*expm*⟩} is a list of any number of expressions or equations, ⟨*exp*⟩ is a single expression or equation.

`idealquotient` calculates the algebraic quotient of the ideal i with the basis $\{exp1, exp2, \ldots, expm\}$ and *exp* with respect to the variables given or extracted. $\{exp1, exp2, \ldots, expm\}$ is not necessarily a Gröbner basis. The result is the Gröbner basis of the quotient.

Saturation: Saturation of an Ideal and an Expression

The `saturation` operator computes the quotient on an ideal and an arbitrary power of an expression $exp * *n$ with arbitrary n. The call is

$$saturation(\{exp1, \ldots, expm\}, exp);$$

where $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations, *exp* is a single expression or equation.

saturation calls *idealquotient* several times, until the result is stable, and returns it.

Operators for Gröbner Bases in all Term Orderings

In some cases where no Gröbner basis with lexical ordering can be calculated, a calculation with a total degree ordering is still possible. Then the Hilbert polynomial gives information about the dimension of the solutions space and for finite sets of solutions univariate polynomials can be calculated. The solutions of the equation system then is contained in the cross product of all solutions of all univariate polynomials.

Hilbertpolynomial: Hilbert Polynomial of an Ideal

This algorithm was contributed by JOACHIM HOLLMAN, Royal Institute of Technology, Stockholm (private communication).

`hilbertpolynomial (`⟨*exp1:expression*⟩`, ... ,`⟨*expm:expression*⟩`)`

where $\{exp1, \ldots, expm\}$ is a list of any number of expressions or equations.

`hilertpolynomial` calculates the Hilbert polynomial of the ideal with basis $\{exp1, \ldots, expm\}$ with respect to the variables given or extracted provided the given term ordering is compatible with the degree, such as the *gradlex*- or *revgradlex*-ordering. The term ordering of the basis must be active and $\{exp1, \ldots, expm\}$ should be a Gröbner basis with respect to this ordering. The Hilbert polynomial gives information about the cardinality of solutions of the system $\{exp1, \ldots, expm\}$: if the Hilbert polynomial is an integer, the system has only a discrete set of solutions and the polynomial is identical with the number of solutions counted with their multiplicities. Otherwise the degree of the Hilbert polynomial is the dimension of the solution space.

If the Hilbert polynomial is not a constant, it is constructed with the variable "x" regardless of whether x is member of $\{var1, \ldots, varn\}$ or not. The value of this polynomial at sufficiently large numbers "x" is the difference of the dimension of the linear vector space of all polynomials of degree $\leq x$ minus the dimension of the subspace of all polynomials of degree $\leq x$ which belong also to the ideal.

x must be an undefined variable or the value of x must be an undefined variable; otherwise a warning is given and a new (generated) variable is taken instead.

Remark: The number of zeros in an ideal and the Hilbert polynomial depend only on the leading terms of the Gröbner basis. So if a subsequent Hilbert calculation is planned, the Gröbner calculation should be performed with `on gltbasis` and the value of `gltb` (or its elements in a `groebnerf` context) should be given to `hilbertpolynomial`. In this manner, a lot of computing time can be saved in the case of long calculations.

20.26.5 Calculations "by Hand"

The following operators support explicit calculations with polynomials in a distributive representation at the REDUCE top level. So they allow one to do Gröbner type evaluations stepwise by separate calls. Note that the normal REDUCE arithmetic can be used for arithmetic combinations of monomials and polynomials.

Representing Polynomials in Distributive Form

`gsort` ⟨*p*⟩

where ⟨*p*⟩ is a polynomial or a list of polynomials.

If ⟨*p*⟩ is a single polynomial, the result is a reordered version of ⟨*p*⟩ in the distributive representation according to the variables and the current term order mode; if ⟨*p*⟩ is a list, its members are converted into distributive representation and the result is the list sorted by the term ordering of the leading terms; zero polynomials are eliminated from the result.

```
torder({alpha,beta,gamma},lex);

dip := gsort(gamma*(alpha-1)**2*(beta+1)**2);
```

```
             2     2                 2
dip := alpha *beta *gamma + 2*alpha *beta*gamma

       2                       2
+ alpha *gamma - 2*alpha*beta *gamma

- 4*alpha*beta*gamma - 2*alpha*gamma

       2
 + beta *gamma + 2*beta*gamma + gamma
```

Splitting of a Polynomial into Leading Term and Reductum

`gsplit` ⟨*p*⟩

where ⟨*p*⟩ is a polynomial.

`gsplit` converts the polynomial ⟨*p*⟩ into distributive representation and splits it into leading monomial and reductum. The result is a list with two elements, the leading monomial and the reductum.

```
gsplit dip;
```

```
       2     2
{alpha *beta *gamma,

         2                   2
```

```
 2*alpha *beta*gamma + alpha *gamma

                2                              2
  - 2*alpha*beta *gamma - 4*alpha*beta*gamma

  - 2*alpha*gamma + beta *gamma + 2*beta*gamma

  + gamma}
```

Calculation of Buchberger's S-polynomial

`gspoly` (⟨*p1*⟩,⟨*p2*⟩)

where ⟨*p1*⟩ and ⟨*p2*⟩ are polynomials.

`gspoly` calculates the *s*-polynomial from ⟨*p1*⟩ and ⟨*p2*⟩.

Example for a complete calculation (taken from DAVENPORT ET AL. [DST93]):

```
torder({x,y,z},lex)$
g1  :=  x**3*y*z - x*z**2;
g2  :=  x*y**2*z - x*y*z;
g3  :=  x**2*y**2 - z;$

% first S-polynomial

g4  :=  gspoly(g2,g3);$

        2        2
 g4 := x *y*z - z

 % next S-polynomial

 p :=  gspoly(g2,g4); $

       2          2
 p := x *y*z - y*z

 % and reducing, here only by g4

 g5  :=  preduce(p,{g4});

             2    2
 g5 :=  - y*z  + z
```

```
% last S-polynomial}

g6  :=  gspoly(g4,g5);

       2  2    3
g6 := x *z  - z

% and the final basis sorted descending

gsort{g2,g3,g4,g5,g6};

  2  2
{x *y  - z,

  2        2
 x *y*z - z ,

  2  2    3
 x *z  - z ,

    2
 x*y *z - x*y*z,

       2    2
  - y*z  + z }
```

20.27 GUARDIAN: Guarded Expressions in Practice

Computer algebra systems typically drop some degenerate cases when evaluating expressions, e.g., x/x becomes 1 dropping the case $x = 0$. We claim that it is feasible in practice to compute also the degenerate cases yielding *guarded expressions*. We work over real closed fields but our ideas about handling guarded expression can be easily transferred to other situations. Using formulas as guards provides a powerful tool for heuristically reducing the combinatorial explosion of cases: equivalent, redundant, tautological, and contradictive cases can be detected by simplification and quantifier elimination. Our approach allows to simplify the expressions on the basis of simplification knowledge on the logical side. The method described in this paper is implemented in the REDUCE package GUARDIAN.

Authors: Andreas Dolzmann and Thomas Sturm

20.27.1 Introduction

It is meanwhile a well-known fact that evaluations obtained with the interactive use of computer algebra systems (CAS) are not entirely correct in general. Typically, some degenerate cases are dropped. Consider for instance the evaluation

$$\frac{x^2}{x} = x,$$

which is correct only if $x \neq 0$. The problem here is that CAS consider variables to be transcendental elements. The user, in contrast, has in mind variables in the sense of logic. In other words: The user does not think of rational functions but of terms.

Next consider the valid expression

$$\frac{\sqrt{x} + \sqrt{-x}}{x}.$$

It is meaningless over the reals. CAS often offer no choice than to interprete surds over the complex numbers even if they distinguish between a *real* and a *complex* mode.

Corless and Jeffrey [CJ92] have examined the behavior of a number of CAS with such input data. They come to the conclusion that simultaneous computation of all cases is exemplary but not feasible due to the combinatorial explosion of cases to be considered. Therefore, they suggest to ignore the degenerate cases but to provide the assumptions to the user on request. We claim, in contrast, that it is in fact feasible to compute all possible cases.

Our setting is as follows: Expressions are evaluated to *guarded expressions* consisting of possibly several conventional expressions guarded by quantifier-free for-

mulas. For the above examples, we would obtain

$$\left[\ \boldsymbol{x} \neq \mathbf{0} \ \middle|\ \boldsymbol{x}\ \right], \quad \left[\ \mathrm{F} \ \middle|\ \frac{\sqrt{\boldsymbol{x}}+\sqrt{-\boldsymbol{x}}}{\boldsymbol{x}}\ \right].$$

As the second example illustrates, we are working in ordered fields, more precisely in real closed fields. The handling of guarded expressions as described in this paper can, however, be easily transferred to other situations.

Our approach can also deal with redundant guarded expressions, such as

$$\left[\begin{array}{c|c} \mathrm{T} & |\boldsymbol{x}| - \boldsymbol{x} \\ x \geq 0 & 0 \\ x < 0 & -2x \end{array}\right]$$

which leads to algebraic simplification techniques based on logical simplification as proposed by Davenport and Faure [DF94].

We use *formulas* over the language of ordered rings as guards. This provides powerful tools for heuristically reducing the combinatorial explosion of cases: equivalent, redundant, tautological, and contradictive cases can be detected by *simplification* [DS97b] and *quantifier elimination* [Tar48, Col75, Wei88, RLW93, Wei97, Wei94]. In certain situations, we will allow the formulas also to contain extra functions such as $\sqrt{\cdot}$ or $|\cdot|$. Then we take care that there is no quantifier elimination applied.

Simultaneous computation of several cases concerning certain expressions being zero or not has been extensively investigated as *dynamic evaluation* [GD96, DR94a, DR94b, BGDW95]. It has also been extended to real closed fields [DGV96]. The idea behind the development of these methods is of a more theoretical nature than to overcome the problems with the interactive usage of CAS sketched above: one wishes to compute in algebraic (or real) extension fields of the rationals. Guarded expressions occur naturally when solving problems parametrically. Consider, e.g., the *Gröbner systems* used during the computation of *comprehensive Gröbner bases* [Wei92].

The algorithms described in this paper are implemented in the REDUCE package GUARDIAN. It is based on the REDUCE [Hea95, Mel95] package REDLOG [DS97a, DS96] implementing a formula data type with corresponding algorithms, in particular including simplification and quantifier elimination.

20.27.2 An outline of our method

Guarded expressions

A *guarded expression* is a scheme

$$\left[\begin{array}{c|c} \boldsymbol{\gamma_0} & \boldsymbol{t_0} \\ \gamma_1 & t_1 \\ \vdots & \vdots \\ \gamma_n & t_n \end{array}\right]$$

where each γ_i is a quantifier-free formula, the *guard*, and each t_i is an associated *conventional expression*. The idea is that some t_i is a valid interpretation iff γ_i holds. Each pair (γ_i, t_i) is called a *case*.

The first case (γ_0, t_0) is the *generic* case: t_0 is the expression the system would compute without our package, and γ_0 is the corresponding guard.

The guards γ_i need neither exclude one another, nor do we require that they form a complete case distinction. We shall, however, assume that all cases covered by a guarded expression are already covered by the generic case; in other words:

$$\bigwedge_{i=1}^{n} (\gamma_i \longrightarrow \gamma_0). \tag{20.79}$$

Consider the following evaluation of $|x|$ to a guarded expression:

$$\left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{|x|} \\ x \geq 0 & x \\ x < 0 & -x \end{array}\right].$$

Here the non-generic cases already cover the whole domain. The generic case is in some way *redundant*. It is just present for keeping track of the system's default behavior. Formally we have

$$\left(\bigvee_{i=1}^{n} \gamma_i\right) \longleftrightarrow \gamma_0. \tag{20.80}$$

As an example for a non-redundant, i.e., *necessary* generic case we have the evaluation of the reciprocal $\frac{1}{x}$:

$$\left[\begin{array}{c|c} \boldsymbol{x \neq 0} & \boldsymbol{\frac{1}{x}} \end{array}\right].$$

In every guarded expression, the generic case is explicitly marked as either necessary or redundant. The corresponding tag is inherited during the evaluation process.

Unfortunately it can happen that guarded expressions satisfy (20.80) without being tagged redundant, e.g., specialization of

$$\left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\sin x} \\ x=0 & 0 \end{array}\right]$$

to $x = 0$ if the system cannot evaluate $\sin(0)$. This does not happen if one claims for necessary generic cases to have, as the reciprocal above, no alternative cases at all. Else, in the sequel "redundant generic case" has to be read as "tagged redundant."

With guarded expressions, the evaluation splits into two independent parts: *Algebraic evaluation* and a subsequent *simplification* of the guarded expression obtained.

Guarding schemes

In the introduction we have seen that certain operators introduce case distinctions. For this, with each operator f there is a *guarding scheme* associated providing information on how to map $f(t_1, \ldots, t_m)$ to a guarded expression provided that one does not have to care for the argument expressions t_1, ..., t_m. In the easiest case, this is a rewrite rule

$$f(a_1, \ldots, a_m) \to G(a_1, \ldots, a_m).$$

The actual terms t_1, ..., t_m are simply substituted for the formal symbols a_1, ..., a_m into the generic guarded expression $G(a_1, \ldots, a_m)$. We give some examples:

$$\begin{aligned}
\frac{a_1}{a_2} &\to \left[\begin{array}{c|c} \boldsymbol{a_2 \neq 0} & \frac{\boldsymbol{a_1}}{\boldsymbol{a_2}} \end{array}\right] \\
\sqrt{a_1} &\to \left[\begin{array}{c|c} \boldsymbol{a_1 \geq 0} & \boldsymbol{\sqrt{a_1}} \end{array}\right] \\
\mathrm{sign}(a_1) &\to \left[\begin{array}{c|c} \mathrm{T} & \mathrm{sign}(\boldsymbol{a_1}) \\ a_1 > 0 & 1 \\ a_1 = 0 & 0 \\ a_1 < 0 & -1 \end{array}\right] \\
|a_1| &\to \left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{|a_1|} \\ a_1 \geq 0 & a_1 \\ a_1 < 0 & -a_1 \end{array}\right]
\end{aligned} \tag{20.81}$$

For functions of arbitrary arity, e.g., min or max, we formally assume infinitely many operators of the same name. Technically, we associate a procedure parameterized with the number of arguments m that generates the corresponding rewrite

rule. As `min_scheme(2)` we obtain, e.g.,

$$\min(a_1, a_2) \to \left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\min(a_1, a_2)} \\ a_1 \leq a_2 & a_1 \\ a_2 \leq a_1 & a_2 \end{array}\right],$$

while for higher arities there are more case distinctions necessary.

For later complexity analysis, we state the concept of a guarding scheme formally: a guarding scheme for an m-ary operator f is a map

$$\mathrm{gscheme}_f : \mathrm{E}^m \to \mathrm{GE}$$

where E is the set of expressions, and GE is the set of guarded expressions. This allows to split $f(t_1, \ldots, t_m)$ in dependence on the form of the parameter expressions $t_1, \ldots, t_m$.

Algebraic evaluation

Evaluating conventional expressions

The evaluation of conventional expressions into guarded expressions is performed recursively: Constants c evaluate to

$$\left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{c} \end{array}\right].$$

For the evaluation of $f(e_1, \ldots, e_m)$ the argument expressions e_1, …, e_m are recursively evaluated to guarded expressions

$$e_i' = \left[\begin{array}{c|c} \boldsymbol{\gamma_{i0}} & \boldsymbol{t_{i0}} \\ \gamma_{i1} & t_{i1} \\ \vdots & \vdots \\ \gamma_{in_i} & t_{in_i} \end{array}\right] \quad \text{for} \quad 1 \leq i \leq m. \tag{20.82}$$

Then the operator f is "moved inside" the e_i' by combining all cases, technically a simultaneous Cartesian product computation of both the sets of guards and the sets of terms:

$$\Gamma = \prod_{i=1}^{m} \{\gamma_{i0}, \ldots, \gamma_{in_i}\}, \quad T = \prod_{i=1}^{m} \{t_{i0}, \ldots, t_{in_i}\}. \tag{20.83}$$

This leads to the intermediate result

$$\left[\begin{array}{c|c} \boldsymbol{\gamma_{10} \wedge \cdots \wedge \gamma_{m0}} & \boldsymbol{f(t_{10}, \ldots, t_{m0})} \\ \vdots & \vdots \\ \gamma_{1n_1} \wedge \cdots \wedge \gamma_{m0} & f(t_{1n_1}, \ldots, t_{m0}) \\ \vdots & \vdots \\ \gamma_{1n_1} \wedge \cdots \wedge \gamma_{mn_m} & f(t_{1n_1}, \ldots, t_{mn_m}) \end{array}\right]. \tag{20.84}$$

The new generic case is exactly the combination of the generic cases of the e'_i. It is redundant if at least one of these combined cases is redundant.

Next, all non-generic cases containing at least one *redundant* generic constituent γ_{i0} in their guard are deleted. The reason for this is that generic cases are only used to keep track of the system default behavior. All other cases get the status of a non-generic case even if they contain necessary generic constituents in their guard.

At this point, we apply the guarding scheme of f to all remaining expressions $f(t_{1i_1}, \ldots, t_{mi_m})$ in the form (20.84) yielding a nested guarded expression

$$\left[\begin{array}{c|c} & \left[\begin{array}{c|c} \boldsymbol{\delta_{00}} & \boldsymbol{u_{00}} \\ \vdots & \vdots \\ \delta_{0k_0} & u_{0k_0} \end{array}\right] \\ \boldsymbol{\Gamma_0} & \\ \vdots & \vdots \\ & \left[\begin{array}{c|c} \boldsymbol{\delta_{N0}} & \boldsymbol{u_{N0}} \\ \vdots & \vdots \\ \delta_{Nk_N} & u_{Nk_N} \end{array}\right] \\ \Gamma_N & \end{array}\right], \tag{20.85}$$

which can be straightforwardly resolved to a guarded expression

$$\left[\begin{array}{c|c} \boldsymbol{\Gamma_0 \wedge \delta_{00}} & \boldsymbol{u_{00}} \\ \vdots & \vdots \\ \Gamma_0 \wedge \delta_{0k_0} & u_{0k_0} \\ \vdots & \vdots \\ \Gamma_N \wedge \delta_{N0} & u_{N0} \\ \vdots & \vdots \\ \Gamma_N \wedge \delta_{Nk_N} & u_{Nk_N} \end{array}\right].$$

This form is treated analogously to the form (20.84): The new generic case $(\Gamma_0 \wedge \delta_{00}, u_{00})$ is redundant if at least one of $\big(\Gamma_0, f(t_{10}, \ldots, t_{m0})\big)$ and (δ_{00}, u_{00}) is redundant. Among the non-generic cases all those containing redundant generic constituents in their guard are deleted, and all those containing necessary generic constituents in their guard get the status of an ordinary non-generic case.

Finally the standard evaluator of the system—`reval` in the case of REDUCE—is applied to all contained expressions, which completes the algebraic part of the evaluation.

Evaluating guarded expressions

The previous section was concerned with the evaluation of pure conventional expressions into guarded expressions. Our system currently combines both conventional and guarded expressions. We are thus faced with the problem of treating guarded subexpressions during evaluation.

When there is a *guarded* subexpression e_i detected during evaluation, all contained expressions are recursively evaluated to guarded expressions yielding a nested guarded expression of the form (20.85). This is resolved as described above yielding the evaluation subresult e_i'.

As a special case, this explains how guarded expressions are (re)evaluated to guarded expressions.

Example

We describe the evaluation of the expression $\min(x, |x|)$. The first argument $e_1 = x$ evaluates recursively to

$$e_1' = \left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{x} \end{array}\right] \tag{20.86}$$

with a necessary generic case. The nested x inside $e_2 = |x|$ evaluates to the same form (20.86). For obtaining e_2', we apply the guarding scheme (20.81) of the absolute value to the only term of (20.86) yielding

$$\left[\begin{array}{c|c} \mathrm{T} & \left[\begin{array}{c|c} \mathrm{T} & |\boldsymbol{x}| \\ x \geq 0 & x \\ x < 0 & -x \end{array}\right] \end{array}\right],$$

where the inner generic case is redundant. This form is resolved to

$$e_2' = \left[\begin{array}{c|c} \mathrm{T} \wedge \mathrm{T} & |\boldsymbol{x}| \\ \mathrm{T} \wedge x \geq 0 & x \\ \mathrm{T} \wedge x < 0 & -x \end{array}\right]$$

with a redundant generic case. The next step is the combination of cases by Cartesian product computation. We obtain

$$\left[\begin{array}{c|c} \mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T}) & \boldsymbol{\min(x, |x|)} \\ \mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0) & \min(x, x) \\ \mathrm{T} \wedge (\mathrm{T} \wedge x < 0) & \min(x, -x) \end{array}\right],$$

which corresponds to (20.84) above. For the outer $\min$, we apply the guarding scheme (20.27.2) to all terms yielding the nested guarded expression

$$\left[\begin{array}{c|c} \mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T}) & \left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\min(x, |x|)} \\ x \leq |x| & x \\ |x| \leq x & |x| \end{array}\right] \\ \mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0) & \left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\min(x, x)} \\ x \leq x & x \\ x \leq x & x \end{array}\right] \\ \mathrm{T} \wedge (\mathrm{T} \wedge x < 0) & \left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\min(x, -x)} \\ x \leq -x & x \\ -x \leq x & -x \end{array}\right] \end{array}\right],$$

which is in turn resolved to

$$\left[\begin{array}{c|c} (\mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T})) \wedge \mathrm{T} & \boldsymbol{\min(x, |x|)} \\ (\mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T})) \wedge x \leq |x| & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T})) \wedge |x| \leq x & |x| \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0)) \wedge \mathrm{T} & \min(x, x) \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x < 0)) \wedge \mathrm{T} & \min(x, -x) \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x < 0)) \wedge x \leq -x & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x < 0)) \wedge -x \leq x & -x \end{array}\right].$$

From this, we delete the two non-generic cases obtained by combination with the redundant generic case of the $\min$. The final result of the algebraic evaluation step is the following:

$$\left[\begin{array}{c|c} (\mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T})) \wedge \mathrm{T} & \boldsymbol{\min(x, |x|)} \\ (\mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T})) \wedge x \leq |x| & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge \mathrm{T})) \wedge |x| \leq x & |x| \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x < 0)) \wedge x \leq -x & x \\ (\mathrm{T} \wedge (\mathrm{T} \wedge x < 0)) \wedge -x \leq x & -x \end{array}\right]. \tag{20.87}$$

Worst-case complexity

Our measure of complexity $|G|$ for guarded expressions G is the number of contained cases:

$$\left|\left[\begin{array}{c|c} \boldsymbol{\gamma_0} & \boldsymbol{t_0} \\ \gamma_1 & t_1 \\ \vdots & \vdots \\ \gamma_n & t_n \end{array}\right]\right| = n + 1.$$

As in Section 20.27.2, consider an m-ary operator f, guarded expression arguments $e'_1, \ldots, e'_m$ as in equation (20.82), and the Cartesian product T as in equation (20.83). Then

$$\begin{aligned} |f(e'_1, \ldots, e'_m)| &\leq \sum_{(t_1,\ldots,t_m)\in T} |\mathrm{gscheme}_f(t_1, \ldots, t_m)| \\ &\leq \max_{(t_1,\ldots,t_m)\in T} |\mathrm{gscheme}_f(t_1, \ldots, t_m)| \cdot \#T \\ &= \max_{(t_1,\ldots,t_m)\in T} |\mathrm{gscheme}_f(t_1, \ldots, t_m)| \cdot \prod_{j=1}^{m} |e'_j| \\ &\leq \max_{(t_1,\ldots,t_m)\in T} |\mathrm{gscheme}_f(t_1, \ldots, t_m)| \cdot \left(\max_{1\leq j\leq m} |e'_j|\right)^m. \end{aligned}$$

In the important special case that the guarding scheme of f is a rewrite rule $f(a_1, \ldots, a_m) \to G$, the above complexity estimation simplifies to

$$|f(e'_1, \ldots, e'_m)| \leq |G| \cdot \prod_{j=1}^{m} |e'_j| \leq |G| \cdot \left(\max_{1 \leq j \leq m} |e'_j|\right)^m.$$

In other words: $|G|$ plays the role of a factor, which, however, depends on f, and $|f(e'_1, \ldots, e'_m)|$ is polynomial in the size of the e_i but exponential in the arity of f.

Simplification

In view of the increasing size of the guarded expressions coming into existence with subsequent computations, it is indispensable to apply simplification strategies. There are two different algorithms involved in the simplification of guarded expressions:

1. A *formula simplifier* mapping quantifier-free formulas to equivalent simpler ones.
2. Effective *quantifier elimination* for real closed fields over the language of ordered rings.

It is not relevant, which simplifier and which quantifier elimination procedure is actually used. We use the formula simplifier described in [DS97b]. Our quantifier elimination uses test point methods developed by Weispfenning [Wei88, RLW93, Wei97]. It is restricted to formulas obeying certain degree restrictions wrt. the quantified variables. As an alternative, REDLOG provides an interface to Hong's QEPCAD quantifier elimination package [HCJE93]. Compared to the simplification, the quantifier elimination is more time consuming. It can be turned off by a *switch*.

The following simplification steps are applied in the given order:

Contraction of cases This is restricted to the non-generic cases of the considered guarded expression. We contract different cases containing the same terms:

$$\left[\begin{array}{c|c} \boldsymbol{\gamma_0} & \boldsymbol{t_0} \\ \vdots & \vdots \\ \gamma_i & t_i \\ \vdots & \vdots \\ \gamma_j & t_i \\ \vdots & \vdots \end{array}\right] \quad \text{becomes} \quad \left[\begin{array}{c|c} \boldsymbol{\gamma_0} & \boldsymbol{t_0} \\ \vdots & \vdots \\ \gamma_i \vee \gamma_j & t_i \\ \vdots & \vdots \end{array}\right].$$

Simplification of the guards The simplifier is applied to all guards replacing them by simplified equivalents. Since our simplifier maps $\gamma \vee \gamma$ to γ, this together with the contraction of cases takes care for the deletion of duplicate cases.

Keep one tautological case If the guard of some non-generic case becomes "T," we delete all other non-generic cases. Else, if quantifier elimination is turned on, we try to detect a tautology by eliminating the universal closures $\underline{\forall}\gamma$ of the guards γ. This quantifier elimination is also applied to the guards of generic cases. These are, in case of success, simply replaced by "T" without deleting the case.

Remove contradictive cases A non-generic case is deleted if its guard has become "F." If quantifier elimination is turned on, we try to detect further contradictive cases by eliminating the existential closure $\underline{\exists}\gamma$ for each guard γ. This quantifier elimination is also applied to generic cases. In case of success they are not deleted but their guards are replaced by "F." Our assumption (20.79) allows then to delete all non-generic cases.

Example revisited

We turn back to the form (20.87) of our example $\min(x, |x|)$. Contraction of cases with subsequent simplification automatically yields

$$\left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\min(x, |x|)} \\ \mathrm{T} & x \\ |x| - x \leq 0 & |x| \\ \mathrm{F} & -x \end{array}\right],$$

of which only the tautological non-generic case survives:

$$\left[\begin{array}{c|c} \mathrm{T} & \boldsymbol{\min(x, |x|)} \\ \mathrm{T} & x \end{array}\right]. \tag{20.88}$$

Output modes

An *output mode* determines which part of the information contained in the guarded expressions is provided to the user. GUARDIAN knows the following output modes:

Matrix Output matrices in the style used throughout this paper. We have already seen that these can become very large in general.

Generic case Output only the generic case.

Generic term Output only the generic term. Thus the output is exactly the same as without the guardian package. If the condition of the generic case becomes "F," a *warning* "`contradictive situation`" is given. The computation can, however, be continued.

Note that output modes are restrictions concerning only the output; internally the system still computes with the complete guarded expressions.

A smart mode

Consider the evaluation result (20.88) of $\min(x, |x|)$. The *generic term* output mode would output $\min(x, |x|)$, although more precise information could be given, namely x. The problem is caused by the fact that generic cases are used to keep track of the system's default behavior. In this section we will describe an optional *smart mode* with a different notion of *generic case*. To begin with, we show why the problem can not be overcome by a "smart output mode."

Assume that there is an output mode which outputs x for (20.88). As the next computation involving (20.88) consider division by y. This would result in

$$\left[\begin{array}{c|c} \boldsymbol{y \neq 0} & \boldsymbol{\frac{\min(x,|x|)}{y}} \\ y \neq 0 & \frac{x}{y} \end{array}\right].$$

Again, there are identic conditions for the generic case and some non-generic case, and, again, the term belonging to the latter is simpler. Our mode would output $\frac{x}{y}$. Next, we apply the absolute value once more yielding

$$\left[\begin{array}{c|c} \boldsymbol{y \neq 0} & \boldsymbol{\frac{|\min(x,|x|)|}{|y|}} \\ xy \geq 0 \wedge y \neq 0 & \frac{x}{y} \\ xy < 0 \wedge y \neq 0 & \frac{-x}{y} \end{array}\right].$$

Here, the condition of the generic case differs from all other conditions. We thus have to output the generic term. For the user, the evaluation of $|\frac{x}{y}|$ results in $\frac{|\min(x,|x|)|}{|y|}$.

The smart mode can turn a non-generic case into a necessary generic one dropping the original generic case and all other non-generic cases. Consider, e.g., (20.88), where the conditions are equal, and the non-generic term is "simpler."

In fact, the relevant relationship between the conditions is that the generic condition *implies* the non-generic one. In other words: Some non-generic condition is not more restrictive than the generic condition, and thus covers the whole domain of the guarded expression. Note that from the implication and (20.79) we may conclude that the cases are even equivalent.

Implication is heuristically checked by simplification. If this fails, quantifier elimination provides a decision procedure. Note that our test point methods are incomplete in this regard due to the degree restrictions. Also it cannot be applied straightforwardly to guards containing operators that do not belong to the language of ordered rings.

Whenever we happen to detect a relevant implication, we actually turn the corresponding non-generic case into the generic one. From our motivation of non-generic cases, we may expect that non-generic expressions are generally more convenient than generic ones.

20.27.3 Examples

We give the results for the following computations as they are printed in the output mode *matrix* providing the full information on the computation result. The reader can derive himself what the output in the mode *generic case* or *generic term* would be.

- Smart mode or not:

$$\frac{1}{x^2+2x+1} = \left[\ \boldsymbol{x+1\neq 0}\ \middle|\ \tfrac{\mathbf{1}}{\boldsymbol{x^2+2x+1}}\ \right].$$

  The simplifier recognizes that the denominator is a square.

- Smart mode or not:

$$\frac{1}{x^2+2x+2} = \left[\ \mathrm{T}\ \middle|\ \tfrac{\mathbf{1}}{\boldsymbol{x^2+2x+2}}\ \right].$$

  Quantifier elimination recognizes the positive definiteness of the denominator.

- Smart mode:

$$|x|-\sqrt{x} = \left[\ \boldsymbol{x\geq 0}\ \middle|\ \boldsymbol{-\sqrt{x}+x}\ \right].$$

  The square root allows to forget about the negative branch of the absolute value.

- Smart mode:

$$|x^2+2x+1| = \left[\ \mathrm{T}\ \middle|\ \boldsymbol{x^2+2x+1}\ \right].$$

  The simplifier recognizes the positive semidefiniteness of the argument. REDUCE itself recognizes squares within absolute values only in very special cases such as $|x^2|$.

- Smart mode:
$$\min\bigl(x, \max(x, y)\bigr) = \left[\ \mathrm{T} \ \middle|\ \boldsymbol{x}\ \right].$$
Note that REDUCE does not know any rules about nested minima and maxima.

- Smart mode:
$$\min\bigl(\mathrm{sign}(x), -1\bigr) = \left[\ \mathrm{T} \ \middle|\ \boldsymbol{-1}\ \right].$$

- Smart mode or not:
$$|x| - x = \left[\begin{array}{c|c} \mathrm{T} & |\boldsymbol{x}| - \boldsymbol{x} \\ x \geq 0 & 0 \\ x < 0 & -2x \end{array}\right].$$
This example is taken from [DF94].

- Smart mode or not:
$$\sqrt{1 + x^2y^2(x^2 + y^2 - 3)} = \left[\ \mathrm{T} \ \middle|\ \sqrt{\boldsymbol{x^4y^2 + x^2y^4 - 3x^2y^2 + 1}}\ \right]$$
The *Motzkin polynomial* is recognized to be positive semidefinite by quantifier elimination.

The evaluation time for the last example is 119 ms on a SUN SPARC-4. This illustrates that efficiency is no problem with such interactive examples.

20.27.4 Outlook

This section describes possible extensions of the GUARDIAN. The extensions proposed in Section 20.27.4 on simplification of terms and Section 20.27.4 on a background theory are clear from a theoretical point of view but not yet implemented. Section 20.27.4 collects some ideas on the application of our ideas to the REDUCE integrator. In this field, there is some more theoretical work necessary.

Simplification of terms

Consider the expression $\mathrm{sign}(x)x - |x|$. It evaluates to the following guarded expression:
$$\left[\begin{array}{c|c} \mathrm{T} & -|\boldsymbol{x}| + \mathrm{sign}(\boldsymbol{x})\boldsymbol{x} \\ x \neq 0 & 0 \\ x = 0 & -x \end{array}\right].$$
This suggests to substitute $-x$ by 0 in the third case, which would in turn allow to contract the two non-generic cases yielding
$$\left[\begin{array}{c|c} \mathrm{T} & -|\boldsymbol{x}| + \mathrm{sign}(\boldsymbol{x})\boldsymbol{x} \\ \mathrm{T} & 0 \end{array}\right].$$

In smart mode second case would then become the only generic case.

Generally, one would proceed as follows: If the guard is a conjunction containing as toplevel equations

$$t_1 = 0, \quad \ldots, \quad t_k = 0,$$

reduce the corresponding expression modulo the set of univariate linear polynomials among $t_1, \ldots, t_k$.

A more general approach would reduce the expression modulo a Gröbner basis of all the $t_1, \ldots, t_k$. This leads, however, to larger expressions in general.

One can also imagine to make use of non-conjunctive guards in the following way:

1. Compute a DNF of the guard.
2. Split the case into several cases corresponding to the conjunctions in the DNF.
3. Simplify the terms.
4. Apply the standard simplification procedure to the resulting guarded expression. Note that it includes *contraction of cases*.

According to experiences with similar ideas in the "Gröbner simplifier" described in [DS97b], this should work well.

Background theory

In practice one often computes with quantities guaranteed to lie in a certain range. For instance, when computing an electrical resistance, one knows in advance that it will not be negative. For such cases one would like to have some facility to provide external information to the system. This can then be used to reduce the complexity of the guarded expressions.

One would provide a function `assert(`φ`)`, which asserts the formula φ to hold. Successive applications of `assert` establish a *background theory*, which is a set of formulas considered conjunctively. The information contained in the background theory can be used with the guarded expression computation. The user must, however, not rely on all the background information to be actually used.

Technically, denote by Φ the (conjunctive) background theory. For the *simplification of the guards*, we can make use of the fact that our simplifier is designed to simplify wrt. a theory, cf. [DS97b]. For proving that some guard γ is *tautological*, we try to prove

$$\underline{\forall}(\Phi \longrightarrow \gamma)$$

instead of $\underline{\forall}\gamma$. Similarly, for proving that γ is *contradictive*, we try to disprove

$$\underline{\exists}(\Phi \wedge \gamma).$$

Instead of proving $\underline{\forall}(\gamma_1 \longrightarrow \gamma_2)$ in smart mode, we try to prove

$$\underline{\forall}\big((\Phi \wedge \gamma_1) \longrightarrow \gamma_2\big).$$

Independently, one can imagine to use a background theory for reducing the *output* with the *matrix* output mode. For this, one simplifies each guard wrt. the theory at the output stage treating contradictions and tautologies appropriately. Using the theory for replacing all cases by one at output stage in a smart mode manner leads once more to the problem of expressions or even guarded expressions "mysteriously" getting more complicated. Applying the theory only at the output stage makes it possible to implement a procedure `unassert(`φ`)` in a reasonable way.

Integration

CAS integrators make "mistakes" similar to those we have examined. Consider, e.g., the typical result

$$\int x^a \, dx = \frac{1}{a+1} x^{a+1}.$$

It does not cover the case $a = -1$, for which one wishes to obtain

$$\int x^{-1} \, dx = \ln x.$$

This problem can also be solved by using guarded expressions for integration results.

Within the framework of this paper, we would have to associate a guarding scheme to the integrator `int`. It is not hard to see that this cannot be done in a reasonable way without putting as much knowledge into the scheme as into the integrator itself. Thus for treating integration, one has to modify the integrator to provide guarded expressions.

Next, we have to clarify what the guarded expression for the above integral would look like. Since we know that the integral is defined for all interpretations of the variables, our assumption (20.79) implies that the generic condition be "T." We obtain the guarded expression

$$\left[\begin{array}{c|c} \mathrm{T} & \int \boldsymbol{x^a}\, \boldsymbol{dx} \\ a \neq -1 & \frac{1}{a+1} x^{a+1} \\ a = -1 & \ln x \end{array}\right].$$

Note that the redundant generic case does not model the system's current behavior.

Combining algebra with logic

Our method, in the described form, uses an already implemented algebraic evaluator. In the previous section, we have seen that this point of view is not sufficient for treating integration appropriately.

Also our approach runs into trouble with built-in knowledge such as

$$\sqrt{x^2} = |x|, \tag{20.89}$$

$$\operatorname{sign}(|x|) = 1. \tag{20.90}$$

Equation (20.89) introduces an absolute value operator within a non-generic term without making a case distinction. Equation (20.90) is wrong when not considering x transcendental. In contrast to the situation with reciprocals, our technique cannot be used to avoid this "mistake." We obtain

$$\operatorname{sign}(|x|) = \left[\begin{array}{c|c} \mathrm{T} & \mathbf{1} \\ x \neq 0 & 1 \\ x = 0 & 0 \end{array}\right]$$

yielding two different answers for $x = 0$.

We have already seen in the example Section 20.27.3 that the implementation of knowledge such as (20.89) and (20.90) is usually quite *ad hoc*, and can be mostly covered by using guarded expressions. This observation gives rise to the following question: When designing a new CAS based on guarded expressions, how should the knowledge be distributed between the algebraic side and the logic side?

20.27.5 Conclusions

Guarded expressions can be used to overcome well-known problems with interpreting expressions as terms. We have explained in detail how to compute with guarded expressions including several simplification techniques. Moreover we gain algebraic simplification power from the logical simplifications. Numerous examples illustrate the power of our simplification methods. The largest part of our ideas is efficiently implemented, and the software is published. The outlook on background theories and on the treatment of integration by guarded expressions points on interesting future extensions.

20.28 IDEALS: Arithmetic for Polynomial Ideals

This package implements the basic arithmetic for polynomial ideals by exploiting the Gröbner bases package of REDUCE. In order to save computing time all intermediate Gröbner bases are stored internally such that time consuming repetitions are inhibited.

Author: Herbert Melenk

20.28.1 Introduction

This package implements the basic arithmetic for polynomial ideals by exploiting the Gröbner bases package of REDUCE. In order to save computing time all intermediate Gröbner bases are stored internally such that time consuming repetitions are inhibited. A uniform setting facilitates the access.

20.28.2 Initialization

Prior to any computation the set of variables has to be declared by calling the operator `I_setting` . E.g. in order to initiate computations in the polynomial ring $Q[x, y, z]$ call

```
I_setting(x,y,z);
```

A subsequent call to `I_setting` allows one to select another set of variables; at the same time the internal data structures are cleared in order to free memory resources.

20.28.3 Bases

An ideal is represented by a basis (set of polynomials) tagged with the symbol `I`, e.g.,

```
u := I(x*z-y**2, x**3-y*z);
```

Alternatively a list of polynomials can be used as input basis; however, all arithmetic results will be presented in the above form. The operator `ideal2list` allows one to convert an ideal basis into a conventional REDUCE list.

Operators

Because of syntactical restrictions in REDUCE, special operators have to be used for ideal arithmetic:

`.*`	ideal product (infix)
`.:`	ideal quotient (infix)
`./`	ideal quotient (infix)
`.=`	ideal equality test (infix)
`subset`	ideal inclusion test (infix)
`intersection`	ideal intersection (prefix,binary)
`member`	test for membership in an ideal (infix: polynomial and ideal)
`gb`	Groebner basis of an ideal (prefix, unary)
`ideal2list`	convert ideal basis to polynomial list (prefix,unary)

Example:

```
I(x+y,x^2) .* I(x-z);

   2                        2    2
i(x  + x*y - x*z - y*z,x*y  - y *z)
```

The test operators return the values 1 (=true) or 0 (=false) such that they can be used in REDUCE `if-then-else` statements directly.

The results of `.+`, `.*`, `.:/./`, and `intersection` are ideals represented by their Gröbner basis in the current setting and term order. The term order can be modified using the operator `torder` from the Gröbner package. Note that ideal equality cannot be tested with the REDUCE equal sign:

```
I(x,y)  = I(y,x)        is false
I(x,y) .= I(y,x)        is true
```

20.28.4 Algorithms

The operators `groebner`, `preduce`, and `idealquotient` of the REDUCE Gröbner package support the basic algorithms:

$\texttt{gb}(Iu_1, u_2...) \rightarrow \texttt{groebner}(\{u_1, u_2...\}, \{x, ...\})$

$p \in I_1 \rightarrow p = 0 \; mod \; I_1$

$I_1 : I(p) \to (I_1 \bigcap I(p))/p\ elementwise$

On top of these the IDEALS package implements the following operations:

$I(u_1, u_2...) + I(v_1, v_2...) \to GB(I(u_1, u_2..., v_1, v_2...))$

$I(u_1, u_2...) * I(v_1, v_2...) \to GB(I(u_1 * v_1, u_1 * v2, ..., u_2 * v_1, u_2 * v_2...))$

$I_1 \bigcap I_2 \to Q[x, ...] \bigcap GB_{lex}(t * I_1 + (1 - t) * I_2, \{t, x, ..\})$

$I_1 : I(p_1, p_2, ...) \to I_1 : I(p_1) \bigcap I_1 : I(p_2) \bigcap ...$

$I_1 = I_2 \to GB(I_1) = GB(I_2)$

$I_1 \subseteq I_2 \to\ u_i \in I_2\ \forall\ u_i \in I_1 = I(u_1, u_2...)$

20.28.5 Examples

Please consult the file `ideals.tst`.

20.29 INVBASE: A Package for Computing Involutive Bases

Involutive bases are a new tool for solving problems in connection with multivariate polynomials, such as solving systems of polynomial equations and analyzing polynomial ideals. An involutive basis of polynomial ideal is nothing but a special form of a redundant Gröbner basis. The construction of involutive bases reduces the problem of solving polynomial systems to simple linear algebra.

Authors: A.Yu. Zharkov and Yu.A. Blinkov

20.29.1 Introduction

Involutive bases are a new tool for solving problems in connection with multivariate polynomials, such as solving systems of polynomial equations and analyzing polynomial ideals, see [ZB96]. An involutive basis of polynomial ideal is nothing but a special form of a redundant Gröbner basis. The construction of involutive bases reduces the problem of solving polynomial systems to simple linear algebra. The INVBASE package [26] calculates involutive bases of polynomial ideals using an algorithm described in [ZB96] which may be considered as an alternative to the well-known Buchberger algorithm [Buc85]. The package can be used over a variety of different coefficient domains, and for different variable and term orderings. The algorithm implemented in the INVBASE package is proved to be valid for any zero-dimensional ideal (finite number of solutions) as well as for positive-dimensional ideals in generic form. However, the algorithm does not terminate for "sparse" positive-dimensional systems. In order to stop the process we use the maximum degree bound for the Gröbner bases of generic ideals in the total-degree term ordering established in [Laz83]. In this case, it is reasonable to call the GROEBNER package with the answer of INVBASE as input information in order to compute the reduced Gröbner basis under the same variable and term ordering.
Though the INVBASE package supports computing involutive bases in any admissible term ordering, it is reasonable to compute them only for the total-degree term orderings. The package includes a special algorithm for conversion of total-degree involutive bases into the triangular bases in the lexicographical term ordering that is desirable for finding solutions. Normally the sum of timings for these two computations is much less than the timing for direct computation of the lexicographical involutive bases. As a rule, the result of the conversion algorithm is a reduced Gröbner basis in the lexicographical term ordering. However, because of some gaps in the current version of the algorithm, there may be rare situations when the resulting triangular set does not possess the formal property of Gröbner bases. Anyway, we recommend using the GROEBNER package with the result of the con-

[26] The REDUCE implementation has been supported by the Konrad-Zuse-Zentrum Berlin

version algorithm as input in order either to check the Gröbner bases property or to transform the result into a lexicographical Gröbner basis.

20.29.2 The Basic Operators

Term Ordering

The following term order modes are available:

```
revgradlex; gradlex; lex .
```

These modes have the same meaning as for the GROEBNER package.
All orderings are based on an ordering among the variables. For each pair of variables an order relation $>$ must be defined, e.g. $x > y$. The term ordering mode as well as the order of variables are set by the operator

`invtorder` ⟨*mode*⟩,{x_1,...,x_n}

where ⟨*mode*⟩ is one of the term order modes listed above. The notion of {x_1,...,x_n} as a list of variables at the same time means $x_1 > ... > x_n$.
Example 1.

```
invtorder revgradlex,{x,y,z}
```

sets the reverse graduated term ordering based on the variable order $x > y > z$.
The operator `invtorder` may be omitted. The default term order mode is `revgradlex` and the default decreasing variable order is alphabetical (or, more generally, the default REDUCE kernel order). Furthermore, the list of variables in the `invtorder` may be omitted. In this case the default variable order is used.

Computing Involutive Bases

To compute the involutive basis of ideal generated by the set of polynomials $\{p_1, ..., p_m\}$ one should type the command

`invbase` {p_1,...,p_n}

where p_i are polynomials in variables listed in the `invtorder` operator. If some kernels in p_i were not listed previously in the `invtorder` operator they are considered as parameters, i.e. they are considered part of the coefficients of polynomials. If `invtorder` was omitted, all the kernels in p_i are considered as variables with the default REDUCE kernel order.

The coefficients of polynomials p_i may be integers as well as rational numbers (or, accordingly, polynomials and rational functions in the parametric case). The computations modulo prime numbers are also available. For this purpose one should type the REDUCE commands

```
on modular; setmod p;
```

where p is a prime number.
The value of the `invbase` function is a list of integer polynomials $\{g_1, ..., g_n\}$ representing an involutive basis of a given ideal.
Example 2.

```
invtorder revgradlex,{x,y,z};
g:= invbase {4*x**2 + x*y**2 - z + 1/4,
             2*x + y**2*z + 1/2,
             x**2*z - 1/2*x - y**2 };
```

The resulting involutive basis in the reverse graduate ordering is

```
              3           2      3         2
g := {8*x*y*z  - 2*x*y*z  + 4*y  - 4*y*z  + 16*x*y

       + 17*y*z - 4*y,

          4         2         2                 2
      8*y  - 8*x*z  - 256*y  + 2*x*z + 64*z

       - 96*x + 20*z - 9,

          3
      2*y *z + 4*x*y + y,

           3         2       2       2
      8*x*z  - 2*x*z  + 4*y  - 4*z  + 16*x + 17*z

       - 4,

             3       3                  2
       - 4*y*z  - 8*y  + 6*x*y*z + y*z  - 36*x*y

       - 8*y,

           2        2       2
      4*x*y  + 32*y  - 8*z  + 12*x - 2*z + 1,
```

```
      2
2*y *z + 4*x + 1,

       3      2            2
 - 4*z  - 8*y  + 6*x*z + z  - 36*x - 8,

   2       2      2
8*x  - 16*y  + 4*z  - 6*x - z}
```

To convert it into a lexicographical Gröbner basis one should type

```
h:=invlex g;
```

The result is

```
                      6        5         4
h := {3976*x + 37104*z  - 600*z  + 2111*z

                 3           2
 + 122062*z  + 232833*z  - 680336*z + 288814

,

       2          6         5         4
1988*y  - 76752*z  + 1272*z  - 4197*z

            3           2
 - 251555*z  - 481837*z  + 1407741*z

 - 595666,

    7      6    5       4       3        2
16*z  - 8*z  + z  + 52*z  + 75*z  - 342*z

 + 266*z - 60}
```

In the case of “sparse” positive-dimensioned system when the involutive basis in the sense of [ZB96] does not exist, you get the error message

```
***** Maximum degree bound exceeded.
```

The resulting list of polynomials which is not an involutive basis is stored in the

share variable `invtempbasis`. In this case it is reasonable to call the GROEBNER package with the value of `invtempbasis` as input under the same variable and term ordering.

20.30 LALR: A Parser Generator

Author: Arthur Norman

This package provides a parser-generator, somewhat styled after `yacc` or the many programs available for use with other languages. You present it with a phrase structure grammar and it generates a set of tables that can then be used by the function `yyparse` to read in material in the syntax that you specified. Internally it uses a very well established technique known "LALR" which takes the grammar are derives the description of a stack automaton that can accept it. Details of the procedure can be found in standard books on compiler construction, such as the one by Aho, Ullman, Lam and Sethi [ALSU06].

At the time of writing this explanation the code is not in its final form, so this will describe the current state and include a few notes on what might chaneg in the future.

Building a parser is done in Reduce symbolic mode, so say "`symbolic;`" or "`lisp;`" before starting your work.

To use the code here you use a function `lalr_create_parser`, giving it two arguments. The first indicates precedence information and will be described later: for now just pass the value `nil`. The second argument is a list of productions, and the first one of these is taken to be the top-level target for the whole grammar.

Each production is in the form

```
(LHS    ((rhs1.1 rhs1.2 ...) a1.1 a1.2 ...)
        ((rhs2.1 rhs2.1 ...) a2.1 a2.2 ...)
        ...)
```

which in regular publication style for grammars might be interpreted as meaning

$$\begin{array}{rl} \text{LHS} \Rightarrow & \text{rhs}_{1,1}\ \ \text{rhs}_{1,2}\ldots\{\text{a}_{1,1}\ \ \text{a}_{1,2}\ldots\} \\ \quad | & \text{rhs}_{2,1}\ \ \text{rhs}_{2,2}\ldots\{\text{a}_{2,1}\ \ \text{a}_{2,2}\ldots\} \\ \ldots & \\ \quad ; & \end{array}$$

The various lines specify different options for what the left hand side (non-terminal symbol) might correspond to, while the items within the braces are sematic actions that get obeyed or evaluated when the production ruls is used.

Each LHS is treated as a non-terminal symbol and is specified as a simple name. Note that by default the Reduce parser will be folding characters within names to lower case and so it will be best to choose names for non-terminals that are unambiguous even when case-folded, but I would like to establish a convention that in source code they are written in capitals.

The RHS items may be either non-terminals (identified because they are present in the left hand side of some production) or terminals. Terminal symbols can be specified in two different ways.

The lexer has built-in recipes that decode certain sequences of characters and return the special markers for !:symbol, !:number, !:string, !:list for commonly used cases. In these cases the variable yylval gets left set to associated data, so for instance in the case of !:symbol it gets set to the particular symbol concerned. The token type :list is used for Lisp or rlisp-like notation where the input contains 'expression or ‘expression so for instance the input ‘(a b c) leads to the lexer returning !:list and yylvel being set to (backquote (a b c)). This treatment is specialised for handling rlisp-like syntax.

Other terminals are indicated by writing a string. That may either consist of characters that would otherwise form a symbol (ie a letter followed by letters, digits and underscores) or a sequence of non-alphanumeric characters. In the latter case if a sequence of three or more punctuation marks make up a terminal then all the shorter prefixes of it will also be grouped to form single entities. So if "<–>" is a terminal then '<', '<-' and '<–' will each by parsed as single tokens, and any of them that are not used as terminals will be classified as !:symbol.

As well as terminals and non-terminals (which are writtent as symbols or strings) it is possible to write one of

(`OPT` *s1 s2 ...*)	0 or 1 instances of the sequence *s1*, ...
(`STAR` *s1 s2 ...*)	0, 1, 2, ... instances.
(`PLUS` *s1 s2 ...*)	1, 2, 3, ... instances.
(`LIST` *sep s1 s2 ...*)	like (`STAR` *s1 s2 ...*) but with the single item *sep* between each instance.
(`LISTPLUS` *sep s1 ...*)	like (`PLUS` *s2 ...*) but with *sep* interleaved.
(`OR` *s1 s2 ...*)	one or other of the tokens shown.

When the lexer processes input it will return a numeric code that identifies the type of the item seen, so in a production one might write (!:symbol ":=" EXPRESSION) and as it recognises the first two tokens the lexer will return a numeric code for !:symbol (and set yylval to the actual symbol as seen) and then a numeric code that it allocates for ":=". In the latter case it will also set yylval to the symbol !:!= in case that is useful. Precedence can be set using `lalr_precedence`. See examples below.

20.30.1 Limitations

1. Grammar rules and semantic actions are specified in fairly raw Lisp.
2. The lexer is hand-written and can not readily be reconfigured for use with languages other than rlisp. For instance it has use of "!" as a character escape built into it.

20.30.2 An example

```
% Here I set up a sample grammar
%    S' -> S
%    S  -> C C        { }
%    C  -> "c" C      { }
%        | "d"        { }
% This is example 4.42 from Aho, Sethi and Ullman's
% Red Dragon book.
% It is example 4.54 in the more recent Purple book.
%
%
grammar := '(
 (s  ((cc cc)  )   % Use default semantic action here
 )
 (cc (("c" cc) (list 'c !$2))  % First production for C
     (("d")    'd           )  % Second production for C
 ))$

parsertables := lalr_create_parser(nil, grammar)$

<< lex_init();
   yyparse() >>;
c c c d c d ;
```

20.31 LAPLACE: Laplace Transforms

This package can calculate ordinary and inverse Laplace transforms of expressions. Documentation is in plain text.

Authors: C. Kazasov, M. Spiridonova, V. Tomov

Reference: [Kaz87].

Some hints on how to use to use this package:

Syntax:

`laplace`(⟨*exp*⟩,⟨*var-s*⟩,⟨*var-t*⟩)

`invlap`(⟨*exp*⟩,⟨*var-s*⟩,⟨*var-t*⟩)

where ⟨*exp*⟩ is the expression to be transformed, ⟨*var-s*⟩ is the source variable (in most cases ⟨*exp*⟩ depends explicitly of this variable) and ⟨*var-t*⟩ is the target variable. If ⟨*var-t*⟩ is omitted, the above operators use an internal variable `lp!&` or `il!&`, respectively.

The following switches can be used to control the transformations:

`lmon`: If on, sin, cos, sinh and cosh are converted by `laplace` into exponentials,

`lhyp`: If on, expressions $e^{\tilde{}x}$ are converted by `invlap` into hyperbolic functions sinh and cosh,

`ltrig`: If on, expressions $e^{\tilde{}x}$ are converted by `invlap` into trigonometric functions sin and cos.

The system can be extended by adding Laplace transformation rules for single functions by rules or rule sets. In such a rule the source variable *must* be free, the target variable *must* be `il!&` for `laplace` and `lp!&` for `invlap` and the third parameter should be omitted. Also rules for transforming derivatives are entered in such a form.

Examples:

```
let {laplace(log(~x),x)
        => -log(Euler_Gamma * il!&)/il!&,

     invlap(log(Euler_Gamma * ~x)/x,x)
        => -log(lp!&)};


operator f;

let{

laplace(df(f(~x),x),x)
       => il!&*laplace(f(x),x) - sub(x=0,f(x)),

laplace(df(f(~x),x,~n),x)
       => il!&**n*laplace(f(x),x) -

for i:=n-1 step -1 until 0 sum

sub(x=0, df(f(x),x,n-1-i)) * il!&**i

when fixp n,

laplace(f(~x),x) = f(il!&)

};
```

Remarks about some functions:

The `delta` and `gamma` functions are known.
ONE is the name of the unit step function.
INTL is a parametrized integral function

`intl`(⟨*expr*⟩,⟨*var*⟩,0,⟨*obj.var*⟩)

which means "Integral of ⟨*expr*⟩ w.r.t. ⟨*var*⟩ taken from 0 to ⟨*obj.var*⟩", e.g. `intl(2*`$y^2, y, 0, x$`)` which is formally a function in x.
We recommend reading the file `laplace.tst` for a further introduction.

20.32 LIE: Functions for the Classification of Real *n*-Dimensional Lie Algebras

LIE is a package of functions for the classification of real n-dimensional Lie algebras. It consists of two modules: **liendmc1** and **lie1234**. With the help of the functions in the **liendmcl** module, real n-dimensional Lie algebras L with a derived algebra $L^{(1)}$ of dimension 1 can be classified.

Authors: Carsten and Franziska Schöbel

20.32.1 liendmc1

With the help of the functions in this module real n-dimensional Lie algebras L with a derived algebra $L^{(1)}$ of dimension 1 can be classified. L has to be defined by its structure constants c_{ij}^k in the basis $\{X_1, \ldots, X_n\}$ with $[X_i, X_j] = c_{ij}^k X_k$. The user must define an `array lienstrucin(n,n,n)` with n being the dimension of the Lie algebra L. The structure constants lienstrucin(i, j, k):=c_{ij}^k for $i < j$ should be given. Then the procedure `liendimcom1` can be called. Its syntax is:

`liendimcom1(`⟨*number*⟩`)` .

⟨*number*⟩ corresponds to the dimension n. The procedure simplifies the structure of L performing real linear transformations. The returned value is a list of the form

(i) {LIE_ALGEBRA(2),COMMUTATIVE(n-2)} or

(ii) {HEISENBERG(k),COMMUTATIVE(n-k)}

with $3 \leq k \leq n$, k odd.

The concepts correspond to the following theorem (`LIE_ALGEBRA(2)` $\rightarrow L_2$, `HEISENBERG(k)` $\rightarrow H_k$ and `COMMUTATIVE(n-k)` $\rightarrow C_{n-k}$):

Theorem. Every real n-dimensional Lie algebra L with a 1-dimensional derived algebra can be decomposed into one of the following forms:

(1) $C(L) \cap L^{(1)} = \{0\} \ : \ L_2 \oplus C_{n-2}$ or

(2) $C(L) \cap L^{(1)} = L^{(1)} \ : \ H_k \oplus C_{n-k} \quad (k = 2r-1,\, r \geq 2)$,

with

1. $C(L) = C_j \oplus (L^{(1)} \cap C(L))$ and $\dim C_j = j$,

2. L_2 is generated by Y_1, Y_2 with $[Y_1, Y_2] = Y_1$,
3. H_k is generated by $\{Y_1, \ldots, Y_k\}$ with
   $[Y_2, Y_3] = \cdots = [Y_{k-1}, Y_k] = Y_1$.

(cf. [Sch93])

The returned list is also stored as `lie_list`. The matrix `lientrans` gives the transformation from the given basis $\{X_1, \ldots, X_n\}$ into the standard basis $\{Y_1, \ldots, Y_n\}$: $Y_j = (\text{LIENTRANS})^k_j X_k$.

A more detailed output can be obtained by turning on the switch `tr_lie`: before the procedure `liendimcom1` is called.

The returned list could be an input for a data bank in which mathematical relevant properties of the obtained Lie algebras are stored.

20.32.2 lie1234

This part of the package classifies real low-dimensional Lie algebras L of the dimension $n :=\dim L = 1, 2, 3, 4$. L is also given by its structure constants c^k_{ij} in the basis $\{X_1, \ldots, X_n\}$ with $[X_i, X_j] = c^k_{ij} X_k$. An ARRAY LIESTRIN$(n, n, n)$ has to be defined and LIESTRIN(i, j, k):=c^k_{ij} for $i < j$ should be given. Then the procedure `lieclass` can be called whose syntax is:

`lieclass(`⟨*number*⟩`)` .

⟨*number*⟩ should be the dimension of the Lie algebra L. The procedure stepwise simplifies the commutator relations of L using properties of invariance like the dimension of the centre, of the derived algebra, unimodularity etc. The returned value has the form:

```
{LIEALG(n),COMTAB(m)},
```

where m corresponds to the number of the standard form (basis: $\{Y_1, \ldots, Y_n\}$) in an enumeration scheme. The corresponding enumeration schemes are listed below (cf. [Sch92],[Mac99]). In case that the standard form in the enumeration scheme depends on one (or two) parameter(s) p_1 (and p_2) the list is expanded to:

```
{LIEALG(n),COMTAB(m),p1,p2}.
```

This returned value is also stored as `lie_class`. The linear transformation from the basis $\{X_1, \ldots, X_n\}$ into the basis of the standard form $\{Y_1, \ldots, Y_n\}$ is given by the matrix `liemat`: $Y_j = (\text{LIEMAT})^k_j X_k$.

By turning on the switch `tr_lie` before the procedure `lieclass` is called the output contains not only the list `lie_class` but also the non-vanishing commutator relations in the standard form.

By the value m and the parameters further examinations of the Lie algebra are possible, especially if in a data bank mathematical relevant properties of the enumerated standard forms are stored.

20.32.3 Enumeration schemes for lie1234

returned list `lie_class`	the corresponding commutator relations
LIEALG(1),COMTAB(0)	commutative case
LIEALG(2),COMTAB(0)	commutative case
LIEALG(2),COMTAB(1)	$[Y_1, Y_2] = Y_2$
LIEALG(3),COMTAB(0)	commutative case
LIEALG(3),COMTAB(1)	$[Y_1, Y_2] = Y_3$
LIEALG(3),COMTAB(2)	$[Y_1, Y_3] = Y_3$
LIEALG(3),COMTAB(3)	$[Y_1, Y_3] = Y_1, [Y_2, Y_3] = Y_2$
LIEALG(3),COMTAB(4)	$[Y_1, Y_3] = Y_2, [Y_2, Y_3] = Y_1$
LIEALG(3),COMTAB(5)	$[Y_1, Y_3] = -Y_2, [Y_2, Y_3] = Y_1$
LIEALG(3),COMTAB(6)	$[Y_1, Y_3] = -Y_1 + p_1 Y_2, [Y_2, Y_3] = Y_1, p_1 \neq 0$
LIEALG(3),COMTAB(7)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = -Y_2, [Y_2, Y_3] = Y_1$
LIEALG(3),COMTAB(8)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = Y_2, [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(0)	commutative case
LIEALG(4),COMTAB(1)	$[Y_1, Y_4] = Y_1$
LIEALG(4),COMTAB(2)	$[Y_2, Y_4] = Y_1$
LIEALG(4),COMTAB(3)	$[Y_1, Y_3] = Y_1, [Y_2, Y_4] = Y_2$
LIEALG(4),COMTAB(4)	$[Y_1, Y_3] = -Y_2, [Y_2, Y_4] = Y_2,$ $[Y_1, Y_4] = [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(5)	$[Y_2, Y_4] = Y_2, [Y_1, Y_4] = [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(6)	$[Y_2, Y_4] = Y_1, [Y_3, Y_4] = Y_2$
LIEALG(4),COMTAB(7)	$[Y_2, Y_4] = Y_2, [Y_3, Y_4] = Y_1$
LIEALG(4),COMTAB(8)	$[Y_1, Y_4] = -Y_2, [Y_2, Y_4] = Y_1$
LIEALG(4),COMTAB(9)	$[Y_1, Y_4] = -Y_1 + p_1 Y_2, [Y_2, Y_4] = Y_1, p_1 \neq 0$
LIEALG(4),COMTAB(10)	$[Y_1, Y_4] = Y_1, [Y_2, Y_4] = Y_2$

returned list `lie_class`	the corresponding commutator relations
LIEALG(4),COMTAB(11)	$[Y_1, Y_4] = Y_2, [Y_2, Y_4] = Y_1$
LIEALG(4),COMTAB(12)	$[Y_1, Y_4] = Y_1 + Y_2, [Y_2, Y_4] = Y_2 + Y_3,$ $[Y_3, Y_4] = Y_3$
LIEALG(4),COMTAB(13)	$[Y_1, Y_4] = Y_1, [Y_2, Y_4] = p_1 Y_2, [Y_3, Y_4] = p_2 Y_3,$ $p_1, p_2 \neq 0$
LIEALG(4),COMTAB(14)	$[Y_1, Y_4] = p_1 Y_1 + Y_2, [Y_2, Y_4] = -Y_1 + p_1 Y_2,$ $[Y_3, Y_4] = p_2 Y_3, p_2 \neq 0$
LIEALG(4),COMTAB(15)	$[Y_1, Y_4] = p_1 Y_1 + Y_2, [Y_2, Y_4] = p_1 Y_2,$ $[Y_3, Y_4] = Y_3, p_1 \neq 0$
LIEALG(4),COMTAB(16)	$[Y_1, Y_4] = 2Y_1, [Y_2, Y_3] = Y_1,$ $[Y_2, Y_4] = (1 + p_1) Y_2, [Y_3, Y_4] = (1 - p_1) Y_3,$ $p_1 \geq 0$
LIEALG(4),COMTAB(17)	$[Y_1, Y_4] = 2Y_1, [Y_2, Y_3] = Y_1,$ $[Y_2, Y_4] = Y_2 - p_1 Y_3, [Y_3, Y_4] = p_1 Y_2 + Y_3,$ $p_1 \neq 0$
LIEALG(4),COMTAB(18)	$[Y_1, Y_4] = 2Y_1, [Y_2, Y_3] = Y_1,$ $[Y_2, Y_4] = Y_2 + Y_3, [Y_3, Y_4] = Y_3$
LIEALG(4),COMTAB(19)	$[Y_2, Y_3] = Y_1, [Y_2, Y_4] = Y_3, [Y_3, Y_4] = Y_2$
LIEALG(4),COMTAB(20)	$[Y_2, Y_3] = Y_1, [Y_2, Y_4] = -Y_3, [Y_3, Y_4] = Y_2$
LIEALG(4),COMTAB(21)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = -Y_2, [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(22)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = Y_2, [Y_2, Y_3] = Y_1$

20.33 LINALG: Linear Algebra Package

This package provides a selection of functions that are useful in the world of linear algebra.

Author: Matt Rebbeck

20.33.1 Introduction

This package provides a selection of functions that are useful in the world of linear algebra. These functions are described alphabetically in subsection 20.33.3 and are labelled 20.33.3.1 to 20.33.3.53. They can be classified into four sections(n.b: the numbers after the dots signify the function label in section 20.33.3).

Contributions to this package have been made by Walter Tietze (ZIB).

20.33.1.1 Basic matrix handling

20.33.1.2 Constructors

Functions that create matrices.

band_matrix	...	20.33.3.6	block_matrix	...	20.33.3.7
char_matrix	...	20.33.3.8	coeff_matrix	...	20.33.3.11
companion	...	20.33.3.13	hessian	...	20.33.3.22
hilbert	...	20.33.3.23	mat_jacobian	...	20.33.3.24
jordan_block	...	20.33.3.25	make_identity	...	20.33.3.27
random_matrix	...	20.33.3.36	toeplitz	...	20.33.3.50
Vandermonde	...	20.33.3.52	Kronecker_Product	...	20.33.3.53

20.33.1.3 High level algorithms

char_poly	...	20.33.3.9	cholesky	...	20.33.3.10
gram_schmidt	...	20.33.3.20	lu_decom	...	20.33.3.26
pseudo_inverse	...	20.33.3.35	simplex	...	20.33.3.41
svd	...	20.33.3.45	triang_adjoint	...	20.33.3.51

There is a separate NORMFORM package described in section 20.40 for computing the following matrix normal forms in REDUCE:

smithex, smithex_int, frobenius, ratjordan, jordansymbolic, jordan.

20.33.1.4 Predicates

matrixp	...	20.33.3.29	squarep	...	20.33.3.42
symmetricp	...	20.33.3.49			

Note on examples:

In the examples the matrix $\mathcal{A}$ will be

$$\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Notation

Throughout $\mathcal{I}$ is used to indicate the identity matrix and $\mathcal{A}^T$ to indicate the transpose of the matrix $\mathcal{A}$.

20.33.2 Getting started

If you have not used matrices within REDUCE before then the following may be helpful.

Creating matrices

Initialisation of matrices takes the following syntax:

```
mat1 := mat((a,b,c),(d,e,f),(g,h,i));
```

will produce

$$mat1 := \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Getting at the entries

The (i, j)th entry can be accessed by:

```
mat1(i,j);
```

Loading the linear_algebra package

The package is loaded by:

```
load_package linalg;
```

20.33.3 What's available

20.33.3.1 add_columns, add_rows

Syntax:

`add_columns(`$\mathcal{A}$`,c1,c2,expr);`

$\mathcal{A}$	:-	a matrix.
$c1, c2$	:-	positive integers.
expr	:-	a scalar expression.

Synopsis:

`add_columns` replaces column $c2$ of $\mathcal{A}$ by
`expr * column(`$\mathcal{A}$`,c1) + column(`$\mathcal{A}$`,c2)`.
`add_rows` performs the equivalent task on the rows of $\mathcal{A}$.

Examples:

$$\texttt{add\_columns}(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & x+2 & 3 \\ 4 & 4*x+5 & 6 \\ 7 & 7*x+8 & 9 \end{pmatrix}$$

$$\texttt{add\_rows}(\mathcal{A},2,3,5)=\begin{pmatrix}1&2&3\\4&5&6\\27&33&39\end{pmatrix}$$

Related functions:
`add_to_columns`, `add_to_rows`, `mult_columns`, `mult_rows`.

20.33.3.2 add_rows

See: `add_columns`.

20.33.3.3 add_to_columns, add_to_rows

Syntax:
`add_to_columns(`$\mathcal{A}$`,column_list,expr);`

$\mathcal{A}$	:-	a matrix.
column_list	:-	a positive integer or a list of positive integers.
expr	:-	a scalar expression.

Synopsis:
`add_to_columns` adds expr to each column specified in column_list of $\mathcal{A}$.

`add_to_rows` performs the equivalent task on the rows of $\mathcal{A}$.

Examples:

$$\texttt{add\_to\_columns}(\mathcal{A},\{1,2\},10)=\begin{pmatrix}11&12&3\\14&15&6\\17&18&9\end{pmatrix}$$

$$\texttt{add\_to\_rows}(\mathcal{A},2,-x)=\begin{pmatrix}1&2&3\\-x+4&-x+5&-x+6\\7&8&9\end{pmatrix}$$

Related functions:
`add_columns`, `add_rows`, `mult_rows`, `mult_columns`.

20.33.3.4 add_to_rows

See: `add_to_columns`.

20.33.3.5 augment_columns, stack_rows

Syntax:
`augment_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$	:-	a matrix.
column_list	:-	either a positive integer or a list of positive integers.

Synopsis:

`augment_columns` gets hold of the columns of $\mathcal{A}$ specified in column_list and sticks them together.

`stack_rows` performs the same task on rows of $\mathcal{A}$.

Examples:

`augment_columns`$(\mathcal{A}, \{1,2\}) = \begin{pmatrix} cc1 & 2 \\ 4 & 5 \\ 7 & 8 \end{pmatrix}$

`stack_rows`$(\mathcal{A}, \{1,3\}) = \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{pmatrix}$

Related functions:

`get_columns`, `get_rows`, `sub_matrix`.

20.33.3.6 band_matrix

Syntax:

```
band_matrix(expr_list,square_size);
```

expr_list	:-	either a single scalar expression or a list of an odd number of scalar expressions.
square_size	:-	a positive integer.

Synopsis:

`band_matrix` creates a square matrix of dimension `square_size`. The diagonal consists of the middle expr of the `expr_list`. The expressions to the left of this fill the required number of sub-diagonals and the expressions to the right the super-diagonals.

Examples: `band_matrix`$(\{x,y,z\},6) = \begin{pmatrix} y & z & 0 & 0 & 0 & 0 \\ x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \\ 0 & 0 & 0 & 0 & x & y \end{pmatrix}$

Related functions:

`diagonal`.

20.33.3.7 block_matrix

Syntax:

`block_matrix(r,c,matrix_list);`

r, c	:-	positive integers.
`matrix_list`	:-	a list of matrices.

Synopsis:

`block_matrix` creates a matrix that consists of $r \times c$ matrices filled from the `matrix_list` row-wise.

Examples:

$$\mathcal{B} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \mathcal{C} = \begin{pmatrix} 5 \\ 5 \end{pmatrix},\ \mathcal{D} = \begin{pmatrix} 22 & 33 \\ 44 & 55 \end{pmatrix}$$

$$\texttt{block\_matrix}(2, 3, \{\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{C}, \mathcal{B}\}) = \begin{pmatrix} 1 & 0 & 5 & 22 & 33 \\ 0 & 1 & 5 & 44 & 55 \\ 22 & 33 & 5 & 1 & 0 \\ 44 & 55 & 5 & 0 & 1 \end{pmatrix}$$

20.33.3.8 char_matrix

Syntax:

`char_matrix(`$\mathcal{A}, \lambda$`);`

$\mathcal{A}$	:-	a square matrix.
λ	:-	a symbol or algebraic expression.

Synopsis:

`char_matrix` creates the characteristic matrix $\mathcal{C}$ of $\mathcal{A}$. This is $\mathcal{C} = \lambda\mathcal{I} - \mathcal{A}$.

Examples: $\texttt{char\_matrix}(\mathcal{A}, x) = \begin{pmatrix} x-1 & -2 & -3 \\ -4 & x-5 & -6 \\ -7 & -8 & x-9 \end{pmatrix}$

Related functions:

`char_poly`.

20.33.3.9 char_poly

Syntax:

`char_poly(`$\mathcal{A}, \lambda$`);`

$\mathcal{A}$	:-	a square matrix.
λ	:-	a symbol or algebraic expression.

Synopsis:

`char_poly` finds the characteristic polynomial of $\mathcal{A}$.

This is the determinant of $\lambda\mathcal{I} - \mathcal{A}$.

Examples:

`char_poly(`$\mathcal{A}, x$`)` $= x^3 - 15 * x^2 - 18 * x$

Related functions:

`char_matrix`.

20.33.3.10 cholesky

Syntax:

`cholesky(`$\mathcal{A}$`);`

$\mathcal{A}$:- a positive definite matrix containing numeric entries.

Synopsis:

`cholesky` computes the cholesky decomposition of $\mathcal{A}$.

It returns $\{\mathcal{L},\mathcal{U}\}$ where $\mathcal{L}$ is a lower matrix, $\mathcal{U}$ is an upper matrix, $\mathcal{A} = \mathcal{L}\mathcal{U}$, and $\mathcal{U} = \mathcal{L}^T$.

Examples:

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$\texttt{cholesky}(\mathcal{F}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 1 & \sqrt{2} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 \\ 0 & \sqrt{2} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \right\}$$

Related functions:

`lu_decom`.

20.33.3.11 coeff_matrix

Syntax:

`coeff_matrix({lin_eqn`$_1$`,lin_eqn`$_2$`, ...,lin_eqn`$_n$`});` [27]

lin_eqn$_1$,lin_eqn$_2$,...,lin_eqn$_n$:- linear equations. Can be of the form *equation* = *number* or just *equation* which is equivalent to *equation* = *0*.

[27] If you're feeling lazy then the {}'s can be omitted.

Synopsis:

`coeff_matrix` creates the coefficient matrix $\mathcal{C}$ of the linear equations. It returns $\{\mathcal{C}, \mathcal{X}, \mathcal{B}\}$ such that $\mathcal{C}\mathcal{X} = \mathcal{B}$.

Examples:

`coeff_matrix`$(\{x + y + 4 * z = 10, y + x - z = 20, x + y + 4\}) =$

$$\left\{ \begin{pmatrix} 4 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} z \\ y \\ x \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \\ -4 \end{pmatrix} \right\}$$

20.33.3.12 column_dim, row_dim

Syntax:

`column_dim(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix.

Synopsis:

`column_dim` finds the column dimension of $\mathcal{A}$.
`row_dim` finds the row dimension of $\mathcal{A}$.

Examples:

`column_dim`($\mathcal{A}$) = 3

20.33.3.13 companion

Syntax:

`companion(poly,x);`

`poly` :- a monic univariate polynomial in x.
x :- the variable.

Synopsis:

`companion` creates the companion matrix $\mathcal{C}$ of `poly`.

This is the square matrix of dimension n, where n is the degree of `poly` w.r.t. x. The entries of $\mathcal{C}$ are: $\mathcal{C}(i, n) = -$`coeffn`$(\texttt{poly}, x, i - 1)$ for $i = 1, \ldots, n$, $\mathcal{C}(i, i - 1) = 1$ for $i = 2, \ldots, n$ and the rest are 0.

Examples: `companion`$(x^4 + 17 * x^3 - 9 * x^2 + 11, x) = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$

Related functions:

`find_companion`.

20.33.3.14 copy_into

Syntax:

`copy_into(`$\mathcal{A}, \mathcal{B}$`,r,c);`

$\mathcal{A}, \mathcal{B}$	:-	matrices.
r, c	:-	positive integers.

Synopsis:

`copy_into` copies matrix $\mathcal{A}$ into $\mathcal{B}$ with $\mathcal{A}(1,1)$ at $\mathcal{B}(r,c)$.

Examples:

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\texttt{copy\_into}(\mathcal{A}, \mathcal{G}, 1, 2) = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 4 & 5 & 6 \\ 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Related functions:

`augment_columns`, `extend`, `matrix_augment`, `matrix_stack`, `stack_rows`, `sub_matrix`.

20.33.3.15 diagonal

Syntax:

`diagonal({mat`$_1$`,mat`$_2$`, ...,mat`$_n$`});`[28]

mat$_1$,mat$_2$, ...,mat$_n$	:-	each can be either a scalar expr or a square matrix.

Synopsis:

`diagonal` creates a matrix that contains the input on the diagonal.

Examples:

$$\mathcal{H} = \begin{pmatrix} 66 & 77 \\ 88 & 99 \end{pmatrix}$$

$$\texttt{diagonal}(\{\mathcal{A}, x, \mathcal{H}\}) = \begin{pmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 66 & 77 \\ 0 & 0 & 0 & 0 & 88 & 99 \end{pmatrix}$$

[28] If you're feeling lazy then the {}'s can be omitted.

Related functions:

`jordan_block`.

20.33.3.16 extend

Syntax:

`extend(`$\mathcal{A}$`,r,c,expr);`

$\mathcal{A}$	:-	a matrix.
r, c	:-	positive integers.
`expr`	:-	algebraic expression or symbol.

Synopsis:

`extend` returns a copy of $\mathcal{A}$ that has been extended by r rows and c columns. The new entries are made equal to `expr`.

Examples: `extend`$(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & 2 & 3 & x & x \\ 4 & 5 & 6 & x & x \\ 7 & 8 & 9 & x & x \\ x & x & x & x & x \end{pmatrix}$

Related functions:

`copy_into`, `matrix_augment`, `matrix_stack`, `remove_columns`, `remove_rows`.

20.33.3.17 find_companion

Syntax:

`find_companion(`$\mathcal{A}$`,x);`

$\mathcal{A}$	:-	a matrix.
x	:-	the variable.

Synopsis:

Given a companion matrix, `find_companion` finds the polynomial from which it was made.

Examples:

$$\mathcal{C} = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

`find_companion`$(\mathcal{C}, x) = x^4 + 17 * x^3 - 9 * x^2 + 11$

Related functions:

`companion`.

20.33.3.18 get_columns, get_rows

Syntax:

`get_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$:- a matrix.
c :- either a positive integer or a list of positive integers.

Synopsis:

`get_columns` removes the columns of $\mathcal{A}$ specified in `column_list` and returns them as a list of column matrices.

`get_rows` performs the same task on the rows of $\mathcal{A}$.

Examples:

$$\texttt{get\_columns}(\mathcal{A},\{1,3\}) = \left\{\begin{pmatrix}1\\4\\7\end{pmatrix},\begin{pmatrix}3\\6\\9\end{pmatrix}\right\}$$

$$\texttt{get\_rows}(\mathcal{A},2) = \left\{\begin{pmatrix}4 & 5 & 6\end{pmatrix}\right\}$$

Related functions:

`augment_columns`, `stack_rows`, `sub_matrix`.

20.33.3.19 get_rows

See: `get_columns`.

20.33.3.20 gram_schmidt

Syntax:

`gram_schmidt({vec`$_1$`,vec`$_2$`, ...,vec`$_n$`});` [29]

vec$_1$,vec$_2$, ...,vec$_n$:- linearly-independent vectors. Each vector must be written as a list, eg:{1,0,0}.

Synopsis:

`gram_schmidt` performs the Gram-Schmidt orthonormalisation on the input vectors. It returns a list of orthogonal normalised vectors.

Examples:

`gram_schmidt({{1,0,0},{1,1,0},{1,1,1}})` =
{{1,0,0},{0,1,0},{0,0,1}}

`gram_schmidt({{1,2},{3,4}})` $= \{\{\frac{1}{\sqrt{5}},\frac{2}{\sqrt{5}}\},\{\frac{2*\sqrt{5}}{5},\frac{-\sqrt{5}}{5}\}\}$

[29] If you're feeling lazy then the {}'s can be omitted.

20.33.3.21 hermitian_tp

Syntax:

`hermitian_tp(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix.

Synopsis:

`hermitian_tp` computes the hermitian transpose of $\mathcal{A}$.

This is a matrix in which the (i,j)th entry is the conjugate of the (j,i)th entry of $\mathcal{A}$.

Examples:

$$\mathcal{J} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 4 & 5 & 2 \\ 1 & i & 0 \end{pmatrix}$$

$$\texttt{hermitian\_tp}(\mathcal{J}) = \begin{pmatrix} -i+1 & 4 & 1 \\ -i+2 & 5 & -i \\ -i+3 & 2 & 0 \end{pmatrix}$$

Related functions:

`tp`[30].

20.33.3.22 hessian

Syntax:

`hessian(expr,variable_list);`

expr :- a scalar expression.
variable_list :- either a single variable or a list of variables.

Synopsis:

`hessian` computes the hessian matrix of expr w.r.t. the varibles in `variable_list`.

This is an $n \times n$ matrix where n is the number of variables and the (i,j)th entry is `df(expr,variable_list(i),variable_list(j))`.

Examples: $\texttt{hessian}(x*y*z+x^2,\{w,x,y,z\}) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & z & y \\ 0 & z & 0 & x \\ 0 & y & x & 0 \end{pmatrix}$

Related functions:

`df`[31].

[30] standard reduce call for the transpose of a matrix - see section 14.4.
[31] standard reduce call for differentiation - see section 7.7.

20.33.3.23 hilbert

Syntax:

`hilbert(square_size,expr);`

square_size	:-	a positive integer.
expr	:-	an algebraic expression.

Synopsis:

`hilbert` computes the square hilbert matrix of dimension `square_size`.

This is the symmetric matrix in which the (i,j)th entry is $1/(i+j-\texttt{expr})$.

Examples: `hilbert`$(3, y+x) =$
$$\begin{pmatrix} \frac{-1}{x+y-2} & \frac{-1}{x+y-3} & \frac{-1}{x+y-4} \\ \frac{-1}{x+y-3} & \frac{-1}{x+y-4} & \frac{-1}{x+y-5} \\ \frac{-1}{x+y-4} & \frac{-1}{x+y-5} & \frac{-1}{x+y-6} \end{pmatrix}$$

20.33.3.24 jacobian

Syntax:

`mat_jacobian(expr_list,variable_list);`

expr_list	:-	either a single algebraic expression or a list of algebraic expressions.
variable_list	:-	either a single variable or a list of variables.

Synopsis:

`mat_jacobian` computes the jacobian matrix of `expr_list` w.r.t. `variable_list`. This is a matrix whose entry at position (i,j) is `df(expr_list(i),variable_list(j))`. The matrix is $n \times m$ where n is the number of variables and m the number of expressions.

Examples:

`mat_jacobian`$(\{x^4, x*y^2, x*y*z^3\}, \{w,x,y,z\}) =$
$$\begin{pmatrix} 0 & 4*x^3 & 0 & 0 \\ 0 & y^2 & 2*x*y & 0 \\ 0 & y*z^3 & x*z^3 & 3*x*y*z^2 \end{pmatrix}$$

Related functions:

`hessian`, `df`[32].

NOTE: The function `mat_jacobian` used to be called just "`jacobian`" however us of that name was in conflict with another Reduce package.

[32] standard reduce call for differentiation - see section 7.7.

20.33.3.25 jordan_block

Syntax:

`jordan_block(expr,square_size);`

expr	:-	an algebraic expression or symbol.
square_size	:-	a positive integer.

Synopsis:

`jordan_block` computes the square jordan block matrix $\mathcal{J}$ of dimension `square_size`.

The entries of $\mathcal{J}$ are: $\mathcal{J}(i,i) =$ `expr` for $i = 1,\ldots,n$, $\mathcal{J}(i,i+1) = 1$ for $i = 1,\ldots,n-1$, and all other entries are 0.

Examples: `jordan_block(x,5)` $=$

$$\begin{pmatrix} x & 1 & 0 & 0 & 0 \\ 0 & x & 1 & 0 & 0 \\ 0 & 0 & x & 1 & 0 \\ 0 & 0 & 0 & x & 1 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Related functions:

`diagonal`, `companion`.

20.33.3.26 lu_decom

Syntax:

`lu_decom(`$\mathcal{A}$`);`

$\mathcal{A}$	:-	a matrix containing either numeric entries or imaginary entries with numeric coefficients.

Synopsis:

`lu_decom` performs LU decomposition on $\mathcal{A}$, ie: it returns $\{\mathcal{L},\mathcal{U}\}$ where $\mathcal{L}$ is a lower diagonal matrix, $\mathcal{U}$ an upper diagonal matrix and $\mathcal{A} = \mathcal{L}\mathcal{U}$.

Caution: The algorithm used can swap the rows of $\mathcal{A}$ during the calculation. This means that $\mathcal{L}\mathcal{U}$ does not equal $\mathcal{A}$ but a row equivalent of it. Due to this, `lu_decom` returns $\{\mathcal{L},\mathcal{U}$,vec$\}$. The call `convert(`$\mathcal{A}$`,vec)` will return the matrix that has been decomposed, ie: $\mathcal{L}\mathcal{U} =$ `convert(`$\mathcal{A}$`,vec)`.

Examples: $\mathcal{K} = \begin{pmatrix} 1 & 3 & 5 \\ -4 & 3 & 7 \\ 8 & 6 & 4 \end{pmatrix}$

`lu` $:=$ `lu_decom(`$\mathcal{K}$`)` $=$

$$\left\{ \begin{pmatrix} 8 & 0 & 0 \\ -4 & 6 & 0 \\ 1 & 2.25 & 1.1251 \end{pmatrix}, \begin{pmatrix} 1 & 0.75 & 0.5 \\ 0 & 1 & 1.5 \\ 0 & 0 & 1 \end{pmatrix}, [\,3\;2\;3\,] \right\}$$

$$\texttt{first lu * second lu} = \begin{pmatrix} 8 & 6 & 4 \\ -4 & 3 & 7 \\ 1 & 3 & 5 \end{pmatrix}$$

$$\texttt{convert(}\mathcal{K}\texttt{,third lu)} = \begin{pmatrix} 8 & 6 & 4 \\ -4 & 3 & 7 \\ 1 & 3 & 5 \end{pmatrix}$$

$$\mathcal{P} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 4 & 5 & 2 \\ 1 & i & 0 \end{pmatrix}$$

$$\texttt{lu} := \texttt{lu\_decom}(\mathcal{P}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 4 & -4*i+5 & 0 \\ i+1 & 3 & 0.41463*i+2.26829 \end{pmatrix}, \right.$$
$$\left. \begin{pmatrix} 1 & i & 0 \\ 0 & 1 & 0.19512*i+0.24390 \\ 0 & 0 & 1 \end{pmatrix}, \ [\ 3\ 2\ 3\ ] \right\}$$

$$\texttt{first lu * second lu} = \begin{pmatrix} 1 & i & 0 \\ 4 & 5 & 2 \\ i+1 & i+2 & i+3 \end{pmatrix}$$

$$\texttt{convert}(\mathcal{P}, \texttt{thirdlu}) = \begin{pmatrix} 1 & i & 0 \\ 4 & 5 & 2 \\ i+1 & i+2 & i+3 \end{pmatrix}$$

Related functions:
`cholesky`.

20.33.3.27 make_identity

Syntax:
`make_identity(square_size);`

square_size :- a positive integer.

Synopsis:
`make_identity` creates the identity matrix of dimension square_size.

Examples: $\texttt{make\_identity}(4) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Related functions:
`diagonal`.

20.33.3.28 matrix_augment, matrix_stack

Syntax:

`matrix_augment({mat1,mat2, ...,matn});`[33]

mat$_1$,mat$_2$,...,mat$_n$:- matrices.

Synopsis:

`matrix_augment` sticks the matrices in `matrix_list` together horizontally.

`matrix_stack` sticks the matrices in `matrix_list` together vertically.

Examples:

`matrix_augment`$(\{\mathcal{A},\mathcal{A}\}) = \begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 4 & 4 & 6 & 4 & 5 & 6 \\ 7 & 8 & 9 & 7 & 8 & 9 \end{pmatrix}$

`matrix_stack`$(\{\mathcal{A},\mathcal{A}\}) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$

Related functions:

`augment_columns`, `stack_rows`, `sub_matrix`.

20.33.3.29 matrixp

Syntax:

`matrixp(test_input);`

test_input :- anything you like.

Synopsis:

`matrixp` is a boolean function that returns t if the input is a matrix and nil otherwise.

Examples:

`matrixp`($\mathcal{A}$) = t

`matrixp`(doodlesackbanana) = nil

Related functions:

`squarep`, `symmetricp`.

[33] If you're feeling lazy then the {}'s can be omitted.

20.33.3.30 matrix_stack

See: `matrix_augment`.

20.33.3.31 minor

Syntax:

`minor(`$\mathcal{A}$`,r,c);`

$\mathcal{A}$	:-	a matrix.
r, c	:-	positive integers.

Synopsis:

`minor` computes the (r, c)th minor of $\mathcal{A}$.

This is created by removing the rth row and the cth column from $\mathcal{A}$.

Examples: `minor`$(\mathcal{A}, 1, 3) = \begin{pmatrix} 4 & 5 \\ 7 & 8 \end{pmatrix}$

Related functions:

`remove_columns`, `remove_rows`.

20.33.3.32 mult_columns, mult_rows

Syntax:

`mult_columns(`$\mathcal{A}$`,column_list,expr);`

$\mathcal{A}$	:-	a matrix.
column_list	:-	a positive integer or a list of positive integers.
expr	:-	an algebraic expression.

Synopsis:

`mult_columns` returns a copy of $\mathcal{A}$ in which the columns specified in column_list have been multiplied by expr.

`mult_rows` performs the same task on the rows of $\mathcal{A}$.

Examples:

`mult_columns`$(\mathcal{A}, \{1, 3\}, x) = \begin{pmatrix} x & 2 & 3*x \\ 4*x & 5 & 6*x \\ 7*x & 8 & 9*x \end{pmatrix}$

`mult_rows`$(\mathcal{A}, 2, 10) = \begin{pmatrix} 1 & 2 & 3 \\ 40 & 50 & 60 \\ 7 & 8 & 9 \end{pmatrix}$

Related functions:

`add_to_columns`, `add_to_rows`.

20.33.3.33 `mult_rows`

See: `mult_columns`.

20.33.3.34 pivot

Syntax:

`pivot(`$\mathcal{A}$`,r,c);`

$\mathcal{A}$:- a matrix.

r, c :- positive integers such that $\mathcal{A}(r, c) \neq 0$.

Synopsis:

`pivot` pivots $\mathcal{A}$ about its (r, c)th entry.

To do this, multiples of the r'th row are added to every other row in the matrix.

This means that the c'th column will be 0 except for the (r,c)'th entry.

Examples: `pivot`$(\mathcal{A}, 2, 3) = \begin{pmatrix} -1 & -0.5 & 0 \\ 4 & 5 & 6 \\ 1 & 0.5 & 0 \end{pmatrix}$

Related functions:

`rows_pivot`.

20.33.3.35 pseudo_inverse

Syntax:

`pseudo_inverse(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix containing only real numeric entries.

Synopsis:

`pseudo_inverse`, also known as the Moore-Penrose inverse, computes the pseudo inverse of $\mathcal{A}$.

Given the singular value decomposition of $\mathcal{A}$, i.e: $\mathcal{A} = \mathcal{U}\Sigma\mathcal{V}^T$, then the pseudo inverse $\mathcal{A}^\dagger$ is defined by $\mathcal{A}^\dagger = \mathcal{V}\Sigma^\dagger\mathcal{U}^T$. For the diagonal matrix Σ, the pseudoinverse $\Sigma^\dagger$ is computed by taking the reciprocal of only the nonzero diagonal elements.

If $\mathcal{A}$ is square and non-singular, then $\mathcal{A}^\dagger = \mathcal{A}$. In general, however, $\mathcal{A}\mathcal{A}^\dagger\mathcal{A} = \mathcal{A}$, and $\mathcal{A}^\dagger\mathcal{A}\mathcal{A}^\dagger = \mathcal{A}^\dagger$.

Perhaps more importantly, $\mathcal{A}^\dagger$ solves the following least-squares problem: given a rectangular matrix $\mathcal{A}$ and a vector b, find the x minimizing $\|\mathcal{A}x - b\|_2$,

and which, in addition, has minimum ℓ_2 (euclidean) Norm, $\|x\|_2$. This x is $\mathcal{A}^\dagger b$.

Examples:

$$\mathcal{R} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 9 & 8 & 7 & 6 \end{pmatrix}, \quad \text{pseudo\_inverse}(\mathcal{R}) = \begin{pmatrix} -0.2 & 0.1 \\ -0.05 & 0.05 \\ 0.1 & 0 \\ 0.25 & -0.05 \end{pmatrix}$$

Related functions:

`svd`.

20.33.3.36 random_matrix

Syntax:

```
random_matrix(r,c,limit);
```

r, c, limit :- positive integers.

Synopsis:

`random_matrix` creates an $r \times c$ matrix with random entries in the range $-\text{limit} < \text{entry} < \text{limit}$.

Switches:

`imaginary`	:-	if on, then matrix entries are $x + iy$ where $-\text{limit} < x, y < \text{limit}$.
`not_negative`	:-	if on then $0 <$ entry $<$ limit. In the imaginary case we have $0 < x, y <$ limit.
`only_integer`	:-	if on then each entry is an integer. In the imaginary case x, y are integers.
`symmetric`	:-	if on then the matrix is symmetric.
`upper_matrix`	:-	if on then the matrix is upper triangular.
`lower_matrix`	:-	if on then the matrix is lower triangular.

Examples:

$$\texttt{random\_matrix}(3,3,10) = \begin{pmatrix} -4.729721 & 6.987047 & 7.521383 \\ -5.224177 & 5.797709 & -4.321952 \\ -9.418455 & -9.94318 & -0.730980 \end{pmatrix}$$

```
on only_integer, not_negative, upper_matrix, imaginary;
```

$\texttt{random\_matrix}(4,4,10) =$

$$\begin{pmatrix} 2*i+5 & 3*i+7 & 7*i+3 & 6 \\ 0 & 2*i+5 & 5*i+1 & 2*i+1 \\ 0 & 0 & 8 & i \\ 0 & 0 & 0 & 5*i+9 \end{pmatrix}$$

20.33.3.37 remove_columns, remove_rows

Syntax:

`remove_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$	:-	a matrix.
column_list	:-	either a positive integer or a list of positive integers.

Synopsis:

`remove_columns` removes the columns specified in column_list from $\mathcal{A}$.

`remove_rows` performs the same task on the rows of $\mathcal{A}$.

Examples:

$\texttt{remove\_columns}(\mathcal{A},2) = \begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{pmatrix}$

$\texttt{remove\_rows}(\mathcal{A},\{1,3\}) = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$

Related functions:

`minor`.

20.33.3.38 remove_rows

See: `remove_columns`.

20.33.3.39 row_dim

See: `column_dim`.

20.33.3.40 rows_pivot

Syntax:

`rows_pivot(`$\mathcal{A}$`,r,c,{row_list});`

$\mathcal{A}$	:-	a matrix.
r,c	:-	positive integers such that $\mathcal{A}$(r,c) neq 0.
row_list	:-	positive integer or a list of positive integers.

Synopsis:

`rows_pivot` performs the same task as `pivot` but applies the pivot only to the rows specified in row_list.

Examples:

$$\mathcal{N} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

$$\texttt{rows\_pivot}(\mathcal{N}, 2, 3, \{4, 5\}) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ -0.75 & 0 & 0.75 \\ -0.375 & 0 & 0.375 \end{pmatrix}$$

Related functions:

`pivot`.

20.33.3.41 simplex

Syntax: `simplex(`⟨*max/min*⟩`,`⟨*objective function*⟩`,{`⟨*linear inequalities*⟩`},[{`⟨*bounds*⟩`}])`

⟨*max/min*⟩	:-	either `max` or `min` (signifying maximise and minimise).
⟨*objective function*⟩	:-	the function you are maximising or minimising.
⟨*linear inequalities*⟩	:-	the constraint inequalities. Each one must be of the form ⟨*linear combination of variables*⟩ ⟨*compop*⟩ ⟨*number*⟩ where ⟨*compop*⟩ is one of <=, =, >=.
⟨*bounds*⟩	:-	bounds on the variables as specified for the LP file format. Each bound has one of the forms $l \leq v$, $v \leq u$, or $l \leq v \leq u$, where v is a variable and l, u are numbers or `infinity` or `-infinity`.

Synopsis:

`simplex` applies the revised simplex algorithm to find the optimal (either maximum or minimum) value of the objective function under the linear inequality constraints.

It returns {*optimal value*, {*values of variables at this optimum*}}.

The {*bounds*} argument is optional and admissible only when the switch fastsimplex is on, which is the default. Without a {*bounds*} argument, the algorithm assumes that all the variables are non-negative.

By default, simplex throws an error if a problem has no feasible solution. However, if the switch noerrsimplex is turned on (it is off by default) then simplex returns the empty list {}, which facilitates use of simplex as a subroutine within other code.

Examples: (assuming on rounded)

```
simplex(max, 10x+5y+5.5z,
        {5x+3z<=200,0.2x+0.1y+0.5z<=12,
         0.1x+0.2y+0.3z<=9,30x+10y+50z<=1500});

{525.0,{x = 40.0,y = 25.0,z = 0}}

simplex(max, x+y, {x>=10,y>=20,x+y<=25});

***** Error in simplex: Problem has no feasible
solution.

on noerrsimplex;

simplex(max, x+y, {x>=10,y>=20,x+y<=25});

{}
```

20.33.3.42 squarep

Syntax:

squarep($\mathcal{A}$);

$\mathcal{A}$:- a matrix.

Synopsis:

squarep is a boolean function that returns t if the matrix is square and nil otherwise.

Examples:

$\mathcal{L} = \begin{pmatrix} 1 & 3 & 5 \end{pmatrix}$

squarep($\mathcal{A}$) = t

squarep($\mathcal{L}$) = nil

Related functions:
`matrixp`, `symmetricp`.

20.33.3.43 stack_rows

See: `augment_columns`.

20.33.3.44 sub_matrix

Syntax:
`sub_matrix(`$\mathcal{A}$`,row_list,column_list);`

$\mathcal{A}$	:-	a matrix.
row_list, column_list	:-	either a positive integer or a list of positive integers.

Synopsis:
`sub_matrix` produces the matrix consisting of the intersection of the rows specified in row_list and the columns specified in column_list.

Examples: `sub_matrix`$(\mathcal{A}, \{1,3\}, \{2,3\}) = \begin{pmatrix} 2 & 3 \\ 8 & 9 \end{pmatrix}$

Related functions:
`augment_columns`, `stack_rows`.

20.33.3.45 svd (singular value decomposition)

Syntax:
`svd(`$\mathcal{A}$`);`

$\mathcal{A}$	:-	a matrix containing only real numeric entries.

Synopsis:
`svd` computes the singular value decomposition of $\mathcal{A}$. If A is an $m \times n$ real matrix of (column) rank r, `svd` returns the 3-element list $\{\mathcal{U}, \Sigma, \mathcal{V}\}$ where $\mathcal{A} = \mathcal{U}\Sigma\mathcal{V}^T$.

Let $k = \min(m, n)$. Then U is $m \times k$, V is $n \times k$, and and $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_k)$, where $\sigma_i \geq 0$ are the singular values of $\mathcal{A}$; only r of these are non-zero. The singular values are the non-negative square roots of the eigenvalues of $\mathcal{A}^T\mathcal{A}$.

$\mathcal{U}$ and $\mathcal{V}$ are such that $\mathcal{U}\mathcal{U}^T = \mathcal{V}\mathcal{V}^T = \mathcal{V}^T\mathcal{V} = \mathcal{I}_k$.

Note: there are a number of different definitions of SVD in the literature, in some of which Σ is square and U and V rectangular, as here, but in others U and V are square, and Σ is rectangular.

Examples:

$$\mathcal{Q} = \begin{pmatrix} 1 & 3 \\ -4 & 3 \\ 3 & 6 \end{pmatrix}$$

$$\texttt{svd}(\mathcal{Q}) = \left\{ \begin{pmatrix} 0.0236042 & 0.419897 \\ -0.969049 & 0.232684 \\ 0.245739 & 0.877237 \end{pmatrix}, \begin{pmatrix} 4.83288 & 0 \\ 0 & 7.52618 \end{pmatrix}, \begin{pmatrix} 0.959473 & 0.281799 \\ -0.281799 & 0.959473 \end{pmatrix} \right\}$$

$$\texttt{svd}(\texttt{TP}(\mathcal{Q})) = \left\{ \begin{pmatrix} 0.959473 & 0.281799 \\ -0.281799 & 0.959473 \end{pmatrix}, \begin{pmatrix} 4.83288 & 0 \\ 0 & 7.52618 \end{pmatrix}, \begin{pmatrix} 0.0236042 & 0.419897 \\ -0.969049 & 0.232684 \\ 0.245739 & 0.877237 \end{pmatrix} \right\}$$

20.33.3.46 swap_columns, swap_rows

Syntax:

`swap_columns(`$\mathcal{A}$`,c1,c2);`

$\mathcal{A}$	:-	a matrix.
c1,c1	:-	positive integers.

Synopsis:

`swap_columns` swaps column c1 of $\mathcal{A}$ with column c2.

`swap_rows` performs the same task on 2 rows of $\mathcal{A}$.

Examples: `swap_columns`$(\mathcal{A}, 2, 3) = \begin{pmatrix} 1 & 3 & 2 \\ 4 & 6 & 5 \\ 7 & 9 & 8 \end{pmatrix}$

Related functions:

`swap_entries`.

20.33.3.47 swap_entries

Syntax:

`swap_entries(`$\mathcal{A}$`,{r1,c1},{r2,c2});`

$\mathcal{A}$	:-	a matrix.
r1,c1,r2,c2	:-	positive integers.

Synopsis:

`swap_entries` swaps $\mathcal{A}$(r1,c1) with $\mathcal{A}$(r2,c2).

Examples: `swap_entries`$(\mathcal{A}, \{1,1\}, \{3,3\}) = \begin{pmatrix} 9 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{pmatrix}$

Related functions:
`swap_columns`, `swap_rows`.

20.33.3.48 swap_rows

See: `swap_columns`.

20.33.3.49 symmetricp

Syntax:
`symmetricp(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix.

Synopsis:
`symmetricp` is a boolean function that returns t if the matrix is symmetric and nil otherwise.

Examples:
$\mathcal{M} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$

`symmetricp`$(\mathcal{A})$ = `nil` `symmetricp`$(\mathcal{M})$ = `t`

Related functions:
`matrixp`, `squarep`.

20.33.3.50 toeplitz

Syntax:
`toeplitz({expr`$_1$`,expr`$_2$`, ...,expr`$_n$`});` [34]

expr$_1$,expr$_2$, ...,expr$_n$:- algebraic expressions.

Synopsis:
`toeplitz` creates the toeplitz matrix from the expression list.

This is a square symmetric matrix in which the first expression is placed on the diagonal and the i'th expression is placed on the (i-1)'th sub and super diagonals.

It has dimension n where n is the number of expressions.

[34] If you're feeling lazy then the {}'s can be omitted.

Examples: `toeplitz`$(\{w,x,y,z\}) = \begin{pmatrix} w & x & y & z \\ x & w & x & y \\ y & x & w & x \\ z & y & x & w \end{pmatrix}$

20.33.3.51 triang_adjoint

Syntax:

`triang_adjoint(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix.

Synopsis: `triang_adjoint` computes the triangularizing adjoint $\mathcal{F}$ of matrix $\mathcal{A}$ due to the algorithm of Arne Storjohann. $\mathcal{F}$ is lower triangular matrix and the resulting matrix $\mathcal{T}$ of $\mathcal{F} * \mathcal{A} = \mathcal{T}$ is upper triangular with the property that the i-th entry in the diagonal of $\mathcal{T}$ is the determinant of the principal i-th submatrix of the matrix $\mathcal{A}$.

Examples:

`triang_adjoint`$(\mathcal{A}) = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -3 & 6 & -3 \end{pmatrix}$

$\mathcal{F} * \mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{pmatrix}$

20.33.3.52 Vandermonde

Syntax:

`vandermonde`({expr$_1$,expr$_2$, ...,expr$_n$}); [35]

expr$_1$,expr$_2$, ...,expr$_n$:- algebraic expressions.

Synopsis:

`Vandermonde` creates the Vandermonde matrix from the expression list. This is the square matrix in which the (i,j)th entry is $\text{expr}_i^{(j-1)}$. It has dimension n, where n is the number of expressions.

Examples: `vandermonde`$(\{x, 2*y, 3*z\}) = \begin{pmatrix} 1 & x & x^2 \\ 1 & 2*y & 4*y^2 \\ 1 & 3*z & 9*z^2 \end{pmatrix}$

[35] If you're feeling lazy then the {}'s can be omitted.

20.33.3.53 kronecker_product

Syntax:

`kronecker_product`(M_1, M_2)

M_1, M_2 :- Matrices

Synopsis:

`kronecker_product` creates a matrix containing the Kronecker product (also called `direct product` or `tensor product`) of its arguments.

Examples:

```
a1 := mat((1,2),(3,4),(5,6))$
a2 := mat((1,1,1),(2,z,2),(3,3,3))$
kronecker_product(a1,a2);
```

$$\begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & z & 2 & 4 & 2*z & 4 \\ 3 & 3 & 3 & 6 & 6 & 6 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 3*z & 6 & 8 & 4*z & 8 \\ 9 & 9 & 9 & 12 & 12 & 12 \\ 5 & 5 & 5 & 6 & 6 & 6 \\ 10 & 5*z & 10 & 12 & 6*z & 12 \\ 15 & 15 & 15 & 18 & 18 & 18 \end{pmatrix}$$

20.33.4 Acknowledgments

Many of the ideas for this package came from the Maple Linalg package.

The algorithms for `cholesky`, `lu_decom`, and `svd` are taken from the book Linear Algebra - J.H. Wilkinson & C. Reinsch.

The `gram_schmidt` code comes from Karin Gatermann's Symmetry package for REDUCE.

20.34 LISTVECOPS: Vector Operations on Lists

Author: Eberhard Schrüfer

This package implements vector operations on lists. Addition, multiplication, division, and exponentiation work elementwise. For example, after

```
A := {a1,a2,a3,a4};
B := {b1,b2,b3,b4};
```

`c*A` will simplify to `{c*a1,..,c*a4}`, `A + B` to `{a1+b1,...,a4+b4}`, and `A*B` to `{a1*b1,...,a4*b4}`. Linear operations work as expected:

```
c1*A + c2*B;

{a1*c1 + b1*c2,

  a2*c1 + b2*c2,

  a3*c1 + b3*c2,

  a4*c1 + b4*c2}
```

A division and an exponentation example:

```
{a,b,c}/{3,g,5};

  a   b   c
{---,---,---}
  3   g   5

ws^3;

   3    3    3
  a    b    c
{----,----,-----}
  27    3   125
       g
```

The new operator `*.` (`ldot`) implements the dot product:

```
{a,b,c,d} *. {5,7,9,11/d};

5*a + 7*b + 9*c + 11
```

For accessing list elements, the new operator _ (`lnth`) can be used instead of the `PART` operator. Note that the infix operator _ must be separated from the name of the list variable to which it is applied, otherwise REDUCE will treat the list name and operator as parts of a single identifier.

```
l := {1,{2,3},4}$

lnth(l,3);

4

l _2*3;

{6,9}

l _2 _2;

3
```

It can also be used to modify a list (unlike `PART`, which returns a modified list):

```
part(l,2,2):=three;

{1,{2,three},4}

l;

{1,{2,3},4}

l _ 2 _2 :=three;

three

l;

{1,{2,three},4}
```

Operators are distributed over lists:

```
a *. log b;

log(b1)*a1 + log(b2)*a2 + log(b3)*a3 + log(b4)*a4

df({sin x*y,x^3*cos y},x,2,y);
```

```
{ - sin(x), - 6*sin(y)*x}

int({sin x,cos x},x);

{ - cos(x),sin(x)}
```

Finally, here are two examples of using vector operations on lists within procedures that return lists:

```
listproc normalize v;
   v / sqrt(v *. v);

listproc spat3(u,v,w);
   begin scalar x,y;
     x := u *. w;
     y := u *. v;
     return v*x - w*y
   end;
```

20.35 LPDO: Linear Partial Differential Operators

Author: Thomas Sturm

20.35.1 Introduction

Consider the field $F = \mathbb{Q}(x_1, \ldots, x_n)$ of rational functions and a set $\Delta = \{\partial_{x_1}, \ldots, \partial_{x_n}\}$ of *commuting derivations* acting on F. That is, for all $\partial_{x_i}, \partial_{x_j} \in \Delta$ and all $f, g \in F$ the following properties are satisfied:

$$\begin{aligned} \partial_{x_i}(f+g) &= \partial_{x_i}(f) + \partial_{x_i}(g), \\ \partial_{x_i}(f \cdot g) &= f \cdot \partial_{x_i}(g) + \partial_{x_i}(f) \cdot g, \qquad (20.91) \\ \partial_{x_i}(\partial_{x_j}(f)) &= \partial_{x_j}(\partial_{x_i}(f)). \qquad (20.92) \end{aligned}$$

Consider now the set $F[\partial_{x_1}, \ldots, \partial_{x_n}]$, where the derivations are used as variables. This set forms a non-commutative *linear partial differential operator ring* with pointwise addition, and multiplication defined as follows: For $f \in F$ and ∂_{x_i}, $\partial_{x_j} \in \Delta$ we have for any $g \in F$ that

$$\begin{aligned} (f\partial_{x_i})(g) &= f \cdot \partial_{x_i}(g), \\ (\partial_{x_i} f)(g) &= \partial_{x_i}(f \cdot g), \qquad (20.93) \\ (\partial_{x_i}\partial_{x_j})(g) &= \partial_{x_i}(\partial_{x_j}(g)). \qquad (20.94) \end{aligned}$$

Here " $\cdot$ " denotes the multiplication in F. From (20.94) and (20.92) it follows that $\partial_{x_i}\partial_{x_j} = \partial_{x_j}\partial_{x_i}$, and using (20.93) and (20.91) the following *commutator* can be proved:

$$\partial_{x_i} f = f\partial_{x_i} + \partial_{x_i}(f).$$

A *linear partial differential operator* (LPDO) of order k is an element

$$D = \sum_{|j| \le k} a_j \partial^j \in F[\partial_{x_1}, \ldots, \partial_{x_n}]$$

in canonical form. Here the expression $|j| \le k$ specifies the set of all tuples of the form $j = (j_1, \ldots, j_n) \in \mathbb{N}^n$ with $\sum_{i=1}^n j_i \le k$, and we define $\partial^j = \partial_{x_1}^{j_1} \cdots \partial_{x_n}^{j_n}$.

A *factorization* of D is a non-trivial decomposition

$$D = D_1 \cdots D_r \in F[\partial_{x_1}, \ldots, \partial_{x_n}]$$

into multiplicative factors, each of which is an LPDO D_i of order greater than 0 and less than k. If such a factorization exists, then D is called *reducible* or *factorable*, else *irreducible*.

For the purpose of factorization it is helpful to temporarily consider as regular commutative polynomials certain summands of the LPDO under consideration. Consider a commutative polynomial ring over F in new indeterminates $y_1, \ldots, y_n$. Adopting the notational conventions above, for $m \leq k$ the *symbol of D of order m* is defined as

$$\mathrm{Sym}_m(D) = \sum_{|j|=m} a_j y^j \in F[y_1, \ldots, y_n].$$

For $m = k$ we obtain as a special case the *symbol* $\mathrm{Sym}(D)$ *of* D.

20.35.2 Operators

20.35.2.1 `partial`

There is a unary operator `partial(·)` denoting ∂.

⟨*partial-term*⟩ → **`partial (`** ⟨*id*⟩ **`)`**

20.35.2.2 `***`

There is a binary operator $\star\star\star$ for the non-commutative multiplication involving partials ∂_x. All expressions involving $\star\star\star$ are implicitly transformed into LPDOs, i.e., into the following normal form:

⟨*normalized-lpdo*⟩ → ⟨*normalized-mon*⟩ [**`+`** ⟨*normalized-lpdo*⟩]
⟨*normalized-mon*⟩ → ⟨*F-element*⟩ [**`***`** ⟨*partial-termprod*⟩]
⟨*partial-termprod*⟩ → ⟨*partial-term*⟩ [**`***`** ⟨*partial-termprod*⟩]

The summands of the *normalized-lpdo* are ordered in some canonical way. As an example consider

```
input: a()***partial(y)***b()***partial(x);

(a()*b()) *** partial(x) *** partial(y)
 + (a()*diff(b(),y,1)) *** partial(x)
```

Here the *F-elements* are polynomials, where the unknowns are of the type *constant-operator* denoting functions from F:

⟨*constant-operator*⟩ → ⟨*id*⟩ **`( )`**

We do not admit division of such constant operators since we cannot exclude that such a constant operator denotes 0.

The operator notation on the one hand emphasizes the fact that the denoted elements are functions. On the other hand it distinguishes `a()` from the variable `a` of a rational function, which specifically denotes the corresponding projection. Consider e.g.

```
input: (x+y)***partial(y)***(x-y)***partial(x);

  2    2
(x  - y ) *** partial(x) *** partial(y) + ( - x - y) *** partial(x)
```

Here we use as *F-elements* specific elements from $F = \mathbb{Q}(x, y)$.

20.35.2.3 `diff`

In our example with constant operators, the transformation into normal form introduces a formal derivative operation `diff(·,·,·)`, which cannot be evaluated. Notice that we do not use the Reduce operator `df(·,·,·)` here, which for technical reasons cannot smoothly handle our constant operators.

In our second example with rational functions as *F-elements*, derivative occurring with commutation can be computed such that `diff` does not occur in the output.

20.35.3 Shapes of F-elements

Besides the generic computations with constant operators, we provide a mechanism to globally fix a certain *shape* for *F-elements* and to expand constant operators according to that shape.

20.35.3.1 `lpdoset`

We give an example for a shape that fixes all constant operators to denote generic bivariate affine linear functions:

```
input: d := (a()+b())***partial(x1)***partial(x2)**2;

                                                    2
d := (a() + b()) *** partial(x1) *** partial(x2)

input: lpdoset {!#10*x1+!#01*x2+!#00,x1,x2};

{-1}

input: d;

(a00 + a01*x2 + a10*x1 + b00 + b01*x2 + b10*x1)
```

```
                                  2
 *** partial(x1) *** partial(x2)
```

Notice that the placeholder # must be escaped with !, which is a general convention for Rlisp/Reduce. Notice that `lpdoset` returns the old shape and that `{-1}` denotes the default state that there is no shape selected.

20.35.3.2 lpdoweyl

The command `lpdoweyl {n,x1,x2,...}` creates a shape for generic polynomials of total degree `n` in variables `x1`, `x2`, ...

```
input: lpdoweyl(2,x1,x2);

                                2
{#_00_ + #_01_*x2 + #_02_*x2  + #_10_*x1

                              2
  + #_11_*x1*x2 + #_20_*x1 ,x1,x2}

input: lpdoset ws;

{#10*x1 + #01*x2 + #00,x1,x2}

input: d;

                                2
(a_00_ + a_01_*x2 + a_02_*x2  + a_10_*x1

                              2
  + a_11_*x1*x2 + a_20_*x1  + b_00_ + b_01_*x2

            2                                       2
  + b_02_*x2  + b_10_*x1 + b_11_*x1*x2 + b_20_*x1

                                    2
 ) *** partial(x1) *** partial(x2)
```

20.35.4 Commands

20.35.4.1 General

lpdoord

The *order* of an lpdo:

```
input: lpdoord((a()+b())***partial(x1)
```

```
          ***partial(x2)**2+3***partial(x1));

3
```

lpdoptl

Returns the list of derivations (partials) occurring in its argument LPDO d.

```
input: lpdoptl(a()***partial(x1)***partial(x2)
               +partial(x4)+diff(a(),x3,1));

{partial(x1),partial(x2),partial(x4)}
```

That is the smallest set $\{\ldots, \partial_{x_i}, \ldots\}$ such that d is defined in $F[\ldots, \partial_{x_i}, \ldots]$. Notice that formal derivatives are not derivations in that sense.

lpdogp

Given a starting symbol a, a list of variables l, and a degree n, `lpdogp(a,l,n)` generates a generic (commutative) polynomial of degree n in variables l with coefficients generated from the starting symbol a:

```
input: lpdogp(a,{x1,x2},2);

                              2                                    2
a_00_ + a_01_*x2 + a_02_*x2  + a_10_*x1 + a_11_*x1*x2 + a_20_*x1
```

lpdogdp

Given a starting symbol a, a list of variables l, and a degree n, `lpdogp(a,l,n)` generates a generic differential polynomial of degree n in variables l with coefficients generated from the starting symbol a:

```
input: lpdogdp(a,{x1,x2},2);

                       2                          2
a_20_ *** partial(x1)  + a_02_ *** partial(x2)

 + a_11_ *** partial(x1) *** partial(x2) + a_10_ *** partial(x1)

 + a_01_ *** partial(x2) + a_00_
```

20.35.4.2 Symbols

lpdosym

The *symbol* of an lpdo. That is the differential monomial of highest order with the partials replaced by corresponding commutative variables:

```
input: lpdosym((a()+b())***partial(x1)***partial(x2)**2
               +3***partial(x1));

           2
y_x1_*y_x2_ *(a() + b())
```

More generally, one can use a second optional arguments to specify a the order of a different differential monomial to form the symbol of:

```
input: lpdosym((a()+b())***partial(x1)***partial(x2)**2
               +3***partial(x1),1);

3*y_x1_
```

Finally, a third optional argument can be used to specify an alternative starting symbol for the commutative variable, which is `y` by default. Altogether, the optional arguments default like `lpdosym(·)=lpdosym(·,lpdoord(·),y)`.

lpdosym2dp

This converts a symbol obtained via `lpdosym` back into an LPDO resulting in the corresponding differential monomial of the original LPDO.

```
input: d := a()***partial(x1)***partial(x2)+partial(x3)$

input: s := lpdosym d;

s := a()*y_x1_*y_x2_

input: lpdosym2dp s;

a() *** partial(x1) *** partial(x2)
```

In analogy to `lpdosym` there is an optional argument for specifying an alternative starting symbol for the commutative variable, which is `y` by default.

lpdos

Given LPDOs p, q and $m \in \mathbb{N}$ the function `lpdos(p,q,m)` computes the commutative polynomial

$$S_m = \sum_{\substack{|j|=m \\ |j|<k}} \left(\sum_{i=1}^{n} p_i \partial_i(q_j) + p_0 q_j \right) y^j.$$

This is useful for the factorization of LPDOs.

```
input: p := a()***partial(x1)+b()$

input: q := c()***partial(x1)+d()***partial(x2)$

input: lpdos(p,q,1);

a()*diff(c(),x1,1)*y_x1_ + a()*diff(d(),x1,1)*y_x2_ + b()*c()*y_x1_

 + b()*d()*y_x2_
```

20.35.4.3 Factorization

lpdofactorize

Factorize the argument LPDO d. The ground field F must be fixed via `lpdoset`. The result is a list of lists $\{\dots,(A_i,L_i),\dots\}$. A_i is is genrally the identifiers `true`, which indicates reducibility. The respective L_i is a list of two differential polynomial factors, the first of which has order 1.

```
input: bk := (partial(x)+partial(y)+(a10-a01)/2) ***
       (partial(x)-partial(y)+(a10+a01)/2);

                2             2
bk := partial(x)  - partial(y)  + a10*partial(x)

                           2      2
                       - a01  + a10
 + a01*partial(y) + -----------------
                           4

input: lpdoset lpdoweyl(1,x,y);

{#_00_ + #_01_*y + #_10_*x,x,y}

input: lpdofactorize bk;

{{true,
```

```
                                      a01 - a10
{ - partial(x) - partial(y) + ------------,
                                        2

                                       - a01 - a10
  - partial(x) + partial(y) + --------------}}}
                                        2
```

If the result is the empty list, then this guarantees that there is no approximate factorization possible. In general it is possible to obtain several sample factorizations. Note, however, that the result does not provide a complete list of possible factorizations with a left factor of order 1 but only at least one such sample factorization in case of reducibility.

Furthermore, the procedure might fail due to polynomial degrees exceeding certain bounds for the extended quantifier elimination by virtual substitution used internally. In this case there is the identifier `failed` returned. This must not be confused with the empty list indicating irreducibility as described above.

Besides

1. the LPDO d,

`lpdofactorize` accepts several optional arguments:

2. An LPDO of order 1, which serves as a template for the left (linear) factor. The default is a generic linear LPDO with generic coefficient functions according from the ground field specified via `lpdoset`. The principle idea is to support the factorization by guessing that certain differential monomials are not present.

3. An LPDO of order $\mathrm{ord}(d)-1$, which serves as a template for the right factor. Similarly to the previous argument the default is fully generic.

lpdofac

This is a low-level entry point to the factorization `lpdofactorize`. It accepts the same arguments as `lpdofactorize`. It generates factorization conditions as a quite large first-order formula over the reals. This can be passed to extended quantifier elimination. For example, consider `bk` as in the example for

`lpdofactorize` above:

```
input: faccond := lpdofac bk$

input: rlqea faccond;

{{true,

                a01 - a10
  {p_00_00_ = -----------,
                     2

   p_00_01_ = 0,

   p_00_10_ = 0,

   p_01_00_ = -1,

   p_01_01_ = 0,

   p_01_10_ = 0,

   p_10_00_ = -1,

   p_10_01_ = 0,

   p_10_10_ = 0,

                 - a01 - a10
   q_00_00_ = ---------------,
                      2

   q_00_01_ = 0,

   q_00_10_ = 0,

   q_01_00_ = 1,

   q_01_01_ = 0,

   q_01_10_ = 0,

   q_10_00_ = -1,

   q_10_01_ = 0,

   q_10_10_ = 0}}}
```

The result of the extended quantifier elimination provides coefficient values for generic factor polynomials p and q. These are automatically interpreted and converted into differential polynomials by `lpdofactorize`.

20.35.4.4 Approximate Factorization

`lpdofactorizex`

Approximately factorize the argument LPDO d. The ground field F must be fixed via `lpdoset`. The result is a list of lists $\{\ldots,(A_i,L_i),\ldots\}$. Each A_i is quantifier-free formula possibly containing a variable `epsilon`, which describes the precision of corresponding factorization L_i. L_i is a list containing two factors, the first of which is linear.

```
input: off lpdocoeffnorm$

input: lpdoset lpdoweyl(0,x1,x2)$

input: f2 := partial(x1)***partial(x2) + 1$

input: lpdofactorizex f2;

{{epsilon - 1 >= 0,{partial(x1),partial(x2)}},

 {epsilon - 1 >= 0,{partial(x2),partial(x1)}}}
```

If the result is the empty list, then this guarantees that there is no approximate factorization possible. In our example we happen to obtain two possible factorizations. Note, however, that the result in general does not provide a complete list of factorizations with a left factor of order 1 but only at least one such sample factorization.

Furthermore, the procedure might fail due to polynomial degrees exceeding certain bounds for the extended quantifier elimination by virtual substitution used internally. If this happens, the corresponding A_i will contain existential quantifiers `ex`, and L_i will be meaningless.
`Da sollte besser ein failed kommen ...`

The first of the two subresults above has the semantics that $\partial_{x_1}\partial_{x_2}$ is an approximate factorization of f_2 for all $\varepsilon \geq 1$. Formally, $||f_2 - \partial_{x_1}\partial_{x_2}|| \leq \varepsilon$ for all $\varepsilon \geq 1$, which is equivalent to $||f_2 - \partial_{x_1}\partial_{x_2}|| \leq 1$. That is, 1 is an upper bound for the approximation error over $\mathbb{R}^2$. Where there are two possible choices for the seminorm $||\cdot||$:

1. ...
2. ...

`explain switch lpdocoeffnorm ...`

Besides

1. the LPDO d,

`lpdofactorizex` accepts several optional arguments:

2. A Boolean combination ψ of equations, negated equations, and (possibly strict) ordering constraints. This ψ describes a (semialgebraic) region over which to factorize approximately. The default is `true` specifying the entire $\mathbb{R}^n$. It is possible to choose ψ parametrically. Then the parameters will in general occur in the conditions A_i in the result.

3., 4. An LPDO of order 1, which serves as a template for the left (linear) factor, and an LPDO of order $\mathrm{ord}(d) - 1$, which serves as a template for the right factor. See the documentation of `lpdofactorize` for defaults and details.

5. A bound ε for describing the desired precision for approximate factorization. The default is the symbol `epsilon`, i.e., a symbolic choice such that the optimal choice (with respect to parameters in ψ) is obtained during factorization. It is possible to fix $\varepsilon \in \mathbb{Q}$. This does, however, not considerably simplify the factorization process in most cases.

```
input: f3 := partial(x1) *** partial(x2) + x1$

input: psi1 := 0<=x1<=1 and 0<=x2<=1$

input: lpdofactorizex(f3,psi1,a()***partial(x1),b()***partial(x2));

{{epsilon - 1 >= 0,{partial(x1),partial(x2)}}}
```

lpdofacx

This is a low-level entry point to the factorization `lpdofactorizex`. It is analogous to `lpdofac` for `lpdofactorize`; see the documentation there for details.

lpdohrect

lpdohcirc

20.36 MATHML: REDUCE-MathML Interface

Mathematical Markup Language (MathML) is an application of XML for describing mathematics. It has two flavours: *Presentation MathML*, like TEX, is intended for displaying mathematics, primarily within web pages, but its meaning can be ambiguous. *Content MathML*, like OpenMath, is intended for conveying meaning, but how it is displayed is determined by the display software. This package allows REDUCE to input and output *Content* MathML.

Author: Luis Alvarez-Sobreviela[36]

20.36.1 Introduction

The MathML interface for REDUCE provides commands to evaluate and output MathML. The principal features of this package are as follows.

- Evaluation of MathML code. This allows REDUCE to parse MathML expressions and evaluate them.
- Generation of MathML compliant code. This provides output of REDUCE expressions in MathML code.

20.36.2 Loading

To use the MathML-REDUCE Interface, the package must first be loaded explicitly by executing the command

```
load_package mathml;
```

20.36.3 Switches

There are two switches that can be used together and are off by default.

`mathml`: When on, all output will be displayed as MathML.

`both`: When on, all output will be displayed as both MathML and normal REDUCE output (which is much easier to read than MathML).

20.36.4 Entering MathML

The MathML-REDUCE Interface allows the user to provide MathML input via a file containing MathML or by entering MathML interactively.

[36] This package was written when the author was a placement student at ZIB Berlin.

20.36.4.1 Reading MathML from a File: MML

When reading from a file the operator mml is used, which takes as argument the name of the file containing the MathML.

mml(FILE:*string*):*expression*

Example: As long as the file given contains valid MathML, no errors should be produced.

```
mml "ex.mml";
```

20.36.4.2 Reading MathML interactively: PARSEML

The operator parseml takes no arguments and expects you to enter MathML markup starting with <mathml> and ending with </mathml>. It then outputs an expression resulting from evaluating the input.

Example: Here is an extract of a REDUCE session where parseml() is used:

```
parseml();
<math>
   <apply><plus/>
      <cn>3</cn>
      <cn>5</cn>
   </apply>
</math>

8
```

20.36.5 The Evaluation of MathML

MathML is always evaluated by REDUCE in algebraic mode before outputting any results. Hence, undefined variables remain undefined, and it is possible to use all normal REDUCE switches and packages.

Example: The following MathML input

```
parseml();
<math>
   <reln><gt/>
```

```
      <ci>x</ci>
      <ci>y</ci>
   </reln>
</math>
```

will evaluate to `x>y`. (Note that this expression cannot be entered directly on its own as REDUCE input!)

Now suppose we execute the following:

```
x:=3; y:=2;
```

If we once again enter and evaluate the above MathML input we will have as result `3>2`. This can be evaluated to `true` or `false` as follows:

```
if ws then true else false;
true
```

Of course, it is possible to set only one of the two variables `x` or `y` used above, say `y:=4`. If we once again enter and evaluate the above MathML we will get the result `x>4`.

When one of the switches `mathml` or `both` is on, the MathML output will be:

```
<math>
   <reln><gt/>
      <ci>x</ci>
      <cn type="integer">4</cn>
   </reln>
</math>
```

Example: Let the file `ex.mml` contain the following MathML:

```
<math>
   <apply><int/>
      <bvar>
         <ci>x</ci>
      </bvar>
      <apply><fn><ci>F</ci><fn>
         <ci>x</ci>
      </apply>
   </apply>
</math>
```

If we execute the command

```
mml "ex.mml";
```

we get `int(f(x),x)` or

$$\int f(x)dx$$

It is clear that this has remained unevaluated. We can now define the function `f(x)` as follows:

```
for all x let f(x) = 2*x^2;
```

If we then enter `mml "ex.mml";` once again, we will have the following result:

$$\frac{2x^3}{3}$$

This shows that the MathML-REDUCE Interface allows normal REDUCE interactions to manipulate the evaluation of MathML input without needing to edit the MathML itself.

20.36.6 Interpretation of Error Messages

The MathML-REDUCE Interface has a set of error messages which aim to help the user understand and correct any invalid MathML. Because there can exist many different causes of errors, such error messages should be considered merely as advice. Here are the most important error messages.

Missing Tag: Many MathML tags are used in pairs, e.g. `<apply> </apply>`, `<reln> </reln>`, `<ci> </ci>`. In the case where the ending tag is missed out, or misspelled, it is very probable that an error of this type will be thrown.

Ambiguous or Erroneous Use of `<apply>`: This error message is non-specific. When it appears, it is not very clear to the interface where exactly the error lies. Probable causes are misuse of the `<apply> </apply>` tags, or misspelling of the tag preceding the `<apply>` tag. However, other types of error may cause this error message to appear.

Tags following an `<apply>` tag may be misspelled without causing an error message, but they will be considered as operators by REDUCE and therefore generate some unexpected expression.

Syntax Errors: It is possible that the input MathML is not syntactically correct. In such a situation, the error will be spotted, and in some cases a resolution might

be presented. There are a variety of syntax error messages, but relying on their advice might not always be helpful.

Despite the verbose nature of the error messages and their recommended resolutions, in most situations reference to the MathML specification is recommended.

20.36.7 Limitations of the Interface

Not all aspects of MathML have been perfectly fitted into the interface. There are still some unsolved problems in the present version of the interface:

- MathML Presentation Markup is not supported. The interface will treat every presentation tag as either an unknown tag or a REDUCE operator.
- Certain MathML tags do not play an important role in the REDUCE environment. Such tags will be parsed correctly, although their action will be ignored. These tags are:
  - `<interval> </interval>`
  - `<inverse> </inverse>`
  - `<condition> </condition>`
  - `<compose> </compose>`
  - `<ident> </ident>`
  - `<forall/>`
  - `<exists/>`

  However, the tags `<condition> </condition>` and `<interval> </interval>` are supported when used within the following tags:

  - `<int/>`
  - `<limit/>`
  - `<sum/>`
  - `<product/>`
- The `<declare>` construct takes one or two arguments. It sets the first argument to the value of the second. In the case where the second argument is a vector or a matrix, an obscure error message is produced.

20.36.8 Examples

Example 1: Enter the following input interactively:

```
on mathml;
solve({z=x*a+1},{z,x});
```

Example 2: Have a file `ex2.mml` containing the following MathML:

```
<mathml>
   <apply><sum/>
      <bvar>
         <ci>x</ci>
      </bvar>
      <apply><fn><ci>F</ci><fn>
         <ci>x</ci>
      </apply>
   </apply>
</mathml>
```

and execute the following command:

```
mml "ex2.mml";
```

Example 3: This example illustrates how practical the switch `both` can be for interpreting verbose MathML. Introduce the following MathML into a file, say `ex3.mml`:

```
<mathml>
   <apply><int/>
      <bvar>
         <ci>x</ci>
      </bvar>
      <apply><sin/>
         <apply><log/>
            <ci>x</ci>
         </apply>
      </apply>
   </apply>
</mathml>
```

then execute the following:

```
on both;
mml "ml";
```

20.36.9 An Overview of How the Interface Works

The interface is primarily built in two parts. A first one which parses and evaluates MathML, and a second one which parses REDUCE's algebraic expressions and

outputs them in MathML format. Both parts use Top-Down Recursive Descent parsing with one token look ahead.

The EBNF description of the MathML grammar is defined informally in APPENDIX E of the original MathML specification, which was used to develop the MathML parser. The MathML parser evaluates all that is possible and returns a valid REDUCE algebraic expression. When either of the switches `mathml` or `both` is on, this algebraic expression is fed into the second part of the program, which parses these expressions and transforms them back into MathML.

The MathML generator parses the algebraic expression produced by either REDUCE itself or the MathML parser. It works in a very similar way to the MathML parser. It is simpler, since no evaluation is involved. All the generated code is MathML compliant. It is important to note that the MathML code generator sometimes introduces Presentation Markup tags, and other tags which are not understood by the MathML parser of the interface.[37]

[37]The set of tags not understood by the MathML parser is detailed in Limitations of the Interface.

20.37 MATHMLOM: REDUCE OpenMath/MathML Interface

OpenMath provides extensible standards for representing the semantics of mathematical objects and communicating them between software systems. This package provides commands to translate OpenMath into content MathML and vice versa.

Author: Luis Alvarez-Sobreviela

To use the OpenMath/MathML Interface, the package must first be loaded explicitly by executing the command

```
load_package mathmlom;
```

The following commands translate subsequent input from one standard to the other:

`om2mml();` translates OpenMath into MathML;

`mml2om();` translates MathML into OpenMath.

Execute one of the above commands, then input one complete expression using the appropriate standard. REDUCE will outputs its intermediate Lisp representation followed by the expression translated to the other standard, and then revert to normal REDUCE input syntax.

Here is a simple example of translating OpenMath into MathML taken from the end of the file `mathmlom.rlg`. The following input

```
om2mml();
<OMOBJ>
  <OMA>
    <OMS name="rational" cd="nums1"/>
    <OMI>4</OMI>
    <OMI>2</OMI>
  </OMA>
</OMOBJ>
```

produces the following output:

```
Intermediate representation:
(rational nil 4 2)

<math>
   <cn type="rational">4<sep/>2</cn>
</math>
```

20.38 MRVLIMIT: A New Exp-Log Limits Package

Author: Neil Langmead

This package was written when the author was a placement student at ZIB Berlin.

20.38.1 The Exp-Log Limits package

This package arises from the PhD thesis of Dominik Gruntz[Gru96], of the ETH Zürich. He developed a new algorithm to compute limits of "exp-log" functions. Many of the examples he gave were unable to be computed by the present LIMITS package in REDUCE, the simplest example being the following, whose limit is obviously 0:

```
load limits;

limit(x^7/e^x,x,infinity);

         7
        x
limit(----,x,infinity)
         x
        e
```

This particular problem arises, because L'Hôpital's rule for the computation of indefinite forms (such as $0/0$, or $\frac{\infty}{\infty}$) can only be applied in a CAS a finite number of times, and in REDUCE, this number is 3. Appling L'Hôpital's rule 7 times to the above problem would have yielded the correct answer 0.

The new algorithm solves this particular problem, and enables the computation of many more limit calculations in REDUCE. We first define the domain in which we work, and then give a statement of the main algorithm that is used in this package.

Definition:
Let $\Re[x]$ be the ring of polynomials in x with real coefficients, and let f be an element in this ring. The field which is obtained from $\Re[x]$ by closing it under the operations $f \to \exp(f)$ and $f \to \log|f|$ is called the $\mathcal{L}$-field (or logarithmico-exponential field, or field of exp-log functions for short).

Hardy proved [Har12] that every $\mathcal{L}$ function is ultimately continuous, of constant sign, monotonic, and tends to $\pm\infty$ or to a finite real constant as $x \to +\infty$.

Here are some examples of exp-log functions, which the package is able to deal

with:

$$f(x) = e^x * \log(\log(x))$$

$$f(x) = \frac{\log(\log(x + e^{-x}))}{e^{x^2} + \log(\log(x))}$$

$$f(x) = \log(x)^{\log(x)}$$

$$f(x) = e^{x*\log(x)}$$

20.38.2 The Algorithm

A complete statement of the algorithm now follows: Let f be a log-exp function in x, whose limit we wish to compute as $x \to x_0$. The main steps of the algorithm to do this are as follows:

- Determine the set Ω of the most rapidly varying subexpressions of $f(x)$. Limits may have to be computed recursively at this stage.
- Choose an expression ω such that $\omega > 0$, $\lim_{x\to\infty} \omega = 0$ and ω is in the same comparability class as any element of Ω. Rewrite the other expressions in Ω as $A(x)\omega^c$, where $A(x)$ only contains subexpressions in lower comparability classes than Ω.
- Let $f(\omega)$ be the function obtained from $f(x)$ by replacing all elements of Ω by their representation in terms of ω. Consider all expressions independent of ω as constants and compute the leading term of the power series of f(ω) around $\omega = 0^+$ as $c_0\omega^{e_0}$.
- If the leading exponent $e_0 > 0$, then the limit is 0, and we stop. If the leading exponent $e_0 < 0$ then the limit is $\pm\infty$. The sign is defined by the sign of the leading coefficient c_0. If the leading exponent $e_0 = 0$ then the limit is the limit of the leading coeficient c_0. If $c_0 \notin C$, where $C = \text{Const}(L)$, the set of exp-log constants, we apply the same algorithm recursively on c_0.

The algorithm to compute the most rapidly varying subset (the mrv set) of a function f is given below:

procedure mrv(f)
% f an exp log function in x
if (not (depend(f,x))) $\to$ return ({})
else if $f = x \to$ return($\{x\}$)
else if $f = gh \to$ return(max(mrv(g),mrv(h)))
else if $f = g + h \to$ return(max(mrv(g),mrv(h)))

else if $f = g^c$ and c $\in C \rightarrow$ return(mrv(g))
else if $f = \log(g) \rightarrow$ return(mrv(g))
else if $f = e^g \rightarrow$
    if $\lim_{x\to\infty} g = \pm\infty \rightarrow$
    return(max($\{e^g\}$, mrv(g)))
    else $\rightarrow$ return mrv(g)
end

The function max() computes the maximum of the two sets of expressions. Max() compares two elements of its argument sets and returns the set which is in the higher comparability class or the union of both if they have the same order of variation.

For example, we have

$$\begin{aligned}
&\text{mrv}(e^x) = \{e^x\}\\
&\text{mrv}(\log(\log(\log(x + x^2 + x^3)))) = \{x\}\\
&\text{mrv}(x) = \{x\}\\
&\text{mrv}(e^x + e^{-x} + x^2 + x\log(x)) = \{e^x, e^{-x}\}\\
&\text{mrv}(e^{e^{-x}}) = \{e^{-x}\}
\end{aligned}$$

For further details, proofs and explanations of the algorithm, please consult Gruntz' thesis[Gru96].

20.38.2.1 Mrv_limit Examples

Consider the following in REDUCE:

```
mrv_limit(e^x,x,infinity);

infinity

mrv_limit(1/log(x),x,infinity);

0

b:=e^x*(e^(1/x-e^-x)-e^(1/x));


            -1         - x
       x + x       - e
b := e         *(e           - 1)


mrv_limit(b,x,infinity);
```

```
-1


ex := (log(-log(x)+log(log(x)))-log(log(x)))
        /(log(log(x)+log(log(log(x)))))*log(x);

       log(x)*( - log(log(x)) + log(log(log(x)) - log(x)))
ex := ---------------------------------------------------
                 log(log(log(log(x))) + log(x))

mrv_limit(ex,x,infinity);

1


(log(x+e^-x)+log(1/x))/(log(x)*e^x);

  - x       -1       -1           - x
e    *log(x)  *(log(x  ) + log(e     + x));

mrv_limit(ws,x,infinity);

0

mrv_limit((log(x)*e^-x)/e^(log(x)+e^(x^2)),x,infinity);

0
```

More examples can be found in the `mrvlimit.tst` file.

20.38.3 Tracing the algorihm

The package provides a means of tracing the `mrv_limit` function at its main steps, and is intended to help the user if he encounters problems. Messages are displayed informing the user which Taylor expansion is being computed, all recursive calls are listed, and the value returned by the `mrv` function is given. This information is displayed when a switch `trlimit` is on.

Note that, due to the recursiveness of the algorithm there is a lot of output when the `trlimit` switch is on.

20.39 NCPOLY: Non-commutative Polynomial Ideals

This package allows the user to set up automatically a consistent environment for computing in an algebra where the non–commutativity is defined by Lie-bracket commutators. The package uses the REDUCE **noncom** mechanism for elementary polynomial arithmetic; the commutator rules are automatically computed from the Lie brackets.

Authors: Herbert Melenk and Joachim Apel

20.39.1 Introduction

REDUCE supports a very general mechanism for computing with objects under a non–commutative multiplication, where commutator relations must be introduced explicitly by rule sets when needed. The package NCPOLY allows you to set up automatically a consistent environment for computing in an algebra where the non–commutativity is defined by Lie-bracket commutators. The package uses the REDUCE `noncom` mechanism for elementary polynomial arithmetic; the commutator rules are automatically computed from the Lie brackets. You can perform polynomial arithmetic directly, including *division* and *factorization*. Additionally NCPOLY supports computations in a one sided ideal (left or right), especially one sided Gröbner bases and *polynomial reduction*.

20.39.2 Setup, Cleanup

Before the computations can start the environment for a non–commutative computation must be defined by a call to `nc_setup`:

`nc_setup(`⟨*vars*⟩[,⟨*comms*⟩][,⟨*dir*⟩]);

where

⟨*vars*⟩ is a list of variables; these must include the non–commutative quantities.

⟨*comms*⟩ is a list of equations `<u>*<v> - <v>*<u>=<rh>` where $< u >$ and $< v >$ are members of $< vars >$, and $< rh >$ is a polynomial.

⟨*dir*⟩ is either $left$ or $right$ selecting a left or a right one sided ideal. The initial direction is $left$.

`nc_setup` generates from ⟨*comms*⟩ the necessary rules to support an algebra where all monomials are ordered corresponding to the given variable sequence. All pairs of variables which are not explicitly covered in the commutator set are considered as commutative and the corresponding rules are also activated.

The second parameter in `nc_setup` may be omitted if the operator is called for the second time, e.g. with a reordered variable sequence. In such a case the last commutator set is used again.

Remarks:

- The variables need not be declared `noncom` - `nc_setup` performs all necessary declarations.
- The variables need not be formal operator expressions; `nc_setup` encapsulates a variable `x` internally as `nc!*(!_x)` expressions anyway where the operator fnc!* keeps the noncom property.
- The commands `order` and `korder` should be avoided because `nc_setup` sets these such that the computation results are printed in the correct term order.

Example:

```
nc_setup({KK,NN,k,n},
            {NN*n-n*NN= NN, KK*k-k*KK= KK});

NN*n;              ->   nn*n
n*NN;              ->   nn*n - nn
nc_setup({k,n,KK,NN});
NN*n - NN          ->   n*nn;
```

Here KK, NN, k, n are non–commutative variables where the commutators are described as $[NN, n] = NN$, $[KK, k] = KK$.

The current term order must be compatible with the commutators: the product $< u > * < v >$ must precede all terms on the right hand side $< rh >$ under the current term order. Consequently

- the maximal degree of $< u >$ or $< v >$ in $< rh >$ is 1,
- in a total degree ordering the total degree of $< rh >$ may be not higher than 1,
- in an elimination degree order (e.g. lex) all variables in $< rh >$ must be below the minimum of $< u >$ and $< v >$.
- If $< rh >$ does not contain any variables or has at most $< u >$ or $< v >$, any term order can be selected.

If you want to use the non–commutative variables or results from non–commutative computations later in commutative operations it might be necessary to switch off the non–commutative evaluation mode because not all operators in REDUCE are prepared for that environment. In such a case use the command

```
nc_cleanup;
```

without parameters. It removes all internal rules and definitions which `nc_setup` had introduced. To reactive non–commutative call `nc_setup` again.

20.39.3 Left and right ideals

A (polynomial) left ideal L is defined by the axioms

$u \in L, v \in L \Longrightarrow u + v \in L$

$u \in L \Longrightarrow k * u \in L$ for an arbitrary polynomial k

where "*" is the non–commutative multiplication. Correspondingly, a right ideal R is defined by

$u \in R, v \in R \Longrightarrow u + v \in R$

$u \in R \Longrightarrow u * k \in R$ for an arbitrary polynomial k

20.39.4 Gröbner bases

When a non–commutative environment has been set up by `nc_setup`, a basis for a left or right polynomial ideal can be transformed into a Gröbner basis by the operator `nc_groebner`:

`nc_groebner(`⟨*plist*⟩`);`

Note that the variable set and variable sequence must be defined before in the `nc_setup` call. The term order for the Gröbner calculation can be set by using the `torder` declaration. The internal steps of the Gröbner calculation can be watched by setting the switches `trgroeb` (=list all internal basis polynomials) or `trgroebs` (=list additionally the S-polynomials) [38].

For details about `torder`, `trgroeb` and `trgroebs` see section 20.26.

```
2: nc_setup({k,n,NN,KK},{NN*n-n*NN=NN,KK*k-k*KK=KK},
            left);
```

[38] The command `lisp(!*trgroebfull:=t);` causes additionally all elementary polynomial operations to be printed.

```
3: p1 := (n-k+1)*NN - (n+1);

p1 :=  - k*nn + n*nn - n + nn - 1

4: p2 := (k+1)*KK -(n-k);

p2 := k*kk + k - n + kk

5: nc_groebner ({p1,p2});

{k*nn - n*nn + n - nn + 1,

 k*kk + k - n + kk,

 n*nn*kk - n*kk - n + nn*kk - kk - 1}
```

Important: Do not use the operators of the GROEBNER package directly as they would not consider the non–commutative multiplication.

20.39.5 Left or right polynomial division

The operator `nc_divide` computes the one sided quotient and remainder of two polynomials:

`nc_divide(`⟨*p1*⟩`,`⟨*p2*⟩`);`

The result is a list with quotient and remainder. The division is performed as a pseudo–division, multiplying $< p1 >$ by coefficients if necessary. The result $\{< q >, < r >\}$ is defined by the relation

$< c > * < p1 >=< q > * < p2 > + < r >$ for direction $left$ and

$< c > * < p1 >=< p2 > * < q > + < r >$ for direction $right$,

where $< c >$ is an expression that does not contain any of the ideal variables, and the leading term of $< r >$ is lower than the leading term of ⟨*p2*⟩ according to the actual term order.

20.39.6 Left or right polynomial reduction

For the computation of the one sided remainder of a polynomial modulo a given set of other polynomials the operator `nc_preduce` may be used:

`nc_preduce(`⟨*polynomial*⟩`,`⟨*plist*⟩`);`

The result of the reduction is unique (canonical) if and only if ⟨*plist*⟩ is a one sided Gröbner basis. Then the computation is at the same time an ideal membership test: if the result is zero, the polynomial is member of the ideal, otherwise not.

20.39.7 Factorization

20.39.7.1 Technique

Polynomials in a non–commutative ring cannot be factored using the ordinary `factorize` command of REDUCE. Instead one of the operators of this section must be used:

`nc_factorize(`⟨*polynomial*⟩`);`

The result is a list of factors of ⟨*polynomial*⟩. A list with the input expression is returned if it is irreducible.

As non–commutative factorization is not unique, there is an additional operator which computes all possible factorizations

`nc_factorize_all(`⟨*polynomial*⟩`);`

The result is a list of factor decompositions of ⟨*polynomial*⟩. If there are no factors at all the result list has only one member which is a list containing the input polynomial.

20.39.7.2 Control of the factorization

In contrast to factoring in commutative polynomial rings, the non–commutative factorization is rather time consuming. Therefore two additional operators allow you to reduce the amount of computing time when you look only for isolated factors in special context, e.g. factors with a limited degree or factors which contain only explicitly specified variables:

`left_factor`(⟨*polynomial*⟩[,⟨*deg*⟩[,⟨*vars*⟩]])
`right_factor`(⟨*polynomial*⟩[,⟨*deg*⟩[,⟨*vars*⟩]])
`left_factors`(⟨*polynomial*⟩[,⟨*deg*⟩[,⟨*vars*⟩]])
`right_factors`(⟨*polynomial*⟩[,⟨*deg*⟩[,⟨*vars*⟩]])

where ⟨*polynomial*⟩ is the form under investigation, ⟨*vars*⟩ is an optional list of variables which must appear in the factor, and ⟨*deg*⟩ is an optional integer degree bound for the total degree of the factor, a zero for an unbounded search, or a monomial (product of powers of the variables) where each exponent is an individual degree bound for its base variable; unmentioned variables are allowed in arbitrary degree. The operators `right_factor` and `left_factor` stop when they have found one factor, while the operators `right_factors` and `left_factors` select all one–sided factors within the given range. If there is no factor of the desired type, an empty list is returned by `right_factors` and `left_factors` while the routines `right_factor` and `left_factor` return the input polynomial.

20.39.7.3 Time of the factorization

The share variable `nc_factor_time` sets an upper limit for the time to be spent for a call to the non–commutative factorizer. If the value is a positive integer, a factorization is terminated with an error message as soon as the time limit is reached. The time units are milliseconds.

20.39.7.4 Usage of SOLVE

The factorizer internally uses `solve`, which is controlled by the REDUCE switch `varopt`. This switch (which per default is set on) allows to reorder the variable sequence, which is favourable for the normal system. It should be avoided to set `varopt` off when using the non–commutative factorizer, unless very small polynomials are used.

20.39.8 Output of expressions

It is often desirable to have the commutative parts (coefficients) in a non–commutative operation condensed by factorization. The operator

`nc_compact(`⟨*polynomial*⟩`)`

collects the coefficients to the powers of the lowest possible non-commutative variable.

```
load ncpoly;

nc_setup({n,NN},{NN*n-n*NN=NN})$
p1 := n**4 + n**2*nn + 4*n**2 + 4*n*nn + 4*nn + 4;

        4    2          2
p1 := n  + n *nn + 4*n  + 4*n*nn + 4*nn + 4

nc_compact p1;

  2     2           2
(n  + 2)  + (n + 2) *nn
```

20.40 NORMFORM: Computation of Matrix Normal Forms

This package contains routines for computing the following normal forms of matrices:

- smithex_int
- smithex
- frobenius
- ratjordan
- jordansymbolic
- jordan.

Author: Matt Rebbeck

20.40.1 Introduction

When are two given matrices similar? Similar matrices have the same trace, determinant, characteristic polynomial, and eigenvalues, but the matrices

$$\mathcal{U} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad \text{and} \quad \mathcal{V} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

are the same in all four of the above but are not similar. Otherwise there could exist a nonsingular $\mathcal{N} \in M_2$ (the set of all 2×2 matrices) such that $\mathcal{U} = \mathcal{N}\mathcal{V}\mathcal{N}^{-1} = \mathcal{N}\,0\,\mathcal{N}^{-1} = 0$, which is a contradiction since $\mathcal{U} \neq 0$.

Two matrices can look very different but still be similar. One approach to determining whether two given matrices are similar is to compute the normal form of them. If both matrices reduce to the same normal form they must be similar.

NORMFORM is a package for computing the following normal forms of matrices:

- smithex
- smithex_int
- frobenius
- ratjordan
- jordansymbolic

- jordan

By default all calculations are carried out in **Q** (the rational numbers). For `smithex`, `frobenius`, `ratjordan`, `jordansymbolic`, and `jordan`, this field can be extended. Details are given in the respective sections.

The `frobenius`, `ratjordan`, and `jordansymbolic` normal forms can also be computed in a modular base. Again, details are given in the respective sections.

The algorithms for each routine are contained in the source code.

NORMFORM has been converted from the normform and Normform packages written by T. M. L. Mulders and A. H. M. Levelt. These have been implemented in Maple [CGG+91].

20.40.2 Smith normal form

Function

`smithex`($\mathcal{A}$, x) computes the Smith normal form $\mathcal{S}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ where $\mathcal{S}, \mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{S}\mathcal{P}^{-1} = \mathcal{A}$.

$\mathcal{A}$ is a rectangular matrix of univariate polynomials in x.

x is the variable name.

Field extensions

Calculations are performed in **Q**. To extend this field the ARNUM package can be used. For details see subsection 20.40.8.

Synopsis:

- The Smith normal form $\mathcal{S}$ of an n by m matrix $\mathcal{A}$ with univariate polynomial entries in x over a field **F** is computed. That is, the polynomials are then regarded as elements of the *Euclidean* domain $\mathbf{F}(x)$.
- The Smith normal form is a diagonal matrix $\mathcal{S}$ where:
  - $\mathrm{rank}(\mathcal{A})$ = number of nonzero rows (columns) of $\mathcal{S}$.
  - $\mathcal{S}(i,i)$ is a monic polynomial for $0 < i \leq \mathrm{rank}(\mathcal{A})$.
  - $\mathcal{S}(i,i)$ divides $\mathcal{S}(i+1,i+1)$ for $0 < i < \mathrm{rank}(\mathcal{A})$.
  - $\mathcal{S}(i,i)$ is the greatest common divisor of all i by i minors of $\mathcal{A}$.

  Hence, if we have the case that $n = m$, as well as $\mathrm{rank}(\mathcal{A}) = n$, then

$$\prod_{i=1}^{n} \mathcal{S}(i,i) = \frac{\det(\mathcal{A})}{\mathrm{lcoeff}(\det(\mathcal{A}), x)}.$$

- The Smith normal form is obtained by doing elementary row and column operations. This includes interchanging rows (columns), multiplying through a row (column) by -1, and adding integral multiples of one row (column) to another.
- Although the rank and determinant can be easily obtained from $\mathcal{S}$, this is not an efficient method for computing these quantities except that this may yield a partial factorization of $\det(\mathcal{A})$ without doing any explicit factorizations.

Example:

$$\mathcal{A} = \begin{pmatrix} x & x+1 \\ 0 & 3*x^2 \end{pmatrix}$$

$$\texttt{smithex}(\mathcal{A}, x) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & x^3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 3*x^2 & 1 \end{pmatrix}, \begin{pmatrix} x & x+1 \\ -3 & -3 \end{pmatrix} \right\}$$

20.40.3 smithex_int

Function

Given an n by m rectangular matrix $\mathcal{A}$ that contains *only* integer entries, `smithex_int`($\mathcal{A}$) computes the Smith normal form $\mathcal{S}$ of $\mathcal{A}$.

It returns $\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ where $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{S}\mathcal{P}^{-1} = \mathcal{A}$.

Synopsis

- The Smith normal form $\mathcal{S}$ of an n by m matrix $\mathcal{A}$ with integer entries is computed.
- The Smith normal form is a diagonal matrix $\mathcal{S}$ where:
  - $\mathrm{rank}(\mathcal{A})$ = number of nonzero rows (columns) of $\mathcal{S}$.
  - $\mathrm{sign}(\mathcal{S}(i,i)) = 1$ for $0 < i \leq \mathrm{rank}(\mathcal{A})$.
  - $\mathcal{S}(i,i)$ divides $\mathcal{S}(i+1, i+1)$ for $0 < i < \mathrm{rank}(\mathcal{A})$.
  - $\mathcal{S}(i,i)$ is the greatest common divisor of all i by i minors of $\mathcal{A}$.

  Hence, if we have the case that $n = m$, as well as rank($\mathcal{A}$) $= n$, then

$$|\det(\mathcal{A})| = \prod_{i=1}^{n} \mathcal{S}(i,i).$$

- The Smith normal form is obtained by doing elementary row and column operations. This includes interchanging rows (columns), multiplying through a row (column) by -1, and adding integral multiples of one row (column) to another.

Example

$$\mathcal{A} = \begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

$$\texttt{smithex\_int}(\mathcal{A}) = \left\{ \begin{pmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}, \begin{pmatrix} -17 & -5 & -4 \\ 64 & 19 & 15 \\ -50 & -15 & -12 \end{pmatrix}, \begin{pmatrix} 1 & -24 & 30 \\ -1 & 25 & -30 \\ 0 & -1 & 1 \end{pmatrix} \right\}$$

20.40.4 frobenius

Function

`frobenius`$(\mathcal{A})$ computes the Frobenius normal form $\mathcal{F}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{F}, \mathcal{P}, \mathcal{P}^{-1}\}$ where $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{F}\mathcal{P}^{-1} = \mathcal{A}$.

$\mathcal{A}$ is a square matrix.

Field extensions

Calculations are performed in $\mathbf{Q}$. To extend this field the ARNUM package can be used. For details see subsection 20.40.8

Modular arithmetic

`frobenius` can be calculated in a modular base. For details see subsection 20.40.9.

Synopsis

- $\mathcal{F}$ has the following structure:

$$\mathcal{F} = \begin{pmatrix} \mathcal{C}p_1 & & & \\ & \mathcal{C}p_2 & & \\ & & \ddots & \\ & & & \mathcal{C}p_k \end{pmatrix}$$

  where the $\mathcal{C}(p_i)$'s are companion matrices associated with polynomials $p_1, p_2, \ldots, p_k$, with the property that p_i divides p_{i+1} for $i = 1 \ldots k-1$. All unmarked entries are zero.
- The Frobenius normal form defined in this way is unique (ie: if we require that p_i divides p_{i+1} as above).

Example

$$\mathcal{A} = \begin{pmatrix} \frac{-x^2+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \\ \frac{-x^2-x+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \end{pmatrix}$$

`frobenius`$(\mathcal{A}) =$

$$\left\{ \begin{pmatrix} 0 & \frac{x*(x^2-x-y^2+y)}{y} \\ 1 & \frac{-2*x^2+x+2*y^2}{y} \end{pmatrix}, \begin{pmatrix} 1 & \frac{-x^2+y^2+y}{y} \\ 0 & \frac{-x^2-x+y^2+y}{y} \end{pmatrix}, \begin{pmatrix} 1 & \frac{-x^2+y^2+y}{x^2+x-y^2-y} \\ 0 & \frac{-y}{x^2+x-y^2-y} \end{pmatrix} \right\}$$

20.40.5 ratjordan

Function

`ratjordan`$(\mathcal{A})$ computes the rational Jordan normal form $\mathcal{R}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{R}, \mathcal{P}, \mathcal{P}^{-1}\}$ where $\mathcal{R}$, $\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{R}\mathcal{P}^{-1} = \mathcal{A}$.

$\mathcal{A}$ is a square matrix.

Field extensions

Calculations are performed in $\mathbf{Q}$. To extend this field the ARNUM package can be used. For details see subsection 20.40.8.

Modular arithmetic

`ratjordan` can be calculated in a modular base. For details see subsection 20.40.9.

Synopsis

- $\mathcal{R}$ has the following structure:

$$\mathcal{R} = \begin{pmatrix} r_{11} & & & & & & \\ & r_{12} & & & & & \\ & & \ddots & & & & \\ & & & r_{21} & & & \\ & & & & r_{22} & & \\ & & & & & \ddots & \end{pmatrix}$$

The r_{ij}'s have the following shape:

$$r_{ij} = \begin{pmatrix} \mathcal{C}(p) & \mathcal{I} & & & & \\ & \mathcal{C}(p) & \mathcal{I} & & & \\ & & \ddots & \ddots & & \\ & & & \mathcal{C}(p) & \mathcal{I} & \\ & & & & \mathcal{C}(p) & \end{pmatrix}$$

where there are eij times $\mathcal{C}(p)$ blocks along the diagonal and $\mathcal{C}(p)$ is the companion matrix associated with the irreducible polynomial p. All unmarked entries are zero.

Example

$$\mathcal{A} = \begin{pmatrix} x+y & 5 \\ y & x^2 \end{pmatrix}$$

`ratjordan`$(\mathcal{A}) =$

$$\left\{ \begin{pmatrix} 0 & -x^3 - x^2 * y + 5 * y \\ 1 & x^2 + x + y \end{pmatrix}, \begin{pmatrix} 1 & x+y \\ 0 & y \end{pmatrix}, \begin{pmatrix} 1 & \frac{-(x+y)}{y} \\ 0 & \frac{1}{y} \end{pmatrix} \right\}$$

20.40.6 jordansymbolic

Function

`jordansymbolic`$(\mathcal{A})$ computes the Jordan normal form $\mathcal{J}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{J}, \mathcal{L}, \mathcal{P}, \mathcal{P}^{-1}\}$, where $\mathcal{J}$, $\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{J}\mathcal{P}^{-1} = \mathcal{A}$. $\mathcal{L} = \{ll, \xi\}$, where ξ is a name and ll is a list of irreducible factors of $p(\xi)$.

$\mathcal{A}$ is a square matrix.

Field extensions

Calculations are performed in $\mathbf{Q}$. To extend this field the ARNUM package can be used. For details see subsection 20.40.8.

Modular arithmetic

`jordansymbolic` can be calculated in a modular base. For details see subsection 20.40.9.

Synopsis

- A *Jordan block* $\jmath_k(\lambda)$ is a k by k upper triangular matrix of the form:

$$\jmath_k(\lambda) = \begin{pmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda & 1 \\ & & & & \lambda \end{pmatrix}$$

There are $k-1$ terms "+1" in the superdiagonal; the scalar λ appears k times on the main diagonal. All other matrix entries are zero, and $\jmath_1(\lambda) = (\lambda)$.

- A Jordan matrix $\mathcal{J} \in M_n$ (the set of all n by n matrices) is a direct sum of *jordan blocks*

$$\mathcal{J} = \begin{pmatrix} \jmath_{n_1}(\lambda_1) & & & \\ & \jmath_{n_2}(\lambda_2) & & \\ & & \ddots & \\ & & & \jmath_{n_k}(\lambda_k) \end{pmatrix}, n_1 + n_2 + \cdots + n_k = n$$

  in which the orders n_i may not be distinct and the values λ_i need not be distinct.

- Here λ is a zero of the characteristic polynomial p of $\mathcal{A}$. If p does not split completely, symbolic names are chosen for the missing zeroes of p. If, by some means, one knows such missing zeroes, they can be substituted for the symbolic names. For this, `jordansymbolic` actually returns $\{\mathcal{J}, \mathcal{L}, \mathcal{P}, \mathcal{P}^{-1}\}$. $\mathcal{J}$ is the Jordan normal form of $\mathcal{A}$ (using symbolic names if necessary). $\mathcal{L} = \{ll, \xi\}$, where ξ is a name and ll is a list of irreducible factors of $p(\xi)$. If symbolic names are used then ξ_{ij} is a zero of ll_i. $\mathcal{P}$ and $\mathcal{P}^{-1}$ are as above.

Example

$$\mathcal{A} = \begin{pmatrix} 1 & y \\ y^2 & 3 \end{pmatrix}$$

`jordansymbolic`$(\mathcal{A}) =$

$$\left\{ \begin{pmatrix} \xi_{11} & 0 \\ 0 & \xi_{12} \end{pmatrix}, \left\{ \left\{ -y^3 + \xi^2 - 4 * \xi + 3 \right\}, \xi \right\}, \right.$$

$$\left. \begin{pmatrix} \xi_{11} - 3 & \xi_{12} - 3 \\ y^2 & y^2 \end{pmatrix}, \begin{pmatrix} \frac{\xi_{11}-2}{2*(y^3-1)} & \frac{\xi_{11}+y^3-1}{2*y^2*(y^3+1)} \\ \frac{\xi_{12}-2}{2*(y^3-1)} & \frac{\xi_{12}+y^3-1}{2*y^2*(y^3+1)} \end{pmatrix} \right\}$$

```
solve(-y^3+xi^2-4*xi+3,xi);
```

$$\{\xi = \sqrt{y^3+1} + 2, \xi = -\sqrt{y^3+1} + 2\}$$

```
J = sub({xi(1,1)=sqrt(y^3+1)+2,
         xi(1,2)=-sqrt(y^3+1)+2},
        first jordansymbolic (A))
```

$$\mathcal{J} = \begin{pmatrix} \sqrt{y^3+1} + 2 & 0 \\ 0 & -\sqrt{y^3+1} + 2 \end{pmatrix}$$

20.40.7 jordan

Function

`jordan`($\mathcal{A}$) computes the Jordan normal form $\mathcal{J}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{J}, \mathcal{P}, \mathcal{P}^{-1}\}$, where $\mathcal{J}$, $\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{J}\mathcal{P}^{-1} = \mathcal{A}$.

$\mathcal{A}$ is a square matrix.

Field extensions

Calculations are performed in **Q**. To extend this field the ARNUM package can be used. For details see subsection 20.40.8.

Note

In certain polynomial cases the switch `fullroots` is turned on to compute the zeroes. This can lead to the calculation taking a long time, as well as the output being very large. In this case a message

```
*****
WARNING: fullroots turned on.  May take a while.
```

will be printed. It may be better to kill the calculation and compute `jordansymbolic` instead.

Synopsis

- The Jordan normal form $\mathcal{J}$ with entries in an algebraic extension of **Q** is computed.
- A *Jordan block* $\jmath_k(\lambda)$ is a k by k upper triangular matrix of the form:

$$\jmath_k(\lambda) = \begin{pmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda & 1 \\ & & & & \lambda \end{pmatrix}$$

  There are $k-1$ terms "+1" in the superdiagonal; the scalar λ appears k times on the main diagonal. All other matrix entries are zero, and $\jmath_1(\lambda) = (\lambda)$.
- A Jordan matrix $\mathcal{J} \in M_n$ (the set of all n by n matrices) is a direct sum of *jordan blocks*.

$$\mathcal{J} = \begin{pmatrix} \jmath_{n_1}(\lambda_1) & & & \\ & \jmath_{n_2}(\lambda_2) & & \\ & & \ddots & \\ & & & \jmath_{n_k}(\lambda_k) \end{pmatrix}, n_1 + n_2 + \cdots + n_k = n$$

  in which the orders n_i may not be distinct and the values λ_i need not be distinct.

- Here λ is a zero of the characteristic polynomial p of $\mathcal{A}$. The zeroes of the characteristic polynomial are computed exactly, if possible. Otherwise they are approximated by floating point numbers.

Example

$$\mathcal{A} = \begin{pmatrix} -9 & -21 & -15 & 4 & 2 & 0 \\ -10 & 21 & -14 & 4 & 2 & 0 \\ -8 & 16 & -11 & 4 & 2 & 0 \\ -6 & 12 & -9 & 3 & 3 & 0 \\ -4 & 8 & -6 & 0 & 5 & 0 \\ -2 & 4 & -3 & 0 & 1 & 3 \end{pmatrix}$$

```
J = first jordan(A);
```

$$\mathcal{J} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & i+2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i+2 \end{pmatrix}$$

20.40.8 Algebraic extensions: Using the ARNUM package

The algebraic field $\mathbf{Q}$ can now be extended. E.g., `defpoly sqrt2**2-2;` will extend it to include $\sqrt{2}$ (defined here by `sqrt2`). The ARNUM package was written by Eberhard Schrüfer and is described in section 9.12.5.

20.40.8.1 Example

```
defpoly sqrt2**2-2;
```
(sqrt2 now changed to $\sqrt{2}$ for looks!)

$$\mathcal{A} = \begin{pmatrix} 4*\sqrt{2}-6 & -4*\sqrt{2}+7 & -3*\sqrt{2}+6 \\ 3*\sqrt{2}-6 & -3*\sqrt{2}+7 & -3*\sqrt{2}+6 \\ 3*\sqrt{2} & 1-3*\sqrt{2} & -2*\sqrt{2} \end{pmatrix}$$

$$\texttt{ratjordan}(\mathcal{A}) = \left\{ \begin{pmatrix} \sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & -3*\sqrt{2}+1 \end{pmatrix}, \right.$$

$$\left. \begin{pmatrix} 7*\sqrt{2}-6 & \frac{2*\sqrt{2}-49}{31} & \frac{-21*\sqrt{2}+18}{31} \\ 3*\sqrt{2}-6 & \frac{21*\sqrt{2}-18}{31} & \frac{-21*\sqrt{2}+18}{31} \\ 3*\sqrt{2}+1 & \frac{-3*\sqrt{2}+24}{31} & \frac{3*\sqrt{2}-24}{31} \end{pmatrix} \right\},$$

$$\left.\begin{pmatrix} 0 & \sqrt{2}+1 & 1 \\ -1 & 4*\sqrt{2}+9 & 4*\sqrt{2} \\ -1 & -\frac{1}{6}*\sqrt{2}+1 & 1 \end{pmatrix}\right\}$$

20.40.9 Modular arithmetic

Calculations can be performed in a modular base by setting the switch `modular` to on. The base can then be set by `setmod p;` (p a prime). The normal form will then have entries in $\mathbf{Z}/\mathrm{p}\mathbf{Z}$.

By also switching on `balanced_mod` the output will be shown using a symmetric modular representation.

Information on this modular manipulation can be found in section 9.12.3.

20.40.9.1 Example

```
on modular;
setmod 23;
```

$$\mathcal{A} = \begin{pmatrix} 10 & 18 \\ 17 & 20 \end{pmatrix}$$

`jordansymbolic`$(\mathcal{A}) =$

$$\left\{\begin{pmatrix} 18 & 0 \\ 0 & 12 \end{pmatrix}, \{\{\lambda+5, \lambda+11\}, \lambda\}, \begin{pmatrix} 15 & 9 \\ 22 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 14 \\ 1 & 15 \end{pmatrix}\right\}$$

```
on balanced_mod;
```

`jordansymbolic`$(\mathcal{A}) =$

$$\left\{\begin{pmatrix} -5 & 0 \\ 0 & -11 \end{pmatrix}, \{\{\lambda+5, \lambda+11\}, \lambda\}, \begin{pmatrix} -8 & 9 \\ -1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & -9 \\ 1 & -8 \end{pmatrix}\right\}$$

20.41 ODESOLVE: Ordinary Differential Equation Solver

ODESOLVE is a solver for ordinary differential equations. It uses only elementary solution techniques. At present, it can handle only a single scalar equation presented as an algebraic expression or equation, and it can solve first-order equations of simple types, linear equations with constant coefficients, Euler equations, and some more complicated types. For example, the evaluation of

```
depend(y, x);
odesolve(df(y,x) = x^2 + e^x, y, x);
```

yields the result

```
                              x     3
    3*arbconst(1) + 3*e   + x
{y=---------------------------}
                  3
```

Main authors: Malcolm MacCallum and Francis Wright

Other contributor: Alan Barnes

20.41.1 Introduction

ODESOLVE [Mac88, Wri97, Wri99] was developed partly under the auspices of the European CATHODE project [CAT]. Various test files that illustrate the capabilities of ODESOLVE, including three versions (with names beginning with `zim`) based on a published review of ODE (ordinary differential equation) solvers [PZ96], are included in the source code distribution in the directory `packages/odesolve`, which you can access online at https://sourceforge.net/p/reduce-algebra/code/HEAD/tree/trunk/packages/odesolve/.

ODESOLVE implements most of the simple and well known solution techniques [Zwi92]. It also provides an extension interface (see §20.41.5), which could be used to support more sophisticated solvers, such as PSODE [Man94, MM97, PS83] and CRACK [BW92], to handle cases where simple techniques fail, although none of these extensions is implemented yet.

The main motivation behind ODESOLVE is pragmatic. It is intended to meet user expectations, to have an easy user-interface that normally does the right thing automatically, and to return solutions in the form that the user wants and expects.

The ODESOLVE package autoloads the first time the normal algebraic-mode

odesolve operator is used.

20.41.2 User interface

The principal interface is via the operator odesolve. (It also has a synonym called dsolve to make porting of examples from Maple easier, but it does not accept general Maple syntax! And if solve is applied to a manifest ODE then it will call odesolve.)

For purposes of description let us refer to the dependent variable as "y" and the independent variable as "x", but of course the names are arbitrary. The general input syntax is

```
odesolve(ode, y, x, conditions, options);
```

All arguments except the first are optional. This is possible because, if necessary, ODESOLVE attempts to deduce the dependent and independent variables used and to make any necessary DEPEND declarations. Messages are output to indicate any assumptions or dependence declarations that are made. Here is an example of what is probably the shortest possible valid input:

```
odesolve(df(y,x));

*** Dependent var(s) assumed to be y

*** Independent var assumed to be x

*** depend y , x

{y=arbconst(1)}
```

Output of ODESOLVE messages is controlled by the standard REDUCE switch msg.

20.41.2.1 Specifying the ODE and its variables

The first argument (ode) is *required*, and must be either an ODE or a variable (or expression) that evaluates to an ODE. Automatic dependence declaration works *only* when the ODE is input *directly* as an argument to the odesolve operator. Here, "ODE" means an equation or expression containing one or more derivatives of y with respect to x. Derivatives of y with respect to other variables are not allowed because ODESOLVE does not solve *partial* differential equations, and symbolic derivatives of variables other than y are treated as symbolic constants.

An expression is implicitly equated to zero, as is usual in equation solvers.

The independent variable may be either an operator that explicitly depends on the independent variable, e.g. $y(x)$ (as required in Maple), or a simple variable that is declared (by the user or automatically by ODESOLVE) to depend on the independent variable. If the independent variable is an operator then it may depend on parameters as well as the independent variable. Variables may be simple identifiers or, more generally, REDUCE "kernels", e.g.

```
operator x, y;
odesolve(df(y(x(a),b),x(a)) = 0);

*** Dependent var(s) assumed to be y(x(a),b)

*** Independent var assumed to be x(a)

{y(x(a),b)=arbconst(1)}
```

The order in which arguments are given must be preserved, but arguments may be omitted, except that if x is specified then y must also be specified, although an empty list `{}` can be used as a "place-holder" to represent "no specified argument". Variables are distinguished from options by requiring that if a variable is specified then it must appear in the ODE, otherwise it is assumed to be an option.

Generally in REDUCE it is not recommended to use the identifier `t` as a variable, since it is reserved in Lisp. However, it is very common practice in applied mathematics to use it as a variable to represent time, and for that reason ODESOLVE provides special support to allow it as either the independent or a dependent variable. But, of course, its use may still cause trouble in other parts of REDUCE!

20.41.2.2 Specifying conditions

If specified, the "conditions" argument must take the form of an (unordered) list of (unordered lists of) equations with either y, x, or a derivative of y on the left. A single list of conditions need not be contained within an outer list. Combinations of conditions are allowed. Conditions within one (inner) list all relate to the same x value. For example:

Boundary conditions:
{{y=y0, x=x0}, {y=y1, x=x1}, ...}

Initial conditions:
{x=x0, y=y0, df(y,x)=dy0, ...}

Combined conditions:

{{y=y0, x=x0}, {df(y,x)=dy1, x=x1}, {df(y,x)=dy2, y=y2, x=x2}, ...}

Here is an example of boundary conditions:

```
odesolve(df(y,x,2) = y, y, x,
         {{x = 0, y = A}, {x = 1, y = B}});

         2*x       2*x           2
      - e    *a + e    *b*e + a*e  - b*e
{y=-----------------------------------}
                 x  2    x
                e *e  - e
```

Here is an example of initial conditions:

```
odesolve(df(y,x,2) = y, y, x,
         {x = 0, y = A, df(y,x) = B});

     2*x       2*x
    e    *a + e    *b + a - b
{y=-------------------------}
                 x
              2*e
```

Here is an example of combined conditions:

```
odesolve(df(y,x,2) = y, y, x,
         {{x=0, y=A}, {x=1, df(y,x)=B}});

     2*x       2*x           2
    e    *a + e    *b*e + a*e  - b*e
{y=-------------------------------}
                x  2    x
               e *e  + e
```

Boundary conditions on the values of y at various values of x may also be specified by replacing the variables by equations with single values or matching lists of values on the right, of the form

y = y0, x = x0

or

y = {y0, y1, ...}, x = {x0, x2, ...}

For example

```
odesolve(df(y,x) = y, y = A, x = 0);

    x
{y=e *a}

odesolve(df(y,x,2) = y, y = {A, B}, x = {0, 1});

         2*x       2*x           2
      - e    *a + e    *b*e + a*e  - b*e
{y=-----------------------------------}
                 x  2    x
                e *e  - e
```

20.41.2.3 Specifying options and defaults

The final arguments may be one or more of the option identifiers listed in the table below, which take precedence over the default settings. All options can also be specified on the right of equations with the identifier "output" on the left, e.g. "output = basis". This facility if provided mainly for compatibility with other systems such as Maple, although it also allows options to be distinguished from variables in case of ambiguity. Some options can be specified on the left of equations that assign special values to the option. Currently, only "trode" and its synonyms can be assigned the value 1 to give an increased level of tracing.

The following switches set default options – they are all off by default. Options set locally using option arguments override the defaults set by switches.

Switch	Option	Effect on solution
odesolve_explicit	explicit	fully explicit
odesolve_expand	expand	expand roots of unity
odesolve_full	full	fully explicit and expanded
odesolve_implicit	implicit	implicit instead of parametric
	algint	turn on algint
odesolve_noint	noint	turn off selected integrations
odesolve_verbose	verbose	display ODE and conditions
odesolve_basis	basis	output basis solution for linear ODE
trode	trode, trace, tracing	turn on algorithm tracing
odesolve_fast	fast	turn off heuristics
odesolve_check	check	turn on solution checking

An "explicit" solution is an equation with y isolated on the left whereas an "implicit" solution is an equation that determines y as one or more of its solutions. A "parametric" solution expresses both x and y in terms of some additional parameter. Some solution techniques naturally produce an explicit solution, but some produce either an implicit or a parametric solution. The "explicit" option causes ODESOLVE to attempt to convert solutions to explicit form, whereas the "implicit" option causes ODESOLVE to attempt to convert parametric solutions (only) to implicit form (by eliminating the parameter). These solution conversions may be slow or may fail in complicated cases.

ODESOLVE introduces two operators used in solutions: `root_of_unity` and `plus_or_minus`, the latter being a special case of the former, i.e. a second root of unity. These operators carry a tag that associates the same root of unity when it appears in more than one place in a solution (cf. the standard `root_of` operator). The "expand" option expands a single solution expressed in terms of these operators into a set of solutions that do not involve them. ODESOLVE also introduces two operators `expand_roots_of_unity` and `expand_plus_or_minus`, that are used internally to perform the expansion described above, and can be used explicitly.

The "algint" option turns on "algebraic integration" locally only within ODESOLVE. It also loads the `algint` package if necessary. Algint allows ODESOLVE to solve some ODEs for which the standard REDUCE integrator hangs (i.e. takes an extremely long time to return). If the resulting solution contains unevaluated integrals then the algint switch should be turned on outside ODESOLVE before the solution is re-evaluated, otherwise the standard integrator may well hang again! For some ODEs, the algint option leads to better solutions than the standard REDUCE integrator.

Alternatively, the "noint" option prevents REDUCE from attempting to evaluate the integrals that arise in some solution techniques. If ODESOLVE takes too long to return a result then you might try adding this option to see if it helps solve this particular ODE, as illustrated in the test files. This option is provided to speed up the computation of solutions that contain integrals that cannot be evaluated, because in some cases REDUCE can spend a long time trying to evaluate such integrals before returning them unevaluated. This only affects integrals evaluated *within* the `odesolve` operator. If a solution containing an unevaluated integral that was returned using the "noint" option is re-evaluated, it may again take REDUCE a very long time to fail to evaluate the integral, so considerable caution is recommended! (A global switch called "noint" is also installed when ODESOLVE is loaded, and can be turned on to prevent REDUCE from attempting to evaluate *any* integrals. But this effect may be very confusing, so this switch should be used only with extreme care. If you turn it on and then forget, you may wonder why REDUCE seems unable to evaluate even trivial integrals!)

The "verbose" option causes ODESOLVE to display the ODE, variables and con-

ditions as it sees them internally, after pre-processing. This is intended for use in demonstrations and possibly for debugging, and not really for general users.

The "basis" option causes ODESOLVE to output the general solutions of linear ODEs in basis format (explained below). Special solutions (of ODEs with conditions) and solutions of nonlinear ODEs are not affected.

The "trode" (or "trace" or "tracing") option turns on tracing of the algorithms used by ODESOLVE. It reports its classification of the ODE and any intermediate results that it computes, such as a chain of progressively simpler (in some sense) ODEs that finally leads to a solution. Tracing can produce a lot of output, e.g. see the test log file "`zimmer.rlg`". The option "`trode = 1`" or the global assignment "`!*trode := 1`" causes ODESOLVE to report every test that it tries in its classification process, producing even more tracing output. This is probably most useful for debugging, but it may give the curious user further insight into the operation of ODESOLVE.

The "fast" option disables all non-deterministic solution techniques (including most of those for nonlinear ODEs of order > 1). It may be most useful if ODESOLVE is used as a subroutine, including calling it recursively in a hook. It makes ODESOLVE behave like the version distributed with REDUCE 3.7, and so does not affect the `odesolve.tst` file. The "fast" option causes ODESOLVE to return no solution fast in cases where, by default, if would return either a solution or no solution more slowly (perhaps much more slowly). Solution of sufficiently simple "deterministically-solvable" ODEs is unaffected.

The "check" option turns on checking of the solution. This checking is performed by code that is largely independent of the solver, so as to perform a genuinely independent check. It is not turned on by default so as to avoid the computational overhead, which is currently of the order of 30%. A check is made that each component solution satisfies the ODE and that a general solution contains at least enough arbitrary constants, or equivalently that a basis solution contains enough basis functions. Otherwise, warning messages are output. It is possible that ODESOLVE may fail to verify a solution because the automatic simplification fails, which indicates a failure in the checker rather than in the solver.

In some cases, in particular in symbolic solutions of Clairaut ODEs, the checker may need to differentiate a composition of operators using the chain rule. In order to do this, it turns on the differentiator switch `expanddf` locally only.

20.41.3 Output syntax

If ODESOLVE is successful it outputs a list of sub-solutions that together represent the solution of the input ODE. Each sub-solution is either an equation that defines a branch of the solution, explicitly or implicitly, or it is a list of equations that define a branch of the solution parametrically in the form $\{y = G(p), x = F(p), p\}$. Here

p is the parameter, which is actually represented in terms of an operator called `arbparam` which has an integer argument to distinguish it from other unrelated parameters, as usual for arbitrary values in REDUCE.

A general solution will contain a number of arbitrary constants represented by an operator called `arbconst` with an integer argument to distinguish it from other unrelated arbitrary constants. A special solution resulting from applying conditions will contain fewer (usually no) arbitrary constants.

The general solution of a linear ODE in basis format is a list consisting of a list of basis functions for the solution space of the reduced ODE followed by a particular solution if the input ODE had a y-independent "driver" term, i.e. was not reduced (which is sometimes ambiguously called "homogeneous"). The particular solution is normally omitted if it is zero. The dependent variable y does not appear in a basis solution. The linear solver uses basis solutions internally.

Currently, there are cases where ODESOLVE cannot solve a linear ODE using its linear solution techniques, in which case it will try nonlinear techniques. These may generate a solution that is not (obviously) a linear combination of basis solutions. In this case, if a basis solution has been requested, ODESOLVE will report that it cannot separate the nonlinear combination, which it will return as the default linear combination solution.

If ODESOLVE fails to solve the ODE then it will return a list containing the input ODE (always in the form of a differential expression equated to 0). At present, ODESOLVE does not return partial solutions. If it fails to solve any part of the problem then it regards this as complete failure. (You can probably see if this has happened by turning on algorithm tracing.)

20.41.4 Solution techniques

The ODESOLVE interface module pre-processes the problem and applies any conditions to the solution. The other modules deal with the actual solution.

ODESOLVE first classifies the input ODE according to whether it is linear or nonlinear and calls the appropriate solver. An ODE that consists of a product of linear factors is regarded as nonlinear. The second main classification is based on whether the input ODE is of first or higher degree.

Solution proceeds essentially by trying to reduce nonlinear ODEs to linear ones, and to reduce higher order ODEs to first order ODEs. Only simple linear ODEs and simple first-order nonlinear ODEs can be solved directly. This approach involves considerable recursion within ODESOLVE.

If all solution techniques fail then ODESOLVE attempts to factorize the derivative of the whole ODE, which sometimes leads to a solution.

20.41.4.1 Linear solution techniques

ODESOLVE splits every linear ODE into a "reduced ODE" and a "driver" term. The driver is the component of the ODE that is independent of y, the reduced ODE is the component of the ODE that depends on y, and the sign convention is such that the ODE can be written in the form "reduced ODE = driver". The reduced ODE is then split into a list of "ODE coefficients".

The linear solver now determines the order of the ODE. If it is 1 then the ODE is immediately solved using an integrating factor (if necessary). For a higher order linear ODE, ODESOLVE considers a sequence of progressively more complicated solution techniques. For most purposes, the ODE is made "monic" by dividing through by the coefficient of the highest order derivative. This puts the ODE into a standard form and effectively deals with arbitrary overall algebraic factors that would otherwise confuse the solution process. (Hence, there is no need to perform explicit algebraic factorization on linear ODEs.) The only situation in which the original non-monic form of the ODE is considered is when checking for exactness, which may depend critically on otherwise irrelevant overall factors.

If the ODE has constant coefficients then it can (in principle) be solved using elementary "D-operator" techniques in terms of exponentials via an auxiliary equation. However, this works only if the polynomial auxiliary equation can be solved. Assuming that it can and there is a driver term, ODESOLVE tries to use a method based on inverse "D-operator" techniques that involves repeated integration of products of the solutions of the reduced ODE with the driver. Experience (by Malcolm MacCallum) suggests that this normally gives the most satisfactory form of solution if the integrals can be evaluated. If any integral fails to evaluate, the more general method of "variation of parameters", based on the Wronskian of the solution set of the reduced ODE, is used instead. This involves only a single integral and so can never lead to nested unevaluated integrals.

If the ODE has non-constant coefficients then it may be of Euler (sometimes ambiguously called "homogeneous") type, which can be trivially reduced to an ODE with constant coefficients. A shift in x is accommodated in this process. Next it is tested for exactness, which leads to a first integral that is an ODE of order one lower. After that it is tested for the explicit absence of y and low order derivatives, which allows trivial order reduction. Then the monic ODE is tested for exactness, and if that fails and the original ODE was non-monic then the original form is tested for exactness.

Finally, pattern matching is used to seek a solution involving special functions, such as Bessel functions. Currently, this is implemented only for second-order ODEs satisfied by Bessel and Airy-integral functions. It could easily be extended to other orders and other special functions. Shifts in x could also be accommodated in the pattern matching.

If all linear techniques fail then ODESOLVE currently calls the variable interchange routine (described below), which takes it into the nonlinear solver. Occasionally, this is successful in producing some, although not necessarily the best, solution of a linear ODE.

20.41.4.2 Nonlinear solution techniques

In order to handle trivial nonlinearity, ODESOLVE first factorizes the ODE algebraically, solves each factor that depends on y and then merges the resulting solutions. Other factors are ignored, but a warning is output unless they are purely numerical.

If all attempts at solution fail then ODESOLVE checks whether the original (unfactored) ODE was exact, because factorization could destroy exactness. Currently, ODESOLVE handles only first and second order nonlinear exact ODEs.

A version of the main solver applied to each algebraic factor branches depending on whether the ODE factor is linear or nonlinear, and the nonlinear solver branches depending on whether the order is 1 or higher and calls one of the solvers described in the next two sections. If that solver fails, ODESOLVE checks for exactness (of the factor). If that fails, it checks whether only a single order derivative is involved and tries to solve algebraically for that. If successful, this decomposes the ODE into components that are, in some sense, simpler and may be solvable. (However, in some cases these components are algebraically very complicated examples of simple types of ODE that the integrator cannot in practice handle, and it can take a very long time before returning an unevaluated integral.)

If all else fails, ODESOLVE interchanges the dependent and independent variables and calls the top-level solver recursively. It keeps a list of all ODEs that have entered the top-level solver in order to break infinite loops that could arise if the solution of the variable-interchanged ODE fails.

First-order nonlinear solution techniques

If the ODE is a first-degree polynomial in the derivative then ODESOLVE represents it in terms of the "gradient", which is a function of x and y such that the ODE can be written as "$dy/dx = \mathit{gradient}$". It then checks *in sequence* for the following special types of ODE, each of which it can (in principle) solve:

Separable The gradient has the form $f(x)g(y)$, leading immediately to a solution by quadrature, i.e. the solution can be immediately written in terms of indefinite integrals. (This is considered to be a solution of the ODE, regardless of whether the integrals can be evaluated.) The solver recognises both explicit and implicit dependence when detecting separable form.

Quasi-separable The gradient has the form $f(y + kx)$, which is (trivially) separable after a linear transformation. It arises as a special case of the "quasi-homogeneous" case below, but is better treated earlier as a case in its own right.

Homogeneous The gradient has the form $f(y/x)$, which is algebraically homogeneous. A substitution of the form "$y = vx$" leads to a first-order linear ODE that is (in principle) immediately solvable.

Quasi-homogeneous The gradient has the form $f\left(\frac{a_1x+b_1y+c_1}{a_2x+b_2y+c_2}\right)$, which is homogeneous after a linear transformation.

Bernoulli The gradient has the form $P(x)y + Q(x)y^n$, in which case the ODE is a first-order linear ODE for y^{1-n}.

Riccati The gradient has the form $a(x)y^2 + b(x)y + c(x)$, in which case the ODE can be transformed into a *linear* second-order ODE that may be solvable.

If the ODE is not first-degree then it may be linear in either x or y. Solving by taking advantage of this leads to a parametric solution of the original ODE, in which the parameter corresponds to y'. It may then be possible to eliminate the parameter to give either an implicit or explicit solution.

An ODE is "solvable for y" if it can be put into the form $y = f(x, y')$. Differentiating with respect to x leads to a first-order ODE for $y'(x)$, which may be easier to solve than the original ODE. The special case that $y = xF(y') + G(y')$ is called a Lagrange (or d'Alembert) ODE. Differentiating with respect to x leads to a first-order linear ODE for $x(y')$. The even more special case that $y = xy' + G(y')$, which may arise in the equivalent implicit form $F(xy' - y) = G(y')$, is called a Clairaut ODE. The general solution is given by replacing y' by an arbitrary constant, and it may be possible to obtain a singular solution by differentiating and solving the resulting factors simultaneously with the original ODE.

An ODE is "solvable for x" if it can be put into the form $x = f(y, y')$. Differentiating with respect to y leads to a first-order ODE for $y'(y)$, which may be easier to solve than the original ODE.

Currently, ODESOLVE recognises the above forms only if the ODE manifestly has the specified form and does not try very hard to actually solve for x or y, which perhaps it should!

Higher-order nonlinear solution techniques

The techniques used here are all special cases of Lie symmetry analysis, which is not yet applied in any general way.

Higher-order nonlinear ODEs are passed through a number of "simplifier" filters that are applied in succession, regardless of whether the previous filter simplifies the ODE or not. Currently, the first filter tests for the explicit absence of y and low order derivatives, which allows trivial order reduction. The second filter tests whether the ODE manifestly depends on $x + k$ for some constant k, in which case it shifts x to remove k.

After that, ODESOLVE tests for each of the following special forms in sequence. The sequence used here is important, because the classification is not unique, so it is important to try the most useful classification first.

Autonomous An ODE is autonomous if it does not depend explicitly on x, in which case it can be reduced to an ODE in y' of order one lower.

Scale invariant or equidimensional in x An ODE is scale invariant if it is invariant under the transformation $x \to ax, y \to a^p y$, where a is an arbitrary indeterminate and p is a constant to be determined. It can be reduced to an autonomous ODE, and thence to an ODE of order one lower. The special case $p = 0$ is called equidimensional in x. It is the nonlinear generalization of the (reduced) linear Euler ODE.

Equidimensional in y An ODE is equidimensional in y if it is invariant under the transformation $y \to ay$. An exponential transformation of y leads to an ODE of the same order that *may* be "more linear" and so easier to solve, but there is no guarantee of this. All (reduced) linear ODEs are trivially equidimensional in y.

The recursive nature of ODESOLVE, especially the thread described in this section, can lead to complicated "arbitrary constant expressions". Arbitrary constants must be included at the point where an ODE is solved by quadrature. Further processing of such a solution, as may happen when a recursive solution stack is unwound, can lead to arbitrary constant expressions that should be re-written as simple arbitrary constants. There is some simple code included to perform this arbitrary constant simplification, but it is rudimentary and not entirely successful.

20.41.5 Extension interface

The idea is that the ODESolve extension interface allows any user to add solution techniques without needing to edit and recompile the main source code, and (in principle) without needing to be intimately familiar with the internal operation of ODESOLVE.

The extension interface consists of a number of "hooks" at various critical places within ODESOLVE. These hooks are modelled in part on the hook mechanism used to extend and customize the Emacs editor, which is a large Lisp-based system

with a structure similar to that of REDUCE. Each ODESOLVE hook is an identifier which can be defined to be a function (i.e. a procedure), or have assigned to it (in symbolic mode) a function name or a (symbolic mode) list of function names. The function should be written to accept the arguments specified for the particular hook, and it should return either a solution to the specified class of ODE in the specified form or nil.

If a hook returns a non-nil value then that value is used by ODESOLVE as the solution of the ODE at that stage of the solution process. (If the ODE being solved was generated internally by ODESOLVE or conditions are imposed then the solution will be re-processed before being finally returned by ODESOLVE.) If a hook returns nil then it is ignored and ODESOLVE proceeds as if the hook function had not been called at all. This is the same mechanism that it used internally by ODESOLVE to run sub-solvers. If a hook evaluates to a list of function names then they are applied in turn to the hook arguments until a non-nil value is returned and this is the value of the hook; otherwise the hook returns nil. The same code is used to run all hooks and it checks that an identifier is the name of a function before it tries to apply it; otherwise the identifier is ignored. However, the hook code does not perform any other checks, so errors within functions run by hooks will probably terminate ODESOLVE and errors in the return value will probably cause fatal errors later in ODESOLVE. Such errors are user errors rather than ODESOLVE errors!

Hooks are defined in pairs which are inserted before and after critical stages of the solver, which currently means the general ODE solver, the nonlinear ODE solver, and the solver for linear ODEs of order greater than one (on the grounds that solving first order linear ODEs is trivial and the standard ODESOLVE code should always suffice). The precise interface definition is as follows.

A reference to an "algebraic expression" implies that the REDUCE representation is a prefix or pseudo-prefix form. A reference to a "variable" means an identifier (and never a more general kernel). The "order" of an ODE is always an explicit positive integer. The return value of a hook function must always be either nil or an algebraic-mode list (which must be represented as a prefix form). Since the input and output of hook functions uses prefix forms (and never standard quotient forms), hook functions can equally well be written in either algebraic or symbolic mode, and in fact ODESOLVE uses a mixture internally. (An algebraic-mode procedure can return nil by returning nothing. The integer zero is *not* equivalent to nil in the context of ODESOLVE hooks.)

Hook names: `ODESolve_Before_Hook`, `ODESolve_After_Hook`.

Run before and after: The general ODE solver.

Arguments: 3

1. The ODE in the form of an algebraic expression with no denominator that must be made identically zero by the solution.
2. The dependent variable.
3. The independent variable.

Return value: A list of equations exactly as returned by ODESOLVE itself.

Hook names: `ODESolve_Before_Non_Hook`, `ODESolve_After_Non_Hook`.

Run before and after: The nonlinear ODE solver.

Arguments: 4

1. The ODE in the form of an algebraic expression with no denominator that must be made identically zero by the solution.
2. The dependent variable.
3. The independent variable.
4. The order of the ODE.

Return value: A list of equations exactly as returned by ODESOLVE itself.

Hook names: `ODESolve_Before_Lin_Hook`, `ODESolve_After_Lin_Hook`.

Run before and after: The general linear ODE solver.

Arguments: 6

1. A list of the coefficient functions of the "reduced ODE", i.e. the coefficients of the different orders (including zero) of derivatives of the dependent variable, each in the form of an algebraic expression, in low to high derivative order. (In general the ODE will not be "monic" so the leading (i.e. last) coefficient function will not be 1. Hence, the ODE may contain an essentially irrelevant overall algebraic factor.)
2. The "driver" term, i.e. the term involving only the independent variable, in the form of an algebraic expression. The sign convention is such that "reduced ODE = driver".
3. The dependent variable.

4. The independent variable.
5. The (maximum) order (> 1) of the ODE.
6. The minimum order derivative present.

Return value: A list consisting of a basis for the solution space of the reduced ODE and optionally a particular integral of the full ODE. This list does not contain any equations, and the dependent variable never appears in it. The particular integral may be omitted if it is zero. The basis is itself a list of algebraic expressions in the independent variable. (Hence the return value is always a list and its first element is also always a list.)

Hook names: `ODESolve_Before_Non1Grad_Hook`, `ODESolve_After_Non1Grad_Hook`.

Run before and after: The solver for first-order first-degree nonlinear ("gradient") ODEs, which can be expressed in the form $dy/dx = \mathrm{gradient}(y, x)$.

Arguments: 3

1. The "gradient", which is an algebraic expression involving (in general) the dependent and independent variables, to which the ODE equates the derivative.
2. The dependent variable.
3. The independent variable.

Return value: A list of equations exactly as returned by ODESOLVE itself. (In this case the list should normally contain precisely one equation.)

The file `extend.tst` contains a very simple test and demonstration of the operation of the first three classes of hook. Beware that this extension interface is experimental and subject to change.

20.42 ORTHOVEC: Manipulation of Scalars and Vectors

ORTHOVEC is a collection of REDUCE procedures and operations which provide a simple-to-use environment for the manipulation of scalars and vectors. Operations include addition, subtraction, dot and cross products, division, modulus, div, grad, curl, laplacian, differentiation, integration, and Taylor expansion.

Author: James W. Eastwood

Version 2 is summarized in [Eas91]. It differs from the original ([Eas87]) in revised notation and extended capabilities.

20.42.1 Introduction

The revised version of ORTHOVEC ([Eas91]) is, like the original ([Eas87]), a collection of REDUCE procedures and operators designed to simplify the machine aided manipulation of vectors and vector expansions frequently met in many areas of applied mathematics. The revisions have been introduced for two reasons: firstly, to add extra capabilities missing from the original and secondly, to tidy up input and output to make the package easier to use.

The changes from Version 1 include:

1. merging of scalar and vector unary and binary operators, $+, -, *, /$
2. extensions of the definitions of division and exponentiation to vectors
3. new vector dependency procedures
4. application of l'Hôpital's rule in limits and Taylor expansions
5. a new component selector operator
6. algebraic mode output of LISP vector components

The LISP vector primitives are again used to store vectors, although with the introduction of LIST types in algebraic mode in REDUCE 3.4, the implementation may have been more simply achieved using lists to store vector components.

The philosophy used in Version 2 follows that used in the original: namely, algebraic mode is used wherever possible. The view is taken that some computational inefficiencies are acceptable if it allows coding to be intelligible to (and thence adaptable by) users other than LISP experts familiar with the internal workings of REDUCE.

Procedures and operators in ORTHOVEC fall into the five classes: initialisation, input-output, algebraic operations, differential operations and integral operations.

Definitions are given in the following sections, and a summary of the procedure names and their meanings are give in Table 1. The final section discusses test examples.

20.42.2 Initialisation

The procedure `vstart` initialises ORTHOVEC. It may be called after ORTHOVEC has been loaded to reset coordinates. `vstart` provides a menu of standard coordinate systems:

1. cartesian (x, y, z) = `(x, y, z)`
2. cylindrical (r, θ, z) = `(r, th, z)`
3. spherical (r, θ, ϕ) = `(r, th, ph)`
4. general (u_1, u_2, u_3) = `(u1, u2, u3)`
5. others

which the user selects by number. Selecting options (1)-(4) automatically sets up the coordinates and scale factors. Selection option (5) shows the user how to select another coordinate system. If `vstart` is not called, then the default cartesian coordinates are used. ORTHOVEC may be re-initialised to a new coordinate system at any time during a given REDUCE session by typing

```
vstart $
```

20.42.3 Input-Output

ORTHOVEC assumes all quantities are either scalars or 3 component vectors. To define a vector a with components (c_1, c_2, c_3) use the procedure `svec` as follows

```
a := svec(c1, c2, c3);
```

The standard REDUCE output for vectors when using the terminator ";" is to list the three components inside square brackets $[\cdots]$, with each component in prefix form. A replacement for the standard REDUCE procedure `maprin` is included in the package to change the output of LISP vector components to algebraic notation. The procedure `vout` (which returns the value of its argument) can be used to give labelled output of components in algebraic form: e.g.,

```
b := svec (sin(x)**2, y**2, z)$
vout(b)$
```

The operator _ can be used to select a particular component (1, 2 or 3) for output e.g.

```
b _1 ;
```

Note the space before the _ operator: otherwise this would be read as identifier `b_1`.

20.42.4 Algebraic Operations

Six infix operators, sum, difference, quotient, times, exponentiation and cross product, and four prefix operators, plus, minus, reciprocal and modulus are defined in ORTHOVEC. These operators can take suitable combinations of scalar and vector arguments, and in the case of scalar arguments reduce to the usual definitions of $+, -, *, /$, etc.

The operators are represented by symbols

```
+, -, /, *, ^, ><
```

The composite >< is an attempt to represent the cross product symbol $\times$ in ASCII characters. If we let **v** be a vector and s be a scalar, then valid combinations of arguments of the procedures and operators and the type of the result are as summarised below. The notation used is
result :=procedure(left argument, right argument) or
result :=(left operand) operator (right operand) .

Vector Addition

v	:=	vectorplus(**v**)	or	**v**	:=	+ **v**
s	:=	vectorplus(s)	or	s	:=	+ s
v	:=	vectoradd(**v**,**v**)	or	**v**	:=	**v** + **v**
s	:=	vectoradd(s,s)	or	s	:=	s + s

Vector Subtraction

v	:=	vectorminus(**v**)	or	**v**	:=	- **v**
s	:=	vectorminus(s)	or	s	:=	- s
v	:=	vectordifference(**v**,**v**)	or	**v**	:=	**v** - **v**
s	:=	vectordifference(s,s)	or	s	:=	s - s

Vector Division

v	:=	vectorrecip(**v**)	or	**v**	:=	/ **v**
s	:=	vectorrecip(s)	or	s	:=	/ s
v	:=	vectorquotient(**v**,**v**)	or	**v**	:=	**v** / **v**
v	:=	vectorquotient(**v**, s)	or	**v**	:=	**v** / s
v	:=	vectorquotient(s ,**v**)	or	**v**	:=	s / **v**
s	:=	vectorquotient(s,s)	or	s	:=	s / s

Vector Multiplication

v	:=	vectortimes(s ,**v**)	or	**v**	:=	s * **v**
v	:=	vectortimes(**v**, s)	or	**v**	:=	**v** * s
s	:=	vectortimes(**v**,**v**)	or	s	:=	**v** * **v**
s	:=	vectortimes(s , s)	or	s	:=	s * s

Vector Cross Product

v	:=	vectorcross(**v**,**v**)	or	**v**	:=	**v** × **v**

Vector Exponentiation

s	:=	vectorexpt (**v**, s)	or	s	:=	**v** ^ s
s	:=	vectorexpt (s , s)	or	s	:=	s ^ s

Vector Modulus

s	:=	vmod (s)
s	:=	vmod (**v**)

All other combinations of operands for these operators lead to error messages being issued. The first two instances of vector multiplication are scalar multiplication of vectors, the third is the product of two scalars and the last is the inner (dot) product. The unary operators `+, -, /` can take either scalar or vector arguments and return results of the same type as their arguments. `vmod` returns a scalar.

In compound expressions, parentheses may be used to specify the order of combination. If parentheses are omitted the ordering of the operators, in increasing order of precedence is

```
+ | - | dotgrad | * | >< | ^ | _
```

and these are placed in the precedence list defined in REDUCE after <. The differential operator `dotgrad` is defined in the following section, and the component selector _ was introduced in section 3.

Vector divisions are defined as follows: If $\mathbf{a}$ and $\mathbf{b}$ are vectors and c is a scalar, then

$$\mathbf{a}/\mathbf{b} = \frac{\mathbf{a}\cdot\mathbf{b}}{|\ \mathbf{b}\ |^2}$$
$$c/\mathbf{a} = \frac{c\mathbf{a}}{|\ \mathbf{a}\ |^2}$$

Both scalar multiplication and dot products are given by the same symbol, braces are advisable to ensure the correct precedences in expressions such as $(\mathbf{a}\cdot\mathbf{b})(\mathbf{c}\cdot\mathbf{d})$.

Vector exponentiation is defined as the power of the modulus:
$\mathbf{a}^n \equiv \mathtt{vmod}(a)^n = |\ \mathbf{a}\ |^n$

s	:=	div (**v**)
v	:=	grad(s)
v	:=	curl(**v**)
v	:=	delsq(**v**)
s	:=	delsq(s)
v	:=	**v** dotgrad **v**
s	:=	**v** dotgrad s

Table 20.18: ORTHOVEC valid combinations of operator and argument

20.42.5 Differential Operations

Differential operators provided are div, grad, curl, delsq, and dotgrad. All but the last of these are prefix operators having a single vector or scalar argument as appropriate. Valid combinations of operator and argument, and the type of the result are shown in table 20.18.

All other combinations of operator and argument type cause error messages to be issued. The differential operators have their usual meanings [Spi59]. The coordinate system used by these operators is set by invoking `vstart` (cf. Sec. 20.42.2). The names `h1`, `h2` and `h3` are reserved for the scale factors, and `u1`, `u2` and `u3` are used for the coordinates.

A vector extension, `vdf`, of the REDUCE procedure DF allows the differentiation of a vector (scalar) with respect to a scalar to be performed. Allowed forms are `vdf`(**v**, s) $\rightarrow$ **v** and `vdf`(s, s) $\rightarrow$ s , where, for example

$$\texttt{vdf(B,x)} \equiv \frac{\partial \mathbf{B}}{\partial x}$$

The standard REDUCE declarations `depend` and `nodepend` have been redefined to allow dependences of vectors to be compactly defined. For example

```
a := svec(a1,a2,a3)$;
depend a,x,y;
```

causes all three components `a1`,`a2` and `a3` of `a` to be treated as functions of `x` and `y`. Individual component dependences can still be defined if desired.

```
depend a3,z;
```

The procedure `vtaylor` gives truncated Taylor series expansions of scalar or vector functions:

```
vtaylor(vex,vx,vpt,vorder);
```

VEX	VX	VPT	VORDER
v	**v**	**v**	**v**
v	**v**	**v**	s
v	s	s	s
s	**v**	**v**	**v**
s	**v**	**v**	s
s	s	s	s

Table 20.19: ORTHOVEC valid combination of argument types.

returns the series expansion of the expression VEX with respect to variable VX about point VPT to order VORDER. Valid combinations of argument types are shown in table 20.19.

Any other combinations cause error messages to be issued. Elements of VORDER must be non-negative integers, otherwise error messages are issued. If scalar VORDER is given for a vector expansion, expansions in each component are truncated at the same order, VORDER.

The new version of Taylor expansion applies l'Hôpital's rule in evaluating coefficients, so handle cases such as $\sin(x)/(x)$, etc. which the original version of ORTHOVEC could not. The procedure used for this is `ov_limit`, which can be used directly to find the limit of a scalar function `ex` of variable `x` at point `pt`:-

```
ans := ov_limit(ex,x,pt);
```

20.42.6 Integral Operations

Definite and indefinite vector, volume and scalar line integration procedures are included in ORTHOVEC. They are defined as follows:

$$\texttt{vint}(\mathbf{v},x) = \int \mathbf{v}(x)dx$$

$$\texttt{dvint}(\mathbf{v},x,a,b) = \int_a^b \mathbf{v}(x)dx$$

$$\texttt{volint}(\mathbf{v}) = \int \mathbf{v}h_1h_2h_3du_1du_2du_3$$

$$\texttt{dvolint}(\mathbf{v},\mathbf{l},\mathbf{u},n) = \int_{\mathbf{l}}^{\mathbf{u}} \mathbf{v}h_1h_2h_3du_1du_2du_3$$

$$\texttt{lineint}(\mathbf{v},\omega,t) = \int \mathbf{v}\cdot\mathbf{dr} \equiv \int v_ih_i\frac{\partial\omega_i}{\partial t}dt$$

$$\texttt{dlineint}(\mathbf{v},\omega t,a,b) = \int_a^b v_ih_i\frac{\partial\omega_i}{\partial t}dt$$

In the vector and volume integrals, $\mathbf{v}$ are vector or scalar, a, b, x and n are scalar. Vectors $\mathbf{l}$ and $\mathbf{u}$ contain expressions for lower and upper bounds to the integrals. The integer index n defines the order in which the integrals over u_1, u_2 and u_3 are performed in order to allow for functional dependencies in the integral bounds:

n	order
1	$u_1\ u_2\ u_3$
2	$u_3\ u_1\ u_2$
3	$u_2\ u_3\ u_1$
4	$u_1\ u_3\ u_2$
5	$u_2\ u_1\ u_3$
otherwise	$u_3\ u_2\ u_1$

The vector ω in the line integral's arguments contain explicit paramterisation of the coordinates u_1, u_2, u_3 of the line $\mathbf{u}(t)$ along which the integral is taken.

20.42.7 Test Cases

To use the REDUCE source version of ORTHOVEC, initiate a REDUCE session and then load the package with the command `load_package orthovec;` (see section 23.2 of the REDUCE manual). If coordinate dependent differential and integral operators other than cartesian are needed, then `vstart` must be used to reset coordinates and scale factors.

Six simple examples are given in the Test Run Output file *orthovec.rlg* to illustrate

Procedures		Description
vstart		select coordinate system
svec		set up a vector
vout		output a vector
vectorcomponent	_	extract a vector component (1-3)
vectoradd	+	add two vectors or scalars
vectorplus	+	unary vector or scalar plus
vectorminus	-	unary vector or scalar minus
vectordifference	-	subtract two vectors or scalars
vectorquotient	/	vector divided by scalar
vectorrecip	/	unary vector or scalar division (reciprocal)
vectortimes	*	multiply vector or scalar by vector/scalar
vectorcross	><	cross product of two vectors
vectorexpt	^	exponentiate vector modulus or scalar
vmod		length of vector or scalar

Table 20.20: Procedures names and operators used in ORTHOVEC (part 1)

the working of ORTHOVEC. The input lines were taken from the file *orthovec.tst* (the Test Run Input), but could equally well be typed in at the Terminal.

Example 1

Show that

$$(\mathbf{a} \times \mathbf{b}) \cdot (\mathbf{c} \times \mathbf{d}) - (\mathbf{a} \cdot \mathbf{c})(\mathbf{b} \cdot \mathbf{d}) + (\mathbf{a} \cdot \mathbf{d})(\mathbf{b} \cdot \mathbf{c}) \equiv 0$$

Example 2

Write the equation of motion

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} + \nabla p - curl(\mathbf{B}) \times \mathbf{B}$$

in cylindrical coordinates.

Example 3

Taylor expand

- $\sin(x)\cos(y) + e^z$ about the point $(0, 0, 0)$ to third order in x, fourth order in y and fifth order in z.
- $\sin(x)/x$ about x to fifth order.
- $\mathbf{v}$ about $\mathbf{x} = (x, y, z)$ to fifth order, where $\mathbf{v} = (x/\sin(x), (e^y - 1)/y, (1 + z)^{10})$.

Procedures	Description
`div`	divergence of vector
`grad`	gradient of scalar
`curl`	curl of vector
`delsq`	laplacian of scalar or vector
`dotgrad`	(vector).grad(scalar or vector)
`vtaylor`	vector or scalar Taylor series of vector or scalar
`vptaylor`	vector or scalar Taylor series of scalar
`taylor`	scalar Taylor series of scalar
`limit`	limit of quotient using l'Hôpital's rule
`vint`	vector integral
`dvint`	definite vector integral
`volint`	volume integral
`dvolint`	definite volume integral
`lineint`	line integral
`dlineint`	definite line integral
`maprin`	vector extension of REDUCE `maprin`
`depend`	vector extension of REDUCE `depend`
`nodepend`	vector extension of REDUCE `nodepend`

Table 20.21: Procedures names and operators used in ORTHOVEC (part 2)

Example 4

Obtain the second component of the equation of motion in example 2, and the first component of the final vector Taylor series in example 3.

Example 5

Evaluate the line integral

$$\int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{A} \cdot d\mathbf{r}$$

from point $\mathbf{r}_1 = (1, 1, 1)$ to point $\mathbf{r}_2 = (2, 4, 8)$ along the path $(x, y, z) = (s, s^2, s^3)$ where

$$\mathbf{A} = (3x^2 + 5y)\mathbf{i} - 12xy\mathbf{j} + 2xyz^2\mathbf{k}$$

and $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ are unit vectors in the (x, y, z) directions.

Example 6

Find the volume V common to the intersecting cylinders $x^2 + y^2 = r^2$ and $x^2 + z^2 = r^2$ i.e. evaluate

$$V = 8 \int_0^r dx \int_0^{ub} dy \int_0^{ub} dz$$

where $ub = \overline{\sqrt{r^2 - x^2}}$

20.43 PHYSOP: Operator Calculus in Quantum Theory

This package has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly of the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space.

Author: Mathias Warns

20.43.1 Introduction

The package PHYSOP has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly in the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space. Since the capabilities of the current REDUCE release to deal with complex expressions containing noncommutative operators are rather restricted, the first step was to enhance these possibilities in order to achieve a better usability of REDUCE for these kind of calculations. This has led to the development of a first package called NONCOM2 which is described in section 2. For more complicated expressions involving both scalar quantities and operators the need for an additional data type has emerged in order to make a clear separation between the various objects present in the calculation. The implementation of this new REDUCE data type is realized by the PHYSOP (for PHYSical OPerator) package described in section 3.

20.43.2 The NONCOM2 Package

The package NONCOM2 redefines some standard REDUCE routines in order to modify the way noncommutative operators are handled by the system. In standard REDUCE declaring an operator to be noncommutative using the `noncom` statement puts a global flag on the operator. This flag is checked when the system has to decide whether or not two operators commute during the manipulation of an expression.

The NONCOM2 package redefines the `noncom` statement in a way more suitable for calculations in physics. Operators have now to be declared noncommutative pairwise, i.e. coding:

```
NONCOM A,B;
```

declares the operators `A` and `B` to be noncommutative but allows them to commute

with any other (noncommutative or not) operator present in the expression. In a similar way if one wants e.g. `A(X)` and `A(Y)` not to commute, one has now to code:

```
NONCOM A,A;
```

Each operator gets a new property list containing the operators with which it does not commute. A final example should make the use of the redefined `NONCOM` statement clear:

```
NONCOM A,B,C;
```

declares `A` to be noncommutative with `B` and `C`, `B` to be noncommutative with `A` and `C` and `C` to be noncommutative with `A` and `B`. Note that after these declaration e.g. `A(X)` and `A(Y)` are still commuting kernels.

Finally to keep the compatibility with standard REDUCE declaring a single identifier using the `NONCOM` statement has the same effect as in standard REDUCE i.e., the identifier is flagged with the `NONCOM` tag.

From the user's point of view there are no other new commands implemented by the package. Commutation relations have to be declared in the standard way as described in the manual i.e. using `LET` statements. The package itself consists of several redefined standard REDUCE routines to handle the new definition of noncommutativity in multiplications and pattern matching processes.

CAVEAT: Due to its nature, the package is highly version dependent. The current version has been designed for the 3.3 and 3.4 releases of REDUCE and may not work with previous versions. Some different (but still correct) results may occur by using this package in conjunction with `let` statements since part of the pattern matching routines have been redesigned. The package has been designed to bridge a deficiency of the current REDUCE version concerning the notion of noncommutativity and it is the author's hope that it will be made obsolete by a future release of REDUCE.

20.43.3 The PHYSOP package

The package PHYSOP implements a new REDUCE data type to perform calculations with physical operators. The noncommutativity of operators is implemented using the NONCOM2 package so this file should be loaded prior to the use of PHYSOP[39]. In the following the new commands implemented by the package are

[39]To build a fast loading version of PHYSOP the NONCOM2 source code should be read in prior to the PHYSOP code

described. Beside these additional commands, the full set of standard REDUCE instructions remains available for performing any other calculation.

20.43.3.1 Type declaration commands

The new REDUCE data type PHYSOP implemented by the package allows the definition of a new kind of operators (i.e. kernels carrying an arbitrary number of arguments). Throughout this manual, the name "operator" will refer, unless explicitly stated otherwise, to this new data type. This data type is in turn divided into 5 subtypes. For each of this subtype, a declaration command has been defined:

SCALOP A; declares `A` to be a scalar operator. This operator may carry an arbitrary number of arguments i.e. after the declaration: `SCALOP A;` all kernels of the form e.g. `A(J), A(1,N), A(N,L,M)` are recognized by the system as being scalar operators.

VECOP V; declares `V` to be a vector operator. As for scalar operators, the vector operators may carry an arbitrary number of arguments. For example `V(3)` can be used to represent the vector operator $\vec{V}_3$. Note that the dimension of space in which this operator lives is arbitrary. One can however address a specific component of the vector operator by using a special index declared as `PHYSINDEX` (see below). This index must then be the first in the argument list of the vector operator.

TENSOP C(3); declares `C` to be a tensor operator of rank 3. Tensor operators of any fixed integer rank larger than 1 can be declared. Again this operator may carry an arbitrary number of arguments and the space dimension is not fixed. The tensor components can be addressed by using special `PHYSINDEX` indices (see below) which have to be placed in front of all other arguments in the argument list.

STATE U; declares `U` to be a state, i.e. an object on which operators have a certain action. The state U can also carry an arbitrary number of arguments.

PHYSINDEX X; declares `X` to be a special index which will be used to address components of vector and tensor operators.

It is very important to understand precisely the way how the type declaration commands work in order to avoid type mismatch errors when using the PHYSOP package. The following examples should illustrate the way the program interprets type declarations. Assume that the declarations listed above have been typed in by the user, then:

- `A,A(1,N),A(N,M,K)` are SCALAR operators.

- `V,V(3),V(N,M)` are VECTOR operators.
- `C, C(5),C(Y,Z)` are TENSOR operators of rank 3.
- `U,U(P),U(N,L,M)` are STATES.

BUT: `V(X),V(X,3),V(X,N,M)` are all scalar operators since the special index `X` addresses a specific component of the vector operator (which is a scalar operator). Accordingly, `C(X,X,X)` is also a scalar operator because the diagonal component C_{xxx} of the tensor operator `C` is meant here (C has rank 3 so 3 special indices must be used for the components).

In view of these examples, every time the following text refers to scalar operators, it should be understood that this means not only operators defined by the `SCALOP` statement but also components of vector and tensor operators. Depending on the situation, in some case when dealing only with the components of vector or tensor operators it may be preferable to use an operator declared with `SCALOP` rather than addressing the components using several special indices (throughout the manual, indices declared with the `PHYSINDEX` command are referred to as special indices).

Another important feature of the system is that for each operator declared using the statements described above, the system generates 2 additional operators of the same type: the adjoint and the inverse operator. These operators are accessible to the user for subsequent calculations without any new declaration. The syntax is as following:

If `A` has been declared to be an operator (scalar, vector or tensor) the adjoint operator is denoted `A!+` and the inverse operator is denoted `A!-1` (an inverse adjoint operator `A!+!-1` is also generated). The exclamation marks do not appear when these operators are printed out by REDUCE (except when the switch `NAT` is set to off) but have to be typed in when these operators are used in an input expression. An adjoint (but no inverse) state is also generated for every state defined by the user. One may consider these generated operators as "placeholders" which means that these operators are considered by default as being completely independent of the original operator. Especially if some value is assigned to the original operator, this value is not automatically assigned to the generated operators. The user must code additional assignement statements in order to get the corresponding values.

Exceptions from these rules are (i) that inverse operators are always ordered at the same place as the original operators and (ii) that the expressions `A!-1*A` and `A*A!-1` are replaced[40] by the unit operator `UNIT`. This operator is defined as a scalar operator during the initialization of the PHYSOP package. It should be used to indicate the type of an operator expression whenever no other PHYSOP occur in it. For example, the following sequence:

[40] This may not always occur in intermediate steps of a calculation due to efficiency reasons.

```
SCALOP A;
A:= 5;
```

leads to a type mismatch error and should be replaced by:

```
SCALOP A;
A:=5*UNIT;
```

The operator UNIT is a reserved variable of the system and should not be used for other purposes.

All other kernels (including standard REDUCE operators) occurring in expressions are treated as ordinary scalar variables without any PHYSOP type (referred to as scalars in the following). Assignement statements are checked to ensure correct operator type assignement on both sides leading to an error if a type mismatch occurs. However an assignement statement of the form A:= 0 or LET A = 0 is always valid regardless of the type of A.

Finally a command CLEARPHYSOP has been defined to remove the PHYSOP type from an identifier in order to use it for subsequent calculations (e.g. as an ordinary REDUCE operator). However it should be remembered that no substitution rule is cleared by this function. It is therefore left to the user's responsibility to clear previously all substitution rules involving the identifier from which the PHYSOP type is removed.

Users should be very careful when defining procedures or statements of the type FOR ALL ... LET ... that the PHYSOP type of all identifiers occurring in such expressions is unambigously fixed. The type analysing procedure is rather restrictive and will print out a "PHYSOP type conflict" error message if such ambiguities occur.

20.43.3.2 Ordering of operators in an expression

The ordering of kernels in an expression is performed according to the following rules:
1. Scalars are always ordered ahead of PHYSOP operators in an expression. The REDUCE statement korder can be used to control the ordering of scalars but has no effect on the ordering of operators.

2. The default ordering of operators follows the order in which they have been declared (and not the alphabetical one). This ordering scheme can be changed using the command OPORDER. Its syntax is similar to the korder statement, i.e. coding: OPORDER A,V,F; means that all occurrences of the operator A are ordered ahead of those of V etc. It is also possible to include operators carrying indices (both normal and special ones) in the argument list of OPORDER. However

including objects not defined as operators (i.e. scalars or indices) in the argument list of the OPORDER command leads to an error.

3. Adjoint operators are placed by the declaration commands just after the original operators on the OPORDER list. Changing the place of an operator on this list means not that the adjoint operator is moved accordingly. This adjoint operator can be moved freely by including it in the argument list of the OPORDER command.

20.43.3.3 Arithmetic operations on operators

The following arithmetic operations are possible with operator expressions:

1. Multiplication or division of an operator by a scalar.

2. Addition and subtraction of operators of the same type.

3. Multiplication of operators is only defined between two scalar operators.

4. The scalar product of two VECTOR operators is implemented with a new function DOT . The system expands the product of two vector operators into an ordinary product of the components of these operators by inserting a special index generated by the program. To give an example, if one codes:

```
VECOP V,W;
V DOT W;
```

the system will transform the product into:

```
V(IDX1) * W(IDX1)
```

where IDX1 is a PHYSINDEX generated by the system (called a DUMMY INDEX in the following) to express the summation over the components. The identifiers IDXn (n is a nonzero integer) are reserved variables for this purpose and should not be used for other applications. The arithmetic operator DOT can be used both in infix and prefix form with two arguments.

5. Operators (but not states) can only be raised to an integer power. The system expands this power expression into a product of the corresponding number of terms inserting dummy indices if necessary. The following examples explain the transformations occurring on power expressions (system output is indicated with an −>):

```
SCALOP A; A**2;
- --> A*A
VECOP V; V**4;
- --> V(IDX1)*V(IDX1)*V(IDX2)*V(IDX2)
```

```
TENSOP C(2); C**2;
- --> C(IDX3,IDX4)*C(IDX3,IDX4)
```

Note in particular the way how the system interprets powers of tensor operators which is different from the notation used in matrix algebra.

6. Quotients of operators are only defined between scalar operator expressions. The system transforms the quotient of 2 scalar operators into the product of the first operator times the inverse of the second one. Example[41]:

```
SCALOP A,B;    A / B;
        -1
 --> (B   )*A
```

7. Combining the last 2 rules explains the way how the system handles negative powers of operators:

```
SCALOP B;
B**(-3);
        -1     -1     -1
 --> (B   )*(B   )*(B   )
```

The method of inserting dummy indices and expanding powers of operators has been chosen to facilitate the handling of complicated operator expressions and particularly their application on states (see section 3.4.3). However it may be useful to get rid of these dummy indices in order to enhance the readability of the system's final output. For this purpose the switch `contract` has to be turned on (`contract` is normally set to `OFF`). The system in this case contracts over dummy indices reinserting the `DOT` operator and reassembling the expanded powers. However due to the predefined operator ordering the system may not remove all the dummy indices introduced previously.

20.43.3.4 Special functions

Commutation relations

If 2 PHYSOPs have been declared noncommutative using the (redefined) `noncom` statement, it is possible to introduce in the environment elementary (anti-) commutation relations between them. For this purpose, 2 scalar operators `comm` and `anticomm` are available. These operators are used in conjunction with `let` statements. Example:

[41] This shows how inverse operators are printed out when the switch `NAT` is on

```
SCALOP A,B,C,D;
LET COMM(A,B)=C;
FOR ALL N,M LET ANTICOMM(A(N),B(M))=D;
VECOP U,V,W; PHYSINDEX X,Y,Z;
FOR ALL X,Y LET COMM(V(X),W(Y))=U(Z);
```

Note that if special indices are used as dummy variables in `FOR ALL ... LET` constructs then these indices should have been declared previously using the `PHYSINDEX` command.

Every time the system encounters a product term involving 2 noncommutative operators which have to be reordered on account of the given operator ordering, the list of available (anti-) commutators is checked in the following way: First the system looks for a commutation relation which matches the product term. If it fails then the defined anticommutation relations are checked. If there is no successful match the product term `A*B` is replaced by:

```
A*B;
 --> COMM(A,B) + B*A
```

so that the user may introduce the commutation relation later on.

The user may want to force the system to look for anticommutators only; for this purpose a switch `anticom` is defined which has to be turned on (`anticom` is normally set to `OFF`). In this case, the above example is replaced by:

```
ON ANTICOM;
A*B;
 -->  ANTICOMM(A,B) - B*A
```

Once the operator ordering has been fixed (in the example above `B` has to be ordered ahead of `A`), there is no way to prevent the system from introducing (anti-)commutators every time it encounters a product whose terms are not in the right order. On the other hand, simply by changing the `OPORDER` statement and reevaluating the expression one can change the operator ordering without the need to introduce new commutation relations. Consider the following example:

```
SCALOP A,B,C;   NONCOM A,B;   OPORDER B,A;
LET COMM(A,B)=C;
A*B;
- -->  B*A + C;
OPORDER A,B;
 B*A;
- --> A*B - C;
```

The functions `comm` and `anticomm` should only be used to define elementary (anti-) commutation relations between single operators. For the calculation of (anti-) commutators between complex operator expressions, the functions `commute` and `anticommute` have been defined. Example (is included as example 1 in the test file):

```
VECOP P,A,K;
PHYSINDEX X,Y;
FOR ALL X,Y LET COMM(P(X),A(Y))=K(X)*A(Y);
COMMUTE(P**2,P DOT A);
```

Adjoint expressions

As has been already mentioned, for each operator and state defined using the declaration commands quoted in section 3.1, the system generates automatically the corresponding adjoint operator. For the calculation of the adjoint representation of a complicated operator expression, a function `adj` has been defined. Example[42]:

```
SCALOP A,B;
ADJ(A*B);
        +     +
 --> (B )*(A )
```

Application of operators on states

For this purpose, a function `opapply` has been defined. It has 2 arguments and is used in the following combinations:

(i) `let opapply(`*operator, state*`)` = *state*; This is to define a elementary action of an operator on a state in analogy to the way elementary commutation relations are introduced to the system. Example:

```
SCALOP A; STATE U;
FOR ALL N,P LET OPAPPLY((A(N),U(P))= EXP(I*N*P)*U(P);
```

(ii) `let opapply(`*state, state*`)` = *scalar exp.*; This form is to define scalar products between states and normalization conditions. Example:

```
STATE U;
FOR ALL N,M LET OPAPPLY(U(N),U(M)) =
   IF N=M THEN 1 ELSE 0;
```

[42] This shows how adjoint operators are printed out when the switch `nat` is on

(iii) *state* `:=` `opapply(`*operator expression, state*); In this way, the action of an operator expression on a given state is calculated using elementary relations defined as explained in **(i)**. The result may be assigned to a different state vector.

(iv) `opapply(`*state*`,` `opapply(`*operator expression, state*`))`; This is the way how to calculate matrix elements of operator expressions. The system proceeds in the following way: first the rightmost operator is applied on the right state, which means that the system tries to find an elementary relation which match the application of the operator on the state. If it fails the system tries to apply the leftmost operator of the expression on the left state using the adjoint representations. If this fails also, the system prints out a warning message and stops the evaluation. Otherwise the next operator occuring in the expression is taken and so on until the complete expression is applied. Then the system looks for a relation expressing the scalar product of the two resulting states and prints out the final result. An example of such a calculation is given in the test file.

The infix version of the `opapply` function is the vertical bar $|$. It is right associative and placed in the precedence list just above the minus $(-)$ operator. Some of the REDUCE implementation may not work with this character, the prefix form should then be used instead[43].

20.43.4 Known problems in the current release of PHYSOP

(i) Some spurious negative powers of operators may appear in the result of a calculation using the PHYSOP package. This is a purely ”cosmetic” effect which is due to an additional factorization of the expression in the output printing routines of REDUCE. Setting off the REDUCE switch `allfac` (`allfac` is normally on) should make these terms disappear and print out the correct result (see example 1 in the test file).

(ii) The current release of the PHYSOP package is not optimized w.r.t. computation speed. Users should be aware that the evaluation of complicated expressions involving a lot of commutation relations requires a significant amount of CPU time and memory. Therefore the use of PHYSOP on small machines is rather limited. A minimal hardware configuration should include at least 4 MB of memory and a reasonably fast CPU (type Intel 80386 or equiv.).

(iii) Slightly different ordering of operators (especially with multiple occurrences of the same operator with different indices) may appear in some calculations due to the internal ordering of atoms in the underlying LISP system (see last example in the test file). This cannot be entirely avoided by the package but does not affect the correctness of the results.

[43] The source code can also be modified to choose another special character for the function

20.43.5 Final remarks

The package PHYSOP has been presented by the author at the IV inter. Conference on Computer Algebra in Physical Research, Dubna (USSR) 1990 (see [War91]). It has been developed with the aim in mind to perform calculations of the type exemplified in the test file included in the distribution of this package. However it should also be useful in some other domains like e.g. the calculations of complicated Feynman diagrams in QCD which could not be performed using the HEPHYS package. The author is therefore grateful for any suggestion to improve or extend the usability of the package. Users should not hesitate to contact the author for additional help and explanations on how to use this package. Some bugs may also appear which have not been discovered during the tests performed prior to the release of this version. Please send in this case to the author a short input and output listing displaying the encountered problem.

Acknowledgements

The main ideas for the implementation of a new data type in the REDUCE environnement have been taken from the VECTOR package developed by Dr. David Harper ([Har89]). Useful discussions with Dr. Eberhard Schrüfer and Prof. John Fitch are also gratefully acknowledged.

20.43.6 Appendix: List of error and warning messages

In the following the error (E) and warning (W) messages specific to the PHYSOP package are listed.

`cannot declare` *x* **`as`** ***data type*** (W): An attempt has been made to declare an object *x* which cannot be used as a PHYSOP operator of the required type. The declaration command is ignored.

x **`already defined as`** ***data type*** (W): The object *x* has already been declared using a REDUCE type declaration command and can therefore not be used as a PHYSOP operator. The declaration command is ignored.

x **`already declared as`** ***data type*** (W): The object x has already been declared with a PHYSOP declaration command. The declaration command is ignored.

x **`is not a PHYSOP`** (E): An invalid argument has been included in an OPORDER command. Check the arguments.

`invalid argument(s) to` ***function*** (E): A function implemented by the PHYSOP package has been called with an invalid argument. Check type of arguments.

Type conflict in *operation* (E): A PHYSOP type conflict has occured during an arithmetic operation. Check the arguments.

invalid call of *function* with args: *arguments* (E): A function of the PHYSOP package has been declared with invalid argument(s). Check the argument list.

type mismatch in *expression* (E): A type mismatch has been detected in an expression. Check the corresponding expression.

type mismatch in *assignement* (E): A type mismatch has been detected in an assignment or in a LET statement. Check the listed statement.

PHYSOP type conflict in *expr* (E): A ambiguity has been detected during the type analysis of the expression. Check the expression.

operators in exponent cannot be handled (E): An operator has occurred in the exponent of an expression.

cannot raise a state to a power (E): states cannot be exponentiated by the system.

invalid quotient (E): An invalid denominator has occurred in a quotient. Check the expression.

physops of different types cannot be commuted (E): An invalid operator has occurred in a call of the COMMUTE/ANTICOMMUTE function.

commutators only implemented between scalar operators (E): An invalid operator has occurred in the call of the COMMUTE/ANTICOMMUTE function.

evaluation incomplete due to missing elementary relations (W): The system has not found all the elementary commutators or application relations necessary to calculate or reorder the input expression. The result may however be used for further calculations.

20.44 PM: A REDUCE Pattern Matcher

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica, and is based on the pattern matcher described in Kevin McIsaac, "Pattern Matching Algebraic Identities", SIGSAM Bulletin, 19 (1985), 4-13.

Author: Kevin McIsaac

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica, and is based on the pattern matcher described in [McI85]. The following is a description of its structure.

A *template* is any expression composed of literal elements, e.g. `5`, `a`, or `a+1`, and specially-denoted pattern variables, e.g. `?a` or `??b`. Atoms beginning with `?` are called *generic variables* and match any expression.

Atoms beginning with `??` are called *multi-generic variables* and match any expression or any sequence of expressions including the null or empty sequence. A sequence is an expression of the form `[a1,a2,...]`. When placed in a function argument list the brackets are removed, i.e. `f([a,1]) -> f(a,1)` and `f(a,[1,2],b) -> f(a,1,2,b)`.

A template is said to match an expression if the template is literally equal to the expression, or if by replacing any of the generic or multi-generic symbols occurring in the template, the template can be made to be literally equal to the expression. These replacements are called the *bindings* for the generic variables. A replacement is an expression of the form `exp1 -> exp2`, which means `exp1` is replaced by `exp2`, or `exp1 -> exp2`, which is the same except `exp2` is not simplified until after the substitution for `exp1` is made. If the expression has any of the properties associativity, commutativity, or an identity element, they are used to determine if the expressions match. If an attempt to match the template to the expression fails the matcher backtracks, unbinding generic variables, until it reaches a place where it can make a different choice. It then proceeds along the new branch.

The current matcher proceeds from left to right in a depth-first search of the template expression tree. Rearrangements of the expression are generated when the match fails and the matcher backtracks.

The matcher also supports *semantic* matching. Briefly, if a subtemplate does not match the corresponding subexpression because they have different structures, then the two are equated and the matcher continues matching the rest of the expression until all the generic variables in the subexpression are bound. The equality is then checked. This is controlled by the switch `semantic`. By default it is `on`.

20.44.1 `M(exp,temp)`

The template `temp` is matched against the expression `exp`. If the template is literally equal to the expression `T` is returned. If the template is literally equal to the expression after replacing the generic variables by their bindings then the set of bindings is returned as a set of replacements. Otherwise `0` (`nil`) is returned.

Examples:

A "literal" template:

```
m(f(a), f(a));
t
```

Not literally equal:

```
m(f(a), f(b));
0
```

Nested operators:

```
m(f(a,h(b)), f(a,h(b)));
t
```

"Generic" templates:

```
m(f(a,b), f(a,?a));
{?a -> b}
m(f(a,b), f(?a,?b));
{?b -> b, ?a -> a}
```

The multi-generic symbol `??a` matches the "rest" of the arguments:

```
m(f(a,b), f(??a));
{??a -> {[a, b]}
```

but the generic symbol `?a` does not:

```
m(f(a,b), f(?a));
0
```

Flag h as "associative":

```
flag('(h), 'assoc);
```

Associativity is used to group terms together:

```
m(h(a,b,d,e), h(?a,d,?b));
{?b -> e, ?a -> h(a,b)}
```

"plus" is a symmetric function:

```
m(a+b+c, c+?a+?b);
{?b -> a, ?a -> b}
```

and it is also associative

```
m(a+b+c, b+?a);
{?a -> c + a}
```

Note that the effect of using a multi-generic symbol is different:

```
m(a+b+c,b+??c);
{??c -> [c,a]}
```

20.44.2 temp _= logical_exp

A template may be qualified by the use of the conditional operator `_=`, `such!-that`. When a `such-that` condition is encountered in a template, it is held until all generic variables appearing in `logical_exp` are bound.

On the binding of the last generic variable, `logical_exp` is simplified and if the result is not `T` the condition fails and the pattern matcher backtracks. When the template has been fully parsed any remaining held `such-that` conditions are evaluated and compared to `T`.

Examples:

```
m(f(a,b), f(?a,?b_=(?a=?b)));
0
m(f(a,a), f(?a,?b_=(?a=?b)));
{?b -> a, ?a -> a}
```

Note that `f(?a,?b_=(?a=?b))` is the same as `f(?a,?a)`.

20.44.3 S(exp,{temp1 → sub1, temp2 → sub2, . . . }, rept, depth)

Substitute the set of replacements into `exp`, re-substituting a maximum of `rept` times and to a maximum depth `depth`. `rept` and `depth` have the default values of 1 and ∞ respectively. Essentially, `S` is a breadth-first search-and-replace. (There is also a depth-first version, `Sd(...)`.) Each template is matched against `exp` until a successful match occurs.

Any replacements for generic variables are applied to the r.h.s. of that replacement and `exp` is replaced by the r.h.s. The substitution process is restarted on the new expression starting with the first replacement. If none of the templates match `exp` then the first replacement is tried against each sub-expression of `exp`. If a matching template is found then the sub-expression is replaced and process continues with the next sub-expression.

When all sub-expressions have been examined, if a match was found, the expression is evaluated and the process is restarted on the sub-expressions of the resulting expression, starting with the first replacement. When all sub-expressions have been examined and no match found the sub-expressions are reexamined using the next replacement. Finally when this has been done for all replacements and no match found then the process recures on each sub-expression. The process is terminated after `rept` replacements or when the expression no longer changes.

The command

```
Si(exp, {temp1 -> sub1, temp2 -> sub2, ...}, depth)
```

means "substitute infinitely many times until expression stops changing". It is short-hand for `S(exp,{temp1 -> sub1, temp2 -> sub2,...},Inf, depth)`.

Examples:

```
s(f(a,b), f(a,?b) -> ?b\^{}2);
2
b
s(a+b, a+b -> a{*}b);
b*a
```

"Associativity" is used to group $a + b + c$ to $(a + b) + c$:

```
s(a+b+c, a+b -> a*b);
b*a + c
```

The next three examples use a rule set that defines the factorial function. Substitute

once:

```
s(nfac(3),
   {nfac(0) -> 1, nfac(?x) -> ?x*nfac(?x-1)});
3*nfac(2)
```

Substitute twice:

```
s(nfac(3),
   {nfac(0) -> 1, nfac(?x) -> ?x*nfac(?x-1)}, 2);
6*nfac(1)
```

Substitute until expression stops changing:

```
si(nfac(3),
   {nfac(0) -> 1, nfac(?x) -> ?x{*}nfac(?x-1)});
6
```

Only substitute at the top level:

```
s(a+b+f(a+b), a+b -> a*b, inf, 0);
f(b+a) + b*a
```

20.44.4 temp :- exp and temp ::- exp

If during simplification of an expression, `temp` matches some sub-expression, then that sub-expression is replaced by `exp`. If there is a choice of templates to apply, the least general is used.

If an old rule exists with the same template, then the old rule is replaced by the new rule. If `exp` is `nil` the rule is retracted.

`temp ::- exp` is the same as `temp :- exp`, but the l.h.s. is not simplified until the replacement is made.

Examples:

Define the factorial function of a natural number as a recursive function and a termination condition. For all other values write it as a gamma function. Note that the order of definition is not important, as the rules are re-ordered so that the most specific rule is tried first. Note the use of `::-` instead of `:-` to stop simplification of the l.h.s. `hold` stops its arguments from being simplified.

```
fac(?x _= Natp(?x)) ::- ?x*fac(?x-1);
```

```
hold(fac(?X-1)*?X)
fac(0) :- 1;
1
fac(?x) :- Gamma(?x+1);
gamma(?X + 1)
fac(3);
6
fac(3/2);
gamma(5/2)
```

20.44.5 Arep({rep1,rep2,...})

In future simplifications automatically apply replacements `rep1, rep2, ...` until the rules are retracted. In effect, this replaces the operator -> by `:-` in the set of replacements `{rep1, rep2, ...}`.

20.44.6 Drep({rep1,rep2,..})

Delete the rules `rep1, rep2, ...`.

As we said earlier, the matcher has been constructed along the lines of the pattern matcher described in McIsaac with the addition of such-that conditions and "semantic matching" as described in Grief. To make a template efficient, some consideration should be given to the structure of the template and the position of such-that statements. In general the template should be constructed so that failure to match is recognized as early as possible. The multi-generic symbol should be used whenever appropriate, particularly with symmetric functions. For further details see McIsaac.

Examples:

`f(?a,?a,?b)` is better than `f(?a,?b,?c_=(?a=?b))`. `?a+??b` is better than `?a+?b+?c...`.

The template `f(?a+?b,?a,?b)`, matched against `f(3,2,1)` is matched as `f(?e_=(?e=?a+?b),?a,?b)` when semantic matching is allowed.

20.44.7 Switches

`trpm` Produces a trace of the rules applied during a substitution. This is useful to see how the pattern matcher works, or to understand an unexpected result.

In general usage the following switches need not be considered:

semantic Allow semantic matches, e.g. `f(?a+?b,?a,?b)` will match `f(3,2,1)`, even though the matcher works from left to right.

sym!-assoc Limits the search space of symmetric associative functions when the template contains multi-generic symbols so that generic symbols will not function. For example `(a+b+c,?a+??b)` will return

```
{?a -> a, ??b -> [b,c]} or
{?a -> b, ??b -> [a,c]} or
{?a -> c, ??b -> [a,b]}
```

but not `{?a -> a+b, ??b -> c}`, etc. No sane template should require these types of matches. However they can be made available by turning the switch off.

20.45 QHULL: Compute the Complex Hull

This package is an interface to qhull (www.qhull.org), which has to be installed externally. There are 3 options for this package to find the qhull program:

1. Put it into the path of your shell (recommended).
2. Set and export an environment variable `QHULL` to the complete path, e.g., in the Bash:

   ```
   export QHULL=/usr/bin/qhull
   ```

3. Inside Reduce set the variable `qhull_call!*` to the complete path, e.g.,

   ```
   symbolic(qhull_call!* := "/usr/bin/qhull");
   ```

Example: Compute the convex hull of a list integer points as a subset of that list as follows:

```
1: qhull {{2,0,0}, {0,2,0}, {0,2,2}, {0,0,0}, {1,1,1}};

{{2,0,0},{0,2,0},{0,2,2},{0,0,0}}

2: symbolic;

nil

3* qhull_qhull
        {{2,0,0}, {0,2,0}, {0,2,2}, {0,0,0}, {1,1,1}};

((2 0 0) (0 2 0) (0 2 2) (0 0 0))
```

Author: Thomas Sturm, March 2013

20.46 QSUM: Indefinite and Definite Summation of q-Hypergeometric Terms

Authors: Harald Böing and Wolfram Koepf

20.46.1 Introduction

This package is an implementation of the q-analogues of Gosper's and Zeilberger's[44] algorithm for indefinite, and definite summation of q-hypergeometric terms, respectively.

An expression a_k is called a *q-hypergeometric term*, if a_k/a_{k-1} is a rational function with respect to q^k. Most q-terms are based on the *q-shifted factorial* or *qpochhammer*. Other typical q-hypergeometric terms are ratios of products of powers, q-factorials, q-binomial coefficients, and q-shifted factorials that are integer-linear in their arguments.

20.46.2 Elementary q-Functions

Our package supports the input of the following elementary q-functions:

- `qpochhammer(a,q,infinity)`

$$(a;\,q)_\infty := \prod_{j=0}^{\infty} \left(1 - a\,q^j\right)$$

- `qpochhammer(a,q,k)`

$$(a;\,q)_k := \begin{cases} \prod_{j=0}^{k-1} \left(1 - a\,q^j\right) & \text{if } k > 0 \\ 1 & \text{if } k = 0 \\ \prod_{j=1}^{k} \left(1 - a\,q^{-j}\right)^{-1} & \text{if } k < 0 \end{cases}$$

- `qbrackets(k,q)`

$$[q,k] := \frac{q^k - 1}{q - 1}$$

- `qfactorial(k,q)`

$$[k]_q! := \frac{(q;\,q)_k}{(1-q)^k}$$

[44] The `ZEILBERG` package (see [Koe95b]) contains the hypergeometric versions. Those algorithms are described in [Gos78],[Zei91],[Zei90] and [Koe95a].

- `qbinomial(n,k,q)`

$$\binom{n}{k}_q := \frac{(q;\,q)_n}{(q;\,q)_k \cdot (q;\,q)_{n-k}}$$

Furthermore it is possible to use an abbreviation for the *generalized q-hypergeometric series* (*basic generalized hypergeometric series*, see e. g. [GR90], Chapter 1) which is defined as:

$$ {}_r\phi_s \left[\begin{array}{c} a_1, a_2, \ldots, a_r \\ b_1, b_2, \ldots, b_s \end{array} \middle| q, z \right] := \sum_{k=0}^{\infty} \frac{(a_1, a_2, \ldots, a_r;\, q)_k}{(b_1, b_2, \ldots, b_s;\, q)_k} \frac{z^k}{(q;\, q)_k} \left[(-1)^k\, q^{\binom{k}{2}} \right]^{1+s-r} \tag{20.95} $$

where $(a_1, a_2, \ldots, a_r;\, q)_k$ is a short form to write the product $\prod_{j=1}^{r} (a_j;\, q)_k$. An ${}_r\phi_s$ series terminates if one of its numerator parameters is of the form q^{-n} with $n \in \mathbb{N}$. The additional factor $\left[(-1)^k\, q^{\binom{k}{2}} \right]^{1+s-r}$ (which does not occur in the corresponding definition of the *generalized hypergeometric function*) is due to a *confluence process*. With this factor one gets the simple formula:

$$ \lim_{a_r \to \infty} {}_r\phi_s \left[\begin{array}{c} a_1, a_2, \ldots, a_r \\ b_1, b_2, \ldots, b_s \end{array} \middle| q, z \right] = {}_{r-1}\phi_s \left[\begin{array}{c} a_1, a_2, \ldots, a_{r-1} \\ b_1, b_2, \ldots, b_s \end{array} \middle| q, z \right]. $$

Another variation is the *bilateral basic hypergeometric series* (see e. g. [GR90], Chapter 5) that is defined as

$$ {}_r\psi_s \left[\begin{array}{c} a_1, a_2, \ldots, a_r \\ b_1, b_2, \ldots, b_s \end{array} \middle| q, z \right] := \sum_{k=-\infty}^{\infty} \frac{(a_1, a_2, \ldots, a_r;\, q)_k}{(b_1, b_2, \ldots, b_s;\, q)_k} z^k \left[(-1)^k\, q^{\binom{k}{2}} \right]^{s-r}. $$

The *summands* of those generalized q-hypergeometric series may be entered by

- `qphihyperterm(a1,a2,...,a3,b1,b2,...,b3,q,z,k)` and
- `qpsihyperterm(a1,a2,...,a3,b1,b2,...,b3,q,z,k)`

respectively.

20.46.3 *q*-Gosper Algorithm

The q-Gosper algorithm[Koo93] is a *decision procedure*, that decides by algebraic calculations whether or not a given q-hypergeometric term a_k has a q-hypergeometric term antidifference g_k, i. e. $a_k = g_k - g_{k-1}$ with g_k/g_{k-1} rational in q^k. The ratio g_k/a_k is also rational in q^k — an important fact which makes the

rational certification (see § 20.46.4) of Zeilberger's algorithm possible. If the procedure is successful it returns g_k, in which case we call a_k *q-Gosper-summable*. Otherwise *no q-hypergeometric antidifference exists*. Therefore if the q-Gosper algorithm does not return a q-hypergeometric antidifference, it has *proved* that no such solution exists, an information that may be quite useful and important.

Any antidifference is uniquely determined up to a constant, and is denoted by

$$g_k = \sum a_k \, \delta_k \ .$$

Finding g_k given a_k is called *indefinite summation*. The antidifference operator Σ is the inverse of the downward difference operator $\nabla a_k = a_k - a_{k-1}$. There is an analogous summation theory corresponding to the upward difference operator $\Delta a_k = a_{k+1} - a_k$.

In case, an antidifference g_k of a_k is known, any sum $\sum_{k=m}^{n} a_k$ can be easily calculated by an evaluation of g at the boundary points like in the integration case:

$$\sum_{k=m}^{n} a_k = g_n - g_{m-1}$$

20.46.4 *q*-Zeilberger Algorithm

The q-Zeilberger algorithm [Koo93] deals with the *definite summation* of q-hypergeometric terms $\mathrm{f}(n,k)$ wrt. n and k:

$$\mathrm{s}(n) := \sum_{k=-\infty}^{\infty} \mathrm{f}(n,k)$$

Zeilberger's idea is to use Gosper's algorithm to find an inhomogeneous recurrence equation with polynomial coefficients for $\mathrm{f}(n,k)$ of the form

$$\sum_{j=0}^{J} \sigma_j(n) \cdot \mathrm{f}(n+j,k) = \mathrm{g}(k) - \mathrm{g}(k-1), \tag{20.96}$$

where $\mathrm{g}(k)/\mathrm{f}(k)$ is rational in q^k and q^n. Assuming finite support of $\mathrm{f}(n,k)$ wrt. k (i. e. $\mathrm{f}(n,k) = 0$ for any n and all sufficiently large k) we can sum equation (20.96) over all $k \in \mathbb{Z}$. Thus we receive a homogeneous recurrence equation with polynomial coefficients (called *holonomic equation*) for $\mathrm{s}(n)$:

$$\sum_{j=0}^{J} \sigma_j(n) \cdot \mathrm{s}(n+j) = 0 \tag{20.97}$$

At this stage the implementation assumes that the summation bounds are infinite and the input term has finite support wrt. k. If those input requirements are not

fulfilled the resulting recursion is probably not valid. Thus we strongly advise the user to check those requirements.

Despite this restriction you may still be able to get valuable information by the program: On request it returns the left hand side of the recurrence equation (20.97) *and* the antidifference $\mathrm{g}(k)$ of equation (20.96).

Once you have the certificate $\mathrm{g}(k)$ it is trivial (at least theoretically) to prove equation (20.97) as long as the input requirements are fulfilled. Let's assume somone gives us equation (20.96). If we divide it by $\mathrm{f}(n,k)$ we get a rational identity (in q^n and q^k) —due to the fact that $\mathrm{g}(k)/\mathrm{f}(n,k)$ is rational in q^n and q^k. Once we confirmed this identity we sum equation (20.96) over $k \in \mathbb{Z}$:

$$\sum_{k\in\mathbb{Z}} \sum_{j=0}^{J} \sigma_j(n) \cdot \mathrm{f}(n+j,k) = \sum_{k\in\mathbb{Z}} (\mathrm{g}(k) - \mathrm{g}(k-1)), \tag{20.98}$$

Again we exploit the fact that $\mathrm{g}(k)$ is a rational multiple of $\mathrm{f}(n,k)$ and thus $\mathrm{g}(k)$ has *finite support* which makes the telescoping sum on the right hand side vanish. If we exchange the order of summation we get equation (20.97) which finishes the proof.

Note that we may relax the requirements for $\mathrm{f}(n,k)$: An infinite support is possible as long as $\lim_{k\to\infty} \mathrm{g}(k) = 0$. (This is certainly true if $\lim_{k\to\infty} \mathrm{p}(k)\,\mathrm{f}(k) = 0$ for all polynomials $\mathrm{p}(k)$.)

For a quite general class of q-hypergeometric terms (*proper q-hypergeometric terms*) the q-Zeilberger algorithm always finds a recurrence equation, not necessarily of lowest order though. Unlike Zeilberger's original algorithm its q-analogue more often fails to determine the recursion of lowest possible order, however (see [PR95]).

If the resulting recurrence equation is of first order

$$\mathrm{a}(n)\,\mathrm{s}(n-1) + \mathrm{b}(n)\,\mathrm{s}(n) = 0\ ,$$

$\mathrm{s}(n)$ turns out to be a q-hypergeometric term (as a and b are polynomials in q^n), and a q-hypergeometric solution can be easily established using a suitable initial value.

If the resulting recurrence equation has order larger than one, this information can be used for identification purposes: Any other expression satisfying the same recurrence equation, and the same initial values, represents the same function.

Our implementation is mainly based on [Koo93] and on the hypergeometric analogue described in [Koe95a]. More examples can be found in [GR90], [Gas95], some of which are contained in the test file `qsum.tst`.

20.46.5 REDUCE operator `qgosper`

The `qgosper` operator is an implementation of the q-Gosper algorithm.

- `qgosper(a,q,k)` determines a q-hypergeometric antidifference. (By default it returns a *downward* antidifference, which may be changed by the switch `qgosper_down`; see also § 20.46.8.) If it does not return a q-hypergeometric antidifference, then such an antidifference does not exist.
- `qgosper(a,q,k,m,n)` determines a closed formula for the definite sum $\sum_{k=m}^{n} a_k$ using the q-analogue of Gosper's algorithm. This is only successful if q-Gosper's algorithm applies.

Examples: The following two examples can be found in [GR90] ((II.3) and (2.3.4)).

```
1: qgosper(qpochhammer(a,q,k)*q^k/qpochhammer(q,q,k),q,k);

   k
 (q *a - 1)*qpochhammer(a,q,k)
------------------------------
  (a - 1)*qpochhammer(q,q,k)

2: qgosper(qpochhammer(a,q,k)*qpochhammer(a*q^2,q^2,k)*
   qpochhammer(q^(-n),q,k)*q^(n*k)/(qpochhammer(a,q^2,k)*
   qpochhammer(a*q^(n+1),q,k)*qpochhammer(q,q,k)),q,k);

    k*n   k          k    n
( - q   *(q *a - 1)*(q  - q )

               1                          2  2
 *qpochhammer(----,q,k)*qpochhammer(a*q ,q ,k)
               n
              q

                          2*k          n
 *qpochhammer(a,q,k))/((q   *a - 1)*(q  - 1)

                  n                           2
   *qpochhammer(q *a*q,q,k)*qpochhammer(a,q ,k)

   *qpochhammer(q,q,k))
```

Here are some other simple examples:

```
3: qgosper(qpochhammer(q^(-n),q,k)*z^k
                    /qpochhammer(q,q,k),q,k);
```

```
***** No q-hypergeometric antidifference exists.

4: off qgosper_down;

5: qgosper(q^k*qbrackets(k,q),q,k);

     k           k
  - q *(q + 1 - q )*qbrackets(k,q)
-----------------------------------
       k
     (q  - 1)*(q + 1)*(q - 1)

6: on qgosper_down;

7: qgosper(q^k,q,k,0,n);

  n
 q *q - 1
----------
  q - 1
```

20.46.6 REDUCE operator `qsumrecursion`

The `qsumrecursion` operator is an implementation of the q-Zeilberger algorithm. It tries to determine a homogeneous recurrence equation for $summ(n)$ wrt. n with polynomial coefficients (in n), where

$$summ(n) := \sum_{k=-\infty}^{\infty} \mathrm{f}(n,k).$$

If successful the left hand side of the recurrence equation (20.97) is returned.

There are three different ways to pass a summand $\mathrm{f}(n,k)$ to `qsumrecursion`:

- `qsumrecursion(f,q,k,n)`, where `f` is a q-hypergeometric term wrt. `k` and `n`, `k` is the summation variable and `n` the recursion variable, `q` is a symbol.
- `qsumrecursion(upper,lower,q,z,n)` is a shortcut for
  `qsumrecursion(qphihyperterm(upper,lower,q,z,k),`
  `q,k,n)`
- `qsumrecursion(f,upper,lower,q,z,n)` is a similar shortcut for
  `qsumrecursion(f*qphihyperterm(upper,lower,q,z,k),`
  `q,k,n),`

i. e. `upper` and `lower` are lists of upper and lower parameters of the generalized q-hypergeometric function. The third form is handy if you have any additional factors.

For all three instances the following variations are allowed:

- If for some reason the recursion order is known in advance you can specify it as an additional (*optional*) argument at the very end of the parameter sequence. There are two ways. If you just specify a positive integer, `qsumrecursion` looks only for a recurrence equation of this order. You can also specify a range by a list of two positive integers, i. e. the first one specifying the lowest and the second one the highest order.

  By default `qsumrecursion` will search for recurrences of order from 1 to 5. (The global variable `qsumrecursion_recrange!*` controls this behavior, see § 20.46.8.)

- Usually `qsumrecursion` uses `summ` as a name for the summ-function defined above. If you want to use another operator, say e. g. `s`, then the following syntax applies: `qsumrecursion(f,q,k,s(n))`

As a first example we want to consider the *q-binomial theorem*:

$$\sum_{k=0}^{\infty} \frac{(a;\,q)_k}{(q;\,q)_k} z^k = \frac{(a\,z;\,q)_\infty}{(z;\,q)_\infty},$$

provided that $|z|, |q| < 1$. It is the q-analogue of the binomial theorem in the sense that

$$\lim_{q\to 1^-} \sum_{k=0}^{\infty} \frac{(q^a;\,q)_k}{(q;\,q)_k} z^k \;=\; \sum_{k=0}^{\infty} \frac{(a)_k}{k!} z^k \;=\; (1-z)^{-a}\,.$$

For $a := q^{-n}$ with $n \in \mathbb{N}$ our implementation gets:

```
8: qsumrecursion(qpochhammer(q^(-n),q,k)*z^k/
   qpochhammer(q,q,k),q,k,n);

      n                          n
 - ((q  - z)*summ(n - 1) - q *summ(n))
```

Notice that the input requirements are fulfilled. For $n \in \mathbb{N}$ the summand is zero for all $k > n$ as $(q^{-n};\,q)_k = 0$ and the $(q;\,q)_k$-term in the denominator makes the summand vanish for all $k < 0$.

With the switch `qsumrecursion_certificate` it is possible to get the antidifference g_k described above. When switched on, `qsumrecursion` returns a list with five entries, see § 20.46.8. For the last example we get:

```
9: on qsumrecursion_certificate;

10: proof:= qsumrecursion(qpochhammer(q^(-n),q,k)*z^k/
    qpochhammer(q,q,k),q,k,n);

                n                          n
proof :=  - ((q  - z)*summ(n - 1) - q *summ(n)),

              k    n
            - (q  - q )*z
           ----------------,
                 n
                q  - 1

             k              1
            z *qpochhammer(----,q,k)
                            n
                           q
           --------------------------,
               qpochhammer(q,q,k)

           k,

           downward_antidifference

11: off qsumrecursion_certificate;
```

Let's define the list entries as `{rec,cert,f,k,dir}`. If you substitute $summ(n+j)$ by $\mathrm{f}(n+j,k)$ in `rec` then you obtain the left hand side of equation (20.96), where `f` is the input summand. The function $\mathrm{g}(k) := \mathtt{f*cert}$ is the corresponding antidifference, where `dir` states which sort of antidifference was calculated `downward_antidifference` or `upward_antidifference`, see also § 20.46.8. Those informations enable you to prove the recurrence equation for the sum or supply you with the necessary informations to determine an inhomogeneous recurrence equation for a sum with nonnatural bounds.

For our last example we can now calculate both sides of equation (20.96):

```
12: lhside:= qsimpcomb(sub(summ(n)=part(proof,3),
    summ(n-1)=sub(n=n-1,part(proof,3)),part(proof,1)));

            k   k   n          n
lhside := (z *(q *(q  - z) + q *(z - 1))

                        1            n
          *qpochhammer(----,q,k))/((q  - 1)
                       n
                      q
```

```
              *qpochhammer(q,q,k))

13: rhside:= qsimpcomb((part(proof,2)*part(proof,3)-
    sub(k=k-1,part(proof,2)*part(proof,3))));

                k    k    n       n   k
rhside := ( - z *((q  - q )*z - q *(q  - 1))

                          1            n
            *qpochhammer(----,q,k))/((q  - 1)
                          n
                         q

              *qpochhammer(q,q,k))

14: qsimpcomb((rhside-lhside)/part(proof,3));

0
```

Thus we have proved the validity of the recurrence equation.

As some other examples we want to consider some generalizations of orthogonal polynomials from the Askey–Wilson–scheme [KS94]: The q-Laguerre (3.21), q-Charlier (3.23) and the continuous q-Jacobi (3.10) polynomials.

```
15: operator qlaguerre,qcharlier;

16: qsumrecursion(qpochhammer(q^(alpha+1),q,n)/
           qpochhammer(q,q,n),
           {q^(-n)}, {q^(alpha+1)}, q, -x*q^(n+alpha+1),
           qlaguerre(n));

           n       alpha + n   n
((q + 1 - q )*q - q          *(q *x + q))

*qlaguerre(n - 1) + (

     alpha + n
   (q          - q)*qlaguerre(n - 2)

        n
    + (q  - 1)*qlaguerre(n))*q

17: qsumrecursion({q^(-n),q^(-x)},{0},q,-q^(n+1)/a,
           qcharlier(n));

      x            n       n       2*n
 - ((q *((q + 1 - q )*a + q )*q - q   )
```

```
                          x
    *qcharlier(n - 1) + q *(

         n          n
       (q  + a*q)*(q  - q)*qcharlier(n - 2)

        - qcharlier(n)*a*q))

18: on qsum_nullspace;

19: term:= qpochhammer(q^(alpha+1),q,n)/qpochhammer(q,q,n)*
       qphihyperterm({q^(-n),q^(n+alpha+beta+1),
                      q^(alpha/2+1/4)*exp(I*theta),
                      q^(alpha/2+1/4)*exp(-I*theta)},
                     {q^(alpha+1),
                      -q^((alpha+beta+1)/2),
                      -q^((alpha+beta+2)/2)},
       q,q,k)$

20: qsumrecursion(term,q,k,n,2);
```

```
      n  i*theta   alpha   beta   n
 - ((q *e       *(q     *(q    *(q *(q + 1) - q)

                   alpha + beta + n


                       n    beta + n
          *(q + 1 - q  - q        )) -

         (alpha + beta)/2   alpha   n
        q                *(q     *(q *(q + 1) - q

                  beta + n           n
              + q        *(q + 1 - q ))

                2*alpha + beta + 2*n
           - (q                     + q)))

                      (2*alpha + 1)/4
    *(sqrt(q) + q) + q

       2*i*theta        alpha + beta + 2*n    2
    *(e          + 1)*(q                   - q )

       alpha + beta + 2*n
    *(q                  - 1))

      alpha + beta + 2*n
```

```
*(q                    - q)*summ(n - 1) -

 i*theta    (alpha + beta + 2*n)/2
e       *((q

              (alpha + beta + 2*n)/2
            *(q                      + q)

              (alpha + beta + 2*n)/2
            *(q                      - q)

            *(sqrt(q) + q) + (

               (2*alpha + 2*beta + 4*n + 1)/2
             q

                       alpha + beta + 2*n    2
              + q)*(q                    - q )

              alpha + beta + n        n
            )*(q                - 1)*(q  - 1)

                         alpha
          *summ(n) + (q     *(sqrt(q)*q

                     alpha + beta + 2*n
                  + q                  ) +

              (3*alpha + beta + 2*n)/2
            q

            *(sqrt(q) + q))

            alpha + beta + 2*n
          *(q                    - 1)

            alpha + n        beta + n
          *(q          - q)*(q         - q)

          *summ(n - 2)))
```

```
21: off qsum_nullspace;
```

The setting of `qsum_nullspace` (see [PR95] and § 20.46.8) results in a faster calculation of the recurrence equation for this example.

20.46.7 Simplification Operators

An essential step in the algorithms introduced above is to decide whether a term a_k is q-hypergeometric, i. e. if the ratio a_k/a_{k-1} is rational in q^k.

The procedure qsimpcomb provides this facility. It tries to simplify all exponential expressions in the given term and applies some transformation rules to the known elementary q-functions as qpochhammer, qbrackets, qbinomial and qfactorial. Note that the procedure may fail to completely simplify some expressions. This is due to the fact that the procedure was designed to simplify ratios of q-hypergeometric terms in the form $\mathrm{f}(k)/\mathrm{f}(k-1)$ and not arbitrary q-hypergeometric terms.

E. g. an expression like $(a;\,q)_{-n} \cdot (a/q^n;\,q)_n$ is not recognized as 1, despite the transformation formula

$$(a;\,q)_{-n} \;=\; \frac{1}{(a/q^n;\,q)_n},$$

which is valid for $n \in \mathbb{N}$.

Note that due to necessary simplification of powers, the switch precise is (locally) turned off in qsimpcomb. This might produce wrong results if the input term contains e. g. complex variables.

The following synomyms may be used:

- up_qratio(f,k) or qratio(f,k) for qsimpcomb(sub(k=k+1,f)/f) and
- down_qratio(f,k) for qsimpcomp(f/sub(k=k-1,f)).

20.46.8 Global Variables and Switches

The following switches can be used in connection with the QSUM package:

- qsum_trace, default setting is off. If it is turned on some intermediate results are printed.
- qgosper_down, default setting is on. It determines whether qgosper returns a downward or an upward antidifference g_k for the input term a_k, i. e. $a_k = g_k - g_{k-1}$ or $a_k = g_{k+1} - g_k$ respectively.
- qsumrecursion_down, default setting is on. If it is switched on a downward recurrence equation will be returned by qsumrecursion. Switching it off leads to an upward recurrence equation.
- qsum_nullspace, default setting is off. The antidifference $\mathrm{g}(k)$ is always a rational multiple (in q^k) of the input term $\mathrm{f}(k)$. qgosper and qsumrecursion determine this certificate, which requires solving a set of linear equations. If the switch qsum_nullspace is turned on a modified nullspace-algorithm will be used for solving those equations. In general this

method is slower. However if the resulting recurrence equation is quite complicated it might help to switch on `qsum_nullspace`. See also [Knu81] and [PR95].

- `qgosper_specialsol`, default setting is on. The antidifference $g(k)$ which is determined by `qgosper` might not be unique. If this switch is turned on, just one special solution is returned. If you want to see all solutions, you should turn the switch off.

- `qsumrecursion_exp`, default setting is off. This switch determines if the coefficients of the resulting recurrence equation should be factored. Turning it off might speed up the calculation (if factoring is complicated). Note that when turning on `qsum_nullspace` usually no speedup occurs by switching `qsumrecursion_exp` on.

- `qsumrecursion_certificate`, default off. As Zeilberger's algorithm delivers a recurrence equation for a q-hypergeometric term $\mathrm{f}(n,k)$, see equation (20.96), this switch is used to get all necessary informations for proving this recurrence equation.

  If it is set on, instead of simply returning the resulting recurrence equation (for the sum)—if one exists—calling `qsumrecursion` returns a list `{rec,cert,f,k,dir}` with five items: The first entry contains the recurrence equation, while the other items enable you to prove the recurrence a posteriori by rational arithmetic.

  If we denote by `r` the recurrence `rec` where we substituted the `summ`-function by the input term
  texttttf (with the corresponding shifts in `n`) then the following equation is valid:

  ```
  r = cert*f - sub(k=k-1,cert*f)
  ```

  if `dir=downward_antidifference` or

  ```
  r = sub(k=k+1,cert*f) - cert*f
  ```

  if `dir=upward_antidifference`.

The global variable `qsumrecursion_recrange!*` controls for which recursion orders the procedure `qsumrecursion` looks. It has to be a list with two entries, the first one representing the lowest and the second one the highest order of a recursion to search for. By default it is set to `{1,5}`.

20.46.9 Messages

The following messages may occur:

- If your call to `qgosper` or `qsumrecursion` reveals some incorrect syntax, e. g. wrong number of arguments or wrong type you may receive the following messages:

  ```
  ***** Wrong number of arguments.
  ```

  or

  ```
  ***** Wrong type of arguments.
  ```

- If you call `qgosper` with a summand term that is free of the summation variable you get

  ```
  WARNING: Summand is independent of summation variable.
  ***** No q-hypergeometric antidifference exists.
  ```

- If `qgosper` finds no antidifference it returns:

  ```
  ***** No q-hypergeometric antidifference exists.
  ```

- If `qsumrecursion` finds no recursion in the specified range it returns:

  ```
  ***** Found no recursion. Use higher order.
  ```

  (If you do not pass a range as an argument to `qsumrecursion` the default range in `qsumrecursion_recrange!*` will be used.)

- If the input term passed to `qgosper` (`qsumrecursion`) is not q-hypergeometric wrt. the summation variable — say k — (and the recursion variable) then you get

  ```
  ***** Input term is probably not q-hypergeometric.
  ```

  With all the examples we tested, our procedures decided properly whether the input term was q-hypergeometric or not. However, we cannot guarantee in general that `qsimpcomb` *always* returns an expression that *looks* rational in q^k if it actually is.

- If the global variable `qsumrecursion_recrange!*` was assigned an invalid value:

  ```
  Global variable qsumrecursion_recrange!* must be a
  list of two positive integers: {lo,hi} with lo<=hi.
  ***** Invalid value of qsumrecursion_recrange!*
  ```

20.47 RANDPOLY: A Random Polynomial Generator

This package is based on a port of the Maple random polynomial generator together with some support facilities for the generation of random numbers and anonymous procedures.

Author: Francis Wright

This package is based on a port of the Maple random polynomial generator together with some support facilities for the generation of random numbers and anonymous procedures.

20.47.1 Introduction

The operator `randpoly` is based on a port of the Maple random polynomial generator. In fact, although by default it generates a univariate or multivariate polynomial, in its most general form it generates a sum of products of arbitrary integer powers of the variables multiplied by arbitrary coefficient expressions, in which the variable powers and coefficient expressions are the results of calling user-supplied functions (with no arguments). Moreover, the "variables" can be arbitrary expressions, which are composed with the underlying polynomial-like function.

The user interface, code structure and algorithms used are essentially identical to those in the Maple version. The package also provides an analogue of the Maple `rand` random-number-generator generator, primarily for use by `randpoly`. There are principally two reasons for translating these facilities rather than designing comparable facilites anew: (1) the Maple design seems satisfactory and has already been "proven" within Maple, so there is no good reason to repeat the design effort; (2) the main use for these facilities is in testing the performance of other algebraic code, and there is an advantage in having essentially the same test data generator implemented in both Maple and REDUCE . Moreover, it is interesting to see the extent to which a facility can be translated without change between two systems. (This aspect will be described elsewhere.)

Sections 20.47.2 and 20.47.3 describe respectively basic and more advanced use of `randpoly`; §20.47.4 describes subsidiary functions provided to support advanced use of `randpoly`; §20.47.5 gives examples; an appendix gives some details of the only non-trivial algorithm, that used to compute random sparse polynomials. Additional examples of the use of `randpoly` are given in the test and demonstration file `randpoly.tst`.

20.47.2 Basic use of `randpoly`

The operator `randpoly` requires at least one argument corresponding to the polynomial variable or variables, which must be either a single expression or a list of expressions.[45] In effect, `randpoly` replaces each input expression by an internal variable and then substitutes the input expression for the internal variable in the generated polynomial (and by default expands the result as usual), although in fact if the input expression is a REDUCE kernel then it is used directly. The rest of this document uses the term "variable" to refer to a general input expression or the internal variable used to represent it, and all references to the polynomial structure, such as its degree, are with respect to these internal variables. The actual degree of a generated polynomial might be different from its degree in the internal variables.

By default, the polynomial generated has degree 5 and contains 6 terms. Therefore, if it is univariate it is dense whereas if it is multivariate it is sparse.

20.47.2.1 Optional arguments

Other arguments can optionally be specified, in any order, after the first compulsory variable argument. All arguments receive full algebraic evaluation, subject to the current switch settings etc. The arguments are processed in the order given, so that if more than one argument relates to the same property then the last one specified takes effect. Optional arguments are either keywords or equations with keywords on the left.

In general, the polynomial is sparse by default, unless the keyword **dense** is specified as an optional argument. (The keyword **sparse** is also accepted, but is the default.) The default degree can be changed by specifying an optional argument of the form

`degree` = ⟨*natural number*⟩ .

In the multivariate case this is the total degree, i.e. the sum of the degrees with respect to the individual variables. The keywords **deg** and **maxdeg** can also be used in place of **degree**. More complicated monomial degree bounds can be constructed by using the coefficient function described below to return a monomial or polynomial coefficient expression. Moreover, `randpoly` respects internally the REDUCE "asymptotic" commands `let`, `weight` etc. described in §11.4 of the REDUCE manual, which can be used to exercise additional control over the polynomial generated.

[45] If it is a single expression then the univariate code is invoked; if it is a list then the multivariate code is invoked, and in the special case of a list of one element the multivariate code is invoked to generate a univariate polynomial, but the result should be indistinguishable from that resulting from specifying a single expression not in a list.

In the sparse case (only), the default maximum number of terms generated can be changed by specifying an optional argument of the form

`terms` = ⟨*natural number*⟩ .

The actual number of terms generated will be the minimum of the value of `terms` and the number of terms in a dense polynomial of the specified degree, number of variables, etc.

20.47.3 Advanced use of `randpoly`

The default order (or minimum or trailing degree) can be changed by specifying an optional argument of the form

`ord` = ⟨*natural number*⟩ .

The keyword is `ord` rather than `order` because `order` is a reserved command name in REDUCE . The keyword `mindeg` can also be used in place of `ord`. In the multivariate case this is the total degree, i.e. the sum of the degrees with respect to the individual variables. The order normally defaults to 0.

However, the input expressions to `randpoly` can also be equations, in which case the order defaults to 1 rather than 0. Input equations are converted to the difference of their two sides before being substituted into the generated polynomial. The purpose of this facility is to easily generate polynomials with a specified zero – for example

```
randpoly(x = a);
```

generates a polynomial that is guaranteed to vanish at $x = a$, but is otherwise random.

Order specification and equation input are extensions of the current Maple version of `randpoly`.

The operator `randpoly` accepts two further optional arguments in the form of equations with the keywords `coeffs` and `expons` on the left. The right sides of each of these equations must evaluate to objects that can be applied as functions of no variables. These functions should be normal algebraic procedures (or something equivalent); the `coeffs` procedure may return any algebraic expression, but the `expons` procedure must return an integer (otherwise `randpoly` reports an error). The values returned by the functions should normally be random, because it is the randomness of the coefficients and, in the sparse case, of the exponents that makes the constructed polynomial random.

A convenient special case is to use the function `rand` on the right of one or both of these equations; when called with a single argument `rand` returns an anonymous function of no variables that generates a random integer. The single argument of `rand` should normally be an integer range in the form a .. b, where a, b are integers such that $a < b$. The spaces around (or at least before) the infix operator ".." are necessary in some cases in REDUCE and generally recommended. For example, the `expons` argument might take the form

```
expons = rand(0 .. n)
```

where `n` will be the maximum degree with respect to each variable *independently*. In the case of `coeffs` the lower limit will often be the negative of the upper limit to give a balanced coefficient range, so that the `coeffs` argument might take the form

```
coeffs = rand(-n .. n)
```

which will generate random integer coefficients in the range $[-n, n]$.

20.47.4 Subsidiary functions: rand, proc, random

20.47.4.1 Rand: a random-number-generator generator

The first argument of `rand` must be either an integer range in the form a .. b, where a, b are integers such that $a < b$, or a positive integer n which is equivalent to the range 0 .. $n - 1$. The operator `rand` constructs a function of no arguments that calls the REDUCE random number generator function `random` to return a random integer in the range specified; in the case that the first argument of `rand` is a single positive integer n the function constructed just calls `random(`n`)`, otherwise the call of `random` is scaled and shifted.

As an additional convenience, if `rand` is called with a second argument that is an identifier then the call of `rand` acts exactly like a procedure definition with the identifier as the procedure name. The procedure generated can then be called with an empty argument list by the algebraic processor.

[Note that `rand()` with no argument is an error in REDUCE and does not return directly a random number in a default range as it does in Maple – use instead the REDUCE function `random` (see below).]

20.47.4.2 Proc: an anonymous procedure generator

The operator `proc` provides a generalization of `rand`, and is primarily intended to be used with expressions involving the `random` function (see below). Essentially,

it provides a mechanism to prevent functions such as `random` being evaluated when the arguments to `randpoly` are evaluated, which is too early. The operator `proc` accepts a single argument which is converted into the body of an anonymous procedure, which is returned as the value of `proc`. (If a named procedure is required then the normal REDUCE `procedure` statement should be used instead.) Examples are given in the following sections, and in the file `randpoly.tst`.

20.47.4.3 Random: a generalized interface

As an additional convenience, this package extends the interface to the standard REDUCE `random` function so that it will directly accept either a natural number or an integer range as its argument, exactly as for the first argument of `rand`. Hence effectively

```
rand(X) = proc random(X)
```

although `rand` is marginally more efficient. However, `proc` and the generalized `random` interface allow expressions such as the following anonymous random fraction generator to be easily constructed:

```
proc(random(-99 .. 99)/random(1 .. 99))
```

20.47.4.4 Further support for procs

`Rand` is a special case of `proc`, and (for either) if the switch `comp` is `on` (and the compiler is available) then the generated procedure body is compiled.

`Rand` with a single argument and `proc` both return as their values anonymous procedures, which if they are not compiled are Lisp lambda expressions. However, if compilation is in effect then they return only an identifier that has no external significance[46] but which can be applied as a function in the same way as a lambda expression.

It is primarily intended that such "proc expressions" will be used immediately as input to `randpoly`. The algebraic processor is not intended to handle lambda expressions. However, they can be output or assigned to variables in algebraic mode, although the output form looks a little strange and is probably best not displayed. But beware that lambda expressions cannot be evaluated by the algebraic processor (at least, not without declaring some internal Lisp functions to be algebraic operators). Therefore, for testing purposes or curious users, this package provides the operators `showproc` and `evalproc` respectively to display and evaluate "proc expressions" output by `rand` or `proc` (or in fact any lambda expression), in the

[46] It is not interned on the oblist.

case of `showproc` provided they are not compiled.

20.47.5 Examples

The file `randpoly.tst` gives a set of test and demonstration examples.

The following additional examples were taken from the Maple `randpoly` help file and converted to REDUCE syntax by replacing [ ] by { } and making the other changes shown explicitly:

```
randpoly(x);

       5       4       3       2
 - 54*x  - 92*x  - 30*x  + 73*x  - 69*x - 67


randpoly({x, y}, terms = 20);

    5       4          4       3  2       3          3
31*x  - 17*x *y - 48*x  - 15*x *y  + 80*x *y + 92*x

       2  3      2          2          4          3
 + 86*x *y  + 2*x *y - 44*x  + 83*x*y  + 85*x*y

         2                         5       4      3
 + 55*x*y - 27*x*y + 33*x - 98*y  + 51*y  - 2*y

       2
 + 70*y  - 60*y - 10


randpoly({x, sin(x), cos(x)});

                    4             3              3
sin(x)*( - 4*cos(x)  - 85*cos(x) *x + 50*sin(x)

                   2
         - 20*sin(x) *x + 76*sin(x)*x + 96*sin(x))


% randpoly(z, expons = rand(-5..5));  % Maple
% A generalized random "polynomial"!
% Note that spaces are needed around .. in REDUCE.
on div; off allfac;
```

```
randpoly(z, expons = rand(-5 .. 5));

       4       3       -3       -4      -5
 - 39*z  + 14*z  - 77*z   - 37*z   - 8*z

off div; on allfac;
% Maple
% randpoly([x], coeffs = proc() randpoly(y) end);
randpoly({x}, coeffs = proc randpoly(y));

    5  5       5  4       5  3       5  2       5
95*x *y  - 53*x *y  - 78*x *y  + 69*x *y  + 58*x *y

       5       4  5       4  4       4  3       4  2
 - 58*x  + 64*x *y  + 93*x *y  - 21*x *y  + 24*x *y

       4         4       3  5       3  4       3  3
 - 13*x *y - 28*x  - 57*x *y  - 78*x *y  - 44*x *y

       3  2       3         3       2  5       2  4
 + 37*x *y  - 64*x *y - 95*x  - 71*x *y  - 69*x *y

    2  3       2  2       2         2         5
 - x *y  - 49*x *y  + 77*x *y + 48*x  + 38*x*y

         4         3         2                         5
 + 93*x*y  - 65*x*y  - 83*x*y  + 25*x*y + 51*x + 35*y

       4       3       2
 - 18*y  - 59*y  + 73*y  - y + 31

% A more conventional alternative is ...
%   procedure r; randpoly(y)$
%   randpoly({x}, coeffs = r);
% or, in fact, equivalently ...
%   randpoly({x}, coeffs = procedure r; randpoly(y));

randpoly({x, y}, dense);

    5       4         4       3  2       3         3
85*x  + 43*x *y + 68*x  + 87*x *y  - 93*x *y - 20*x

       2  2       2        2         4         3
 - 74*x *y  - 29*x *y + 7*x  + 10*x*y  + 62*x*y
```

```
         2                          5       4       3
 - 86*x*y  + 15*x*y - 97*x - 53*y  + 71*y  - 46*y

       2
 - 28*y  + 79*y + 44
```

20.47.6 Appendix: Algorithmic background

The only part of this package that involves any mathematics that is not completely trivial is the procedure to generate a sparse set of monomials of specified maximum and minimum total degrees in a specified set of variables. This involves some combinatorics, and the Maple implementation calls some procedures from the Maple Combinatorial Functions Package `combinat` (of which I have implemented restricted versions in REDUCE).

Given the maximum possible number N of terms (in a dense polynomial), the required number of terms (in the sparse polynomial) is selected as a random subset of the natural numbers up to N, where each number indexes a term. In the univariate case these indices are used directly as monomial exponents, but in the multivariate case they are converted to monomial exponent vectors using a lexicographic ordering.

20.47.6.1 Numbers of polynomial terms

By explicitly enumerating cases with 1, 2, etc. variables, as indicated by the inductive proof below, one deduces that:

Proposition 1. *In n variables, the number of distinct monomials having total degree precisely r is ${}^{r+n-1}C_{n-1}$, and the maximum number of distinct monomials in a polynomial of maximum total degree d is ${}^{d+n}C_n$.*

Proof Suppose the first part of the proposition is true, namely that there are at most

$$N_h(n,r) = {}^{r+n-1}C_{n-1}$$

distinct monomials in an n-variable *homogeneous* polynomial of total degree r. Then there are at most

$$N(d,r) = \sum_{r=0}^{d} {}^{r+n-1}C_{n-1} = {}^{d+n}C_n$$

distinct monomials in an n-variable polynomial of maximum total degree d.

The sum follows from the fact that

$$^{r+n}C_n = \frac{(r+n)^{\underline{n}}}{n!}$$

where $x^{\underline{n}} = x(x-1)(x-2)\cdots(x-n+1)$ denotes a falling factorial, and

$$\sum_{a \le x < b} x^{\underline{n}} = \left.\frac{x^{\underline{n+1}}}{n+1}\right|_a^b .$$

(See, for example [GK82, equation (1.37)]. Hence the second part of the proposition follows from the first.

The proposition holds for 1 variable ($n = 1$), because there is clearly 1 distinct monomial of each degree precisely r and hence at most $d+1$ distinct monomials in a polynomial of maximum degree d.

Suppose that the proposition holds for n variables, which are represented by the vector X. Then a homogeneous polynomial of degree r in the $n+1$ variables X together with the single variable x has the form

$$x^r P_0(X) + x^{r-1} P_1(X) + \cdots + x^0 P_r(X)$$

where $P_s(X)$ represents a polynomial of maximum total degree s in the n variables X, which therefore contains at most $^{s+n}C_n$ distinct monomials. The homogeneous polynomial of degree r in $n+1$ terms therefore contains at most

$$\sum_{s=0}^{r} {}^{s+n}C_n = {}^{r+n+1}C_{n+1}$$

distinct monomials. Hence the proposition holds for $n+1$ variables, and therefore by induction it holds for all n. □

20.47.6.2 Mapping indices to exponent vectors

The previous proposition is also the basis of the algorithm to map term indices $m \in \mathbb{N}$ to exponent vectors $v \in \mathbb{N}^n$, where n is the number of variables.

Define a norm $\|\cdot\|$ on exponent vectors by $\|v\| = \sum_{i=1}^{n} v_i$, which corresponds to the total degree of the monomial. Then, from the previous proposition, the number of exponent vectors of length n with norm $\|v\| \le d$ is $N(n,d) = {}^{d+n}C_n$. The elements of the m^{th} exponent vector are constructed recursively by applying the algorithm to successive tail vectors, so let a subscript denote the length of the vector to which a symbol refers.

The aim is to compute the vector of length n with index $m = m_n$. If this vector has norm d_n then the index and norm must satisfy

$$N(n, d_n - 1) \le m_n < N(n, d_n),$$

which can be used (as explained below) to compute d_n given n and m_n. Since there are $N(n, d_n - 1)$ vectors with norm less than d_n, the index of the $(n-1)$-element tail vector must be given by $m_{n-1} = m_n - N(n, d_n - 1)$, which can be used recursively to compute the norm d_{n-1} of the tail vector. From this, the first element of the exponent vector is given by $v_1 = d_n - d_{n-1}$.

The algorithm therefore has a natural recursive structure that computes the norm of each tail subvector as the recursion stack is built up, but can only compute the first term of each tail subvector as the recursion stack is unwound. Hence, it constructs the exponent vector from right to left, whilst being applied to the elements from left to right. The recursion is terminated by the observation that $v_1 = d_1 = m_1$ for an exponent vector of length $n = 1$.

The main sub-procedure, given the required length n and index m_n of an exponent vector, must return its norm d_n and the index of its tail subvector of length $n - 1$. Within this procedure, $N(n, d)$ can be efficiently computed for values of d increasing from 0, for which $N(n, 0) = {}^nC_n = 1$, until $N(n, d) > m$ by using the observation that

$$N(n, d) = {}^{d+n}C_n = \frac{(d+n)(d-1+n)\cdots(1+n)}{d!}.$$

20.48 RATAPRX: Rational Approximations Package for REDUCE

Authors: Lisa Temme, Wolfram Koepf and Alan Barnes

20.48.1 Periodic Decimal Representation

The division of one integer by another often results in a period in the decimal part. The `rational2periodic` function in this package can recognise and represent such an answer in a periodic representation. The inverse function, `periodic2rational`, converts a periodic representation back to a rational number.

Periodic Representation of a Rational Number

SYNTAX: `rational2periodic(`⟨*n*⟩`)`
`rational2periodic(`⟨*n*⟩`,`⟨*b*⟩`)`

INPUT: ⟨*n*⟩ is a rational number
⟨*b*⟩ is the number base, if absent the default is 10.

RESULT: `periodic({`⟨*a1*⟩`,...,`⟨*an*⟩`},`
`{`⟨*b1*⟩`,...,`⟨*bm*⟩`},`
`{`⟨*c1*⟩`,...,`⟨*ck*⟩`},`
$\pm\langle b\rangle$)

where `{`⟨*a1*⟩`,...,`⟨*an*⟩`}` is a list of the digits in the integer part,
`{`⟨*b1*⟩`,...,`⟨*bm*⟩`}` is a list of the digits in the non-periodic part,
`{`⟨*c1*⟩`,...,`⟨*ck*⟩`}` is a list of the digits in the periodic part
and $\pm\langle b\rangle$ where $\langle b\rangle$ is the number base $2 \leq \mathrm{b} \leq 16$,
a minus indicating the rational number $\langle n\rangle$ was negative.

EXAMPLES:

$-59/70$ written as $-0.8\overline{428571}$

```
1: rational2periodic(-59/70);

periodic({0}, {8}, {4,2,8,5,7,1}, -10)}

2: rational2periodic(1/80,16);

periodic({0}, {0}, {3}, 16)
```

Normally the operator `periodic` will not be seen as the output will be prettyprinted as $-0.8\overline{428571}$ and $0.0\overline{3}$ (base 16) respectively.

Rational Number of a Periodic Representation

SYNTAX: `periodic2rational(`
`periodic({`⟨*a1*⟩,...,⟨*an*⟩`},`
`{`⟨*b1*⟩,...,⟨*bm*⟩`},`
`{`⟨*c1*⟩,...,⟨*ck*⟩`},`
`±`⟨*b*⟩`))`

`periodic2rational({`⟨*a1*⟩,...,⟨*an*⟩`},`
`{`⟨*b1*⟩,...,⟨*bm*⟩`},`
`{`⟨*c1*⟩,...,⟨*ck*⟩`},`
`±`⟨*b*⟩`)`

INPUT: `{`⟨*a1*⟩`,...,`⟨*an*⟩`}` is a list of the digits in the integer part,
`{`⟨*b1*⟩`,...,`⟨*bm*⟩`}` is a list of the digits in the non-periodic part,
`{`⟨*c1*⟩`,...,`⟨*ck*⟩`}` is a list of the digits in the periodic part
and where ⟨*b*⟩ is the number base $2 \leq \mathrm{b} \leq 16$, a minus
indicating the rational number result should be negative.
If the base is omitted, 10 is assumed.

RESULT: A rational number.

EXAMPLES:
$0.8\overline{428571}$ written as $59/70$

```
3: periodic2rational(
       periodic({0},{8},{4,2,8,5,7,1}));
 59
----
 70

4: periodic2rational(
       {0},{8},{4,2,8,5,7,1}, -10);

    59
 - ----
    70
```

Note that `periodic2rational` will produce the correct rational result when passed a parameter for the periodic part which is not minimal. Similarly, a parameter for the periodic part which consists of all 9's (or in base b, all $(b-1)$'s) is treated correctly although such periodic parts are not canonical and are never generated by calls to `rational2periodic`.

For example,

```
periodic2rational({0}, {}, {1, 2, 1, 2});
periodic2rational({0}, {1}, {2, 1});
periodic2rational({0}, {1, 2}, {1, 2, 1, 2});
```

all produce the same rational result, namely $\frac{4}{33}$, as the canonical input

```
periodic2rational({0}, {}, {1, 2});
```

Similarly,

```
periodic2rational({0}, {}, {9});
periodic2rational({0}, {9}, {9});
periodic2rational({0}, {}, {9, 9, 9, 9});
```

all produce the same rational result, namely 1, as the canonical input

```
periodic2rational({1}, {}, {});
```

Although the operators `periodic2rational` and `rational2periodic` work even when `ROUNDED` is `ON`, they are best used when `ROUNDED` is `OFF`. The input to `rational2periodic` should not be a rounded number, otherwise an error results.

For example, the input `rational2periodic(1/7);` will produce the intended periodic representation even with `ROUNDED ON`. However, the input

```
a := 1/7; rational2periodic(a);
```

will result in an error as the simplifier is applied in the assignment and rounds the rational number.

Similarly, although the result of `periodic2rational` will always be a rational number (represented by a `QUOTIENT` prefix form), if the simplifier is applied to the result a rounded value will be produced.

20.48.2 Continued Fractions

A continued fraction (see [JT80]) has the general form

$$a_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \ldots}}} \; .$$

A more compact way of writing this is as

$$a_0 + \frac{a_1|}{|b_1} + \frac{a_2|}{|b_2} + \frac{a_3|}{|b_3} + \ldots \; .$$

Even more succinctly:

$$\{a_0,\ \{a_1, b_1\},\ \{a_2, b_2\},\ \ldots\}$$

This is represented in REDUCE as

```
contfrac(Expression, Rational approximant,
                {a0,{a1,b1},{a2,b2},...})
```

The operator `cfrac` is used to generate a generalised continued fraction expansion of an algebraic expression.

`cfrac(`⟨*num*⟩`)`
`cfrac(`⟨*num*⟩`,` ⟨*length*⟩`)`
`cfrac(`⟨*func*⟩`,` ⟨*var*⟩`)`
`cfrac(`⟨*func*⟩`,` ⟨*var*⟩`,` ⟨*length*⟩`)`

INPUT:
⟨*num*⟩ is any real number
⟨*func*⟩ is a function
⟨*var*⟩ is the function main variable
⟨*length*⟩ is the maximum number of terms (continuents) to be generated and is **optional**.

For non-rational function or irrational number input the ⟨*length*⟩ argument specifies the number of continuents (ordered pairs, $\{a_i, b_i\}$), to be returned. Its default value is five. For rational function or rational number input the `length` argument can only truncate the answer, it cannot return additional pairs even if the precision is increased. The default for rational function or rational number input is the complete continued fraction.

For a non-rational function, power series expansion is necessary. The new switch `cf_taylor` controls whether the TAYLOR or the TPS package is used to produce the power series required. By default this switch is OFF and so the TPS package is normally employed. In most cases the choice is not important, but the TPS option is somewhat better at handling cases where the series expansion is rather sparse. In a few cases TPS may fail to produce a series expansion when TAYLOR succeeds and vice-versa.

For numerical input the default value is exact for rational number arguments whilst for irrational or rounded input it is dependent on the precision of the session. The `length` argument will only take effect if is smaller than the number of ordered pairs which the default value would return.

If the number of continuent pairs returned does not exceed twelve, the result will usually be pretty-printed as a two element list consisting of the convergent followed

by a rendering of the traditional continued fraction expansion. For a larger number of pairs the output is of the second element is printed as a list of pairs. Thus, usually the operator `contfrac` is not seen in the output.

EXAMPLES

```
cfrac(pi, 4);
```

```
     355             1
{pi,-----,3 + ----------------}
     113               1
               7 + ----------
                             1
                    15 + ---
                             1
```

```
cfrac(sqrt 2, 5);
```

```
          41                  1
{sqrt(2),----,1 + ---------------------}
          29                    1
                  2 + ---------------
                                  1
                       2 + ---------
                                    1
                            2 + ---
                                    2
```

```
cfrac(23.696, 4);
```

```
  2962   237               1
{------,-----,23 + ---------------}
  125    10                  1
                    1 + ---------
                                1
                         2 + ---
                                3
```

```
cfrac((x+2/3)^2/(6*x-5), x, 10);
```

```
     2
  9*x  + 12*x + 4
{-----------------,   exact,
     54*x - 45
```

```
   6*x + 13          1
---------- + ------------}
    36          24*x - 20
              -----------
                   9
```

```
cfrac(e^x, x);
```

```
       3      2
  x   x  + 9*x  + 36*x + 60
{e , ----------------------,
           2
        3*x  - 24*x + 60

                    x
 1 + ---------------------------}
                     x
      1 - ---------------------
                       x
            2 + ---------------
                         x
                 3 - ---------
                           x
                      2 + ---
                           5
```

The operator `CF` is a synonym for the operator `continued_fraction`.

`cf(`⟨*num*⟩`)`
`cf(`⟨*num*⟩`,` ⟨*size*⟩`)`
`cf(`⟨*num*⟩`,` ⟨*size*⟩`,` ⟨*numterms*⟩`)`

The operator takes the same arguments as the operator `continued_fraction`: the original number to be expanded ⟨*num*⟩, an optional maximum size ⟨*size*⟩ permitted for the denominator of the convergent and an optional maximum number of continuents ⟨*numterms*⟩ to be generated.

The output is in the same format as that of `cfrac` described above. As with the operator `cfrac` output of `CF` is normally pretty-printed so the operator `confract` will not be seen.

The operators `cf_expression`, `cf_convergent` and `cf_continuents` are accessors and allow the various parts of a continued fraction object ⟨*cf_object*⟩

(as returned by any of the operators `cf`, `cfrac`, `continued_fraction` and `cf_euler`) to be extracted.

These three operators return, respectively, the originating expression of the continued fraction object, the last convergent of the continued fraction, a list of its continuents (that is a list of pairs of partial numerators and denominators).

The operator `cf_convergents` returns a list of all the convergents of the expansion.

`cf_expression(`⟨*cf_object*⟩`)`
`cf_convergent(`⟨*cf_object*⟩`)`
`cf_continuents(`⟨*cf_object*⟩`)`
`cf_convergents(`⟨*cf_object*⟩`)`

EXAMPLES

```
2: cf(6/11);
```

```
  6    6           1
{----,----,---------------}
  11   11            1
            1 + ---------
                      1
                 1 + ---
                      5
```

```
3: a := cf(pi,1000);
```

```
          355              1
a := {pi,-----,3 + ----------------}
          113                1
                     7 + ----------
                                1
                           15 + ---
                                1
```

```
4: cf_convergents a;
```

```
    22   333   355
{3,----,-----,-----}
    7    106   113
```

```
5: cf_continuents a;
```

```
{3,7,15,1}

6: precision 20;

12

7: cf pi;

{pi,

  21053343141
 -------------,
  6701487259

 {3,

  {1,7},

  {1,15},

  {1,1},

  {1,292},

  {1,1},

  {1,1},

  {1,1},

  {1,2},

  {1,1},

  {1,3},

  {1,1},

  {1,14},

  {1,2},

  {1,1},
```

```
{1,1},

{1,2},

{1,2},

{1,2},

{1,2},

{1,1}}}
```

The operator `cf_euler` is used to generate a generalised continued fraction expansion of an algebraic expression using a formula due to Leonhard Euler ([Eul48]).

`cf_euler(`⟨*func*⟩`,` ⟨*var*⟩`)`
`cf_euler(`⟨*func*⟩`,` ⟨*var*⟩`,` ⟨*length*⟩`)`

INPUT:
⟨*func*⟩ is a function
⟨*var*⟩ is the function main variable
⟨*length*⟩ is the maximum number of continuents to be generated and is **optional**.

The meaning of the parameters is similar to those of `cfrac`, but the continued fraction expansion generated will usually be different. Note that unlike `cfrac`, `cf_euler` cannot currently generate continued fraction expansion of numbers and for a rational function argument (with a non-constant denominator) the expansion will not be exact.

A number of operators are provided for transforming their continued fraction argument ⟨*cf_object*⟩ into an equivalent expansion, that is one with exactly the same convergents. They all accept as their single argument any continued fraction object ⟨*cf_object*⟩. These are:

`cf_unit_denominators`
converts all partial denominators to 1.

`cf_unit_numerators`
converts all partial numerators to 1.

`cf_remove_fractions`
converts the denominators of the partial numerators and partial denominators in the continuents to 1.

`cf_remove_constant`

removes the zeroth continuent (if non-zero) absorbing it into the first continuent pair.

The operator `cf_transform` is a general purpose function for transforming its continued fraction argument ⟨*cf_object*⟩ into an equivalent expansion. Unlike the four preceding operators it requires a second argument: a list of multipliers used to modify the partial numerators and denominators of the original expansion.

`cf_transform(`⟨*cf_object*⟩`,` ⟨*multiplier-list*⟩`)`

To understand the operation of `cf_transform` consider first the special case where ⟨*multiplier-list*⟩ is a list of the form $\{1, 1, \ldots, 1, l_n, 1, \ldots, 1\}$ whose nth element is l_n. Only the nth continuent pair $\{a_n,\ b_n\}$ and (n+1)th partial numerator a_{n+1} are altered and become $\{l_n a_n,\ l_n b_n\}$ and $l_n a_{n+1}$ respectively. For a ⟨*multiplier-list*⟩ that has more than one non-unit element, the above transformations are applied sequentially from left to right.

If the number of continuent pairs in the ⟨*cf_object*⟩ is greater than the length of the ⟨*multiplier-list*⟩, the latter is (in effect) padded with 1's. Conversely if it is shorter, the surplus elements of ⟨*multiplier-list*⟩ are ignored.

The operator `cf_even_odd` splits its continued fraction argument ⟨*cf_object*⟩ into two continued fraction objects: namely its even and odd parts (in that order) which are returned as a two-element list.

`cf_even_odd(`⟨*cf_object*⟩`)`

The convergents of the even part are the even-numbered convergents of the original expansion and those of the odd part are the odd-numbered ones (except the zeroth convergent which is necessarily zero). For the continued fraction expansions generated by the operators `cf` and `cfrac` with a numerical first argument ⟨*num*⟩. The convergents of the even part form a monotonically increasing sequence whilst those of the odd part (after the zeroth) form a monotonically decreasing sequence.

EXAMPLES

```
cf_remove_fractions(cf_euler(e^x, x, 4));
```

```
                3       2
  x          x  + 3*x  + 6*x + 6
{e , ----------------------,
                    6

                        1
 -------------------------------------}
                        x
```

```
  1 - ------------------------------
                          x
       (x + 1) - -------------------
                               2*x
                  (x + 2) - -------
                                x + 3
```

```
a := cf_remove_fractions(cf_euler(4*atan x, x, 4));
```

```
a := {4*atan(x),

                7       5        3
          - 60*x  + 84*x  - 140*x  + 420*x
         ----------------------------------,
                        105

                     2       2
       (4*x)/(1 + x /(( - x  + 3)

                                      2
                                   9*x
                  + --------------------------------
                                              2
                             2             25*x
                     ( - 3*x  + 5) + -------------
                                             2
                                    - 5*x  + 7

                  ))}
```

```
b := (a where x => 1);
```

```
            304             4
b := {pi,-----,----------------------}
            105               1
                  1 + ----------------
                                9
                       2 + ----------
                                  25
                            2 + ----
                                  2
```

```
c := cf(pi, 0, 6);
```

```
          104348                    1
c := {pi,--------,3 + ------------------------------}
          33215                       1
                        7 + ------------------------
                                            1
                                 15 + -----------------
                                                1
                                       1 + -----------
                                                    1
                                            292 + ---
                                                    1

cf_remove_constant c;

     104348                 22
{pi,--------,--------------------------------}
     33215                      1
               7 + -------------------------
                                  22
                       333 + -----------------
                                      1
                             1 + -----------
                                          1
                                  292 + ---
                                          1

c:= cf(pi, 0, 8)$
d := cf_even_odd c;

           208341                  15
d := {{pi,--------,3 + ----------------------},
           66317                     292
                          106 - --------------
                                       15
                                 4687 - -----
                                       585

           312689              22
      {pi,--------,--------------------------}}
           99532                    1
                     7 + -------------------
                                      22
                           355 - -----------
                                      1
```

```
                                  294 - ---
                                        3
cf_convergents c;

    22   333   355   103993   104348   208341
{3,----,-----,-----,--------,--------,--------,
    7    106   113   33102    33215    66317

  312689
 --------}
  99532

cf_convergents first d;

    333   103993   208341
{3,-----,--------,--------}
    106   33102    66317

cf_convergents second d;

    22   355   104348   312689
{0,----,-----,--------,--------}
    7    113   33215    99532
```

20.48.3 Padé Approximation

The Padé approximant represents a function by the ratio of two polynomials. The coefficients of the powers occuring in the polynomials are determined by the coefficients in the Taylor series expansion of the function (see [BGM96]). Given a power series

$$f(x) = c_0 + c_1(x-h) + c_2(x-h)^2 \ldots$$

and the degree of numerator, n, and of the denominator, d, the `pade` function finds the unique coefficients a_i, b_i in the Padé approximant

$$\frac{a_0 + a_1 x + \cdots + a_n x^n}{b_0 + b_1 x + \cdots + b_d x^d} \ .$$

SYNTAX: `pade(⟨f⟩, ⟨x⟩, ⟨h⟩, ⟨n⟩, ⟨d⟩)`

INPUT:

$\langle f \rangle$	the funtion to be approximated
$\langle x \rangle$	the function variable
$\langle h \rangle$	the point at which the approximation is evaluated
$\langle n \rangle$	the (specified) degree of the numerator
$\langle d \rangle$	the (specified) degree of the denominator

RESULT: Padé Approximant, ie. a rational function.

ERROR MESSAGES:

`***** not yet implemented`
The Taylor series expansion for the function, f, has not yet been implemented in the REDUCE Taylor Package.

`***** no Pade Approximation exists`
A Padé Approximant of this function does not exist.

`***** Pade Approximation of this order does not exist`
A Padé Approximant of this order (ie. the specified numerator and denominator orders) does not exist but one of a different order may exist.

EXAMPLES

```
23: pade(sin(x),x,0,3,3);

          2
 x*( - 7*x  + 60)
------------------
       2
   3*(x  + 20)

24: pade(tanh(x),x,0,5,5);

     4         2
 x*(x  + 105*x  + 945)
-----------------------
      4        2
 15*(x  + 28*x  + 63)

25: pade(atan(x),x,0,5,5);

        4         2
 x*(64*x  + 735*x  + 945)
--------------------------
         4        2
 15*(15*x  + 70*x  + 63)

26: pade(exp(1/x),x,0,5,5);

***** no Pade Approximation exists
```

```
27: pade(factorial(x),x,1,3,3);

***** not yet implemented

28: pade(asech(x),x,0,3,3);

            2                          2                 2
- 3*log(x)*x  + 8*log(x) + 3*log(2)*x  - 8*log(2) + 2*x
------------------------------------------------------
                         2
                      3*x  - 8

29: taylor(ws-asech(x),x,0,10);

                11
log(x)*(0 + O(x  ))

     13    6     43    8    1611    10       11
 + (-----*x  + ------*x  + -------*x   + O(x  ))
     768        2048        81920

30: pade(sin(x)/x^2,x,0,10,0);

***** Pade Approximation of this order does not exist

31:  pade(sin(x)/x^2,x,0,10,2);

     10        8         6           4            2
( - x   + 110*x  - 7920*x  + 332640*x  - 6652800*x

  + 39916800)/(39916800*x)

32: pade(exp(x),x,0,10,10);

  10        9         8           7            6
(x   + 110*x  + 5940*x  + 205920*x  + 5045040*x

               5               4               3
  + 90810720*x  + 1210809600*x  + 11762150400*x

                 2
  + 79394515200*x  + 335221286400*x + 670442572800)/

      10        9         8           7            6
```

```
    (x   - 110*x  + 5940*x  - 205920*x  + 5045040*x

                    5               4
         - 90810720*x  + 1210809600*x

                      3               2
         - 11762150400*x + 79394515200*x

         - 335221286400*x + 670442572800)
```

```
33: pade(sin(sqrt(x)),x,0,3,3);

(sqrt(x)*
            3            2
  (56447*x  - 4851504*x  + 132113520*x - 885487680))\

              3         2
    (7*(179*x  - 7200*x  - 2209680*x - 126498240))
```

20.49 RATINT: Integrate Rational Functions using the Minimal Algebraic Extension to the Constant Field

Author: Neil Langmead

This package was written when the author was a placement student at ZIB Berlin.

20.49.1 Rational Integration

This package implements the Horowitz/ Rothstein/ Trager algorithms [GCL92] for the integration of rational functions in REDUCE. We work within a field K of characteristic 0 and functions $p, q \in K[x]$. K is normally the field Q of rational numbers, but not always. These procedures return $\int \frac{p}{q} dx$. The aim is to be able to integrate any function of the form p/q in x, where p and q are polynomials in the field Q. The algorithms used avoid algebraic number extensions wherever possible, and in general, express the integral using the minimal algebraic extension field.

20.49.1.1 Syntax of `ratint`

This function has the following syntax:

`ratint(`⟨*p*⟩`,`⟨*q*⟩`,`⟨*var*⟩`)`

where ⟨*p*⟩ and ⟨*q*⟩ are polynomials in ⟨*var*⟩, so that p/q is a rational function in var. The output of `ratint` is a list of two elements: the first is the polynomial part of the integral, the second is the logarithmic part. The integral is the sum of these parts.

20.49.1.2 Examples

Consider the following examples in REDUCE (the meaning of the `log_sum` operator will be explained in the next section).

```
ratint(1,x^2-2,x);

{0,

                    2     1
 log_sum(beta,beta  - ---,0,log(2*beta*x - 1)*beta)}
                         8

p:=441*x^7+780*x^6-2861*x^5+4085*x^4+7695*x^3+3713*x^2
```

```
          -43253*x+24500;

q:=9*x^6+6*x^5-65*x^4+20*x^3+135*x^2-154*x+49;

ratint(p,q,x);


   49    6    226   5    268   4    1608   3
{(----*(x  + -----*x  - -----*x  - ------*x
   2          147         49           49

            6011   2    536       256      4
         + ------*x  + -----*x - -----))/(x
            147         21         9

        2   3      2          7
     - ---*x  - 4*x  + 6*x - ---),
        3                     3

 0}

k:=36*x^6+126*x^5+183*x^4+(13807/6)*x^3-407*x^2
      -(3242/5)*x+(3044/15);
l:=(x^2+(7/6)*x+(1/3))^2*(x-(2/5))^3;

ratint(k,l,x);

   5271    3    39547   2    31018       7142
  ------*(x  + -------*x  - -------*x + -------)
    5           52710        26355       26355
{-----------------------------------------------,
       4    11   3    11   2    2        4
      x  + ----*x  - ----*x  - ----*x + ----
           30        25        25       75

  37451            2      91125           2
 -------*(log(x - ---) + -------*log(x + ---)
   16              5      37451           3

             128000           1
          - --------*log(x + ---))}
             37451            2
```

```
ratint(1,x^2+1,x);

                      2    1
{0,log_sum(beta,beta  + ---,0,log(2*beta*x - 1)*beta)}
                           4
```

20.49.2 The Algorithm

The following main algorithm is used:

procedure *ratint*(p, q, x);
% p and q are polynomials in x, with coefficients in the
% constant field Q
solution_list $\leftarrow HorowitzReduction(p, q, x)$
$c/d \leftarrow$ part(solution_list,1)
$poly\_part \leftarrow$ part(solution_list,2)
$rat\_part \leftarrow$ part(solution_list,3)
$rat\_part \leftarrow LogarithmicPartIntegral(rat\_part, x)$
return($rat\_part + c/d + poly\_part$)
end

The algorithm contains two subroutines, *HorowitzReduction* and *rt*. *HorowitzReduction* is an implementation of Horowitz' method to reduce a given rational function into a polynomial part and a logarithmic part. The integration of the polynomial part is a trivial task, and is done by the *int* operator in REDUCE. The integration of the logarithmic part is done by the routine *rt*, which is an implementation of the Rothstein and Trager method. These two answers are outputed in a list, the complete answer being the sum of these two parts.
These two algorithms are as follows:

procedure $how(p, q, x)$

for a given rational function p/q in x, this algorithm calculates the reduction of $\int(p/q)$ into a polynomial part and logarithmic part.

$poly\_part \leftarrow quo(p, q)$; $\quad p \leftarrow rem(p, q)$;

$d \leftarrow GCD(q, q')$; $\quad b \leftarrow quo(q, d)$; $\quad m \leftarrow deg(b)$;
$n \leftarrow deg(d)$;

$a \leftarrow \sum_{i=1}^{m-1} a_i x^i$; $\quad c \leftarrow \sum_{i=1}^{n-1} c_i x^i$;
$r \leftarrow b * c' - quo(b * d', d) + d * a$;

for i from 0 to $m + n - 1$ do
{

$eqns(i) \leftarrow coeff(p,i) = coeff(r,i)$;
};

$solve(eqns, \{a(0), ..., a(m-1), c(0), ..., c(n-1)\})$;

return($c/d + \int poly\_part + a/b$);
end;

procedure RothsteinTrager(a, b, x)

% Given a rational function a/b in x with $deg(a) < deg(b)$,
with b monic and square free, we calculate $\int(a/b)$
$R(z) \leftarrow resultant(a - zb', b)$
$(r_1(z)...r_k(z)) \leftarrow factors(R(z))$
integral $\leftarrow 0$

for i from 1 to k do
{
$d \leftarrow degree(r_i(z))$
if $d = 1$ then {
c $\leftarrow solve(r_i(z) = 0, z)$
v $\leftarrow GCD(a - cb', b)$
v $\leftarrow v/lcoeff(v)$
$integral \leftarrow integral + c * log(v)$
}
else {
% we need to do a GCD over algebraic number field
v $\leftarrow GCD(a - \alpha * b', b)$
v $\leftarrow v/lcoff(v)$, where $\alpha = roof\_of(r_i(z))$
if $d = 2$ then {
% give answer in terms of radicals
c $\leftarrow solve(r_i(z) = 0, z)$
for j from 1 to 2 do {
$v[j] \leftarrow substitute(\alpha = c[j], v)$
$integral \leftarrow integral + c[j] * log(v[j])$
}
else {
% Need answer in terms of root_of notation
for j from 1 to d do {
v[j] $\leftarrow substitute(\alpha = c[j], v)$
integral $\leftarrow integral + c[j] * log(v[j])$
% where $c[j] = root\_of(r_i(z))$ }
}
}
}
return(integral)

end

20.49.3 The `log_sum` operator

The algorithms above returns a sum of terms of the form

$$\sum_{\alpha|R(\alpha)=0} \log(S(\alpha, x)),$$

where $R \in K[z]$ is square free, and $S \in K[z, x]$. In the cases where the degree of $R(\alpha)$ is less than two, this is merely a sum of logarithms. For cases where the degree is two or more, I have chosen to adopt this notation as the answer to the original problem of integrating the rational function. For example, consider the integral

$$\int \frac{a}{b} = \int \frac{2x^5 - 19x^4 + 60x^3 - 159 + x^2 + 50x + 11}{x^6 - 13x^5 + 58x^4 - 85x^3 - 66x^2 - 17x + 1}\, dx$$

Calculating the resultant $R(z) = res_x(a - zb', b)$ and factorising gives

$$R(z) = -190107645728000(z^3 - z^2 + z + 1)^2$$

Making the result monic, we have

$$R_2(z) = z^3 - z^2 + z + 1$$

which does not split over the constant field Q. Continuting with the Rothstein Trager algorithm, we now calculate

$$gcd(a - \alpha\, b', b) = z^2 + (2 * \alpha - 5) * z + \alpha^2,$$

where α is a root of $R_2(z)$.
Thus we can write

$$\int \frac{a}{b} = \sum_{\alpha|\alpha^3-\alpha^2+\alpha+1=0} \alpha * \log(x^2 + 2\alpha x - 5x + \alpha^2),$$

and this is the answer now returned by REDUCE, via a function called `log_sum`. This has the following syntax:

$$\text{log\_sum}(\alpha, eqn(\alpha), 0, sum\_term, var)$$

where α satisfies $eqn = 0$, and sum_term is the term of the summation in the variable var. Thus in the above example, we have

$$\int \frac{a}{b}\, dx = \text{log\_sum}(\alpha, \alpha^3 - \alpha^2 + \alpha + 1, 0, \alpha * \log(x^2 + 2\alpha x - 5x + \alpha^2), x)$$

Many rational functions that could not be integrated by REDUCE previously can now be integrated with this package. The above is one example; some more are given on the next page.

20.49.3.1 More examples

$$\int \frac{1}{x^5+1}\,dx = \frac{1}{5}\log(x+1)$$

$$+ 5\,\mathrm{log\_sum}(\beta, \beta^4 + \frac{1}{5}\beta^3 + \frac{1}{25}\beta^2 + \frac{1}{125}\beta + \frac{1}{625}, 0, \log(5*\beta+x)*\beta)$$

which should be read as

$$\int \frac{1}{x^5+1}\,dx = \frac{1}{5}\log(x+1) + \sum_{\beta \mid \beta^4+\frac{1}{5}\beta^3+\frac{1}{25}\beta^2+\frac{1}{125}\beta+\frac{1}{625}=0} \log(5*\beta+x)\beta$$

$$\int \frac{7x^{13}+10x^8+4x^7-7x^6-4x^3-4x^2+3x+3}{x^{14}-2x^8-2x^7-2x^4-4x^3-x^2+2x+1}\,dx =$$

$$\mathrm{log\_sum}(\alpha, \alpha^2-\alpha-\frac{1}{4}, 0, \log(-2\alpha x^2-2\alpha x+x^7+x^2-1)*\alpha, x),$$

$$\int \frac{1}{x^3+x+1}\,dx = \mathrm{log\_sum}(\beta, \beta^3-\frac{3}{31}\beta^2-\frac{1}{31}, 0, \beta\log(-\frac{62}{9}\beta^2+\frac{31}{9}\beta+x+\frac{4}{9})).$$

20.49.4 Options

There are several alternative forms that the answer to the integration problem can take. One output is the `log_sum` form shown in the examples above. There is an option with this package to convert this to a "normal" sum of logarithms in the case when the degree of eqn in α is two, and α can be expressed in surds. To do this, use the function `convert`, which has the following syntax:

`convert(`⟨*exp*⟩`)`

If ⟨*exp*⟩ is free of `log_sum` terms, then ⟨*exp*⟩ itself is returned. If ⟨*exp*⟩ contains `log_sum` terms, then α is represented as surds, and substituted into the `log_sum` expression. For example, using the last example, we have in REDUCE:

```
2: ratint(a,b,x);

{0,

                      2            1
 log_sum(alpha,alpha  - alpha - ---,0,log(
                                 4
```

```
                    2                   7    2
            - 2*alpha*x  - 2*alpha*x + x  + x

            - 1)*alpha,x)}

3: convert(ws);


 1
---*(sqrt(2)
 2

                       2                  7
     *log( - sqrt(2)*x  - sqrt(2)*x + x  - x - 1)

     - sqrt(2)

                   2                  7
     *log(sqrt(2)*x  + sqrt(2)*x + x  - x - 1) +

                    2                  7
     log( - sqrt(2)*x  - sqrt(2)*x + x  - x - 1)

                    2                  7
      + log(sqrt(2)*x  + sqrt(2)*x + x  - x - 1))
```

20.49.4.1 `LogtoAtan` function

The user could then combine these to form a more elegant answer, using the switch combinelogs if one so wished. Another option is to convert complex logarithms to real arctangents [Bro97], which is recommended if definite integration is the goal. This is implemented in REDUCE via a function `convert_log`, which has the following syntax:

`convert_log(`⟨*exp*⟩`)`

where ⟨*exp*⟩ is any expression containing `log_sum` terms.

The procedure to convert complex logarithms to real arctangents is based on an algorithm by Rioboo. Here is what it does:

Given a field K of characteristic 0 such that $\sqrt{-1} \notin K$ and $A, B \in K[x]$ with $B \neq 0$, return a sum f of arctangents of polynomials in $K[x]$ such that

$$\frac{df}{dx} = \frac{d}{dx} i \log(\frac{A + iB}{A - iB})$$

Example:

$$\int \frac{x^4 - 3 * x^2 + 6}{x^6 - 5 * x^4 + 5 * x^2 + 4}\, dx = \sum_{\alpha | 4\alpha + 1 = 0} \alpha \log(x^3 + 2\alpha x^2 - 3x - 4\alpha)$$

Substituting $\alpha = i/2$ and $\alpha = -i/2$ gives the result

$$\frac{i}{2} \log(\frac{(x^3 - 3x) + i(x^2 - 2)}{(x^3 - 3x) - i(x^2 - 2)})$$

Applying logtoAtan now with $A = x^3 - 3x$, and $B = x^2 - 2$ we obtain

$$\int \frac{x^4 - 3 * x^2 + 6}{x^6 - 5 * x^4 + 5 * x^2 + 4}\, dx$$
$$= \arctan(\frac{x^5 - 3x^3 + x}{2}) + \arctan(x^3) + \arctan(x),$$

and this is the formula which should be used for definite integration.

Another example in REDUCE is given below:

```
1: ratint(1,x^2+1,x);

                      2    1
{0,log_sum(beta,beta  + ---,0,log(2*beta*x - 1)*beta)}
                         4

13: part(ws,2);

                    2    1
log_sum(beta,beta  + ---,0,log(2*beta*x - 1)*beta)
                       4

14: on combinelogs;

15: convertlog(ws);

 1        - i*x + 1
---*log(------------)*i
 2         i*x + 1


logtoAtan(-x,1,x);

 - 2*atan(x)
```

20.49.5 Hermite's method

The package also implements Hermite's method to reduce the integral into its polynomial and logarithmic parts, but occasionally, REDUCE returns the incorrect answer when this algorithm is used. This is due to the REDUCE operator pf, which performs a complete partial fraction expansion when given a rational function as input. Work is presently being done to give the pf operator a facility which tells it that the input is already factored. This would then enable REDUCE to perform a partial fraction decomposition with respect to a square free denominator, which may not necessarily be fully factored over Q.

For a complete explanation of this and the other algorithms used in this package, including the theoretical justification and proofs, please consult [GCL92].

20.49.6 Tracing the *ratint* program

The package includes a facility to trace in some detail the inner workings of the `ratint` program. Messages are given at the key stages of the algorithm, together with the results obtained. These messages are displayed when the switch `traceratint` is on, which is done in REDUCE with the command

```
on traceratint;
```

This switch is off by default. Here is an example of the output obtained with this switch on:

```
1: on traceratint;


2: ratint(1+x,x^2-2*x+1,x);

                                      x + 1
performing Howoritz reduction on --------------
                                        2
                                       x  - 2*x + 1

                    - 2         1
Howoritz gives: {-------,0,-------}
                  x - 1     x - 1

                                  1
computing Rothstein Trager on -------
                               x - 1
```

```
integral in Rothstein T is log(x - 1)

   - 2
{-------,log(x - 1)}
  x - 1
```

20.49.7 Bugs, suggestions and comments

This package was written when the author was working as a placement student at ZIB Berlin.

20.50 REACTEQN: Support for Chemical Reaction Equation Systems

This package allows a user to transform chemical reaction systems into ordinary differential equation systems (ODE) corresponding to the laws of pure mass action.

Author: Herbert Melenk

A single reaction equation is an expression of the form

$$\langle n1\rangle\langle s1\rangle + \langle n2\rangle\langle s2\rangle + \ldots \text{-}> \langle n3\rangle\langle s3\rangle + \langle n4\rangle\langle s4\rangle + \ldots$$

or

$$\langle n1\rangle\langle s1\rangle + \langle n2\rangle\langle s2\rangle + \ldots <> \langle n3\rangle\langle s3\rangle + \langle n4\rangle\langle s4\rangle + \ldots$$

where the $\langle si\rangle$ are arbitrary names of species (REDUCE symbols) and the $\langle ni\rangle$ are positive integer numbers. The number 1 can be omitted. The connector -> describes a one way reaction, while <> describes a forward and backward reaction.

A reaction system is a list of reaction equations, each of them optionally followed by one or two expressions for the rate constants. A rate constant can a number, a symbol or an arbitrary REDUCE expression. If a rate constant is missing, an automatic constant of the form RATE(n) (where n is an integer counter) is generated. For double reactions the first constant is used for the forward direction, the second one for the backward direction.

The names of the species are collected in a list bound to the REDUCE share variable `species`. This list is automatically filled during the processing of a reaction system. The species enter in an order corresponding to their appearance in the reaction system and the resulting ode's will be ordered in the same manner.

If a list of species is preassigned to the variable `species` either explicitly or from previous operations, the given order will be maintained and will dominate the formatting process. So the ordering of the result can be easily influenced by the user.

Syntax:

`reac2ode {`$\langle$*reaction*$\rangle$ [,$\langle$*rate*$\rangle$ [,$\langle$*rate*$\rangle$]] [,$\langle$*reaction*$\rangle$ [,$\langle$*rate*$\rangle$ [,$\langle$*rate*$\rangle$]]] ... `};`

where two rates are applicable only for <> reactions.

Result is a system of explicit ordinary differential equations with polynomial right-hand sides. As side effect the following variables are set:

Lists: **`rates`** list of the rates in the system

species list of the species in the system

Matrices: **inputmat** matrix of the input coefficients

outputmat matrix of the output coefficients

In the matrices the row number corresponds to the input reaction number, while the column number corresponds to the species index. Note: if the rates are numerical values, it will be in most cases appropriate to switch on REDUCE rounded mode for floating point numbers. That is

```
on rounded;
```

Inputmat and outputmat can be used for linear algebra type investigations of the reaction system. The classical reaction matrix is the difference of these matrices; however, the two matrices contain more information than their differences because the appearance of a species on both sides is not reflected by the reaction matrix.

EXAMPLES: This input

```
% Example taken from Feinberg (Chemical Engineering):

   species := {A1,A2,A3,A4,A5};

   reac2ode { A1 + A4 <> 2A1, rho, beta,
              A1 + A2 <> A3, gamma, epsilon,
              A3      <> A2 + A5, theta, mue};
```

gives the output

```
{df(a1,t)

                      2
 =rho*a1*a4 - beta*a1  - gamma*a1*a2 + epsilon*a3,

 df(a2,t)= - gamma*a1*a2 + epsilon*a3 + theta*a3

  - mue*a2*a5,

 df(a3,t)

 =gamma*a1*a2 - epsilon*a3 - theta*a3 + mue*a2*a5,

                                    2
 df(a4,t)= - rho*a1*a4 + beta*a1 ,
```

```
 df(a5,t)=theta*a3 - mue*a2*a5}
```

The corresponding matrices are

```
 inputmat;

[1  0  0  1  0]
[              ]
[1  1  0  0  0]
[              ]
[0  0  1  0  0]

 outputmat;

[2  0  0  0  0]
[              ]
[0  0  1  0  0]
[              ]
[0  1  0  0  1]

% computation of the classical reaction matrix as
% difference of output and input matrix:

  reactmat := outputmat-inputmat;

             [1   0   0   -1  0]
             [                  ]
reactmat := [-1  -1  1   0   0]
             [                  ]
             [0   1   -1  0   1]

% Example with automatic generation of rate constants
% and automatic extraction of species

   species := {};

   reac2ode { A1 + A4 <> 2A1,
              A1 + A2 <> A3,
                   a3 <> A2 + A5};

new species: a1
new species: a4
```

```
new species: a2
new species: a3
new species: a5

                2
{df(a1,t)= - a1 *rate(2) + a1*a4*rate(1)

  - a1*a2*rate(3) + a3*rate(4),

            2
 df(a4,t)=a1 *rate(2) - a1*a4*rate(1),

 df(a2,t)= - a1*a2*rate(3) - a2*a5*rate(6)

  + a3*rate(5) + a3*rate(4),

 df(a3,t)=a1*a2*rate(3) + a2*a5*rate(6)

  - a3*rate(5) - a3*rate(4),

 df(a5,t)= - a2*a5*rate(6) + a3*rate(5)}

% Example with rates computed from numerical expressions

   species := {};

   reac2ode { A1 + A4 <> 2A1, 17.3* 22.4^1.5,
                              0.04* 22.4^1.5 };

new species: a1
new species: a4

{df(a1,t)

                      2
 = - 4.24064598853*a1  + 1834.07939004*a1*a4,

                           2
 df(a4,t)=4.24064598853*a1  - 1834.07939004*a1*a4}
```

20.51 REDLOG: Extend REDUCE to a Computer Logic System

The name REDLOG stand for REDuce LOGic system. Redlog implements symbolic algorithms on first-order formulas with respect to user-chosen first-order languages and theories. The available domains include real numbers, integers, complex numbers, p-adic numbers, quantified propositional calculus, term algebras.

Documentation for this package can be found online.

Authors: Andreas Dolzmann and Thomas Sturm

20.52 RLFI: REDUCE LaTeX Formula Interface

This package adds LaTeX syntax to REDUCE. Text generated by REDUCE in this mode can be directly used in LaTeX source documents. Various mathematical constructions are supported by the interface including subscripts, superscripts, font changing, Greek letters, divide-bars, integral and sum signs, derivatives, and so on.

Author: Richard Liska

High quality typesetting of mathematical formulas is a quite tedious task. One of the most sophisticated typesetting programs for mathematical text TeX [Knu84], together with its widely used macro package LaTeX [Lam86], has a strange syntax of mathematical formulas, especially of the complicated type. This is the main reason which lead us to designing the formula interface between the computer algebra system REDUCE and the document preparation system LaTeX. The other reason is that all available syntaxes of the REDUCE formula output are line oriented and thus not suitable for typesetting in mathematical text. The idea of interfacing a computer algebra system to a typesetting program has already been used, eg. in [Fat87] presenting the TeX output of the MACSYMA computer algebra system.

The formula interface presented here adds to REDUCE the new syntax of formula output, namely LaTeX syntax, and can also be named REDUCE - LaTeX translator. Text generated by REDUCE in this syntax can be directly used in LaTeX source documents. Various mathematical constructions are supported by the interface including subscripts, superscripts, font changing, Greek letters, divide-bars, integral and sum signs, derivatives etc.

The interface can be used in two ways:

- for typesetting of results of REDUCE algebraic calculations.
- for typesetting of users formulas.

The latter can even be used by users unfamiliar with the REDUCE system, because the REDUCE input syntax of formulas is almost the same as the syntax of the majority of programming languages. We aimed at speeding up the process of formula typesetting, because we are convinced, that the writing of correct complicated formulas in the REDUCE syntax is a much more simpler task than writing them in the LaTeX syntax full of keywords and special characters `\`, `{`, `^`, etc. It is clear that not every formula produced by the interface is typeset in the best format from an aesthetic point of view. When a user is not satisfied with the result, he can add some LaTeX commands to the REDUCE output - LaTeX input.

The interface is connected to REDUCE by three new switches and several statements. To activate the LaTeX output mode the switch `latex` must be set `on`. this switch, similar to the switch `fort` producing FORTRAN output, being `on` causes all outputs to be written in the LaTeX syntax of formulas. The switch `verbatim`

is used for input printing control. If it is `on` input to REDUCE system is typeset in LaTeX verbatim environment after the line containing the string `REDUCE Input:`.

The switch `lasimp` controls the algebraic evaluation of input formulas. If it is `on` every formula is evaluated, simplified and written in the form given by ordinary REDUCE statements and switches such as `factor`, `order`, `rat` etc. In the case when the `lasimp` switch is `off` evaluation, simplification or reordering of formulas is not performed and REDUCE acts only as a formula parser and the form of the formula output is exactly the same as that of the input, the only difference remains in the syntax. The mode `off lasimp` is designed especially for typesetting of formulas for which the user needs preservation of their structure. This switch has no meaning if the switch `Latex` is `off` and thus is working only for LaTeX output.

For every identifier used in the typeset REDUCE formula the following properties can be defined by the statement `defid`:

- its printing symbol (Greek letters can be used).
- the font in which the symbol will be typeset.
- accent which will be typeset above the symbol.

Symbols with indexes are treated in REDUCE as operators. Each index corresponds to an argument of the operator. The meaning of operator arguments (where one wants to typeset them) is declared by the statement `defindex`. This statement causes the arguments to be typeset as subscripts or superscripts (on left or right-hand side of the operator) or as arguments of the operator.

ttindextype[RLFI]laline"!*variable The statement `mathstyle` defines the style of formula typesetting. The variable `laline!*` defines the length of output lines.

The fractions with horizontal divide bars are typeset by using the new REDUCE infix operator `//`. This operator is not algebraically simplified. During typesetting of powers the checking on the form of the power base and exponent is performed to determine the form of the typeset expression (eg. sqrt symbol, using parentheses).

Some special forms can be typeset by using REDUCE prefix operators. These are as follows:

- `int` - integral of an expression.
- `dint` - definite integral of an expression.
- `df` - derivative of an expression.
- `pdf` - partial derivative of an expression.
- `sum` - sum of expressions.

- `product` - product of expressions.
- `sqrt` - square root of expression.

There are still some problems unsolved in the present version of the interface as follows:

- breaking the formulas which do not fit on one line.
- automatic decision where to use divide bars in fractions.
- distinction of two- or more-character identifiers from the product of one-character symbols.
- typesetting of matrices.

Remark

After finishing presented interface, we have found another work [ASW89], which solves the same problem. The RLFI package has been described in [DLS90] too.

20.52.1 APPENDIX: Summary and syntax

Warning

The RLFI package can be used only on systems supporting lower case letters with `off raise` statement. The package distinquishes the upper and lower case letters, so be carefull in typing them. In REDUCE 3.6 the REDUCE commands have to be typed in lower-case while the switch `latex` is `on`, in previous versions the commands had to be typed in upper-case.

Switches

`latex` - If `on` output is in LaTeX format. It turns `off` the `raise` switch if it is set `on` and `on` the `raise` switch if it is set `off`. By default is `off`.

`lasimp` - If `on` formulas are evaluated (simplified), REDUCE works as usually. If `off` no evaluation is performed and the structure of formulas is preserved. By default is `on`.

`verbatim` - If `on` the REDUCE input, while `latex` switch being `on`, is printed in LaTeX verbatim environment. The actual REDUCE input is printed after the line containing the string `"REDUCE Input:"`. It turns `on` resp. `off` the `echo` switch when turned `on` resp. `off`. by default is `off`.

Operators

infix - `//`

prefix - `int`,`dint`,`df`,`pdf`,`sum`,`product`,`sqrt` and all REDUCE prefix operators defined in the REDUCE kernel and the SOLVE module.

⟨*alg. expression*⟩ `//` ⟨*alg. expression*⟩
`int` (⟨*function*⟩,⟨*variable*⟩)
`dint` (⟨*from*⟩,⟨*to*⟩,⟨*function*⟩,⟨*variable*⟩)
`df` (⟨*function*⟩,⟨*variables*⟩)
⟨*variables*⟩ ::= ⟨*o-variable*⟩ | ⟨*o-variable*⟩,⟨*variables*⟩
⟨*o-variable*⟩ ::= ⟨*variable*⟩ | ⟨*variable*⟩ ⟨*order*⟩
⟨*variable*⟩ ::= ⟨*kernel*⟩
⟨*order*⟩ ::= ⟨*integer*⟩
⟨*function*⟩ ::= ⟨*alg. expression*⟩
⟨*from*⟩ ::= ⟨*alg. expression*⟩
⟨*to*⟩ ::= ⟨*alg. expression*⟩
`pdf` (⟨*function*⟩,⟨*variables*⟩)
`sum` (⟨*from*⟩,⟨*to*⟩,⟨*function*⟩)
`product` (⟨*from*⟩,⟨*to*⟩,⟨*function*⟩)
`sqrt` (⟨*alg. expression*⟩)

⟨*alg. expression*⟩ is any algebraic expression. Where appropriate, it can include also relational operators (e.g. argument ⟨*from*⟩ of `sum` or `product` operators is usually equation). ⟨*kernel*⟩ is an identifier or prefix operator with arguments as described in section 8.1. The interface supports typesetting lists of algebraic expressions.

Statements

```
mathstyle ⟨m-style>⟩ ;
⟨m-style⟩          ::=  math | displaymath | equation
defid ⟨identifier⟩,⟨d-equations⟩
⟨d-equations⟩      ::=  ⟨d-equation⟩ | ⟨d-equation⟩,⟨d-equations⟩
⟨d-equation⟩       ::=  ⟨d-print symbol⟩ | ⟨d-font⟩ | ⟨d-accent⟩
⟨d-print symbol⟩   ::=  name = ⟨print symbol⟩
⟨d-font⟩           ::=  font = ⟨font⟩
⟨d-accent⟩         ::=  accent = ⟨accent⟩
⟨print symbol⟩     ::=  ⟨character⟩ | ⟨special symbol⟩
⟨special symbol⟩   ::=  alpha | beta | gamma | delta | epsilon |
                        varepsilon | zeta | eta | theta | vartheta |
                        iota | kappa | lambda | mu | nu | xi | pi |
                        varpi | rho | varrho | sigma | varsigma |
                        tau | upsilon | phi | varphi | chi | psi |
                        omega | Gamma | Delta | Theta | Lambda | Xi |
                        Pi | Sigma | Upsilon | Phi | Psi | Omega |
                        infty | hbar
⟨font⟩             ::=  bold | roman
⟨accent⟩           ::=  hat | check | breve | acute | grave | tilde |
                        bar | vec | dot | ddot
```

For special symbols and accents see [Lam86], p. 43, 45, 51.

```
defindex ⟨d-operators⟩ ;
⟨d-operators⟩      ::=  ⟨d-operator⟩ | ⟨d-operator⟩,⟨d-operators⟩
⟨d-operator⟩       ::=  ⟨prefix operator⟩(⟨descriptions⟩)
⟨prefix operator⟩  ::=  ⟨identifier⟩
⟨descriptions⟩     ::=  ⟨description⟩ | ⟨description⟩,⟨descriptions⟩
⟨description⟩      ::=  arg | up | down | leftup | leftdown
```

The meaning of the statements is briefly described in the preceding text.

20.53 SCOPE: REDUCE Source Code Optimization Package

SCOPE is a package for the production of an optimized form of a set of expressions. It applies an heuristic search for common (sub)expressions to almost any set of proper REDUCE assignment statements. The output is obtained as a sequence of assignment statements. GENTRAN is used to facilitate expression output.

Author: J.A. van Hulzen

Further documentation is available at https://reduce-algebra.sourceforge.io/extra-docs/scope.pdf.

20.54 SETS: A Basic Set Theory Package

Author: Francis Wright

The SETS package for REDUCE provides algebraic-mode support for set operations on lists regarded as sets (or representing explicit sets) and on implicit sets represented by identifiers. It provides the set-valued infix operators (with synonyms) `union`, `intersection` (`intersect`) and `setdiff` (`\`, `minus`) and the Boolean-valued infix operators (predicates) `member`, `subset_eq`, `subset`, `set_eq`. The union and intersection operators are n-ary and the rest are binary. A list can be explicitly converted to the canonical set representation by applying the operator `mkset`. (The package also provides an operator not specifically related to set theory called `evalb` that allows the value of any Boolean-valued expression to be displayed in algebraic mode.)

20.54.1 Introduction

REDUCE has no specific representation for a set, neither in algebraic mode nor internally, and any object that is mathematically a set is represented in REDUCE as a list. The difference between a set and a list is that in a set the ordering of elements is not significant and duplicate elements are not allowed (or are ignored). Hence a list provides a perfectly natural and satisfactory representation for a set (but not vice versa). Some languages, such as Maple, provide different internal representations for sets and lists, which may allow sets to be processed more efficiently, but this is not *necessary*.

This package supports set theoretic operations on lists and represents the results as normal algebraic-mode lists, so that all other REDUCE facilities that apply to lists can still be applied to lists that have been constructed by explicit set operations. The algebraic-mode set operations provided by this package have all been available in symbolic mode for a long time, and indeed are used internally by the rest of REDUCE, so in that sense set theory facilities in REDUCE are far from new. What this package does is make them available in algebraic mode, generalize their operation by extending the arity of union and intersection, and allow their arguments to be implicit sets represented by unbound identifiers. It performs some simplifications on such symbolic set-valued expressions, but this is currently rather *ad hoc* and is probably incomplete.

For examples of the operation of the SETS package see (or run) the test file `sets.tst`. This package is experimental and developments are under consideration; if you have suggestions for improvements (or corrections) then please send them to me (FJW), preferably by email. The package is intended to be run under REDUCE 3.5 and later versions; it may well run correctly under earlier versions

although I cannot provide support for such use.

20.54.2 Infix operator precedence

The set operators are currently inserted into the standard REDUCE precedence list (see page 44, §2.7, of the REDUCE manual) as follows:

```
or and not member memq = set_eq neq eq >= > <= <
subset_eq subset freeof + - setdiff union intersection
* / ^ .
```

20.54.3 Explicit set representation and `mkset`

Explicit sets are represented by lists, and this package does not require any restrictions at all on the forms of lists that are regarded as sets. Nevertheless, duplicate elements in a set correspond by definition to the same element and it is conventional and convenient to represent them by a single element, i.e. to remove any duplicate elements. I will call this a normal representation. Since the order of elements in a set is irrelevant it is also conventional and may be convenient to sort them into some standard order, and an appropriate ordering of a normal representation gives a canonical representation. This means that two identical sets have identical representations, and therefore the standard REDUCE equality predicate (=) correctly determines set equality; without a canonical representation this is not the case.

Pre-processing of explicit set-valued arguments of the set-valued operators to remove duplicates is always done because of the obvious efficiency advantage if there were any duplicates, and hence explicit sets appearing in the values of such operators will never contain any duplicate elements. Such sets are also currently sorted, mainly because the result looks better. The ordering used satisfies the `ordp` predicate used for most sorting within REDUCE, except that explicit integers are sorted into increasing numerical order rather than the decreasing order that satisfies `ordp`.

Hence explicit sets appearing in the result of any set operator are currently returned in a canonical form. Any explicit set can also be put into this form by applying the operator `mkset` to the list representing it. For example

```
mkset {1,2,y,x*y,x+y};

{x + y,x*y,y,1,2}
```

The empty set is represented by the empty list `{}`.

20.54.4 Union and intersection

The operator `intersection` (the name used internally) has the shorter synonym `intersect`. These operators will probably most commonly be used as binary infix operators applied to explicit sets, e.g.

```
{1,2,3} union {2,3,4};

{1,2,3,4}

{1,2,3} intersect {2,3,4};

{2,3}
```

They can also be used as n-ary operators with any number of arguments, in which case it saves typing to use them as prefix operators (which is possible with all REDUCE infix operators), e.g.

```
{1,2,3} union {2,3,4} union {3,4,5};

{1,2,3,4,5}

intersect({1,2,3}, {2,3,4}, {3,4,5});

{3}
```

For completeness, they can currently also be used as unary operators, in which case they just return their arguments (in canonical form), and so act as slightly less efficient versions of `mkset` (but this may change), e.g.

```
union {1,5,3,5,1};

{1,3,5}
```

20.54.5 Symbolic set expressions

If one or more of the arguments evaluates to an unbound identifier then it is regarded as representing a symbolic implicit set, and the union or intersection will evaluate to an expression that still contains the union or intersection operator. These two operators are symmetric, and so if they remain symbolic their arguments will be sorted as for any symmetric operator. Such symbolic set expressions are simplified, but the simplification may not be complete in non-trivial cases. For example:

```
a union b union {} union b union {7,3};

{3,7} union a union b

a intersect {};

{}
```

In implementations of REDUCE that provide fancy display using mathematical notation, the empty set, union, intersection and set difference are all displayed using their conventional mathematical symbols, namely $\emptyset$, $\cup$, $\cap$, $\setminus$.

A symbolic set expression is a valid argument for any other set operator, e.g.

```
a union (b intersect c);

b intersection c union a
```

Intersection distributes over union, which is not applied by default but is implemented as a rule list assigned to the variable `set_distribution_rule`, e.g.

```
a intersect (b union c);

(b union c) intersection a

a intersect (b union c) where set_distribution_rule;

a intersection b union a intersection c
```

20.54.6 Set difference

The set difference operator is represented by the symbol `\` and is always output using this symbol, although it can also be input using either of the two names `setdiff` (the name used internally) or `minus` (as used in Maple). It is a binary operator, its operands may be any combination of explicit or implicit sets, and it may be used in an argument of any other set operator. Here are some examples:

```
{1,2,3} \ {2,4};

{1,3}

{1,2,3} \ {};
```

```
{1,2,3}

a \ {1,2};

a\{1,2}

a \ a;

{}

a \ {};

a

{} \ a;

{}
```

20.54.7 Predicates on sets

These are all binary infix operators. Currently, like all REDUCE predicates, they can only be used within conditional statements `if`, `while`, `repeat`) or within the argument of the `evalb` operator provided by this package, and they cannot remain symbolic – a predicate that cannot be evaluated to a Boolean value causes a normal REDUCE error.

The `evalb` operator provides a convenient shorthand for an `if` statement designed purely to display the value of any Boolean expression (not only predicates defined in this package). It has some similarity with the `evalb` function in Maple, except that the values returned by `evalb` in REDUCE (the identifiers `true` and `false`) have no significance to REDUCE itself. Hence, in REDUCE, use of `evalb` is *never* necessary.

```
if a = a then true else false;

true

evalb(a = a);

true

if a = b then true else false;

false
```

```
evalb(a = b);

false

evalb 1;

true

evalb 0;

false
```

I will use the `evalb` operator in preference to an explicit `if` statement for purposes of illustration.

20.54.7.1 Set membership

Set membership is tested by the predicate `member`. Its left operand is regarded as a potential set element and its right operand *must* evaluate to an explicit set. There is currently no sense in which the right operand could be an implicit set; this would require a mechanism for declaring implicit set membership (akin to implicit variable dependence) which is currently not implemented. Set membership testing works like this:

```
evalb(1 member {1,2,3});

true

evalb(2 member {1,2} intersect {2,3});

true

evalb(a member b);

***** b invalid as list
```

20.54.7.2 Set inclusion

Set inclusion is tested by the predicate `subset_eq` where `a subset_eq b` is true if the set a is either a subset of or equal to the set b; strict inclusion is tested by the predicate `subset` where `a subset b` is true if the set a is *strictly* a subset of the set b and is false is a is equal to b. These predicates provide some support

for symbolic set expressions, but this is not yet correct as indicated below. Here are some examples:

```
evalb({1,2} subset_eq {1,2,3});

true

evalb({1,2} subset_eq {1,2});

true

evalb({1,2} subset {1,2});

false


evalb(a subset a union b);

true

evalb(a\b subset a);

true

evalb(a intersect b subset a union b);  %%% BUG

false
```

An undecidable predicate causes a normal REDUCE error, e.g.

```
evalb(a subset_eq {b});

***** Cannot evaluate a subset_eq {b}
as Boolean-valued set expression

evalb(a subset_eq b);  %%% BUG

false
```

20.54.7.3 Set equality

As explained above, equality of two sets in canonical form can be reliably tested by the standard REDUCE equality predicate (=). This package also provides the

predicate `set_eq` to test equality of two sets not represented canonically. The two predicates behave identically for operands that are symbolic set expressions because these are always evaluated to canonical form (although currently this is probably strictly true only in simple cases). Here are some examples:

```
evalb({1,2,3} = {1,2,3});

true

evalb({2,1,3} = {1,3,2});

false

evalb(mkset{2,1,3} = mkset{1,3,2});

true

evalb({2,1,3} set_eq {1,3,2});

true

evalb(a union a = a\{});

true
```

20.54.8 Possible future developments

- Unary union/intersection to implement repeated union/intersection on a set of sets.
- More symbolic set algebra, canonical forms for set expressions, more complete simplification.
- Better support for Boolean variables via a version (evalb10?) of `evalb` that returns 1/0 instead of `true`/`false`, or predicates that return 1/0 directly.

20.55 SPARSE: Sparse Matrix Calculations

Author: Stephen Scowcroft

20.55.1 Introduction

A very powerful feature of REDUCE is the ease with which matrix calculations can be performed. This package extends the available matrix feature to enable calculations with sparse matrices. This package also provides a selection of functions that are useful in the world of linear algebra with respect to sparse matrices.

Loading the Package

The package is loaded by: `load_package sparse;`

20.55.2 Sparse Matrix Calculations

To extend the the syntax to this class of calculations we need to add an expression type `sparse`.

20.55.2.1 Sparse Variables

An identifier may be declared a sparse variable by the declaration `SPARSE`. The size of the sparse matrix must be declared explicitly in the matrix declaration. For example,

```
sparse aa(10,1),bb(200,200);
```

declares `aa` to be a 10 x 1 (column) sparse matrix and `y` to be a 200 x 200 sparse matrix. The declaration `sparse` is similar to the declaration `matrix`. Once a symbol is declared to name a sparse matrix, it can not also be used to name an array, operator, procedure, or used as an ordinary variable. For more information see the Matrix Variables section (14.2).

20.55.2.2 Assigning Sparse Matrix Elements

Once a matix has been declared a sparse matrix all elements of the matrix are initialized to 0. Thus when a sparse matrix is initially referred to the message

```
"Empty matrix"
```

is returned. When printing out a matrix only the non-zero elements are printed. This is due to the fact that only the non-zero elements of the matrix are stored. To assign the elements of the declared matrix we use the following syntax. Assuming `aa` and `bb` have been declared as spasre matrices, we simply write,

```
aa(1,1):=10;
bb(100,150):=a;
```

etc. This then sets the element in the first row and first column to 10, or the element in the 100th row and 150th column to `a`.

20.55.2.3 Evaluating Sparse Matrix Elements

Once an element of a sparse matrix has been assingned, it may be referred to in standard array element notation. Thus `aa(2,1)` refers to the element in the second row and first column of the sparse matrix `aa`.

20.55.3 Sparse Matrix Expressions

These follow the normal rules of matrix algebra. Sums and products must be of compatible size; otherwise an error will result during evaluation. Similarly, only square matrices may be raised to a power. A negative power is computed as the inverse of the matrix raised to the corresponding positive power. For more information and the syntax for matrix algebra see the Matrix Expressions section (14.3).

20.55.4 Operators with Sparse Matrix Arguments

The operators in the Sparse Matrix Package are the same as those in the Matrix Package with the exception that the `nullspace` operator is not defined. See section Operators with Matrix Arguments (14.4) for more details.

20.55.4.1 Examples

In the examples the matrix $\mathcal{AA}$ will be

$$\mathcal{AA} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \end{pmatrix}$$

```
det aa;
```

```
135

trace aa;

18

rank aa;

4

spmateigen(aa,eta);

{{eta - 1,1,

  spm(1,1) := arbcomplex(1)$
  },

 {eta - 3,1,

  spm(2,1) := arbcomplex(2)$
  },

 {eta - 5,1,

  spm(3,1) := arbcomplex(3)$
  },

 {eta - 9,1,

  spm(4,1) := arbcomplex(4)$
  }}
```

20.55.5 The Linear Algebra Package for Sparse Matrices

This package is an extension of the Linear Algebra Package for REDUCE described in section 20.33. These functions are described alphabetically in section 20.55.6. They can be classified into four sections(n.b: the numbers after the dots signify the function label in section 6).

20.55.5.1 Basic matrix handling

20.55.5.2 Constructors

Functions that create sparse matrices.

20.55.5.3 High level algorithms

20.55.5.4 Predicates

Note on examples:

In the examples the matrix $\mathcal{A}$ will be

$$\mathcal{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

Unfortunately, due to restrictions of size, it is not practical to use "large" sparse matrices in the examples. As a result the examples shown may appear trivial, but they give an idea of how the functions work.

Notation

Throughout $\mathcal{I}$ is used to indicate the identity matrix and $\mathcal{A}^T$ to indicate the transpose of the matrix $\mathcal{A}$.

20.55.6 Available Functions

20.55.6.1 spadd_columns, spadd_rows

Syntax:

`spadd_columns(`$\mathcal{A}$`,c1,c2,expr);`

$\mathcal{A}$	:-	a sparse matrix.
$c1, c2$	:-	positive integers.
expr	:-	a scalar expression.

Synopsis:

`spadd_columns` replaces column $c2$ of $\mathcal{A}$ by
`expr` $*$ `column(`$\mathcal{A}$`,c1)` $+$ `column(`$\mathcal{A}$`,c2)`.
`spadd_rows` performs the equivalent task on the rows of $\mathcal{A}$.

Examples:

$$\texttt{spadd\_columns}(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & x & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{spadd\_rows}(\mathcal{A}, 2, 3, 5) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 25 & 9 \end{pmatrix}$$

Related functions:

`spadd_to_columns`, `spadd_to_rows`, `spmult_columns`, `spmult_rows`.

20.55.6.2 spadd_rows

See: `spadd_columns`.

20.55.6.3 spadd_to_columns, spadd_to_rows

Syntax:

`spadd_to_columns(`$\mathcal{A}$`,column_list,expr);`

$\mathcal{A}$	:-	a sparse matrix.
column_list	:-	a positive integer or a list of positive integers.
expr	:-	a scalar expression.

Synopsis:

`spadd_to_columns` adds expr to each column specified in column_list of $\mathcal{A}$.

`spadd_to_rows` performs the equivalent task on the rows of $\mathcal{A}$.

Examples:

$$\texttt{spadd\_to\_columns}(\mathcal{A}, \{1,2\}, 10) = \begin{pmatrix} 11 & 10 & 0 \\ 10 & 15 & 0 \\ 10 & 10 & 9 \end{pmatrix}$$

$$\texttt{spadd\_to\_rows}(\mathcal{A}, 2, -x) = \begin{pmatrix} 1 & 0 & 0 \\ -x & -x+5 & -x \\ 0 & 0 & 9 \end{pmatrix}$$

Related functions:

`spadd_columns`, `spadd_rows`, `spmult_rows`, `spmult_columns`.

20.55.6.4 spadd_to_rows

See: `spadd_to_columns`.

20.55.6.5 spaugment_columns, spstack_rows

Syntax:

`spaugment_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$	:-	a sparse matrix.
column_list	:-	either a positive integer or a list of positive integers.

Synopsis:

`spaugment_columns` gets hold of the columns of $\mathcal{A}$ specified in column_list and sticks them together.

`spstack_rows` performs the same task on rows of $\mathcal{A}$.

Examples:

$$\texttt{spaugment\_columns}(\mathcal{A}, \{1,2\}) = \begin{pmatrix} 1 & 0 \\ 0 & 5 \\ 0 & 0 \end{pmatrix}$$

$\texttt{spstack\_rows}(\mathcal{A}, \{1,3\}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 9 \end{pmatrix}$

Related functions:
`spget_columns`, `spget_rows`, `spsub_matrix`.

20.55.6.6 spband_matrix

Syntax:
`spband_matrix(expr_list,square_size);`

expr_list	:-	either a single scalar expression or a list of an odd number of scalar expressions.
square_size	:-	a positive integer.

Synopsis:
`spband_matrix` creates a sparse square matrix of dimension square_size.

Examples: $\texttt{spband\_matrix}(\{x,y,z\},6) = \begin{pmatrix} y & z & 0 & 0 & 0 & 0 \\ x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \\ 0 & 0 & 0 & 0 & x & y \end{pmatrix}$

Related functions:
`spdiagonal`.

20.55.6.7 spblock_matrix

Syntax:
`spblock_matrix(r,c,matrix_list);`

r,c	:-	positive integers.
matrix_list	:-	a list of matrices of either sparse or matrix type.

Synopsis:
`spblock_matrix` creates a sparse matrix that consists of r by c matrices filled from the matrix_list row wise.

Examples:

$\mathcal{B} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \mathcal{C} = \begin{pmatrix} 5 \\ 0 \end{pmatrix},\ \mathcal{D} = \begin{pmatrix} 22 & 0 \\ 0 & 0 \end{pmatrix}$

$\texttt{spblock\_matrix}(2,3,\{\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{D},\mathcal{C},\mathcal{B}\}) = \begin{pmatrix} 1 & 0 & 5 & 22 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 22 & 0 & 5 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$

20.55.6.8 spchar_matrix

Syntax:

`spchar_matrix(`$\mathcal{A}, \lambda$`);`

$\mathcal{A}$:- a square sparse matrix.
λ :- a symbol or algebraic expression.

Synopsis:

`spchar_matrix` creates the characteristic matrix $\mathcal{C}$ of $\mathcal{A}$.

This is $\mathcal{C} = \lambda * \mathcal{I} - \mathcal{A}$.

Examples: `spchar_matrix`$(\mathcal{A}, x) = \begin{pmatrix} x-1 & 0 & 0 \\ 0 & x-5 & 0 \\ 0 & 0 & x-9 \end{pmatrix}$

Related functions:

`spchar_poly`.

20.55.6.9 spchar_poly

Syntax:

`spchar_poly(`$\mathcal{A}, \lambda$`);`

$\mathcal{A}$:- a sparse square matrix.
λ :- a symbol or algebraic expression.

Synopsis:

`spchar_poly` finds the characteristic polynomial of $\mathcal{A}$.

This is the determinant of $\lambda * \mathcal{I} - \mathcal{A}$.

Examples:

`spchar_poly(`$\mathcal{A}, x$`)` $= x^3 - 15 * x^2 - 59 * x - 45$

Related functions:

`spchar_matrix`.

20.55.6.10 spcholesky

Syntax:

`spcholesky(`$\mathcal{A}$`);`

$\mathcal{A}$:- a positive definite sparse matrix containing numeric entries.

Synopsis:

`spcholesky` computes the cholesky decomposition of $\mathcal{A}$.

It returns $\{\mathcal{L},\mathcal{U}\}$ where $\mathcal{L}$ is a lower matrix, $\mathcal{U}$ is an upper matrix, $\mathcal{A} = \mathcal{L}\mathcal{U}$, and $\mathcal{U} = \mathcal{L}^T$.

Examples:

$$\mathcal{F} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{cholesky}(\mathcal{F}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 0 & \sqrt{5} & 0 \\ 0 & 0 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & \sqrt{5} & 0 \\ 0 & 0 & 3 \end{pmatrix} \right\}$$

Related functions:

`splu_decom`.

20.55.6.11 spcoeff_matrix

Syntax:

`spcoeff_matrix({lin_eqn`$_1$`,lin_eqn`$_2$`, ...,lin_eqn`$_n$`});`

lin_eqn$_1$,lin_eqn$_2$, ...,lin_eqn$_n$:- linear equations. Can be of the form *equation* = *number* or just *equation* which is equivalent to *equation* = *0*.

Synopsis:

`spcoeff_matrix` creates the coefficient matrix $\mathcal{C}$ of the linear equations.

It returns $\{\mathcal{C}, \mathcal{X}, \mathcal{B}\}$ such that $\mathcal{C}\mathcal{X} = \mathcal{B}$.

Examples:

`spcoeff_matrix`$(\{y - 20 * w = 10, y - z = 20, y + 4 + 3 * z, w + x + 50\}) =$

$$\left\{ \begin{pmatrix} 1 & -20 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} y \\ w \\ z \\ x \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \\ -4 \\ 50 \end{pmatrix} \right\}$$

20.55.6.12 spcol_dim, sprow_dim

Syntax:

`column_dim(`$\mathcal{A}$`);`

$\mathcal{A}$:- a sparse matrix.

Synopsis:

`spcol_dim` finds the column dimension of $\mathcal{A}$.
`sprow_dim` finds the row dimension of $\mathcal{A}$.

Examples:

`spcol_dim`($\mathcal{A}$) = 3

20.55.6.13 spcompanion

Syntax:

`spcompanion(poly,x);`

poly	:-	a monic univariate polynomial in x.
x	:-	the variable.

Synopsis:

`spcompanion` creates the companion matrix $\mathcal{C}$ of poly.

This is the square matrix of dimension n, where n is the degree of poly w.r.t. x. The entries of $\mathcal{C}$ are: $\mathcal{C}(i,n) = -\texttt{coeffn}(\texttt{poly}, x, i-1)$ for $i = 1 \ldots n$, $\mathcal{C}(i, i-1) = 1$ for $i = 2 \ldots n$ and the rest are 0.

Examples:

$$\texttt{spcompanion}(x^4 + 17 * x^3 - 9 * x^2 + 11, x) = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

Related functions:

`spfind_companion`.

20.55.6.14 spcopy_into

Syntax:

`spcopy_into(`$\mathcal{A}, \mathcal{B}$`,r,c);`

$\mathcal{A}, \mathcal{B}$	:-	matrices of type sparse or matrix.
r,c	:-	positive integers.

Synopsis:

`spcopy_into` copies matrix $\mathcal{A}$ into $\mathcal{B}$ with $\mathcal{A}$(1,1) at $\mathcal{B}$(r,c).

Examples:

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\texttt{spcopy\_into}(\mathcal{A},\mathcal{G},1,2) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Related functions:

`spaugment_columns`, `spextend`, `spmatrix_augment`, `spmatrix_stack`, `spstack_rows`, `spsub_matrix`.

20.55.6.15 spdiagonal

Syntax:

`spdiagonal({mat`$_1$`,mat`$_2$`, ...,mat`$_n$`});`[47]

mat$_1$,mat$_2$, ...,mat$_n$:- each can be either a scalar expr or a square matrix of sparse or matrix type.

Synopsis:

`spdiagonal` creates a sparse matrix that contains the input on the diagonal.

Examples:

$$\mathcal{H} = \begin{pmatrix} 66 & 77 \\ 88 & 99 \end{pmatrix}$$

$$\texttt{spdiagonal}(\{\mathcal{A},x,\mathcal{H}\}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 66 & 77 \\ 0 & 0 & 0 & 0 & 88 & 99 \end{pmatrix}$$

Related functions:

`spjordan_block`.

20.55.6.16 spextend

Syntax:

`spextend(`$\mathcal{A}$`,r,c,expr);`

$\mathcal{A}$:- a sparse matrix.
r,c :- positive integers.
expr :- algebraic expression or symbol.

[47] The {}'s can be omitted.

Synopsis:

`spextend` returns a copy of $\mathcal{A}$ that has been extended by r rows and c columns. The new entries are made equal to expr.

Examples: `spextend`$(\mathcal{A}, 1, 2, 0) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

Related functions:

`spcopy_into`, `spmatrix_augment`, `spmatrix_stack`, `spremove_columns`, `spremove_rows`.

20.55.6.17 spfind_companion

Syntax:

`spfind_companion(`$\mathcal{A}$`,x);`

$\mathcal{A}$	:-	a sparse matrix.
x	:-	the variable.

Synopsis:

Given a sparse companion matrix, `spfind_companion` finds the polynomial from which it was made.

Examples:

$$\mathcal{C} = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

`spfind_companion`$(\mathcal{C}, x) = x^4 + 17 * x^3 - 9 * x^2 + 11$

Related functions:

`spcompanion`.

20.55.6.18 spget_columns, spget_rows

Syntax:

`spget_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$	:-	a sparse matrix.
c	:-	either a positive integer or a list of positive integers.

Synopsis:

`spget_columns` removes the columns of $\mathcal{A}$ specified in column_list and returns them as a list of column matrices.

`spget_rows` performs the same task on the rows of $\mathcal{A}$.

Examples:

`spget_columns`$(\mathcal{A}, \{1,3\}) = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 9 \end{pmatrix} \right\}$

`spget_rows`$(\mathcal{A}, 2) = \left\{ \begin{pmatrix} 0 & 5 & 0 \end{pmatrix} \right\}$

Related functions:

`spaugment_columns`, `spstack_rows`, `spsub_matrix`.

20.55.6.19 spget_rows

See: `spget_columns`.

20.55.6.20 spgram_schmidt

Syntax:

`spgram_schmidt({vec`$_1$`,vec`$_2$`, ...,vec`$_n$`});`

vec$_1$,vec$_2$, ...,vec$_n$:- linearly independent vectors. Each vector must be written as a list of predefined sparse (column) matrices, eg: sparse a(4,1);, a(1,1):=1;

Synopsis:

`spgram_schmidt` performs the gram_schmidt orthonormalisation on the input vectors.

It returns a list of orthogonal normalised vectors.

Examples:

Suppose a,b,c,d correspond to sparse matrices representing the following lists: {{1,0,0,0},{1,1,0,0},{1,1,1,0},{1,1,1,1}}.

`spgram_schmidt({{a},{b},{c},{d}})` =

{{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}}

20.55.6.21 sphermitian_tp

Syntax:

`sphermitian_tp(`$\mathcal{A}$`);`

$\mathcal{A}$:- a sparse matrix.

Synopsis:

`sphermitian_tp` computes the hermitian transpose of $\mathcal{A}$.

Examples:

$$\mathcal{J} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 0 & 0 & 0 \\ 0 & i & 0 \end{pmatrix}$$

$$\texttt{sphermitian\_tp}(\mathcal{J}) = \begin{pmatrix} -i+1 & 0 & 0 \\ -i+2 & 0 & -i \\ -i+3 & 0 & 0 \end{pmatrix}$$

Related functions:

`tp`[48].

20.55.6.22 sphessian

Syntax:

```
sphessian(expr,variable_list);
```

expr	:-	a scalar expression.
variable_list	:-	either a single variable or a list of variables.

Synopsis:

`sphessian` computes the hessian matrix of expr w.r.t. the variables in variable_list.

Examples: $\texttt{sphessian}(x*y*z+x^2, \{w,x,y,z\}) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & z & y \\ 0 & z & 0 & x \\ 0 & y & x & 0 \end{pmatrix}$

20.55.6.23 spjacobian

Syntax:

```
spjacobian(expr_list,variable_list);
```

expr_list	:-	either a single algebraic expression or a list of algebraic expressions.
variable_list	:-	either a single variable or a list of variables.

Synopsis:

`spjacobian` computes the jacobian matrix of expr_list w.r.t. variable_list.

Examples:

$\texttt{spjacobian}(\{x^4, x*y^2, x*y*z^3\}, \{w,x,y,z\}) =$

$$\begin{pmatrix} 0 & 4*x^3 & 0 & 0 \\ 0 & y^2 & 2*x*y & 0 \\ 0 & y*z^3 & x*z^3 & 3*x*y*z^2 \end{pmatrix}$$

[48] standard reduce call for the transpose of a matrix - see section 14.4.

Related functions:
`sphessian`, `df`[49].

20.55.6.24 spjordan_block

Syntax:
`spjordan_block(expr,square_size);`

expr	:-	an algebraic expression or symbol.
square_size	:-	a positive integer.

Synopsis:
`spjordan_block` computes the square jordan block matrix $\mathcal{J}$ of dimension square_size.

Examples: `spjordan_block(x,5)` $= \begin{pmatrix} x & 1 & 0 & 0 & 0 \\ 0 & x & 1 & 0 & 0 \\ 0 & 0 & x & 1 & 0 \\ 0 & 0 & 0 & x & 1 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$

Related functions:
`spdiagonal`, `spcompanion`.

20.55.6.25 splu_decom

Syntax:
`splu_decom(`$\mathcal{A}$`);`

$\mathcal{A}$	:-	a sparse matrix containing either numeric entries or imaginary entries with numeric coefficients.

Synopsis:
`splu_decom` performs LU decomposition on $\mathcal{A}$, ie: it returns $\{\mathcal{L},\mathcal{U}\}$ where $\mathcal{L}$ is a lower diagonal matrix, $\mathcal{U}$ an upper diagonal matrix and $\mathcal{A} = \mathcal{L}\mathcal{U}$.

Caution: The algorithm used can swap the rows of $\mathcal{A}$ during the calculation. This means that $\mathcal{L}\mathcal{U}$ does not equal $\mathcal{A}$ but a row equivalent of it. Due to this, `splu_decom` returns $\{\mathcal{L},\mathcal{U}$,vec$\}$. The call `spconvert(`$\mathcal{A}$`,vec)` will return the sparse matrix that has been decomposed, ie: $\mathcal{L}\mathcal{U}$ = `spconvert(`$\mathcal{A}$`,vec)`.

Examples: $\mathcal{K} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$

[49] standard reduce call for differentiation - see section 7.7.

$$\texttt{lu} := \texttt{splu\_decom}(\mathcal{K}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, [\,1\;2\;3\,] \right\}$$

$$\texttt{first lu} \star \texttt{second lu} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{convert}(\mathcal{K}, \texttt{third lu}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

Related functions:
`spcholesky`.

20.55.6.26 spmake_identity

Syntax:
`spmake_identity(square_size);`

square_size :- a positive integer.

Synopsis:
`spmake_identity` creates the identity matrix of dimension square_size.

Examples: $\texttt{spmake\_identity}(4) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Related functions:
`spdiagonal`.

20.55.6.27 spmatrix_augment, spmatrix_stack

Syntax:
`spmatrix_augment({mat`$_1$`, mat`$_2$`, ..., mat`$_n$`});`[50]

mat$_1$,mat$_2$, ...,mat$_n$:- matrices.

Synopsis:
`spmatrix_augment` joins the matrices in matrix_list together horizontally.

`spmatrix_stack` joins the matrices in matrix_list together vertically.

[50] The {}'s can be omitted.

Examples:

$$\texttt{spmatrix\_augment}(\{\mathcal{A},\mathcal{A}\}) = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 5 & 0 & 0 & 5 & 0 \\ 0 & 0 & 9 & 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{spmatrix\_stack}(\{\mathcal{A},\mathcal{A}\}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \\ 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

Related functions:

`spaugment_columns`, `spstack_rows`, `spsub_matrix`.

20.55.6.28 matrixp

Syntax:

`matrixp(test_input);`

test_input :- anything you like.

Synopsis:

`matrixp` is a boolean function that returns t if the input is a matrix of type sparse or matrix and nil otherwise.

Examples:

`matrixp`($\mathcal{A}$) = t

`matrixp`(doodlesackbanana) = nil

Related functions:

`squarep`, `symmetricp`, `sparsematp`.

20.55.6.29 spmatrix_stack

See: `spmatrix_augment`.

20.55.6.30 spminor

Syntax:

`spminor(`$\mathcal{A}$`,r,c);`

$\mathcal{A}$:- a sparse matrix.
r,c :- positive integers.

Synopsis:

spminor computes the (r,c)'th minor of $\mathcal{A}$.

Examples: $\texttt{spminor}(\mathcal{A},1,3) = \begin{pmatrix} 0 & 5 \\ 0 & 0 \end{pmatrix}$

Related functions:

spremove_columns, spremove_rows.

20.55.6.31 spmult_columns, spmult_rows

Syntax:

spmult_columns($\mathcal{A}$,column_list,expr);

$\mathcal{A}$	:-	a sparse matrix.
column_list	:-	a positive integer or a list of positive integers.
expr	:-	an algebraic expression.

Synopsis:

spmult_columns returns a copy of $\mathcal{A}$ in which the columns specified in column_list have been multiplied by expr.

spmult_rows performs the same task on the rows of $\mathcal{A}$.

Examples:

$$\texttt{spmult\_columns}(\mathcal{A},\{1,3\},x) = \begin{pmatrix} x & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9*x \end{pmatrix}$$

$$\texttt{spmult\_rows}(\mathcal{A},2,10) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 50 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

Related functions:

spadd_to_columns, spadd_to_rows.

20.55.6.32 spmult_rows

See: spmult_columns.

20.55.6.33 sppivot

Syntax:

sppivot($\mathcal{A}$,r,c);

$\mathcal{A}$	:-	a sparse matrix.
r,c	:-	positive integers such that $\mathcal{A}$(r,c) neq 0.

Synopsis:

`sppivot` pivots $\mathcal{A}$ about it's (r,c)'th entry.

To do this, multiples of the r'th row are added to every other row in the matrix.

This means that the c'th column will be 0 except for the (r,c)'th entry.

Related functions:

`sprows_pivot`.

20.55.6.34 sppseudo_inverse

Syntax:

`sppseudo_inverse(`$\mathcal{A}$`);`

$\mathcal{A}$:- a sparse matrix containing only real numeric entries.

Synopsis:

`sppseudo_inverse`, also known as the Moore-Penrose inverse, computes the pseudo inverse of $\mathcal{A}$.

Given the singular value decomposition of $\mathcal{A}$, i.e: $\mathcal{A} = \mathcal{U}\Sigma\mathcal{V}^T$, then the pseudo inverse $\mathcal{A}^\dagger$ is defined by $\mathcal{A}^\dagger = \mathcal{V}\Sigma^\dagger\mathcal{U}^T$. For the diagonal matrix Σ, the pseudoinverse $\Sigma^\dagger$ is computed by taking the reciprocal of only the nonzero diagonal elements.

If $\mathcal{A}$ is square and non-singular, then $\mathcal{A}^\dagger = \mathcal{A}$. In general, however, $\mathcal{A}\mathcal{A}^\dagger\mathcal{A} = \mathcal{A}$, and $\mathcal{A}^\dagger\mathcal{A}\mathcal{A}^\dagger = \mathcal{A}^\dagger$.

Perhaps more importantly, $\mathcal{A}^\dagger$ solves the following least-squares problem: given a rectangular matrix $\mathcal{A}$ and a vector b, find the x minimizing $\|\mathcal{A}x-b\|_2$, and which, in addition, has minimum ℓ_2 (euclidean) Norm, $\|x\|_2$. This x is $\mathcal{A}^\dagger b$.

Examples:

$$\mathcal{R} = \begin{pmatrix} 0 & 0 & 3 & 0 \\ 9 & 0 & 7 & 0 \end{pmatrix}$$

$$\texttt{sppseudo\_inverse}(\mathcal{R}) = \begin{pmatrix} -0.26 & 0.11 \\ 0 & 0 \\ 0.33 & 0 \\ 0.25 & -0.05 \end{pmatrix}$$

Related functions:

`spsvd`.

20.55.6.35 spremove_columns, spremove_rows

Syntax:

`spremove_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$	:-	a sparse matrix.
column_list	:-	either a positive integer or a list of positive integers.

Synopsis:

`spremove_columns` removes the columns specified in column_list from $\mathcal{A}$.

`spremove_rows` performs the same task on the rows of $\mathcal{A}$.

Examples:

`spremove_columns`$(\mathcal{A}, 2) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 9 \end{pmatrix}$

`spremove_rows`$(\mathcal{A}, \{1, 3\}) = \begin{pmatrix} 0 & 5 & 0 \end{pmatrix}$

Related functions:

`spminor`.

20.55.6.36 spremove_rows

See: `spremove_columns`.

20.55.6.37 sprow_dim

See: `spcolumn_dim`.

20.55.6.38 sprows_pivot

Syntax:

`sprows_pivot(`$\mathcal{A}$`,r,c,{row_list});`

$\mathcal{A}$	:-	a sparse matrix.
r,c	:-	positive integers such that $\mathcal{A}$(r,c) neq 0.
row_list	:-	positive integer or a list of positive integers.

Synopsis:

`sprows_pivot` performs the same task as `sppivot` but applies the pivot only to the rows specified in row_list.

Related functions:

`sppivot`.

20.55.6.39 sparsematp

Syntax:

`sparsematp(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix.

Synopsis:

`sparsematp` is a boolean function that returns t if the matrix is declared sparse and nil otherwise.

Examples:

$\mathcal{L}$:= `mat((1,2,3),(4,5,6),(7,8,9));`

`sparsematp(`$\mathcal{A}$`)` = `t`

`sparsematp(`$\mathcal{L}$`)` = `nil`

Related functions:

`matrixp`, `symmetricp`, `squarep`.

20.55.6.40 squarep

Syntax:

`squarep(`$\mathcal{A}$`);`

$\mathcal{A}$:- a matrix.

Synopsis:

`squarep` is a boolean function that returns t if the matrix is square and nil otherwise.

Examples:

$\mathcal{L} = \begin{pmatrix} 1 & 3 & 5 \end{pmatrix}$

`squarep(`$\mathcal{A}$`)` = `t`

`squarep(`$\mathcal{L}$`)` = `nil`

Related functions:

`matrixp`, `symmetricp`, `sparsematp`.

20.55.6.41 spstack_rows

See: `spaugment_columns`.

20.55.6.42 spsub_matrix

Syntax:

`spsub_matrix(`$\mathcal{A}$`,row_list,column_list);`

$\mathcal{A}$	:-	a sparse matrix.
row_list, column_list	:-	either a positive integer or a list of positive integers.

Synopsis:

`spsub_matrix` produces the matrix consisting of the intersection of the rows specified in row_list and the columns specified in column_list.

Examples: `spsub_matrix`$(\mathcal{A}, \{1,3\}, \{2,3\}) = \begin{pmatrix} 5 & 0 \\ 0 & 9 \end{pmatrix}$

Related functions:

`spaugment_columns`, `spstack_rows`.

20.55.6.43 spsvd (singular value decomposition)

Syntax:

`spsvd(`$\mathcal{A}$`);`

$\mathcal{A}$	:-	a sparse matrix containing only real numeric entries.

Synopsis:

`spsvd` computes the singular value decomposition of $\mathcal{A}$.

If A is an $m \times n$ real matrix of (column) rank r, `svd` returns the 3-element list $\{\mathcal{U}, \Sigma, \mathcal{V}\}$ where $\mathcal{A} = \mathcal{U}\Sigma\mathcal{V}^T$.

Let $k = \min(m, n)$. Then U is $m \times k$, V is $n \times k$, and and $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_k)$, where $\sigma_i \geq 0$ are the singular values of $\mathcal{A}$; only r of these are non-zero. The singular values are the non-negative square roots of the eigenvalues of $\mathcal{A}^T\mathcal{A}$.

$\mathcal{U}$ and $\mathcal{V}$ are such that $\mathcal{U}\mathcal{U}^T = \mathcal{V}\mathcal{V}^T = \mathcal{V}^T\mathcal{V} = \mathcal{I}_k$.

Note: there are a number of different definitions of SVD in the literature, in some of which Σ is square and U and V rectangular, as here, but in others U and V are square, and Σ is rectangular.

Examples:

$\mathcal{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$

`svd`$(\mathcal{Q}) = \left\{ \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1.0 & 0 \\ 0 & 5.0 \end{pmatrix}, \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \right\}$

20.55.6.44 spswap_columns, spswap_rows

Syntax:

`spswap_columns(`$\mathcal{A}$`,c1,c2);`

$\mathcal{A}$	:-	a sparse matrix.
c1,c1	:-	positive integers.

Synopsis:

`spswap_columns` swaps column c1 of $\mathcal{A}$ with column c2.

`spswap_rows` performs the same task on 2 rows of $\mathcal{A}$.

Examples: `spswap_columns`$(\mathcal{A}, 2, 3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 5 \\ 0 & 9 & 0 \end{pmatrix}$

Related functions:

`spswap_entries`.

20.55.6.45 swap_entries

Syntax:

`spswap_entries(`$\mathcal{A}$`,{r1,c1},{r2,c2});`

$\mathcal{A}$	:-	a sparse matrix.
r1,c1,r2,c2	:-	positive integers.

Synopsis:

`spswap_entries` swaps $\mathcal{A}$(r1,c1) with $\mathcal{A}$(r2,c2).

Examples: `spswap_entries`$(\mathcal{A}, \{1,1\}, \{3,3\}) = \begin{pmatrix} 9 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Related functions:

`spswap_columns`, `spswap_rows`.

20.55.6.46 spswap_rows

See: `spswap_columns`.

20.55.6.47 symmetricp

Syntax:

`symmetricp(`$\mathcal{A}$`);`

$\mathcal{A}$	:-	a matrix.

Synopsis:

`symmetricp` is a boolean function that returns t if the matrix is symmetric and nil otherwise.

Examples:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

$\texttt{symmetricp}(\mathcal{A}) = \texttt{nil}$

$\texttt{symmetricp}(\mathcal{M}) = \texttt{t}$

Related functions:

`matrixp`, `squarep`, `sparsematp`.

20.55.7 Fast Linear Algebra

By turning the `fast_la` switch on, the speed of the following functions will be increased:

spadd_columns	spadd_rows	spaugment_columns	spcol_dim
spcopy_into	spmake_identity	spmatrix_augment	spmatrix_stack
spminor	spmult_column	spmult_row	sppivot
spremove_columns	spremove_rows	sprows_pivot	squarep
spstack_rows	spsub_matrix	spswap_columns	spswap_entries
spswap_rows	symmetricp		

The increase in speed will be insignificant unless you are making a significant number(i.e: thousands) of calls. When using this switch, error checking is minimised. This means that illegal input may give strange error messages. Beware.

20.55.8 Acknowledgments

This package is an extention of the code from the Linear Algebra Package for REDUCE by Matt Rebbeck (cf. section 20.33).

The algorithms for `spcholesky`, `splu_decom`, and `spsvd` are taken from the book Linear Algebra – J.H. Wilkinson & C. Reinsch[3].

The `spgram_schmidt` code comes from Karin Gatermann's Symmetry package[4] for REDUCE.

20.56 SPDE: Finding Symmetry Groups of PDEs

The package SPDE provides a set of functions which may be used to determine the symmetry group of Lie- or point-symmetries of a given system of partial differential equations. In many cases the determining system is solved completely automatically. In other cases the user has to provide additional input information for the solution algorithm to terminate.

Author: Fritz Schwarz

The package SPDE provides a set of functions which may be applied to determine the symmetry group of Lie- or point-symmetries of a given system of partial differential equations. Preferably it is used interactively on a computer terminal. In many cases the determining system is solved completely automatically. In some other cases the user has to provide some additional input information for the solution algorithm to terminate. The package should only be used in compiled form.

For all theoretical questions, a description of the algorithm and numerous examples the following articles should be consulted: [Sch85c, Sch88, Sch87].

20.56.1 Description of the System Functions and Variables

The symmetry analysis of partial differential equations logically falls into three parts. Accordingly the most important functions provided by the package are:

Function name	Operation
`cresys`(⟨ *arguments* ⟩)	Constructs determining system
`simpsys`()	Solves determining system
`result`()	Prints infinitesimal generators and commutator table

Table 20.22: SPDE Functions

Some other useful functions for obtaining various kinds of output are:

Function name	Operation
`prsys()`	Prints determining system
`prgen()`	Prints infinitesimal generators
`comm`(U,V)	Prints commutator of generators U and V

Table 20.23: SPDE Useful Output Functions

There are several global variables defined by the system which should not be used for any other purpose than that given in Table 20.24 and 20.25. The three globals of the type integer are:

Variable name	Meaning
`nn`	Number of independent variables
`mm`	Number of dependent variables
`pclass`=0, 1 or 2	Controls amount of output

Table 20.24: SPDE Integer valued globals

In addition there are the following global variables of type operator:

Variable name	Meaning
X(I)	Independent variable x_i
U(ALFA)	Dependent variable u^{alfa}
U(ALFA,I)	Derivative of u^{alfa} w.r.t. x_i
DEQ(I)	i-th differential equation
SDER(I)	Derivative w.r.t. which DEQ(I) is resolved
GL(I)	i-th equation of determining system
GEN(I)	i-th infinitesimal generator
XI(I), ETA(ALFA) ZETA(ALFA,I)	See definition given in the references quoted in the introduction.
C(I)	i-th function used for substitution

Table 20.25: SPDE Operator type global variables

The differential equations of the system at issue have to be assigned as values to the operator deq i applying the notation which is defined in Table 20.25. The entries in the third and the last line of that Table have obvious extensions to higher derivatives.

The derivative w.r.t. which the i-th differential equation deq i is resolved has to be assigned to sder i. Exception: If there is a single differential equation and no assignment has been made by the user, the highest derivative is taken by default.

When the appropriate assignments are made to the variable deq, the values of nn and mm (Table 20.23) are determined automatically, i.e. they have not to be assigned by the user.

The function CRESYS may be called with any number of arguments, i.e.

```
cresys(); or cresys(deq 1,deq 2,... );
```

are legal calls. If it is called without any argument, all current assignments to deq are taken into account. Example: If deq 1, deq 2 and deq 3 have been assigned a differential equation and the symmetry group of the full system comprising all three equations is desired, equivalent calls are

```
cresys();   or   cresys(deq 1,deq 2,deq 3);
```

The first alternative saves some typing. If later in the session the symmetry group of deq 1 alone has to be determined, the correct call is

```
cresys deq 1;
```

after the determining system has bee created, simpsys which has no arguments may be called for solving it. The amount of intermediate output produced by simpsys is controlled by the global variable pclass with the default value 0. With pclass equal to 0, no intermediate steps are shown. With pclass equal to 1, all intermediate steps are displayed so that the solution algorithm may be followed through in detail. Each time the algorithm passes through the top of the main solution loop the message

```
Entering main loop
```

is written. pclass equal 2 produces a lot of LISP output and is of no interest for the normal user.

If with pclass=0 the procedure simpsys terminates without any response, the determining system is completely solved. In some cases simpsys does not solve the determining system completely in a single run. In general this is true if there are only genuine differential equations left which the algorithm cannot handle at present. If a case like this occurs, simpsys returns the remaining equations of the determining system. To proceed with the solution algorithm, appropriate assignments have to be transmitted by the user, e.g. the explicit solution for one of the returned differential equations. Any new functions which are introduced thereby must be operators of the form c(k) with the correct dependencies generated by a depend statement (see section 7.27). Its enumeration has to be chosen in agreement with the current number of functions which have alreday been introduced.

This value is returned by `simpsys` too.

After the determining system has been solved, the procedure `result`, which has no arguments, may be called. It displays the infinitesimal generators and its non-vanishing commutators.

20.56.2 How to Use the Package

In this Section it is explained by way of several examples how the package SPDE is used interactively to determine the symmetry group of partial differential equations. Consider first the diffusion equation which in the notation given above may be written as

```
deq 1:=u(1,1)+u(1,2,2);
```

It has been assigned as the value of deq 1 by this statement. There is no need to assign a value to sder 1 here because the system comprises only a single equation.

The determining system is constructed by calling

```
cresys(); or cresys deq 1;
```

The latter call is compulsory if there are other assignments to the operator deq i than for i=1.

The error message

```
***** Differential equations not defined
```

appears if there are no differential equations assigned to any deq.

If the user wants the determining system displayed for inspection before starting the solution algorithm he may call

```
prsys();
```

and gets the answer

```
GL(1):=2*df(eta(1),u(1),x(2)) - df(xi(2),x(2),2)

        - df(xi(2),x(1))

GL(2):=df(eta(1),u(1),2) - 2*df(xi(2),u(1),x(2))
```

```
GL(3):=df(eta(1),x(2),2) + df(eta(1),x(1))

GL(4):=df(xi(2),u(1),2)

GL(5):=df(xi(2),u(1)) - df(xi(1),u(1),x(2))

GL(6):=2*df(xi(2),x(2)) - df(xi(1),x(2),2)

        - df(xi(1),x(1))

GL(7):=df(xi(1),u(1),2)

GL(8):=df(xi(1),u(1))

GL(9):=df(xi(1),x(2))


The remaining dependencies

xi(2) depends on u(1),x(2),x(1)

xi(1) depends on u(1),x(2),x(1)

eta(1) depends on u(1),x(2),x(1)
```

The last message means that all three functions xi(1), xi(2) and eta(1) depend on x(1), x(2) and u(1). Without this information the nine equations gl(1) to gl(9) forming the determining system are meaningless. Now the solution algorithm may be activated by calling

```
simpsys();
```

If the print flag `pclass` has its default value which is 0 no intermediate output is produced and the answer is

```
Determining system is not completely solved

The remaining equations are

gl(1):=df(c(1),x(2),2) + df(c(1),x(1))

Number of functions is 16
```

```
The remaining dependencies

c(1) depends on x(2),x(1)
```

With `pclass` equal to 1 about 6 pages of intermediate output are obtained. It allows the user to follow through each step of the solution algorithm.

In this example the algorithm did not solve the determining system completely as it is shown by the last message. This was to be expected because the diffusion equation is linear and therefore the symmetry group contains a generator depending on a function which solves the original differential equation. In cases like this the user has to provide some additional information to the system so that the solution algorithm may continue. In the example under consideration the appropriate input is

```
df(c(1),x(1)) := - df(c(1),x(2),2);
```

If now the solution algorithm is activated again by

```
simpsys();
```

the solution algorithm terminates without any further message, i.e. there are no equations of the determining system left unsolved. To obtain the symmetry generators one has to say finally

```
result();
```

and obtains the answer

```
The differential equation

DEQ(1):=u(1,2,2) + u(1,1)


The symmetry generators are

GEN(1):= dx(1)

GEN(2):= dx(2)

GEN(3):= 2*dx(2)*x(1) + du(1)*u(1)*x(2)

GEN(4):= du(1)*u(1)
```

```
GEN(5):= 2*dx(1)*x(1) + dx(2)*x(2)

                        2
GEN(6):= 4*dx(1)*x(1)

       + 4*dx(2)*x(2)*x(1)

                             2
         + du(1)*u(1)*(x(2)  - 2*x(1))

GEN(7):= du(1)*c(1)

The remaining dependencies

c(1) depends on x(2),x(1)


Constraint

df(c(1),x(1)):= - df(c(1),x(2),2)


The non-vanishing commutators of the finite subgroup

COMM(1,3):= 2*dx(2)

COMM(1,5):= 2*dx(1)

COMM(1,6):= 8*dx(1)*x(1) + 4*dx(2)*x(2) - 2*du(1)*u(1)

COMM(2,3):= du(1)*u(1)

COMM(2,5):= dx(2)

COMM(2,6):= 4*dx(2)*x(1) + 2*du(1)*u(1)*x(2)

COMM(3,5):=  - (2*dx(2)*x(1) + du(1)*u(1)*x(2))

                           2
COMM(5,6):= 8*dx(1)*x(1)

          + 8*dx(2)*x(2)*x(1)
```

```
                                        2
   + 2*du(1)*u(1)*(x(2)  - 2*x(1))
```

The message "Constraint" which appears after the symmetry generators are displayed means that the function c(1) depends on x(1) and x(2) and satisfies the diffusion equation.

More examples which may used for test runs are given in the final section.

If the user wants to test a certain ansatz of a symmetry generator for given differential equations, the correct proceeding is as follows. Create the determining system as described above. Make the appropriate assignments for the generator and call PRSYS() after that. The determining system with this ansatz substituted is returned. Example: Assume again that the determining system for the diffusion equation has been created. To check the correctness for example of generator GEN 3 which has been obtained above, the assignments

```
xi(1):=0;  xi(2):=2*x(1);  eta(1):=x(2)*u(1);
```

have to be made. If now `prsys()` is called all gl(k) are zero proving the correctness of this generator.

Sometimes a user only wants to know some of the functions `zeta` for for various values of its possible arguments and given values of `mm` and `nn`. In these cases the user has to assign the desired values of `mm` and `nn` and may call the ZETAs after that. Example:

```
mm:=1;  nn:=2;

factor u(1,2),u(1,1),u(1,1,2),u(1,1,1);

on list;

zeta(1,1);

-u(1,2)*u(1,1)*df(xi(2),u(1))

-u(1,2)*df(xi(2),x(1))

        2
-u(1,1) *df(xi(1),u(1))

+u(1,1)*(df(eta(1),u(1)) -df(xi(1),x(1)))

+df(eta(1),x(1))
```

```
zeta(1,1,1);

-2*u(1,1,2)*u(1,1)*df(xi(2),u(1))

-2*u(1,1,2)*df(xi(2),x(1))

-u(1,1,1)*u(1,2)*df(xi(2),u(1))

-3*u(1,1,1)*u(1,1)*df(xi(1),u(1))

+u(1,1,1)*(df(eta(1),u(1)) -2*df(xi(1),x(1)))

                2
-u(1,2)*u(1,1)  *df(xi(2),u(1),2)

-2*u(1,2)*u(1,1)*df(xi(2),u(1),x(1))

-u(1,2)*df(xi(2),x(1),2)

        3
-u(1,1)  *df(xi(1),u(1),2)

        2
+u(1,1)  *(df(eta(1),u(1),2) -2*df(xi(1),u(1),x(1)))

+u(1,1)*(2*df(eta(1),u(1),x(1)) -df(xi(1),x(1),2))

+df(eta(1),x(1),2)
```

If by error no values to `mm` or `nn` and have been assigned the message

```
***** Number of variables not defined
```

is returned. Often the functions zeta are desired for special values of its arguments eta(alfa) and xi(k). To this end they have to be assigned first to some other variable. After that they may be evaluated for the special arguments. In the previous example this may be achieved by

```
z11:=zeta(1,1)$   z111:=zeta(1,1,1)$
```

Now assign the following values to xi 1, xi 2 and eta 1:

```
  xi 1:=4*x(1)**2; xi 2:=4*x(2)*x(1);

  eta 1:=u(1)*(x(2)**2  - 2*x(1));
```

They correspond to the generator gen 6 of the diffusion equation which has been obtained above. Now the desired expressions are obtained by calling

```
  z11;

                                    2
 - (4*u(1,2)*x(2) - u(1,1)*x(2)  + 10*u(1,1)*x(1)

 + 2*u(1))

  z111;

                                          2
 - (8*u(1,1,2)*x(2) - u(1,1,1)*x(2)  + 18*u(1,1,1)*x(1)

 +  12*u(1,1))
```

20.56.3 Test File

This appendix is a test file. The symmetry groups for various equations or systems of equations are determined. The variable `pclass` has the default value 0 and may be changed by the user before running it. The output may be compared with the results which are given in the references.

```
%The Burgers equations

deq 1:=u(1,1)+u 1*u(1,2)+u(1,2,2)$

cresys deq 1$ simpsys()$ result()$

%The Kadomtsev-Petviashvili equation

deq 1:=3*u(1,3,3)+u(1,2,2,2,2)+6*u(1,2,2)*u 1

       +6*u(1,2)**2+4*u(1,1,2)$

cresys deq 1$ simpsys()$ result()$

%The modified Kadomtsev-Petviashvili equation
```

```
deq 1:=u(1,1,2)-u(1,2,2,2,2)-3*u(1,3,3)

       +6*u(1,2)**2*u(1,2,2)+6*u(1,3)*u(1,2,2)$

cresys deq 1$ simpsys()$ result()$

%The real- and the imaginary part of the nonlinear
%Schroedinger equation

deq 1:= u(1,1)+u(2,2,2)+2*u 1**2*u 2+2*u 2**3$

deq 2:=-u(2,1)+u(1,2,2)+2*u 1*u 2**2+2*u 1**3$

%Because this is not a single equation
% the two assignments

sder 1:=u(2,2,2)$  sder 2:=u(1,2,2)$

%are necessary.

cresys()$ simpsys()$ result()$

%The symmetries of the system comprising
% the four equations

deq 1:=u(1,1)+u 1*u(1,2)+u(1,2,2)$

deq 2:=u(2,1)+u(2,2,2)$

deq 3:=u 1*u 2-2*u(2,2)$

deq 4:=4*u(2,1)+u 2*(u 1**2+2*u(1,2))$

sder 1:=u(1,2,2)$ sder 2:=u(2,2,2)$ sder 3:=u(2,2)$
sder 4:=u(2,1)$

%is obtained by calling

cresys()$ simpsys()$

df(c 5,x 1):=-df(c 5,x 2,2)$

df(c 5,x 2,x 1):=-df(c 5,x 2,3)$
```

```
simpsys()$  result()$

% The symmetries of the subsystem comprising equation 1
%  and 3 are obtained by

cresys(deq 1,deq 3)$ simpsys()$ result()$

% The result for all possible subsystems is discussed in
% detail in ``Symmetries and Involution Systems: Some
% Experiments in Computer Algebra'', contribution to the
% Proceedings of the Oberwolfach Meeting on Nonlinear
% Evolution Equations, Summer 1986, to appear.
```

20.57 SPECFN: Package for Special Functions

This special function package is separated into two portions to make it easier to handle. The packages are called SPECFN and SPECFN2. The first one is more general in nature, whereas the second is devoted to special special functions. Additional examples can be found in the files `specfn.tst` and `specfn2.tst` in the `packages/specfn` directory.

Authors: Chris Cannam, with contributions from Winfried Neun, Herbert Melenk, Victor Adamchik, Francis Wright, Alan Barnes and several others

20.57.1 Special Functions: Introduction

The package SPECFN is designed to provide algebraic and numeric manipulations of many common special functions, namely:

- The Exponential Integral, Sine & Cosine Integrals;
- The Hyperbolic Sine & Cosine Integrals;
- The Fresnel Integrals & Error function;
- The Gamma function;
- The Beta function;
- The psi function & its derivatives;
- The Bessel functions J and Y of the first and second kinds;
- The modified Bessel functions I and K;
- The Hankel functions $H^{(1)}$ and $H^{(2)}$;
- The Airy functions;
- The Kummer hypergeometric functions M and U;
- The Struve, Lommel and Whittaker functions;
- The Riemann Zeta function;
- The Dilog function;
- The Polylog and Lerch Phi functions;
- Lambert's W function;
- Associated Legendre Functions (Spherical and Solid Harmonics);

- 3j and 6j symbols, Clebsch-Gordan coefficients;
- Stirling Numbers;
- and some well-known constants.

All of the above functions (except Stirling numbers) are autoloading.

More information on all these functions may be found on the website DLMF:NIST although currently not all functions may conform to these standards.

All algorithms whose sources are uncredited are culled from series or expressions found in the Dover Handbook of Mathematical Functions[AS72].

A nice collection of example plots for special functions can be found in the file `specplot.tst` in the subfolder `plot` of the packages folder. These examples will reproduce a number of well-known pictures from [AS72].

20.57.2 Polynomial Functions: Introduction

Most of these polynomial functions are not autoloading. This package needs to be loaded before they may be used with the command:

```
load_package specfn;
```

20.57.2.1 Orthogonal Polynomial Functions

The polynomial function sets available are:

- Hermite Polynomials;
- Legendre Polynomials;
- Laguerre Polynomials;
- Chebyshev Polynomials;
- Jacobi Polynomials;
- Gegenbauer Polynomials;

20.57.2.2 Other Polynomial Functions

- Bernoulli Numbers & Polynomials;
- Euler Numbers & Polynomials;
- Fibonnacci Numbers & Polynomials;

20.57.3 Simplification and Approximation

All of the operators supported by this package have certain algebraic simplification rules to handle special cases, poles, derivatives and so on. Such rules are applied whenever they are appropriate. However, if the `rounded` switch is on, numeric evaluation is also carried out. Unless otherwise stated below, the result of an application of a special function operator to real or complex numeric arguments in rounded mode will be approximated numerically whenever it is possible to do so. All approximations are to the current precision.

Most algebraic simplifications within the special function package are defined in the form of a REDUCE ruleset. Therefore, in order to get a quick insight into the simplification rules one can use the ShowRules operator, e.g.

```
ShowRules BesselI;

                          1          ~z     - ~z
{besseli(~n,~z) => ----------------*(e   - e     )
                   sqrt(pi*2*~z)

                           1
  when numberp(~n) and ~n=---,
                           2

                          1          ~z     - ~z
 besseli(~n,~z) => ----------------*(e   + e     )
                   sqrt(pi*2*~z)

                              1
  when numberp(~n) and ~n= - ---,
                              2

 besseli(~n,~z) => 0

  when numberp(~z) and ~z=0
   and numberp(~n) and ~n neq 0,

 besseli(~n,~z) => besseli( - ~n,~z) when numberp(~n)

  and impart(~n)=0 and ~n=floor(~n) and ~n<0,

 besseli(~n,~z) => do*i(~n,~z)

  when numberp(~n) and numberp(~z) and *rounded,
```

```
df(besseli(~n,~z),~z)

     besseli(~n - 1,~z) + besseli(~n + 1,~z)
 => ----------------------------------------,
                        2

df(besseli(~n,~z),~z)

 => besseli(1,~z) when numberp(~n) and ~n=0}
```

Several REDUCE packages (such as Sum or Limits) obtain different (hopefully better) results for the algebraic simplifications when the SPECFN package is loaded, because the latter package contains some information which may be useful and directly applicable for other packages, e.g.:

```
sum(1/k^s,k,1,infinity);  % evaluates to zeta(s)
```

A record is kept of all values previously approximated, so that should a value be required which has already been computed to the current precision or greater, it can be simply looked up. This can result in some storage overheads, particularly if many values are computed which will not be needed again. In this case, the switch `savesfs` may be turned off in order to inhibit the storage of approximated values. The switch is on by default.

20.57.4 Integral Functions

The SPECFN package includes manipulation and limited numerical evaluation for some integral functions, namely

`erf`, `erfc`, `Si`, `Shi`, `si`, `Ci`, `Chi`, `Ei`, `Li`, `Fresnel_C`, and `Fresnel_S`.

The error function, its complement and the two Fresnel integrals are defined by:

$$\begin{aligned}
\operatorname{erf}(z) &= \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\,\mathrm{d}t \\
\operatorname{erfc}(z) &= \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2}\,\mathrm{d}t = 1 - \operatorname{erf}(z) \\
C(z) &= \int_0^z \cos\left(\frac{\pi}{2}t^2\right)\,\mathrm{d}t \\
S(z) &= \int_0^z \sin\left(\frac{\pi}{2}t^2\right)\,\mathrm{d}t
\end{aligned}$$

respectively.

The exponential and related integrals are defined by the following:

$$\begin{aligned}
\mathrm{Ei}(z) &= e^{-z}\int_z^\infty \frac{e^{-t}}{t+z}\,\mathrm{d}t \\
\mathrm{Li}(z) &= \int_0^z \frac{\mathrm{d}t}{\log t} \\
\mathrm{Si}(z) &= \int_0^z \frac{\sin t}{t}\,\mathrm{d}t \\
\mathrm{si}(z) &= -\int_z^\infty \frac{\sin t}{t}\,\mathrm{d}t = \mathrm{Si}(z) - \frac{\pi}{2} \\
\mathrm{Ci}(z) &= -\int_z^\infty \frac{\cos t}{t}\,\mathrm{d}t = \int_0^z \frac{\cos t - 1}{t}\,\mathrm{d}t + \log z + \gamma \\
\mathrm{Shi}(z) &= \int_0^z \frac{\sinh t}{t}\,\mathrm{d}t \\
\mathrm{Chi}(z) &= \int_0^z \frac{\cosh t - 1}{t}\,\mathrm{d}t + \log z + \gamma
\end{aligned}$$

where γ is Euler's constant (`Euler_gamma`).

The definitions of the exponential and related integrals, the derviatives and some limits are known, together with some simple properties such as symmetry conditions.

The numerical approximations for the integral functions suffer from the fact that the precision is not set correctly for values of the argument above 10.0 (approx.) and from the usage of summations even for large arguments.

$\mathrm{Li}(z)$ is simplified to $\mathrm{Ei}(\ln(z))$.

20.57.5 The Γ Function and Related Functions

20.57.5.1 The Γ Function

This is represented by the unary operator `Gamma`. The `Gamma` function is defined by the integral:

$$\Gamma(a) = \int_0^\infty e^{-t}t^{a-1}\,\mathrm{d}t.$$

Initial transformations applied with `rounded` off are: $\Gamma(n)$ for integral n is computed, $\Gamma(n+1/2)$ for integral n is rewritten to an expression in $\sqrt{\pi}$, $\Gamma(n+1/m)$ for natural n and m a positive integral power of 2 less than or equal to 64 is rewritten to an expression in $\Gamma(1/m)$, expressions with arguments at which there is a pole are replaced by `infinity`, and those with a negative (real) argument are rewritten so as to have positive arguments.

The algorithm used for numerical approximation is an implementation of an asymptotic series for $\ln(\Gamma)$, with a scaling factor obtained from the Pochhammer symbols.

An expression for $\Gamma'(z)$ in terms of Γ and ψ is included.

20.57.5.2 Incomplete Gamma Functions

There are two incomplete gamma functions defined in the literature, see the DLMF:NIST chapter on Incomplete Gamma functions:

$$\gamma(a,z) = \int_0^z e^{-t}t^{a-1}\,\mathrm{d}t,$$
$$\Gamma(a,z) = \int_z^\infty e^{-t}t^{a-1}\,\mathrm{d}t.$$

called (unnormalised) lower and upper incomplete gamma function, respectively. $\gamma(a,z)$ is provided by the binary function `m_gamma`, $\Gamma(a,z)$ by the two argument version of the function `Gamma`.

The normalised incomplete gamma function $P(a,z)$ is provided by the binary function `iGamma` and is defined as

$$P(a,z) = \frac{\gamma(a,z)}{\Gamma(a)}.$$

20.57.5.3 The Beta Functions

The binary function $B(a,b)$ is related to the Γ function[AS72] and is defined by

$$B(a,b) = \int_0^1 t^a(1-t)^b\,\mathrm{d}t = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

It is represented by the binary function `Beta`.

The unnormalised and nomalised incomplete Beta funtions are defined by

$$B_x(a,b) = \int_0^x t^a(1-t)^b\,\mathrm{d}t,$$
$$I_x(a,b) = \frac{B_x(a,b)}{B(a,b)}$$

respectively. $I_x(a,b)$ is represented by the ternary function `ibeta(a,b,x)`.

20.57.5.4 The Digamma Function, ψ

This is represented by the unary operator `psi`. It is defined as the logarithmic derivative of the Γ function:

$$\psi(z) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

Initial transformations for ψ are applied on a similar basis to those for Γ; where possible, $\psi(x)$ is rewritten in terms of $\psi(1)$ and $\psi(\frac{1}{2})$, and expressions with negative arguments are rewritten to have positive ones.

The algorithm for numerical evaluation of ψ is based upon an asymptotic series, with a suitable scaling.

Relations for the derivative and integral of ψ are included.

20.57.5.5 The Polygamma Functions, $\psi^{(n)}$

The nth derivative of the ψ function is represented by the binary operator `Polygamma`, whose first argument is n.

Initial manipulations on $\psi^{(n)}$ are few; where the second argument is 1 or $3/2$, the expression is rewritten to one involving the Riemann ζ function, and when the first is zero it is rewritten to ψ; poles are also handled.

Numerical evaluation is available for real and complex arguments. The algorithm used is again an asymptotic series with a scaling factor; for negative (second) arguments, a Reflection Formula is used, introducing a term in the nth derivative of $\cot(z\pi)$.

Simple relations for derivatives and integrals are provided.

20.57.6 Bessel Functions

Support is provided for the Bessel functions J and Y, the modified Bessel functions I and K, and the Hankel functions of the first and second kinds. The relevant operators are, respectively, `BesselJ`, `BesselY`, `BesselI`, `BesselK`, `Hankel1` and `Hankel2`, which are all binary operators.

The Bessel functions $J_\nu(z)$ and $Y_\nu(z)$ are solutions of the Bessel equation:

$$z^2\frac{\mathrm{d}^2w}{\mathrm{d}z^2} + z\frac{\mathrm{d}w}{\mathrm{d}z} + (z^2 - \nu^2)w = 0.$$

Bessel's function of the first kind, $J_\nu(z)$, has the series expansion:

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} (-1)^k \frac{(z/2)^{2k}}{k!\Gamma(\nu + k + 1)}\,.$$

Bessel's function of the second kind, $Y_\nu(z)$, (for non-integral ν) is defined by:

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

or by its limiting value:

$$Y_\nu(z) = \frac{1}{\pi}\frac{\partial J_\nu(z)}{\partial \nu}\bigg|_{\nu=n} + \quad \frac{(-1)^n}{\pi}\frac{\partial J_\nu(z)}{\partial \nu}\bigg|_{\nu=-n}.$$

It is sometimes known as Weber's function.

The Hankel functions are alternative solutions of the Bessel equation distinguished by their asymptotic behaviour as $z \to \infty$:

$$H_\nu^{(1)}(z) \sim \sqrt{\frac{2}{\pi z}}\exp\left(i\left(z - \frac{\nu\pi}{2} - \frac{\pi}{4}\right)\right),$$
$$H_\nu^{(2)}(z) \sim \sqrt{\frac{2}{\pi z}}\exp\left(-i\left(z - \frac{\nu\pi}{2} - \frac{\pi}{4}\right)\right).$$

The modified Bessel functions $I_\nu(z)$ and $K_\nu(z)$ are solutions of the modified Bessel equation:

$$z^2\frac{\mathrm{d}^2 w}{\mathrm{d}z^2} + z\frac{\mathrm{d}w}{\mathrm{d}z} - (z^2 + \nu^2)w = 0\,.$$

Since they may be obtained by replacing z by $\pm iz$ the modified Bessel functions are sometimes called *Bessel functions of imaginary argument*. $I_\nu(z)$ has the series expansion:

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{(z/2)^{2k}}{k!\Gamma(\nu + k + 1)},$$

whereas $K_\nu(z)$ is distinguished by its asymptotic behaviour:

$$K_\nu(z) \sim \sqrt{\frac{\pi}{2z}}e^{-z}$$

as $z \to \infty$. For more information, see the DLMF:NIST chapters on Hankel & Bessel functions and Modified Bessel functions.

The following initial transformations are performed:

- trivial cases or poles of J, Y, I and K are handled;
- J, Y, I and K with negative first argument are transformed to have positive first argument;
- J with negative second argument is transformed to have positive second argument;

- Y or K with non-integral or complex second argument is transformed into an expression in J or I respectively;
- derivatives of J, Y and I are carried out;
- derivatives of K with zero first argument are carried out;
- derivatives of Hankel functions are carried out.

Also, if the `complex` switch is on and `rounded` is off, expressions in Hankel functions are rewritten in terms of Bessel functions.

No numerical approximation is provided for the Bessel K function, or for the Hankel functions for anything other than special cases. The algorithms used for the other Bessel functions are generally implementations of standard ascending series for J, Y and I, together with asymptotic series for J and Y; usually, the asymptotic series are tried first, and if the argument is too small for them to attain the current precision, the standard series are applied. An obvious optimization prevents an attempt with the asymptotic series if it is clear from the outset that it will fail.

There are no rules for the integration of Bessel and Hankel functions.

20.57.7 Airy Functions

Support is provided for the Airy Functions Ai and Bi and for their derivatives Ai' and Bi'. The relevant operators are respectively `Airy_Ai`, `Airy_Bi`, `Airy_Aiprime` and `Airy_Biprime`, which are all unary.

Airy functions are solutions of the differential equation:

$$\frac{\mathrm{d}^2 w}{\mathrm{d}z^2} = zw.$$

Trivial cases of Airy_Ai and Airy_Bi and their primes are evaluated, and all functions accept both real and complex arguments.

The Airy Functions can also be represented in terms of Bessel Functions by activating an inactive rule set:

```
let Airy2Bessel_rules;
```

As a result the `Airy_Ai` function will be evaluated using the formula:

$$\mathtt{Ai}(z) = \frac{1}{3}\sqrt{z}\left[I_{-1/3}(\zeta) - I_{1/3}(\zeta)\right], \text{ where } \zeta = \frac{2}{3}z^{\frac{2}{3}}.$$

Note: In order to obtain satisfactory approximations to numerical values both the `complex` and `rounded` switches must be on.

The algorithms used for the Airy Functions are implementations of standard ascending series, together with asymptotic series. At some point it is better to use the asymptotic rather than the ascending series, which is calculated by the program and depends on the given precision.

There are no rules for the integration of Airy Functions.

20.57.8 Hypergeometric and Other Functions

This package also provides some support for other functions, in the form of algebraic simplifications:

- The Struve **H** and **L** functions, through the binary operators `StruveH` and `StruveL`, for which manipulations are provided to handle special cases, simplify to more readily handled functions where appropriate, and differentiate with respect to the second argument. These functions with arguments ν and x are solutions of the differential equation:
$$\frac{\mathrm{d}^2w}{\mathrm{d}x^2} + \frac{1}{x}\frac{\mathrm{d}w}{\mathrm{d}x} + \left(1 - \frac{\nu^2}{x^2}\right) w = \frac{(z/2)^{\nu-1}}{\sqrt{\pi}\Gamma(\nu + 1/2)}\,.$$

- The Lommel functions of the first and second kinds, through the ternary operators `Lommel1` and `Lommel2` with arguments ν, μ and x may be considered generalisations of the Struve functions satisfying the differential equation:
$$\frac{\mathrm{d}^2w}{\mathrm{d}x^2} + \frac{1}{x}\frac{\mathrm{d}w}{\mathrm{d}x} + \left(1 - \frac{\nu^2}{x^2}\right) w = z^{\mu-1}\,.$$
Manipulations are provided to handle special cases and simplify where appropriate.

- The Kummer confluent hypergeometric functions M and U (the hypergeometric ${}_1F_1$ or Φ, and $z^{-a}{}_2F_0$ or Ψ, respectively), represented by the ternary operators `KummerM` and `KummerU` with arguments a, b and x, are solutions of the differential equation:
$$\frac{\mathrm{d}^2w}{\mathrm{d}x^2} + (b - x)\frac{\mathrm{d}w}{\mathrm{d}x} - aw = 0\,.$$
There are manipulations for special cases and simplifications, derivatives and, for the M function, numerical approximations for real arguments.

- The Whittaker M and W functions are variations upon the Kummer functions, which are represented by the ternary operators `WhittakerM` and `WhittakerW` with arguments κ, μ and x. They satisfy the Whittaker differential equation:
$$\frac{\mathrm{d}^2W}{\mathrm{d}x^2} + \left(\frac{1 - 4\mu^2}{4x^2} + \frac{\kappa}{x} - \frac{1}{4}\right) W = 0\,,$$

which is obtained from the Kummer differential equation via the substituions

$$W = e^{z/2} z^{\mu+1/2} w, \qquad \kappa = b/2 - a \qquad \mu = (b-1)/2\,.$$

The Whittaker M and W functions with non-numeric arguments are simplified to expressions involving the Kummer M and U functions respectively.

20.57.9 The Riemann Zeta Function

This is represented by the unary operator `Zeta` and defined by the formula:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

With `rounded` off, $\zeta(z)$ is evaluated numerically for even integral arguments in the range $-31 < z < 31$, and for odd integral arguments in the range $-30 < z < 16$. Outside this range the values become a little unwieldy.

Numerical evaluation of ζ is only carried out if the argument is real. The algorithms used for ζ are: for odd integral arguments, an expression relating $\zeta(n)$ with $\psi^{n-1}(3)$; for even arguments, a trivial relationship with the Bernoulli numbers; and for other arguments the approach is either (for larger arguments) to take the first few primes in the standard over-all-primes expansion, and then continue with the defining series with natural numbers not divisible by these primes, or (for smaller arguments) to use a fast-converging series obtained from [BO78].

There are no rules for differentiation or integration of ζ.

20.57.10 Polylogarithm and Related Functions

The dilogarithm function $Li_2(z)$ is defined by

$$\mathrm{Li}_2(z) \equiv \sum_{n=1}^{\infty} \frac{z^n}{n^2} = -\int_0^z \frac{\log(1-t)}{t}\,\mathrm{d}t$$

and represented by the unary function `dilog`.

The polylogarithm function $Li_s(z)$ is defined by

$$\mathrm{Li}_s(z) \equiv \sum_{n=1}^{\infty} \frac{z^n}{n^s} = \frac{z}{\Gamma(s)} \int_0^{\infty} \frac{t^{s-1}}{e^t - z}\,\mathrm{d}t.$$

and represented by the binary function `Polylog`. The case $s = 2$ is, of course, the dilogarithm function and the special case when $z = 1$ gives the Riemann zeta

function $\zeta(s)$. For $s = 1$, the polylogarithm reduces to the elementary function: $-\log(1-t)$.

Lerch's transcendent or Lerch Phi function is defined by

$$\Phi(z,s,a) = \sum_{n=0}^{\infty} \frac{z^n}{(n+a)^s}.$$

It is represented by the ternary function `Lerch_Phi(z,s,a)`. For the special case $a = 1$, Lerch's function is related to a polylogarithm: $zLi_s(z) = \Phi(z,s,1)$.

20.57.11 Lambert's W Function

Lambert's function $\omega(x)$, represented by the unary operator `Lambert_W`, is the inverse of the function $x = we^w$. Therefore it is an important contribution for the solve package.

For real-valued arguments $\omega(x)$ is only real-valued in the interval $(-1/e, \infty)$. In the interval $(-1/e, 0)$, it is double-valued with a branch point at the point (-1/e, -1) where $\omega'(x)$ is singular. The positive branch is defined on the interval $(-1/e, \infty)$ where it is monotonically increasing with $\omega(x) > -1$. The negative branch is defined on the interval $(-1/e, 0)$ where it is monotonically decreasing with $\omega(x) < -1$.

Simplification rules for $\omega(x)$ are provided for the special arguments 0 and $-1/e$ and for its logarithm, derivative and integral. A previous rule for its exponential caused problems with power series expansions about zero and has been deactivated. This does not seem to impact on the SOLVE package. However, this rule may be reactivated if required by

```
   let lambert_exp_rule;
%  and deactivated again by
   clear lambert_exp_rule;
```

The function is studied extensively in [HCGJ92]. The current implementation will compute values on the principal branch for all complex numerical arguments only if the switch `rounded` is on. However, since the numerical computations are carried out in `complex-rounded` mode, it is also better to turn the switch `complex` to on to avoid repeated irritating mode change warnings.

The real positive branch is part of the principal branch and currently there is no way of computing values on the real negative branch or indeed any non-principal values.

20.57.12 Spherical and Solid Harmonics

The relevant operators are, respectively,
`SolidHarmonicY` and `SphericalHarmonicY`.

The `SolidHarmonicY` operator implements the Solid Harmonics described below. It expects 6 parameter, namely n, m, x, y, z and $r2$ and returns a polynomial in x, y, z and $r2$.

The operator `SphericalHarmonicY` is a special case of `SolidHarmonicY` with the usual definition:

```
algebraic procedure SphericalHarmonicY(n,m,theta,phi);
        SolidHarmonicY(n,m,sin(theta)*cos(phi),
                sin(theta)*sin(phi),cos(theta),1)$
```

Solid Harmonics of order n (Laplace polynomials) are homogeneous polynomials of degree n in x, y, z which are solutions of the Laplace equation:-

```
df(P,x,2) + df(P,y,2) + df(P,z,2) = 0.
```

There are $2n + 1$ independent such polynomials for any given $n \geq 0$ and with:-

```
w!0 = z, w!+ = i*(x-i*y)/2, w!- = i*(x+i*y)/2,
```

they are given by the Fourier integral:-

```
S(n,m,w!-,w!0,w!+) =
       (1/(2*pi)) *
       for u:=-pi:pi integrate(w!0 + w!+ * exp(i*u)
            + w!- * exp(-i*u))^n * exp(i*m*u) * du;
```

which is obviously zero if $|m| > n$ since then all terms in the expanded integrand contain the factor e^{iku} with $k \neq 0$.

$S(n, m, x, y, z)$ is proportional to

```
r^n * Legendre(n,m,cos theta) * exp(i*phi)
```

where $r^2 = x^2 + y^2 + z^2$.

The spherical harmonics are simply the restriction of the solid harmonics to the surface of the unit sphere and the set of all spherical harmonics with $n \geq 0, -n \leq m \leq n$ form a complete orthogonal basis on it, i.e. $\langle n, m|n', m'\rangle = \delta_{n,n'}\delta_{m,m'}$ using $\langle \ldots | \ldots \rangle$ to designate the scalar product of functions over the spherical sur-

face.

The coefficients of the solid harmonics are normalised in what follows to yield an orthonormal system of spherical harmonics.

Given their polynomial nature, there are many recursions formulae for the solid harmonics and any recursion valid for Legendre functions can be 'translated' into solid harmonics. However the direct proof is usually far simpler using Laplace's definition.

It is also clear that all differentiations of solid harmonics are trivial, qua polynomials.

Some substantial reduction in the symbolic form would occur if one maintained throughout the recursions the symbol $r2$ (r cannot occur as it is not rational in x, y, z). Formally the solid harmonics appear in this guise as more compact polynomials in $x, y, z, r2$.

Only two recursions are needed:-

(i) along the diagonal (n, n);

(ii) along a line of constant n: $(m, m), (m + 1, m), \ldots, (n, m)$.

Numerically these recursions are stable.

For $m < 0$ one has:-

$$S(n, m, x, y, z) = (-1)^m S(n, -m, x, -y, z).$$

20.57.13 3j symbols and Clebsch-Gordan Coefficients

The operators `ThreeJSymbol` and `Clebsch_Gordan` are defined as in [LB68] or [Edm57] and expect as arguments three lists of values $\{j_i, m_i\}$, e.g.

```
ThreeJSymbol({J+1,M},{J,-M},{1,0});
Clebsch_Gordan({2,0},{2,0},{2,0});
```

20.57.14 6j symbols

The operator `SixJSymbol` is defined as in [LB68] or [Edm57] and expects two lists of values $\{j_1, j_2, j_3\}$ and $\{l_1, l_2, l_3\}$ as arguments, e.g.

```
SixJSymbol({7,6,3},{2,4,6});
```

In the current implementation of `SixJSymbol` there is only limited reasoning about the minima and maxima of the summation using the INEQ package, such that in most cases the special 6j-symbols (see e.g. [LB68]) will not be found.

20.57.15 Stirling Numbers

The Stirling numbers of the first and second kind are computed by calling the binary operators `Stirling1` and `Stirling2` respectively.

Stirling numbers of the first kind have the generating function:

$$\sum_{m=0}^{n} s_n^m x^m = (x-n+1)_n$$

where $(x-n+1)_n$ is the Pochhammer symbol. This provides a convenient way of calculating these Stirling numbers by extracting coefficients of the polynomial obtained by evaluating the Pochhammer symbol. REDUCE however uses an explicit summation.

Stirling numbers of the second kind are defined by the formula:

$$S_n^m = \frac{1}{m!}\sum_{k=0}^{m}(-1)^{m-k}\binom{m}{k}k^n.$$

REDUCE uses this explicit summation to evaluate Stirling numbers of the second kind.

20.57.16 Constants

The following well-known constants are defined in the REDUCE core, but the code for computing their numerical value when the switch `ROUNDED` is on is contained in the special function package.

- `Euler_Gamma` : Euler's constant, also available as $-\psi(1)$;
- `Catalan` : Catalan's constant;
- `Khinchin` : Khinchin's constant, defined in [Khi64] (which takes a lot of time to compute);
- `Golden_Ratio` : $\dfrac{1+\sqrt{5}}{2}$

20.57.17 Orthogonal Polynomials

All the polynomials in this section take two or more parameters; the first is the degree of the polynomial and the last is its argument. Any remaining arguments are parameters which in the literature are normally rendered as subscripts and superscripts. First, the definitions appropriate to all the sets of orthogonal polynomials in the following subsections are listed.

A set of polynomials $\{p_n(x)\}$, $n = 0, 1, \ldots$ are said to be *orthogonal* on open interval (a, b) (where a and/or b may be infinite) with positive weight function $w(x)$ if

$$\int_a^b p_n(x)p_m(x)w(x)\mathrm{d}x = 0 \qquad \text{when } m \neq n.$$

This defines each polynomial $p_n(x)$ up to a constant factor c_n which is usually fixed by normalisation. If these factors are chosen so that

$$h_n = \int_a^b (p_n(x))^2 w(x)\mathrm{d}x = 1 \qquad \text{i.e. } c_n = \sqrt{h_n}$$

then the polynomial set is said to be *orthonormal*. An alternative normalisation, that is sometimes used, is to set the leading term of each polynomial $k_n = 1$. The polynomial set is then said to be *monic*.

In REDUCE the normalisation is chosen so that the polynomial sets are orthonormal and hence $k_n \neq 1$ in general. In the subsections below on each of the polynomial sets, the interval (a, b) over which the polynomials are orthogonal, the weight function $w(x)$ and the leading coefficient k_n of the polynomial of degree n are given together with any constraints on any additional parameters. Also given are what might be called the 'first moment' $\tilde{h}_n$ of the nth polynomial defined by:

$$\tilde{h}_n = \int_a^b x(p_n(x))^2 w(x)\,\mathrm{d}x$$

and the ratio

$$r_n = \frac{\tilde{k}_n}{k_n} \quad \text{where } p_n(x) = k_n x^n + \tilde{k}_n x^{n-1} \ldots$$

These quantities may be used in recurrence relations when generating the polynomials.

20.57.17.1 Legendre Polynomials

The function call `LegendreP(n,x)` will return the nth Legendre polynomial if n is a non-negative integer; otherwise the result will involve the original operator `LegendreP` or on graphical interfaces $P_n(x)$ will be output.

The interval of definition is $(-1, 1)$, the weight function $w(x) = 1$ and, for the orthonormal case, the leading coefficients are given by $k_n = 2^n(\frac{1}{2})_n/n!$ where $(\frac{1}{2})_n$ is the Pochhammer symbol. Also $\tilde{h}_n = \frac{2}{2n+1}$ and $r_n = 0$.

20.57.17.2 Associated Legendre Functions

The function call `LegendreP(n,m,x)` will return the nth associated Legendre function if n and m are integers with $0 \leq m \leq n$; otherwise the result will in-

volve the original operator `LegendreP` or on graphical interfaces $P_n^{(m)}(x)$ will be output.

They are defined by

$$P_n^{(m)}(x) = (-1)^m (1-x^2)^{m/2} \frac{\mathrm{d}^m P_n(x)}{\mathrm{d}x^m} ;$$

it should be noted that they are only polynomials if m is even. Currently the extension of these functions to negative n and m is not implemented in REDUCE.

For fixed m these functions are orthogonal over the interval $(-1, 1)$; the weight function being $w(x) = 1$. However, unlike the polynomials in the rest of this section, they are *not* orthonormal:

$$\int_{-1}^{1} \left(P_n^{(m)}(x)\right)^2 \mathrm{d}x = h_n = \frac{2(l+m)!}{(2l+1)!(l-m)!} .$$

20.57.17.3 Chebyshev Polynomials

The function call `ChebyshevT(n,x)` will return the nth Chebyshev polynomial of the first kind if n is a non-negative integer; otherwise the result will involve the original operator `ChebyshevT` or on graphical interfaces $T_n(x)$ will be output.

The interval of definition is $(-1, 1)$, the weight function $w(x) = (1-x^2)^{-1/2}$ and, for the orthonormal case, the leading coefficients are given by $k_n = 2^{n-1}$ for $n > 0$; $k_0 = 1$. Also $\tilde{h}_n = \pi/2$ for $n > 0$; $\tilde{h}_0 = \pi$ and $r_n = 0$.

The function call `ChebyshevU(n,x)` will return the nth Chebyshev polynomial of the second kind if n is a non-negative integer; otherwise the result will involve the original operator `ChebyshevU` or on graphical interfaces $U_n(x)$ will be output.

The interval of definition is $(-1, 1)$, the weight function $w(x) = (1-x^2)^{-1/2}$ and, for the orthonormal case, the leading coefficients are given by $k_n = 2^n$, $\tilde{h}_n = \pi/2$ and $r_n = 0$.

20.57.17.4 Gegenbauer Polynomials

The function call `GegenbauerP(n,a,x)` will return the Gegenbauer polynomial of degree n and parameter a if n is a non-negative integer and a is numerical; otherwise the result will involve the original operator `GegenbauerP` or on graphical interfaces $C_n^{(a)}(x)$ will be output.

The interval of definition is $(-1, 1)$, the weight function $w(x) = (1-x^2)^{a-1/2}$ and, for the orthonormal case, the leading coefficients are given by $k_n = 2^n (a)_n / n!$

where $(a)_n$ is the Pochhammer symbol. The parameter a should satisfy $a > -1/2,\ \ a \neq 0$. Also

$$\tilde{h}_n = \frac{2^{1-2a}\pi\Gamma(n+2a)}{(n+a)(\Gamma(a))^2 n!} \qquad \text{and} \quad r_n = 0\,.$$

20.57.17.5 Jacobi Polynomials

The function call `JacobiP(n,a,b,x)` will return the Jacobi polynomial of degree n and parameters a and b if n is a non-negative integer and a and b are numerical; otherwise the result will involve the original operator `JacobiP` or on graphical interfaces $P_n^{(a,b)}(x)$ will be output.

The interval of definition is $(-1, 1)$, the weight function $w(x) = (1-x)^a(1+x)^b$ and, for the orthonormal case, the leading coefficients are given by

$$\tilde{h}_n = \frac{(n+a+b+1)_n}{2^n n!}$$

where $(n+a+b+1)_n$ is the Pochhammer symbol. The parameters a and b should satisfy $a > -1,\ \ b > -1$. Also

$$\begin{aligned}
\tilde{h}_0 &= 2^{a+b+1}\frac{\Gamma(a+1)\Gamma(b+1)}{\Gamma(a+b+2)} \\
\tilde{h}_n &= 2^{a+b+1}\frac{\Gamma(n+a+1)\Gamma(n+b+1)}{(2n+a+b+1)\Gamma(n+a+b+1)n!} \quad \text{for } n > 0 \\
r_n &= \frac{n(a-b)}{2n+a+b}.
\end{aligned}$$

The Legendre, Chebyshev and Gegenbauer polynomials are all, in fact, special cases of the Jacobi polynomials.

20.57.17.6 Laguerre Polynomials

The function call `LaguerreP(n,x)` will return the nth Laguerre polynomial if n is a non-negative integer; otherwise the result will involve the original operator `LaguerreP` or on graphical interfaces $L_n(x)$ will be output.

The interval of definition is $(0, \infty)$, the weight function $w(x) = e^{-x}$ and, for the orthonormal case, the leading coefficients are given by $k_n = (-1)^n/n!$, $\tilde{h}_n = 1$ and $r_n = -n^2$.

20.57.17.7 Generalised Laguerre Polynomials

If used with three arguments `LaguerreP(n,a,x)` returns the nth generalised (or associated) Laguerre polynomial if n is a non-negative integer and a is nu-

meric; otherwise the result will involve the original operator `LaguerreP` or on graphical interfaces $L_n^{(a)}(x)$ will be output. These are more properly called *Sonin polynomials* after their discoverer N. Y. Sonin.

The interval of definition is $(0, \infty)$, the weight function $w(x) = e^{-x}x^a$ and, for the orthonormal case, the leading coefficients are given by $k_n = (-1)^n/n!$, $\tilde{h}_n = \Gamma(n + a + 1)/n!$ and $r_n = -n(n + a)$. The parameter a should satisfy $a > -1$.

20.57.17.8 Hermite Polynomials

The function call `HermiteP(n,x)` will return the nth Hermite polynomial if n is a non-negative integer; otherwise the result will involve the original operator `HermiteP` or on graphical interfaces $H_n(x)$ will be output.

The interval of definition is $(-\infty, +\infty)$, the weight function $w(x) = e^{-x^2}$ and, for the orthonormal case, the leading coefficients are given by $k_n = 2^n$, $\tilde{h}_n = \sqrt{\pi}2^n n!$ and $r_n = 0$.

20.57.18 Other Polynomials and Related Numbers

20.57.18.1 Fibonacci Polynomials

`FibonacciP(n,x)` returns the nth Fibonacci polynomial in the variable x. If n is an integer, it will be evaluated using the recursive definition:

$$F_0(x) = 0; \qquad F_1(x) = 1; \qquad F_n(x) = xF_{n-1}(x) + F_{n-2}(x)\,.$$

The recursion is, of course, optimised as a simple loop to avoid repeated computation of lower-order polynomials.

20.57.18.2 Euler Numbers and Polynomials

Euler numbers are computed by the unary operator `Euler`; the call `Euler(n)` returns the nth Euler number; all the odd Euler numbers are zero. The computation is derived directly from Pascal's triangle of binomial coefficients.

The Euler numbers and polynomials have the following generating functions:

$$\frac{2e^t}{1+e^{2t}} = \sum_{n=0}^{\infty} \frac{E_n t^n}{n!}, \qquad \frac{e^{xt}}{1+e^t} = \sum_{n=0}^{\infty} \frac{E_n(x) t^n}{n!}$$

respectively. Thus $E_0 = 1$ and $E_1 = 0$. Furthermore the numbers and polynomials

are related by the equations:

$$E_n = 2^n E_n\left(\frac{1}{2}\right), \qquad E_n(x) = \sum_{k=0}^{n} \binom{n}{k} \frac{E_k}{2^k} \left(x - \frac{1}{2}\right)^{n-k}.$$

The Euler polynomials are evaluated for non-negative integer n by using the summation immediately above.

20.57.18.3 Bernoulli Numbers & Polynomials

The call `Bernoulli(n)` evaluates to the nth Bernoulli number; all of the odd Bernoulli numbers, except `Bernoulli(1)`, are zero.

The algorithms for Bernoulli numbers used are based upon those by Herbert Wilf, presented by Sandra Fillebrown [Fil92]. If the `rounded` switch is off, the algorithms are exactly those; if it is on, some further rounding may be done to prevent computation of redundant digits. Hence, these functions are particularly fast when used to approximate the Bernoulli numbers in rounded mode.

The Bernoulli numbers and polynomials have the following generating functions:

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} \frac{B_n t^n}{n!}, \qquad \frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} \frac{B_n(x) t^n}{n!}$$

respectively. Thus $B_0 = 1$ and $B_1 = -\frac{1}{2}$. Furthermore the numbers and polynomials are related by the equations:

$$B_n = B_n(0), \qquad B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}.$$

The Bernoulli polynomials are evaluated for non-negative integer n by using the summation immediately above.

Both the Bernoulli and Euler numbers and polynomials may also be calculated directly by expanding the corresponding generating function as a power series in t using either the TPS or TAYLOR package, extracting the nth term and multiplying by $n!$. The use of the TPS package is probably preferable here as the series for the generating function is extendible and need only be calculated once; it will be extended automatically if higher order numbers or polynomials are required.

20.57.19 Function Bases

The following procedures compute sets of functions e.g. to be used for approximation. All procedures have two parameters, the expression to be used as $variable$

(an identifier in most cases) and the order of the desired system. The functions are not scaled to a specific interval, but the $variable$ can be accompanied by a scale factor and/or a translation in order to map the generic interval of orthogonality to another (e.g. $(x - 1/2) * 2pi$). The result is a function list with ascending order, such that the first element is the function of order zero and (for the polynomial systems) the function of order n is the $n + 1$-th element.

```
monomial_base(x,n)          {1,x,...,x**n}
trigonometric_base(x,n)     {1,sin x,cos x,
                               sin(2x),cos(2x)...}
Bernstein_base(x,n)         Bernstein polynomials
Legendre_base(x,n,a,b)      Legendre polynomials
Laguerre_base(x,n,a)        Laguerre polynomials
Hermite_base(x,n)           Hermite polynomials
Chebyshev_base_T(x,n)       Chebyshev polynomials
                             of the first kind
Chebyshev_base_U(x,n)       Chebyshev polynomials
                             of the second kind
Gegenbauer_base(x,n,a)      Gegenbauer polynomials
```

Example:

```
Bernstein_base(x,5);

        5      4       3       2
   { - X  + 5*X  - 10*X  + 10*X  - 5*X + 1,

       4      3      2
    5*X*(X  - 4*X  + 6*X  - 4*X + 1),

        2      3      2
    10*X *( - X  + 3*X  - 3*X + 1),

        3   2
    10*X *(X  - 2*X + 1),

       4
    5*X *( - X + 1),

     5
    X }
```

20.57.20 Acknowledgements

The contributions of Kerry Gaskell, Matthew Rebbeck, Lisa Temme, Stephen Scowcroft and David Hobbs (all students from the University of Bath on placement in ZIB Berlin for one year) were very helpful to augment the package. The advice of René Grognard (CSIRO, Australia) for the development of the module for Clebsch-Gordan and 3j, 6j symbols and the module for spherical and solid harmonics was very much appreciated.

20.57.21 Tables of Operators and Constants

Special Functions

Function	Operator
$\mathrm{Si}(z)$	`Si(z)`
$\mathrm{Si}(z) - \pi/2$	`s_i(z)`
$\mathrm{Ci}(z)$	`Ci(z)`
$\mathrm{Shi}(z)$	`Shi(z)`
$\mathrm{Chi}(z)$	`Chi(z)`
$\mathrm{erf}(z)$	`Erf(z)`
$1 - \mathrm{erf}(z)$	`erfc(z)`
$\mathrm{Ei}(z)$	`Ei(z)`
$\mathrm{Ei}(\log(z))$	`Li(z)`
$C(x)$	`Fresnel_C(x)`
$S(x)$	`Fresnel_S(x)`
$B(a,b)$	`Beta(a,b)`
$\Gamma(a)$	`Gamma(a)`
normalized incomplete Beta $I_x(a,b) = \frac{B_x(a,b)}{B(a,b)}$	`iBeta(a,b,x)`
normalized (lower) incomplete Gamma $P(a,z) = \frac{\gamma(a,z)}{\Gamma(a)}$	`iGamma(a,z)`
incomplete (lower) Gamma $\gamma(a,z)$	`m_gamma(a,z)`
incomplete (upper) Gamma $\Gamma(a,z)$	`Gamma(a,z)`
$(a)_k$	`Pochhammer(a,k)`
$\psi(z)$	`Psi(z)`
$\psi^{(n)}(z)$	`Polygamma(n,z)`
$J_\nu(z)$	`BesselJ(nu,z)`
$Y_\nu(z)$	`BesselY(nu,z)`
$I_\nu(z)$	`BesselI(nu,z)`
$K_\nu(z)$	`BesselK(nu,z)`
$H_n^{(1)}(z)$	`Hankel1(n,z)`
$H_n^{(2)}(z)$	`Hankel2(n,z)`

More Special Functions

Function	Operator
$\mathrm{Ai}(z)$	`Airy_Ai(z)`
$\mathrm{Bi}(z)$	`Airy_Bi(z)`
$\mathrm{Ai}'(z)$	`Airy_Aiprime(z)`
$\mathrm{Bi}'(z)$	`Airy_Biprime(z)`
$\mathbf{H}_\nu(z)$	`StruveH(nu,z)`
$\mathbf{L}_\nu(z)$	`StruveL(nu,z)`
$s_{a,b}(z)$	`Lommel1(a,b,z)`
$S_{a,b}(z)$	`Lommel2(a,b,z)`
$M(a,b,z)$ or ${}_1F_1(a,b;z)$ or $\Phi(a,b;z)$	`KummerM(a,b,z)`
$U(a,b,z)$ or $z^{-a}{}_2F_0(a,b;z)$ or $\Psi(a,b;z)$	`KummerU(a,b,z)`
Expression in Kummer_M	`WhittakerM(kappa,mu,z)`
Expression in Kummer_U	`WhittakerW(kappa,mu,z)`
Riemann's $\zeta(z)$	`zeta(z)`
Lambert $\omega(z)$	`Lambert_W(z)`
$\mathrm{Li}_2(z)$	`dilog(z)`
$\mathrm{Li}_n(z)$	`Polylog(n,z)`
Lerch's transcendent $\Phi(z,s,a)$	`Lerch_Phi(z,s,a)`

Function	Operator
$Y_n^m(x,y,z,r2)$	`SolidHarmonicY(n,m,x,y,z,r2)`
$Y_n^m(\theta,\phi)$	`SphericalHarmonicY(n,m,theta,phi)`
$\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix}$	`ThreeJSymbol({j1,m1},{j2,m2},{j3,m3})`
$(j_1 m_1 j_2 m_2 \mid j_1 j_2 j_3 - m_3)$	`Clebsch_Gordan({j1,m1},{j2,m2},{j3,m3})`
$\begin{Bmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{Bmatrix}$	`SixJSymbol({j1,j2,j3},{l1,l2,l3})`

Polynomial Functions

Function	Operator
Fibonacci Polynomials $F_n(x)$	`FibonacciP(n,x)`
$B_n(x)$	`BernoulliP(n,x)`
$E_n(x)$	`EulerP(n,x)`
$H_n(x)$	`HermiteP(n,x)`
$L_n(x)$	`LaguerreP(n,x)`
Generalised Laguerre $L_n^{(m)}(x)$	`LaguerreP(n,m,x)`
$P_n(x)$	`LegendreP(n,x)`
Associated Legendre $P_n^{(m)}(x)$	`LegendreP(n,m,x)`
$U_n(x)$	`ChebyshevU(n,x)`
$T_n(x)$	`ChebyshevT(n,x)`
$C_n^{(\alpha)}(x)$	`GegenbauerP(n,alpha,x)`
$P_n^{(\alpha,\beta)}(x)$	`JacobiP(n,alpha,beta,x)`

Well-known Numbers and Reserved Constants

Function	Operator
$\binom{n}{m}$	`Binomial(n,m)`
Fibonacci Numbers F_n	`Fibonacci(n)`
s_n^m	`Stirling1(n,m)`
S_n^m	`Stirling2(n,m)`
Bernoulli(n) or B_n	`Bernoulli(n)`
Euler(n) or E_n	`Euler(n)`
Motzkin(n) or M_n	`Motzkin(n)`
Constant	**REDUCE name**
Square root of (-1)	`i`
π	`pi`
Base of natural logarithms	`e`
Euler's γ constant	`Euler_gamma`
Catalan's constant	`Catalan`
Khinchin's constant	`Khinchin`
Golden ratio	`Golden_ratio`

20.58 SPECFN2: Package for Special Special Functions

This package provides algebraic manipulations of generalized hypergeometric functions and Meijer's G function. Generalized hypergeometric functions are simplified towards special functions and Meijer's G function is simplified towards special functions or generalized hypergeometric functions.

Author: Victor Adamchik, with major updates by Winfried Neun

The package SPECFN2 is designed to provide algebraic and numeric manipulations for some less commonly used special functions:

- Hypergeometric function;
- Meijer's G function.

These functions are from the non-core package SPECFN2, which needs to be loaded before use with the command:

```
load_package specfn2;
```

More information on the functions provided may be found on the website DLMF:NIST although currently not all functions may conform to these standards.

20.58.1 Hypergeometric Functions: Introduction

The (generalised) hypergeometric functions

$$ {}_pF_q\left(\left.\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\right|z\right) $$

are defined in textbooks on special functions as

$$ {}_pF_q\left(\left.\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\right|z\right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \ldots (a_p)_n}{(b_1)_n \ldots (b_q)_n} \frac{z^n}{n!} $$

where $(a)_n$ is the Pochhammer symbol

$$ (a)_n = \prod_{k=0}^{n-1} (a+k). $$

The function

$$ G_{pq}^{mn}\left(z \left| \begin{matrix}(a_p)\\ (b_q)\end{matrix}\right.\right) $$

has been studied by C. S. Meijer beginning in 1936 and has been called Meijer's G function later on. The complete definition of Meijer's G function can be found in [PBM89]. Many well-known functions can be written as G functions, e.g. exponentials, logarithms, trigonometric functions, Bessel functions and hypergeometric functions.

Several hundreds of particular values can be found in [PBM89].

20.58.2 The Hypergeometric Operator

The operator `hypergeometric` expects 3 arguments, namely the list of upper parameters (which may be empty), the list of lower parameters (which may be empty too), and the argument, e.g. the input:

```
hypergeometric ({},{},z);
```

yields the output

```
 z
e
```

and the input

```
hypergeometric ({1/2,1},{3/2},-x^2);
```

gives

```
 atan(abs(x))
--------------
    abs(x)
```

Since hundreds of particular cases for the generalised hypergeometric functions can be found in the literature, one cannot expect that all cases are known to the `hypergeometric` operator. Nevertheless the set of special cases can be augmented by adding rules to the REDUCE system, e.g.

```
let {hypergeometric({1/2,1/2},{3/2},-(~x)^2)
        => asinh(x)/x};
```

20.58.3 Meijer's G Function

The operator `MeijerG` expects 3 arguments, namely the list of upper parameters (which may be empty), the list of lower parameters (which may be empty too), and

the argument.

The first element of the lists has to be the list of the first n or m respective parameters, e.g. to describe

$$G_{11}^{10}\left(x \,\middle|\, \begin{matrix}1\\0\end{matrix}\right)$$

one has to write

```
MeijerG({{},1},{{0}},x); % and the result is:
```

```
 sign( - x + 1) + sign(x + 1)
------------------------------
              2
```

and for

$$G_{02}^{10}\left(\frac{x^2}{4} \,\middle|\, 1+\tfrac{1}{4}, 1-\tfrac{1}{4}\right)$$

```
MeijerG({{}},{{1+1/4},1-1/4},(x^2)/4) * sqrt pi;
```

```
                      2                       2
 sqrt(pi)*sqrt(-----------)*sin(abs(x))*x
                 abs(x)*pi
-----------------------------------------
                      4
```

20.59 SSTOOLS: Computations with Supersymmetric Algebraic and Differential Expressions

Authors: Thomas Wolf and Eberhard Schruefer

A detailed description is available after loading SSTOOLS and issuing the command

```
sshelp()$
```

The correct functioning of all procedures is tested through reading in and running `sstools.tst`. This test also illustrates the commutator rules for products of the different fields and their derivatives with respect to bosonic and fermionic variables.

The topics in `sshelp()$` are:

- Purpose
- Interactive Session
- Loading Files
- Notation
- Initializations
- The command `coeffn`
- The procedure `SSym`
- The procedure `SSConL`
- The procedure `FindSSWeights`
- The procedure `Linearize`
- The procedure `GenSSPoly`
- The procedure `ToCoo`
- The procedure `ToField`
- Discovery of recursion operators
- Verification of symmetries

SSTOOLS has a results page at https://lie.ac.brocku.ca/twolf/bl/susy/.

20.60 SUM: A Package for Series Summation

This package implements the Gosper algorithm for the summation of series. It defines operators `SUM` and `PROD`. The operator `SUM` returns the indefinite or definite summation of a given expression, and `PROD` returns the product of the given expression.

This package loads automatically.

Author: Fujio Kako

This package implements the Gosper algorithm for the summation of series. It defines operators `sum` and `prod`. The operator `sum` returns the indefinite or definite summation of a given expression, and the operator `prod` returns the product of the given expression. These are used with the syntax:

`sum`(⟨*expr:expression*⟩,⟨*k:kernel*⟩[,⟨*lolim:expression*⟩[,⟨*uplim:expression*⟩]])
`prod`(⟨*expr:expression*⟩,⟨*k:kernel*⟩[,⟨*lolim:expression*⟩[,⟨*uplim:expression*⟩]])

If there is no closed form solution, these operators return the input unchanged. ⟨*lolim*⟩ and ⟨*uplim*⟩ are optional parameters specifying the lower limit and upper limit of the summation (or product), respectively. If ⟨*uplim*⟩ is not supplied, the upper limit is taken as ⟨*k*⟩ (the summation variable itself).

For example:

```
sum(n**3,n);

sum(a+k*r,k,0,n-1);

sum(1/((p+(k-1)*q)*(p+k*q)),k,1,n+1);

prod(k/(k-2),k);
```

Gosper's algorithm succeeds whenever the ratio

$$\frac{\sum_{k=n_0}^{n} f(k)}{\sum_{k=n_0}^{n-1} f(k)}$$

is a rational function of n. The function `sum!-sq` handles basic functions such as polynomials, rational functions and exponentials.

The trigonometric functions sin, cos, etc. are converted to exponentials and then Gosper's algorithm is applied. The result is converted back into sin, cos, sinh and cosh.

Summations of logarithms or products of exponentials are treated by the formula:

$$\sum_{k=n_0}^{n} \log f(k) = \log \prod_{k=n_0}^{n} f(k)$$
$$\prod_{k=n_0}^{n} \exp f(k) = \exp \sum_{k=n_0}^{n} f(k)$$

Other functions, as shown in the test file for the case of binomials and formal products, can be summed by providing LET rules which must relate the functions evaluated at k and $k-1$ (k being the summation variable).

There is a switch `trsum` (default off). If this switch is on, trace messages are printed out during the course of Gosper's algorithm.

20.61 SUSY2: Supersymmetric Functions and Algebra of Supersymmetric Operators

This package deals with supersymmetric functions and with the algebra of supersymmetric operators in the extended $N = 2$ as well as in the non-extended $N = 1$ supersymmetry (SuSy). It allows us to realize the SuSy algebra of differential operators, compute the gradients of given SuSy Hamiltonians and obtain the SuSy version of soliton equations using the SuSy Lax approach. There are also many additional procedures encountered in the SuSy soliton approach, as for example: conjugation of a given SuSy operator, computation of the general form of SuSy Hamiltonians (up to SuSy divergence equivalence), and checking the validity of the Jacobi identity for SuSy Hamiltonian operators.

Author: Ziemowit Popowicz

20.61.1 Introduction

The main idea of supersymmetry (SuSy) is to treat boson and fermion operators equally [1,2]. This has been realised by introducing so-called supermultiplets constructed from the boson and fermion operators and additionally from the Mayorana spinors. Such supermultiplets possess the proper transformation property under the transformation of the Lorentz group. At the moment we have no experimental confirmation that supersymmetry appears in nature.

The idea of using supersymmetry for the generalization of the soliton equations [3–7] appeared almost in parallel to the usage of SuSy in the quantum field theory. The first results, concerning the construction of classical field theories with fermionic and bosonic fields depending on time and one space variable can be found in [8–12]. In many cases, the addition of fermion fields does not guarantee that the final theory becomes SuSy invariant and therefore this method was named the fermionic extension in order to distinguish it from the fully SuSy method.

In order to get a SuSy theory we have to add to a system of k bosonic equations kN fermion and $k(N-1)$ boson fields $(k = 1, 2, \ldots,\ N = 1, 2, \ldots)$ in such a way that the final theory becomes SuSy invariant. From the soliton point of view we can distinguish two important classes of the supersymmetric equations: the non-extended $(N = 1)$ and extended $(N > 1)$ cases. Consideration of the extended case may imply new bosonic equations whose properties need further investigation. This may be viewed as a bonus, but this extended case is no more fundamental than the non-extended one. The problem of the supersymmetrization of the nonlinear partial differential equations has its own history, and at the moment we have no unique solution [13–40]. We can distinguish three different methods of supersymmetrization: algebraic, geometric and direct.

In the first two cases we are looking for the symmetry group of the given equat-

ion and then we replace this group by the corresponding SuSy group. As a final product we are able to obtain a SuSy generalization of the given equation. The classification into the algebraic or geometric approach is connected with the kind of symmetry which appears in the classical case. For example, if our classical equation could be described in terms of the geometrical object then the simple exchange of the classical symmetry group of this object with its SuSy partner justifies the name geometric. In the algebraic case we are looking for the symmetry group of the equation without any reference to its geometrical origin. This strategy could be applied to the so-called hidden symmetry as for example in the case of the Toda lattice. These methods each have advantages and disadvantages. For example, sometimes we obtain the fermionic extensions. In the case of the extended supersymmetric Korteweg-de Vries equation we have three different fully SuSy extensions; however only one of them fits these two classifications.

In the direct approach we simply replace all objects which appear in the evolution equation by all possible combinations of the supermultiplets and their superderivatives so as to conserve the conformal dimensions. This is non-unique and yields many different possibilities. However, the arbitrariness is reduced if we additionally investigate super-bi-hamiltonian structure or try to find its supersymmetric Lax pair. In many cases this approach is successful.

The utilization of the above methods can be helped by symbolic computer algebraic and for this reason we developed the package SuSy2 in the symbolic language REDUCE [41].

We have implemented and ordered the superfunctions in our program, extensively using the concept of "noncom operator" in order to implement the supersymmetric integro-differential operators. The program is meant to perform the symbolic calculations using either fully supersymmetric supermultiplets or the component version of our supersymmetry. We have constructed 25 different commands to allow us to compute almost all objects encountered in the supersymmetrization procedure of the soliton equation.

20.61.2 Supersymmetry

The basic object in the supersymmetric analysis is the superfield and the supersymmetric derivative. The superfields are the superfermions or the superbosons [1]. These fields, in the case of extended $N = 2$ supersymmetry, depend, in addition to x and t, upon two anticommuting variables, θ_1 and θ_2 (such that $\theta_2\theta_1 = -\theta_1\theta_2,\ \theta_1^2 = \theta_2^2 = 0$). Their Taylor expansion with respect to θ_1, θ_2 is

$$b(x, t, \theta_1, \theta_2) := w + \theta_1\zeta_1 + \theta_2\zeta_2 + \theta_2\theta_1 u$$

for superbosons and

$$f(x, t, \theta_1, \theta_2) := \zeta_1 + \theta_1 w + \theta_2 u + \theta_2\theta_1\zeta_2$$

for superfermions, where w and u are classical (commuting) functions depending on x and t, and ζ_1 and ζ_2 are odd Grassmann-valued functions depending on x and t.

On the set of these superfunctions we can define the usual derivative and the superderivative. Usually, we encounter two different realizations of the superderivative: the first we call "traditional" and the second "chiral".

The traditional realization can be defined by introducing two superderivatives D_1 and D_2:

$$D_1 = \partial_{\theta_1} + \theta_1 \partial,$$
$$D_2 = \partial_{\theta_2} + \theta_2 \partial,$$

with the properties:

$$D_1 D_1 = D_2 D_2 = \partial,$$
$$D_1 D_2 + D_2 D_1 = 0.$$

The chiral realization is defined by

$$D_1 = \partial_{\theta_1} - \frac{1}{2}\theta_2 \partial,$$
$$D_2 = \partial_{\theta_2} - \frac{1}{2}\theta_1 \partial,$$

with the properties:

$$D_1 D_1 = D_2 D_2 = 0,$$
$$D_1 D_2 + D_2 D_1 = -\partial.$$

Below we shall use the names "traditional", "chiral" or "chiral1" algebras to denote the kind of commutation relations assumed for the superderivatives. The chiral1 algebras possess, additionally to the chiral algebra, the commutator of D_1 and D_2 defined by

$$D_3 = D_1 D_2 - D_2 D_1.$$

In the SUSY2 package we have implemented the superfunctions and the algebra of superderivatives. Moreover, we have defined many additional procedures which are useful in the supersymmetrization of the classical nonlinear system of partial differential equation. Different applications of this package to physical problems can be found in the papers [34–38].

20.61.3 Superfunctions

In this manual entry, REDUCE procedure (function) and let-rule names are usually displayed in a bold font, while all other input and output is usually displayed

as normal typeset mathematics. The value returned by a procedure (function) is indicated by the notation

$$\mathbf{function}(arguments) \Rightarrow function\ value.$$

However, REDUCE input without corresponding output and REDUCE command names are usually displayed in a typewriter font.

Superfunctions are represented in this package by

$$\mathbf{bos}(f, 0, 0) \tag{20.99}$$

for superbosons and

$$\mathbf{fer}(g, 0, 0)$$

for superfermions.

The first argument denotes the name of the given superobject, the second denotes the value of the SuSy derivative, and the last denotes the value of the usual derivative. The **bos** and **fer** objects are declared as `noncom` operators in REDUCE. The first argument can take an arbitrary value but with the following restrictions:

$$\mathbf{bos}(0, m, n) = 0,$$
$$\mathbf{fer}(0, m, n) = 0,$$

for all values of m, n.

The program has the capability to compute the coordinates of arbitrary SuSy expressions, using expansions in powers of θ. We have here four commands:

1. In order to have the given expression in components use

   $$\mathbf{fpart}(expression).$$

   The output is in the form of a list, in which the first element is the zero-order term in θ, the second is the first-order term in θ_1, the third is the first-order term in θ_2 and the fourth is the term in $\theta_2\theta_1$. For example, the superfunction (20.99) has the representation

   $$\mathbf{fpart}(\mathbf{bos}(f, 0, 0)) \Rightarrow \{\mathbf{fun}(f_0, 0), \mathbf{gras}(ff_1, 0), \mathbf{gras}(ff_2, 0), \mathbf{fun}(f_1, 0)\},$$

   where **fun** denotes the classical function and **gras** denotes the Grassmann function. The first argument in **fun** or **gras** denotes the name of the given object, while the second denotes the usual derivative.

2. In order to have the bosonic sector only, in which all odd Grassmann functions disappear, use

   $$\mathbf{bpart}(expression).$$

   Example:

   $$\mathbf{bpart}(\mathbf{fer}(g, 0, 0)) \Rightarrow \{0, \mathbf{fun}(g_0, 0), \mathbf{fun}(g_1, 0), 0\}$$

3. In order to have the given coordinates use

$$\mathbf{bf\_part}(expression, n),$$

where $n = 0, 1, 2, 3$.

Example:

$$\mathbf{bf\_part}(\mathbf{bos}(f, 0, 0), 3) \Rightarrow \mathbf{fun}(f_1, 0)$$

4. In order to have the given coordinates in the bosonic sector use

$$\mathbf{b\_part}(expression, n),$$

where $n = 0, 1, 2, 3$.

Example:

$$\mathbf{b\_part}(\mathbf{fer}(g, 0, 0), 1) \Rightarrow \mathbf{fun}(g_0, 0)$$

Notice that the program switches on factoring of $\mathbf{fer}, \mathbf{bos}, \mathbf{gras}, \mathbf{fun}$. If you remove this factoring then many commands give wrong results (for example the commands $\mathbf{lyst}$, $\mathbf{lyst1}$ and $\mathbf{lyst2}$).

20.61.4 The Inverse and Exponentials of Superfunctions

In addition to our definitions of the superfunctions we can also define the inverse and exponential of the superboson.

The inverse of the given $\mathbf{bos}$ function (not to be confused with the "inverse function" encountered in the usual analysis) is defined as

$$\mathbf{bos}(f, n, m, -1),$$

for arbitrary f, n, m with the property $\mathbf{bos}(f, n, m, -1)\,\mathbf{bos}(f, n, m, 1) = 1$. The object $\mathbf{bos}(f, n, m, k)$, in general, denotes the k-th power of the $\mathbf{bos}(f, n, m)$ superfunction. If we use the command `let inverse` then three-index $\mathbf{bos}$ objects are transformed into four-index $\mathbf{bos}$ objects.

The exponential of the superboson function is

$$\mathbf{axp}(\mathbf{bos}(f, 0, 0)).$$

It is also possible to use $\mathbf{axp}(f)$, but then we should specify what is f.

We have the following representation in components for the inverse and $\mathbf{axp}$ superfunctions:

$$\begin{aligned}\mathbf{fpart}(\mathbf{bos}(f, 0, 0, -1)) = \{&\mathbf{fun}(f_0, 0, -1), -\mathbf{fun}(f_0, 0, -1)\,\mathbf{gras}(\mathit{ff}_1, 0),\\ &-\mathbf{fun}(f_0, 0, -1)\,\mathbf{gras}(\mathit{ff}_2, 0), -\mathbf{fun}(f_0, 0, -2)\,\mathbf{fun}(f_1, 0, 1)\\ &+ 2\,\mathbf{fun}(f_0, 0, -3)\,\mathbf{gras}(\mathit{ff}_1, 0)\,\mathbf{gras}(\mathit{ff}_2, 0)\},\end{aligned}$$

$$\begin{aligned}\mathbf{fpart}(\mathbf{axp}(f)) = \{&\mathbf{axx}(\mathbf{bf\_part}(f,0)), \mathbf{axx}(\mathbf{bf\_part}(f,0))\,\mathbf{bf\_part}(f,1),\\ &\mathbf{axx}(\mathbf{bf\_part}(f,0))\,\mathbf{bf\_part}(f,2), \mathbf{axx}(\mathbf{bf\_part}(f,0))\\ &(\mathbf{bf\_part}(f,3) + 2\,\mathbf{bf\_part}(f,1)\,\mathbf{bf\_part}(f,2))\},\end{aligned}$$

where $\mathbf{axx}(f)$ denotes the exponentiation of the given classical function while $\mathbf{fun}(f,m,n)$ denotes the n-th power of the function $\mathbf{fun}(f,m)$.

20.61.5 Ordering

The three different superfunctions $\mathbf{fer}, \mathbf{bos}, \mathbf{axp}$ are ordered among themselves as

$$\begin{aligned}&\mathbf{fer}(f,n,m)\,\mathbf{bos}(h,j,k)\,\mathbf{axp}(g),\\ &\mathbf{fer}(f,n,m)\,\mathbf{bos}(h,j,k,l)\,\mathbf{axp}(g),\end{aligned}$$

independently of the arguments. The superfunctions $\mathbf{bos}$ and $\mathbf{axp}$ commute with themselves, while the superfunctions $\mathbf{fer}$ anticommute with themselves. For these superfunctions we introduce the following ordering.

- The $\mathbf{bos}$ objects with three and four arguments are ordered as follows: the first argument anti-lexicographically, the second and third by decreasing order of natural numbers; the last (fourth) is not ordered because

  $$\mathbf{bos}(f,n,m,k) * \mathbf{bos}(f,n,m,l) \Rightarrow \mathbf{bos}(f,n,m,k+l)$$

- The anticommuting $\mathbf{fer}$ objects are ordered as follows: the first argument anti-lexicographically, the second and third by decreasing order of natural numbers.

  Example:

  $$\mathbf{fer}(f,n,m) * \mathbf{fer}(g,k,l) \Rightarrow -\mathbf{fer}(g,k,l)\,\mathbf{fer}(f,n,m)$$

  for arbitrary n, m, k, l.

  $$\mathbf{fer}(f,n,m) * \mathbf{fer}(f,n,m) \Rightarrow 0$$

  for arbitrary f, n, m.

  $$\begin{aligned}\mathbf{bos}(f,2,3,7) * \mathbf{bos}(a,0,3) * \mathbf{bos}(f,2,3,-7) &\Rightarrow \mathbf{bos}(a,0,3),\\ \mathbf{bos}(f,2,3,2) * \mathbf{bos}(z,0,3,2) * \mathbf{bos}(f,2,3,-2) &\Rightarrow \mathbf{bos}(z,0,3,2).\end{aligned}$$

- For all exponential functions we have

  $$\mathbf{axp}(f) * \mathbf{axp}(g) \Rightarrow \mathbf{axp}(f+g).$$

20.61.6 (Super)Differential Operators

We have implemented three different realizations of the supersymmetric derivatives. In order to select the traditional realization declare `let trad`. In order to select the chiral or chiral1 algebra declare `let chiral` or `let chiral1`. By default we have the traditional algebra.

We have introduced three different types of SuSy operators which act on the superfunctions and are considered as noncommuting operators in REDUCE.

For the usual differentiation we introduced two types of operators:

- right differentations

$$\mathbf{d}(1) * \mathbf{bos}(f,0,0) \Rightarrow \mathbf{bos}(f,0,1) + \mathbf{bos}(f,0,0)\,\mathbf{d}(1);$$

- left differentations

$$\mathbf{fer}(f,0,0) * \mathbf{d}(2) \Rightarrow -\mathbf{fer}(f,0,1) + \mathbf{d}(2)\,\mathbf{fer}(f,0,0).$$

This example illustrates that the third argument in the **bos** and **fer** objects can take an arbitrary integer value.

We denote SuSy derivatives as **der** and **del**, which represent the right and left operations respectively, and are one-argument operators. The action of these objects on the superfunctions depends on the choice of the supersymmetric algebra.

Explicitly, we have for the traditional algebra:

right SuSy derivative

$$\begin{aligned}
\mathbf{der}(1) * \mathbf{bos}(f,0,0) &\Rightarrow \mathbf{fer}(f,1,0) + \mathbf{bos}(f,0,0)\,\mathbf{der}(1),\\
\mathbf{der}(2) * \mathbf{fer}(g,0,0) &\Rightarrow \mathbf{bos}(g,2,0) - \mathbf{fer}(g,0,0)\,\mathbf{der}(2),\\
\mathbf{der}(1) * \mathbf{fer}(f,2,0) &\Rightarrow \mathbf{bos}(f,3,0) - \mathbf{fer}(f,2,0)\,\mathbf{der}(1),\\
\mathbf{der}(2) * \mathbf{bos}(f,3,0) &\Rightarrow -\mathbf{fer}(f,1,1) + \mathbf{bos}(f,3,0)\,\mathbf{der}(2),\\
\mathbf{der}(1) * \mathbf{bos}(f,0,0,-1) &\Rightarrow -\mathbf{fer}(f,1,0)\,\mathbf{bos}(f,0,0,-2) +\\
&\quad \mathbf{bos}(f,0,0,-1)\,\mathbf{der}(1),\\
\mathbf{der}(2) * \mathbf{axp}(\mathbf{bos}(f,0,0)) &\Rightarrow \mathbf{fer}(f,2,0)\,\mathbf{axp}(\mathbf{bos}(f,0,0)) +\\
&\quad \mathbf{axp}(\mathbf{bos}(f,0,0))\,\mathbf{der}(2).
\end{aligned}$$

left SuSy derivative

$$\begin{aligned}
\mathbf{bos}(f,0,0)*\mathbf{del}(\mathbf{1}) &\Rightarrow -\mathbf{fer}(f,1,0)+\mathbf{del}(1)\,\mathbf{bos}(f,0,0),\\
\mathbf{fer}(g,0,0)*\mathbf{del}(\mathbf{2}) &\Rightarrow \mathbf{bos}(g,2,0)-\mathbf{del}(2)\,\mathbf{fer}(g,0,0),\\
\mathbf{fer}(f,2,0)*\mathbf{del}(2) &\Rightarrow \mathbf{bos}(f,3,0)-\mathbf{del}(1)\,\mathbf{fer}(f,2,0),\\
\mathbf{bos}(f,3,0)*\mathbf{del}(2) &\Rightarrow \mathbf{fer}(f,1,1)+\mathbf{del}(2)\,\mathbf{bos}(f,3,0),\\
\mathbf{bos}(f,0,0,-1)*\mathbf{del}(1) &\Rightarrow \mathbf{fer}(f,1,0)\,\mathbf{bos}(f,0,0,-2)+\\
&\quad\ \mathbf{del}(1)\,\mathbf{bos}(f,0,0,-1),\\
\mathbf{axp}(\mathbf{bos}(f,0,0))*\mathbf{del}(2) &\Rightarrow -\mathbf{fer}(f,2,0)\,\mathbf{axp}(\mathbf{bos}(f,0,0))+\\
&\quad\ \mathbf{del}(2)\,\mathbf{axp}(\mathbf{bos}(f,0,0)).
\end{aligned}$$

These examples illustrate that the second argument in the **fer** and **bos** objects can take values 0, 1, 2, 3 only with the following meaning: 0 – no SuSy derivatives, 1 – first SuSy derivative, 2 – second SuSy derivative, 3 – first and second SuSy derivative.

Using the results above we obtain

$$\begin{aligned}
&\mathbf{der}(1)*\mathbf{der}(2)*\mathbf{bos}(f,0,0)\Rightarrow\\
&\qquad \mathbf{bos}(f,3,0)+\mathbf{bos}(f,0,0)\,\mathbf{der}(1)\,\mathbf{der}(2)+\\
&\qquad\qquad \mathbf{fer}(f,1,0)\,\mathbf{der}(2)-\mathbf{fer}(f,2,0)\,\mathbf{der}(1).
\end{aligned}$$

For the chiral representation, the meaning of the second argument in the **bos** or **fer** object is the same as for the traditional representation while the actions of SuSy operators on the superfunctions are different. For example, we have

$$\begin{aligned}
\mathbf{der}(1)*\mathbf{fer}(f,1,0) &\Rightarrow -\mathbf{fer}(f,1,0)\,\mathbf{der}(1),\\
\mathbf{der}(1)*\mathbf{fer}(f,2,0) &\Rightarrow \mathbf{bos}(g,3,0)-\mathbf{fer}(f,2,0)\,\mathbf{der}(1),\\
\mathbf{der}(2)*\mathbf{bos}(g,3,0) &\Rightarrow -\mathbf{fer}(g,2,1)+\mathbf{bos}(g,3,0)\,\mathbf{der}(2),\\
\mathbf{bos}(g,2,0)*\mathbf{del}(2) &\Rightarrow \mathbf{del}(2)\,\mathbf{bos}(g,2,).
\end{aligned}$$

For the chiral1 representation we have a different meaning of the second argument in the **bos** and **fer** objects: the values $0,1,2$ for this second argument denote the values of the SuSy derivatives while 3 denotes the value of the commutator. Explicitly, we have

$$\begin{aligned}
\mathbf{der}(3)*\mathbf{bos}(f,0,0) &\Rightarrow \mathbf{bos}(f,3,0)+2\,\mathbf{fer}(f,1,0,0)\,\mathbf{der}(2)\\
&\qquad -2\,\mathbf{fer}(f,2,0)\,\mathbf{der}(1)+\mathbf{bos}(f,0,0)\,\mathbf{der}(3),\\
\mathbf{der}(1)*\mathbf{fer}(f,2,0) &\Rightarrow (\mathbf{bos}(f,3,0)-\mathbf{bos}(f,0,1))/2-\mathbf{fer}(f,2,0)\,\mathbf{der}(1).
\end{aligned}$$

The supersymmetric operators are always ordered in the case of traditional algebra

as

$$\begin{aligned}
\mathbf{der}(2) * \mathbf{der}(1) &\Rightarrow -\mathbf{der}(1)\,\mathbf{der}(2),\\
\mathbf{del}(2) * \mathbf{del}(1) &\Rightarrow -\mathbf{del}(1)\,\mathbf{del}(2),\\
\mathbf{der}(1) * \mathbf{del}(1) &\Rightarrow \mathbf{d}(1),\\
\mathbf{der}(1) * \mathbf{del}(2) &\Rightarrow -\mathbf{del}(2)\,\mathbf{der}(1);
\end{aligned}$$

for the chiral algebra we have

$$\begin{aligned}
\mathbf{der}(2) * \mathbf{der}(1) &\Rightarrow -\mathbf{d}(1) - \mathbf{der}(1)\,\mathbf{der}(2),\\
\mathbf{del}(2) * \mathbf{del}(1) &\Rightarrow -\mathbf{d}(1) - \mathbf{del}(1)\,\mathbf{del}(2),\\
\mathbf{der}(1) * \mathbf{del}(1) &\Rightarrow 0,\\
\mathbf{der}(1) * \mathbf{del}(2) &\Rightarrow -\mathbf{d}(1) - \mathbf{del}(2)\,\mathbf{der}(1);
\end{aligned}$$

while for chiral1 additionally we have

$$\begin{aligned}
\mathbf{der}(3) * \mathbf{der}(1) &\Rightarrow -\mathbf{der}(1)\,\mathbf{d}(1),\\
\mathbf{der}(1) * \mathbf{der}(3) &\Rightarrow \mathbf{der}(1)\,\mathbf{d}(1),\\
\mathbf{der}(3) * \mathbf{der}(2) &\Rightarrow \mathbf{der}(2)\,\mathbf{d}(1),\\
\mathbf{der}(2) * \mathbf{der}(3) &\Rightarrow -\mathbf{der}(2)\,\mathbf{d}(1).
\end{aligned}$$

Please notice that if we would like to have the components of some $\mathbf{bos}(f,3,0,-1)$ superfunction in the chiral representation then new objects appear:

$$\mathbf{b\_part}(\mathbf{bos}(f,3,0,-1),1) \Rightarrow \mathbf{fun}(f1,0,f0,1,-1),$$

We should consider the five-argument object **fun** as

$$\mathbf{fun}(f,n,g,m,-k) \Rightarrow (\mathbf{fun}(f,n) - \mathbf{fun}(g,m)/2)^{-k}.$$

Similar interpretation is valid for other commands containing objects like $\mathbf{bos}(f,3,n,-k)$.

20.61.7 Action of the Operators

In order to have the value of the action of the given operator on some superfunction we introduce two operators **pr** and **pg**. The operator

$$\mathbf{pr}(n,\, expression)$$

where $n = 0,1,2,3$ denotes the value itself of the action of the SuSy derivatives on the given expression. For $n = 0$ there is no SuSy derivative, $n = 1$ corresponds to $\mathbf{der}(1)$, $n = 2$ to $\mathbf{der}(2)$, and $n = 3$ to $\mathbf{der}(1) * \mathbf{der}(2)$.

Example:

$$\mathbf{pr}(1, \mathbf{bos}(f,0,0)) \Rightarrow \mathbf{fer}(f,1,0),$$
$$\mathbf{pr}(3, \mathbf{fer}(g,0,0)) \Rightarrow \mathbf{fer}(f,3,0).$$

For the usual derivative we reserve the command

$$\mathbf{pg}(n, expression)$$

where $n = 0, 1, 2, \ldots$ denotes the value of the usual derivative on the expression.

Example:

$$\mathbf{pg}(2, \mathbf{bos}(f,0,0)) \Rightarrow \mathbf{bos}(f,0,2)$$

20.61.8 Supersymmetric Integration

There is one command $\mathbf{s\_int}(number, expression, list)$ only. This allows us to compute the supersymmetric integral of arbitrary polynomial expressions constructed from **fer** and **bos** objects. It is valid in the traditional representation of supersymmetry. The argument $number$ takes the following values: 0 corresponds to the usual x integration, 1 or 2 to integration over the first or second supersymmetric argument, while 3 corresponds to integration over both the first and second arguments. The argument $list$ is a list of the names of the superfunctions over which we would like to integrate. The output of this command is in the form of the integrated part and non-integrated part. The non-integrated part is denoted by $\mathbf{del}(-number)$ for $number = 1, 2, 3$ and by $\mathbf{d}(-3)$ for $number = 0$.

Example:

$$\mathbf{s\_int}(0, 2 * \mathbf{bos}(f,0,1) * \mathbf{bos}(f,0,1), \{f\}) \Rightarrow \mathbf{bos}(f,0,0)^2,$$
$$\mathbf{s\_int}(1, 2 * \mathbf{fer}(f,1,0) * \mathbf{bos}(f,0,0), \{f\}) \Rightarrow \mathbf{bos}(f,0,0)^2,$$

$$\mathbf{s\_int}(3, \mathbf{bos}(f,3,0) * \mathbf{bos}(g,0,0) + \mathbf{bos}(f,0,0) * \mathbf{bos}(g,3,0), \{f,g\}) \Rightarrow$$
$$\mathbf{bos}(f,0,0)\,\mathbf{bos}(g,0,0) -$$
$$\mathbf{del}(-3)\left(\mathbf{fer}(f,1,0)\,\mathbf{fer}(g,2,0) - \mathbf{fer}(f,2,0)\,\mathbf{bos}(g,1,0)\right).$$

20.61.9 Integration Operators

We introduced four different types of integration operators: right and left denoted by $\mathbf{d}(-1)$ and $\mathbf{d}(-2)$ respectively and moreover two different types of neutral integration operators $\mathbf{d}(-3)$ and $\mathbf{d}(-4)$. In the first two cases they act according to the formula

$$\mathbf{d}(-1)\,\mathbf{bos}(f,0,0) = \sum_{i=1}^{\infty} (-1)^i\,\mathbf{bos}(f,0,i-1)\,\mathbf{d}(-1)^i$$

for the right integration and

$$\mathbf{bos}(f,0,0)\,\mathbf{d}(-2)=\sum_{i=1}^{\infty}\mathbf{d}(-2)^i\,\mathbf{bos}(f,0,i-1)$$

for the left integration.

Before using these operators the precision of the integration must be specified by an assignment of the form `ww := number`, which sets the actual upper limit to be used on the sums above instead of infinity. If required this precision can be changed by reassignment. Both operators are defined by their action and by the properties

$$\begin{aligned}\mathbf{d}(1)\,\mathbf{d}(-1)=\mathbf{d}(-1)\,\mathbf{d}(1)=\mathbf{d}(2)\,\mathbf{d}(-1)=\mathbf{d}(2)\,\mathbf{d}(-1)=1,\\ \mathbf{der}(1)\,\mathbf{d}(-1)=\mathbf{d}(-1)\,\mathbf{der}(1),\\ \mathbf{d}(-1)\,\mathbf{del}(1)=\mathbf{del}(1)\,\mathbf{d}(-1),\end{aligned}$$

and analogously for $\mathbf{d}(-2)$ and $\mathbf{der}(2),\mathbf{del}(2)$.

The neutral operator does not show any action on an expression but has several properties. More precisely

$$\begin{aligned}\mathbf{d}(1)\,\mathbf{d}(-3)=\mathbf{d}(-3)\,\mathbf{d}(1)=\mathbf{d}(2)\,\mathbf{d}(-3)=\mathbf{d}(-3)\,\mathbf{d}(2)=1,\\ \mathbf{der}(k)\,\mathbf{d}(-3)=\mathbf{d}(-3)\,\mathbf{der}(k),\\ \mathbf{d}(-3)\,\mathbf{del}(k)=\mathbf{del}(k)\,\mathbf{d}(-3),\end{aligned}$$

while for $\mathbf{d}(-4)$

$$\begin{aligned}\mathbf{d}(1)\,\mathbf{d}(-4)=\mathbf{d}(-4)\,\mathbf{d}(1)=\mathbf{d}(2)\,\mathbf{d}(-4)=\mathbf{d}(-4)\,\mathbf{d}(2)=1,\\ \mathbf{der}(k)\,\mathbf{d}(-4)=\mathbf{d}(-4)\,\mathbf{der}(k),\end{aligned}$$

where $k=1,2$.

From the last two formulas we see that the $\mathbf{d}(-3)$ operator is transparent under $\mathbf{del}$ operators while the $\mathbf{d}(-4)$ operator stops the $\mathbf{del}$ action.

Similarly to $\mathbf{d}(-3)$ or $\mathbf{d}(-4)$ it is also possible to use the neutral differentiation operator denoted by $\mathbf{d}(3)$. It has the properties

$$\begin{aligned}\mathbf{d}(3)\,\mathbf{d}(-4)=\mathbf{d}(-4)\,\mathbf{d}(3)=\mathbf{d}(3)\,\mathbf{d}(-3)=\mathbf{d}(-3)\,\mathbf{d}(3)=1,\\ \mathbf{der}(k)\,\mathbf{d}(3)=\mathbf{d}(3)\,\mathbf{der}(k),\\ \mathbf{d}(3)\,\mathbf{del}(k)=\mathbf{del}(k)\,\mathbf{d}(3),\end{aligned}$$

where $k=1,2$.

We can have also "accelerated" integration operators denoted by $\mathbf{dr}(-n)$ where n is a natural number. The action of these operators is exactly the same as $\mathbf{d}(-1)^n$ but

instead of using the integration formulas n times in the case of $\mathbf{d}(-1)^n$, $\mathbf{dr}(-n)$ uses the following formula only once:

$$\mathbf{dr}(-n)\,\mathbf{bos}(f,0,0) = \sum_{s=0}^{ww}(-1)^s \binom{n+s-1}{n-1}\,\mathbf{bos}(f,0,s)\,\mathbf{dr}(-n-s).$$

Similarly to the $\mathbf{d}(-1)$ case, we have to declare also the "precision" of integration if we would like to use the accelerated integration operators. The command `let cutoff` and assignment of the form `cut := number` allow us to annihilate the higher-order terms in the $\mathbf{dr}$ integration procedure. Moreover, the command `let drr` automatically changes the usual integration $\mathbf{d}(-1)$ into accelerated integration $\mathbf{dr}$. The command `let nodrr` changes $\mathbf{dr}$ integration into $\mathbf{d}(-1)$.

20.61.10 Useful Commands

Combinations

We encounter, in many practical applications, the problem of constructing different possible combinations of superfunction and super-pseudo-differential elements with given conformal dimensions. We provide three different procedures in order to realize this requirement:

$$\mathbf{w\_comb}(\mathit{list}, n, m, x),$$
$$\mathbf{fcomb}(\mathit{list}, n, m, x),$$
$$\mathbf{pse\_ele}(n, \mathit{list}, m).$$

All these commands are based on the gradations trick, to associate with superfunctions and superderivatives the scaling parameter conformal dimension. We consider here $k/2$ and k (k a positive integer) gradation only.

The command $\mathbf{w\_comb}$ gives the most general form of superfunction combinations of given gradation. It is a four-argument procedure in which:

1. the first argument is a list in which each element is a three-element list in which the first element is the name of the superfunction from which we would like to construct our combinations, the second denotes its gradation, and the last can take two values: f or b to indicate that the superfunction is respectively superfermionic or superbosonic;

2. the second argument is a number, the desired gradation;

3. the third argument is an arbitrary non-numerical value which enumerates the free parameters in our combinations;

4. the fourth argument takes one of two values: f or b to indicate that whole combinations should be respectively fermionic or bosonic.

Examples:

$$\begin{aligned}\mathbf{w\_comb}(\{\{f,1,b\},\{g,1,b\}\},2,z,b) &\Rightarrow z1\,\mathbf{bos}(f,3,0)+z2\,\mathbf{bos}(f,0,1)+\\ &\quad z3\,\mathbf{bos}(f,0,0)^2\\ \mathbf{w\_comb}(\{\{f,1,b\}\},3/2,g,f) &\Rightarrow g1\,\mathbf{fer}(f,1,0)+g2\,\mathbf{fer}(f,2,0)\end{aligned}$$

The command **fcomb**, similarly to **w_comb**, gives the general form of an arbitrary combination of superfunctions modulo divergence terms. It is a four-argument command with the same meaning of arguments as for **w_comb**. This command first calls **w_comb**, then eliminates in the canonical way SuSy derivatives by integration by parts of **w_comb**. By canonical we understand that (SuSy) derivatives are removed first from the superfunction which is first in the list of superfunctions in the **fcomb** command, next from the second, etc.

In order to illustrate the canonical manner of elimination of (SuSy) derivatives let us consider some expression which is constructed from f, g and h superfunctions and their (SuSy) derivatives. This expression is first split into three sub-expressions called the *f-expression*, *g-expression* and *h-expression*. The *f-expression* contains only combinations of f with f or g or (and) h, while the *g-expression* contains only combinations of g with g or h and the *h-expression* contains only combinations of h with h. The command **fcomb** removes first (SuSy) derivatives from f in *f-expression*, then from g in *g-expression*, and finally from h in *h-expression*. Consider this example:

$$\mathbf{fer}(f,1,0)\,\mathbf{fer}(g,2,0)+\mathbf{bos}(g,0,0)\,\mathbf{bos}(g,3,0).$$

Let us now assume that we have f, g order; then the *f-expression* is $\mathbf{fer}(f,1,0)\,\mathbf{fer}(g,2,0)$, while the *g-expression* is $\mathbf{bos}(g,0,1)\,\mathbf{bos}(g,3,0)$. Now canonical elimination gives

$$-\mathbf{bos}(f,0,0)\,\mathbf{bos}(g,3,0)+2\,\mathbf{bos}(g,0,0)\,\mathbf{bos}(g,3,1),$$

while assuming g, f order gives

$$-\mathbf{bos}(f,3,0)\,\mathbf{bos}(g,0,0)+2\,\mathbf{bos}(g,0,0)\,\mathbf{bos}(g,3,1).$$

Example:

$$\begin{aligned}&\mathbf{fcomb}(\{\{u,1\}\},4,h)\Rightarrow\\ &h(1)\,\mathbf{fer}(u,2,0)\,\mathbf{fer}(u,1,0)\,\mathbf{bos}(u,0,0)+h(2)\,\mathbf{bos}(u,3,0)\,\mathbf{bos}(u,0,0)^2+\\ &\qquad h(3)\,\mathbf{bos}(u,0,2)\,\mathbf{bos}(u,0,0)+h(4)\,\mathbf{bos}(u,0,0)^4\end{aligned}$$

Finally, the command **pse_ele** gives the general form of an element of the pseudo-SuSy derivative algebra [3]. Such an element can be written down symbolically as

$$(\mathbf{bos}+\mathbf{fer}\,\mathbf{der}(1)+\mathbf{fer}\,\mathbf{der}(2)+\mathbf{bos}\,\mathbf{der}(1)\,\mathbf{der}(2))\,\mathbf{d}(1)^n$$

for the traditional and chiral representations, or

$$(\mathbf{bos} + \mathbf{fer}\,\mathbf{der}(1) + \mathbf{fer}\,\mathbf{der}(2) + \mathbf{bos}\,\mathbf{der}(3))\,\mathbf{d}(1)^n$$

for the chiral1 representation, where **bos** and **fer** denote arbitrary superfunctions. The mentioned command allows us to obtain such an element of the given gradation which is constructed from some set of superfunctions of given gradation. This command takes three arguments:

$$\mathbf{pse\_ele}(wx, wy, wz).$$

The first argument denotes the gradation of the SuSy-pseudo-element, and the second denotes the names and gradations of the superfunctions from which we would like to construct our element. This second argument wy is in the form of a list exactly the same as in the **w_comb** command. The last argument denotes the names which enumerate the free parameters in our combination.

Parts of the pseudo-SuSy-differential elements

In order to obtain the components of the (pseudo)-SuSy element we have three different commands:

$$\mathbf{s\_part}(expression, m),$$
$$\mathbf{d\_part}(expression, n),$$
$$\mathbf{sd\_part}(expression, m, n),$$

where $m, n = 0, 1, 2, 3, \ldots$.

The **s_part** command gives the coefficient standing in the m-th SuSy derivative. However, notice that for $m = 3$ we should consider the coefficients standing in the $\mathbf{der}(1)\,\mathbf{der}(2)$ operator for the traditional or chiral representations while for the chiral1 representation the terms standing in the $\mathbf{der}(3)$ operator. The **d_part** command give the coefficients standing in the same power of $\mathbf{d}(1)$, while **sd_part** gives the term standing in the m-th SuSy derivative and n-th power of the usual derivative.

Example: Given the REDUCE input

```
ala := bos(g,0,0) + fer(f,3,0)*der(1) +
  (fer(h,2,0)*der(2) + bos(r,0,0)*der(1)*der(2))*d(1);
```

we have

$$\begin{aligned}
\mathbf{s_part}(ala, 3) &\Rightarrow \mathbf{fer}(f, 3, 0)\\
\mathbf{d_part}(ala, 1) &\Rightarrow \mathbf{fer}(h, 2, 0)\,\mathbf{der}(2) + \mathbf{bos}(r, 0, 0)\,\mathbf{der}(1)\,\mathbf{der}(2)\\
\mathbf{sd_part}(ala, 0, 0) &\Rightarrow \mathbf{bos}(g, 0, 0)
\end{aligned}$$

Adjoint

The *adjoint* PP^* of some SuSy operator PP is defined in standard form by

$$\langle \alpha, PP\, \beta \rangle = \langle \beta, PP^*\, \alpha \rangle$$

where α and β are test superboson functions and the scalar product is defined by

$$\langle \alpha, \beta \rangle = \int \alpha\beta \, d\theta_1 \, d\theta_2,$$

where we use the Berezin integral definition [1]

$$\int \theta_i \, d\theta_j = \delta_{i,j},$$
$$\int d\theta_i = 0.$$

For this operation we have the command

$$\mathbf{cp}(expression).$$

Examples:

$$\begin{aligned} \mathbf{cp}(\mathbf{der}(1)) &\Rightarrow -\mathbf{der}(1), \\ \mathbf{cp}(\mathbf{del}(1) * \mathbf{fer}(r,1,0) * \mathbf{der}(1)) &\Rightarrow \mathbf{fer}(r,1,1) + \mathbf{fer}(r,1,0)\, \mathbf{d}(1) - \\ &\quad \mathbf{del}(1)\, \mathbf{bos}(r,0,1), \end{aligned}$$

The last example illustrates that it is possible to define $\mathbf{cp}(\mathbf{del}(1)\, \mathbf{fer}(r,1,0)\, \mathbf{der}(1))$ in the different but equivalent manner as $\mathbf{fer}(r,1,0)\, \mathbf{d}(1) - \mathbf{bos}(r,0,1)\, \mathbf{der}(1)$.

From a practical point of view, we do not define conjugation for the $\mathbf{d}(-1)$ and $\mathbf{d}(-2)$ operators, because then we should define the precision of the action of the operators $\mathbf{d}(-1)$ and $\mathbf{d}(-2)$, and even then we would obtain very complicated formulas. However, if somebody decides to apply this conjugation to $\mathbf{d}(-1)$ or $\mathbf{d}(-2)$, it is recommended first to change by hand these operators into $\mathbf{d}(-3)$, next to compute $\mathbf{cp}$ and change $\mathbf{d}(-3)$ back into $\mathbf{d}(-1)$ or $\mathbf{d}(-2)$ together with the declaration of the precision.

Projection

In many cases, especially in the SuSy approach to soliton theory, we have to obtain the projection onto the invariant subspace (with respect to the commutator) of the pseudo-SuSy-differential algebra. There are three different subspaces [4] and hence we have the two-argument command

$$\mathbf{rzut}(expression, n)$$

in which $n = 0, 1, 2$.

Example: Given the REDUCE input

```
ewa := bos(f,0,0) + bos(f3,0,0)*der(1)*der(2) +
   bos(g,0,0)*d(1) + bos(g3,0,0)*d(1)*der(1)*der(2) +
   fer(f1,1,0)*der(1) + fer(f2,2,0)*der(2) +
   fer(g1,1,0)*d(1)*der(1) + fer(g2,2,0)*d(1)*der(2);
```

we have

$$\mathbf{rzut}(ewa, 0) \Rightarrow ewa,$$

$$\mathbf{rzut}(ewa, 1) \Rightarrow ewa - \mathbf{bos}(f, 0, 0),$$

$$\mathbf{rzut}(ewa, 2) \Rightarrow \mathbf{bos}(f3, 0, 0)\,\mathbf{der}(1)\,\mathbf{der}(2) + \big(\mathbf{fer}(g1, 1, 0)\,\mathbf{der}(1) + \mathbf{fer}(g2, 2, 0)\,\mathbf{der}(2) + \mathbf{bos}(g3, 0, 0)\,\mathbf{der}(1)\,\mathbf{der}(2)\big)\,\mathbf{d}(1).$$

Analogue of `coeff`

Motivated by practical applications, we constructed for our supersymmetric functions three commands, which allow us to obtain a list of the same combinations of some superfunctions and (SuSy) derivatives from some given operator-valued expression. Each command takes one argument and returns a list. We use the following REDUCE input to illustrate each command:

```
magda := fer(f,1,0)*fer(f,2,0)*a1 + der(1);
```

The first command is

$$\mathbf{lyst}(expression).$$

For example

$$\mathbf{lyst}(magda) \Rightarrow \{\mathbf{fer}(f, 1, 0)\,\mathbf{fer}(f, 2, 0)\,a1, \mathbf{der}(1)\}.$$

The second command is

$$\mathbf{lyst1}(expression)$$

with the output in the form of a list in which each element is constructed from the coefficients and (SuSy) operators of the corresponding element in the **lyst** list. For example

$$\mathbf{lyst1}(magda) \Rightarrow \{a1, \mathbf{der}(1)\}.$$

The third command is

$$\mathbf{lyst2}(expression)$$

with the output in the form of a list in which each element is constructed from coefficients in the given expression. For example

$$\mathbf{lyst2}(magda) \Rightarrow \{a1, 1\}.$$

Simplification

If we encounter during the process of computation an expression such as

$$\mathbf{fer}(f, 1, 0)\, \mathbf{d}(-3)\, \mathbf{fer}(f, 2, 0)\, \mathbf{d}(1),$$

it is not reduced further. To facilitate simplification, we can replace $\mathbf{d}(1)$ with $\mathbf{d}(2)$, or vice versa. In order to do this replacement we have the command

$$\mathbf{chan}(expression)$$

Example:

$$\begin{aligned}\mathbf{chan}(\mathbf{fer}(f, 1, 0) * \mathbf{d}(-3) * \mathbf{fer}(f, 2, 0) * \mathbf{d}(1)) \Rightarrow \\ - \mathbf{fer}(f, 2, 0)\, \mathbf{fer}(f, 1, 0) - \mathbf{fer}(f, 1, 0)\, \mathbf{d}(-3)\, \mathbf{fer}(f, 2, 1).\end{aligned}$$

Notice that as a result we remove the $\mathbf{d}(1)$ operator.

O(2) Invariance

In many cases in supersymmetric theories we deal with the O(2) invariance of SuSy indices. This invariance follows from the physical assumption of nonprivileging the "fermionic" coordinates in the superspace. In order to check whether our formula possesses such invariance we can use

$$\mathbf{odwa}(expression)$$

This procedure replaces in the given expression $\mathbf{der}(1)$ with $-\mathbf{der}(2)$ and $\mathbf{der}(2)$ with $\mathbf{der}(1)$. Next, it changes, in the same manner, the values of the action of these operators on the superfunctions.

Macierz

We can define the supercomponent form for the $\mathbf{pse\_ele}$ objects similarly to the representation of the superfunctions. Usually we can consider such an object as the matrix which acts on the components of the superfunction. It is realized in our program using the command

$$\mathbf{macierz}(expression, x, y),$$

where *expression* is the formula under consideration. The argument x can take two values, b or f, depending on whether we would like to consider the bosonic (b) part or fermionic (f) part of the expression. The last argument denotes the option in which we act on the bosonic or fermionic superfunction. It takes two values: f for fermionic test superfunction or b for bosonic. More explicitly, we obtain

$$\mathbf{macierz}(\mathbf{der}(1) * \mathbf{der}(2), b, f) \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{d}(1) & 0 \\ 0 & -\mathbf{d}(1) & 0 & 0 \\ -\mathbf{d}(1)^2 & 0 & 0 & 0 \end{pmatrix},$$

$$\mathbf{macierz}(\mathbf{der}(1) * \mathbf{der}(2), f, b) \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{d}(1) \\ -\mathbf{d}(1) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

20.61.11 Functional Gradients

In the SuSy soliton approach we very frequently encounter the problem of computing the gradient of the given functional. The usual definition of the gradient [2] is adopted in the supersymmetry also:

$$H'[v] = \langle \mathrm{grad}\, H, v \rangle = \frac{\partial}{\partial \epsilon} H(u + \epsilon v) \mid_{\epsilon=0},$$

where H denotes some functional which depends on u, v denotes a vector along which we compute the gradient, and $\langle \cdot, \cdot \rangle$ denotes the relevant scalar product.

We implemented all that in our package for the traditional algebra only. In order to compute the gradient with respect to some superfunction use

$$\mathbf{gra}(\mathit{expression}, f),$$

where *expression* is the given density of the functional and f denotes the first argument in the superfunction operator (name of the superfunction).

Example:

$$\mathbf{gra}(\mathbf{bos}(f, 3, 0) * \mathbf{fer}(f, 1, 0), f) \Rightarrow -2\,\mathbf{fer}(f, 2, 1)$$

For practical use we provide two additional commands:

$$\mathbf{dyw}(\mathit{expression}, f),$$
$$\mathbf{war}(\mathit{expression}, f).$$

The first computes the variation of *expression* with respect to superfunction f; the second removes (via integration by parts) SuSy derivatives from various functions

and finally produces a list of factorized **fer** and **bos** superfunctions. When the given expression is a full (SuSy) derivative, the result of the **dyw** command is 0 and hence this command is very useful in verifications of (SuSy) divergences of expressions.

When the result of applications of the **dyw** command is not zero then we would like to have the system of equations on the coefficients standing in the same factorized **fer** and **bos** superfunction. We can quickly obtain such a list using the command $\mathbf{war}(expression, f)$ with the same meaning for the arguments as in the **dyw** command.

Examples: Given the REDUCE input

```
xxx := fer(f,1,0)*fer(f,2,0) + x*bos(f,3,0)^2;
```

we obtain

$$\mathbf{dyw}(xxx, f) \Rightarrow \{-2\,\mathbf{bos}(f,3,0)\,\mathbf{bos}(f,0,0), -2x\,\mathbf{bos}(f,0,2)\,\mathbf{bos}(f,0,0)\},$$
$$\mathbf{war}(xxx, f) \Rightarrow \{-2, -2x\}.$$

20.61.12 Conservation Laws

In many cases we would like to know whether a given expression is a conservation law for some Hamiltonian equation. We can quickly check it using

$$\mathbf{dot\_ham}(equation, expression)$$

where *equation* is a list of two-element lists in which the first element denotes the function while the second denotes its flow. The second argument should be understood as the density of some conserved current. For example, for the SuSy version of the Nonlinear Schrödinger Equation [7] we could use the following REDUCE input:

```
ew := {{q, -bos(q,0,2) + bos(q,0,0)^3*bos(r,0,0)^2
          2*bos(q,0,0)*pr(3,bos(q,0,0)*bos(r,0,0))},
        {r, bos(r,0,2) - bos(q,0,0)^2*bos(r,0,0)^3 +
          2*bos(r,0,0)*pr(3,bos(q,0,0)*bos(r,0,0))}};
ham := bos(q,0,1)*bos(r,0,0) +
          x*bos(q,0,0)^2*bos(r,0,0)^2;
yyy := dot_ham(ew,ham);
```

The result of the previous computation is a complicated expression that is not zero. We would like to interpret it as a full (SuSy) divergence and we can quickly verify it by using the command **war**. We can solve the resulting list of equations using

known techniques. For example, in our previous case we obtain

$$\mathbf{war}(yyy, q) \Rightarrow \{-4x, -8x, -4x\},$$
$$\mathbf{war}(yyy, r) \Rightarrow \{4x, 8x, 4x\},$$

and we conclude that ham is a constant of motion if $x = 0$.

It is also possible to apply the command $\mathbf{dot\_ham}$ to the pseudo-SuSy-differential element. This is very useful in the SuSy approach to the Lax operator in which we would like to check the validity of the formula

$$\partial_t L := [L, A],$$

where A is some (SuSy) operator.

20.61.13 Jacobi Identity

The Jacobi identity for some SuSy Hamiltonian operators is verified using the relation

$$\langle \alpha, P'_{P(\beta)}\gamma \rangle + \text{all cyclic permutations of } \alpha, \beta, \gamma = 0,$$

where P' denotes the directional derivative along the $P(\beta)$ vector and $\langle \cdot, \cdot \rangle$ denotes the scalar product. The directional derivative is defined in the standard manner as [44]:

$$F^{'}(u)[v] = \frac{\partial}{\partial \epsilon} F(u + \epsilon v) \mid_{\epsilon=0},$$

where F is some functional depending on u, and v is a directional vector.

In this package we have several commands that allow us to verify the Jacobi identity. We have the possibility to compute, independently of verifying the Jacobi identity, the directional derivative for the given Hamiltonian operator along the given vector using

$$\mathbf{n\_gat}(pp, wim),$$

where pp is a scalar or matrix Hamiltonian operator and wim denotes the components of a vector along which we compute the derivative. It has the form of a list in which each element has the representation

$$\mathbf{bos}(f) \Rightarrow expression.$$

The expression $\mathbf{bos}(f)$ above denotes the shift of the $\mathbf{bos}(f, 0, 0)$ superfunction according to the definition of the directional derivative.

In order to compute the Jacobi identity we use the command

$$\mathbf{fjacob}(pp, wim)$$

with the same meaning for pp and wim as in the $\mathbf{n\_gat}$ command.

Notice that the ordering of the components in the wim list is important and is connected with the interpretation of the components of the Hamiltonian operator pp as a set of Poisson brackets constructed just from elements of the wim list. For example, in our scheme, the first component of wim is always connected with the element from which we create the Poisson bracket and which corresponds to the first element on the diagonal of pp, the second element of wim with the second element on the diagonal of pp, etc.

As the result of the application of the **fjacob** command to some Hamiltonian operator we obtain a complicated formula, not necessarily equal to zero but which should be expressed as a (SuSy) divergence. However, we can quickly verify it using the same method as for the **dot_ham** command, which was described in the previous subsection.

Usually, after the application of the **fjacob** command to some matrix Hamiltonian operator we obtain a huge expression, which is too complicated to analyze even when we would like to check its (SuSy) divergence. In this case we could extract from the **fjacob** expression terms containing given components of vector test functions fixed by us. We can use the command

$$\mathbf{jacob}(pp, wim, mm)$$

where pp and wim have the same meaning as for the **fjacob** command while mm is a three-element list denoting the components: $\{\alpha, \beta, \gamma\}$.

This command is not prepared to compute in full the Jacobi identity, which contains the integration operators. We do not implement here the symbolic integration of superfunctions in order to simplify the final results.

20.61.14 Objects, Commands and Let Rules

Objects

$\mathbf{bos}(f,n,m)$	$\mathbf{bos}(f,n,m,k)$	$\mathbf{fer}(f,n,m)$	$\mathbf{axp}(f)$	$\mathbf{fun}(f,n)$
$\mathbf{fun}(f,n,m)$	$\mathbf{gras}(f,n)$	$\mathbf{axx}(f)$	$\mathbf{d}(1)$	$\mathbf{d}(2)$
$\mathbf{d}(3)$	$\mathbf{d}(-1)$	$\mathbf{d}(-2)$	$\mathbf{d}(-3)$	$\mathbf{d}(-4)$
$\mathbf{dr}(-n)$	$\mathbf{der}(1)$	$\mathbf{der}(2)$	$\mathbf{del}(1)$	$\mathbf{del}(2)$

Commands

fpart(*expression*)
bpart(*expression*)
bf_part(*expression*, n)
b_part(*expression*, n)
pr(n, *expression*)
pg(n, *expression*)
w_comb($\{\{f, n, x\}, \ldots\}, m, z, y$)
fcomb($\{\{f, n, x\}, \ldots\}, m, z, y$)
pse_ele($n, \{\{f, n\}, \ldots\}, z$)
s_part(*expression*, n)
d_part(*expression*, n)
sd_(*expression*, n, m)
cp(*expression*)
rzut(*expression*, n)
lyst(*expression*)
lyst1(*expression*)
lyst2(*expression*)
chan(*expression*)
odwa(*expression*)
gra(*expression*, f)
dyw(*expression*, f)
war(*expression*, f)
dot_ham(*equations*, *expression*)
n_gat(*operator*, *list*)
fjacob(*operator*, *list*)
jacob(*operator*, *list*, $\{\alpha, \beta, \gamma\}$)
macierz(*expression*, x, y)
s_int(*numbers*, *expression*, *list*)

Let Rules

`trad` `chiral` `chiral1` `inverse` `drr` `nodrr`

Acknowledgement

The author would like to thank to Dr W. Neun for valuable remarks.

Bibliography

Please see the original version of this document, which is available formatted as https://reduce-algebra.sourceforge.io/extra-docs/susy2.pdf and as the LaTeX source file `susy2.tex` (in the REDUCE `packages/susy2` directory).

20.62 SYMMETRY: Operations on Symmetric Matrices

This package computes symmetry-adapted bases and block diagonal forms of matrices which have the symmetry of a group. The package is the implementation of the theory of linear representations for small finite groups such as the dihedral groups.

Author: Karin Gatermann

20.62.1 Introduction

The exploitation of symmetry is a very important principle in mathematics, physics and engineering sciences. The aim of the SYMMETRY package is to give an easy access to the underlying theory of linear representations for small groups. For example the dihedral groups D_3, D_4, D_5, D_6 are included. For an introduction to the theory see SERRE [Ser77] or STIEFEL and FÄSSLER [SF79]. For a given orthogonal (or unitarian) linear representation

$$\vartheta : G \longrightarrow GL(K^n), \qquad K = R, C.$$

the character $\psi \rightarrow K$, the canonical decomposition or the bases of the isotypic components are computed. A matrix A having the symmetry of a linear representation,e.g.

$$\vartheta_t A = A\vartheta_t \quad \forall\, t \in G,$$

is transformed to block diagonal form by a coordinate transformation. The dependence of the algorithm on the field of real or complex numbers is controled by the switch `complex`. An example for this is given in the testfile *symmetry.tst*.

As the algorithm needs information concerning the irreducible representations this information is stored for some groups (see the operators in Section 3). It is assumed that only orthogonal (unitar) representations are given.

The package is loaded by

```
load symmetry;
```

20.62.2 Operators for linear representations

First the data structure for a linear representation has to be explained. *representation* is a list consisting of the group identifier and equations which assign matrices to the generators of the group.

Example:

```
rr:=mat((0,1,0,0),
```

```
            (0,0,1,0),
            (0,0,0,1),
            (1,0,0,0));

   sp:=mat((0,1,0,0),
            (1,0,0,0),
            (0,0,0,1),
            (0,0,1,0));

   representation:={D4,rD4=rr,sD4=sp};
```

For orthogonal (unitarian) representations the following operators are available.

`canonicaldecomposition(representation);`

returns an equation giving the canonical decomposition of the linear representation.

`character(representation);`

computes the character of the linear representation. The result is a list of the group identifier and of lists consisting of a list of group elements in one equivalence class and a real or complex number.

`symmetrybasis(representation,nr);`

computes the basis of the isotypic component corresponding to the irreducible representation of type nr. If the nr-th irreducible representation is multidimensional, the basis is symmetry adapted. The output is a matrix.

`symmetrybasispart(representation,nr);`

is similar as `symmetrybasis`, but for multidimensional irreducible representations only the first part of the symmetry adapted basis is computed.

`allsymmetrybases(representation);`

is similar as `symmetrybasis` and `symmetrybasispart`, but the bases of all isotypic components are computed and thus a complete coordinate transformation is returned.

`diagonalize(matrix,representation);`

returns the block diagonal form of matrix which has the symmetry of the given linear representation. Otherwise an error message occurs.

`on complex;`

Of course the property of irreducibility depends on the field K of real or complex numbers. This is why the algorithm depends on K. The type of computation is set by the switch `complex`.

20.62.3 Display Operators

In this section the operators are described which give access to the stored information for a group. First the operators for the abstract groups are given. Then it is described how to get the irreducible representations for a group.

`availablegroups();`

returns the list of all groups for which the information such as irreducible representations is stored. In the following `group` is always one of these group identifiers.

`printgroup(group);`

returns the list of all group elements;

`generators(group);`

returns a list of group elements which generates the group. For the definition of a linear representation matrices for these generators have to be defined.

`charactertable(group);`

returns a list of the characters corresponding to the irreducible representations of this group.

`charactern(group,nr);`

returns the character corresponding to the nr-th irreducible representation of this group as a list (see also `character`).

`irreducibleреptable(group);`

returns the list of irreducible representations of the group.

`irreduciblerepnr(group,nr);`

returns an irreducible representation of the group. The output is a list of the group identifier and equations assigning the representation matrices to group elements.

20.62.4 Storing a new group

If the user wants to do computations for a group for which information is not predefined, the package SYMMETRY offers the possibility to supply information for this group.

For this the following data structures are used.

elemlist = list of identifiers.

relationlist = list of equations with identifiers and operators @ and $**$.

grouptable = matrix with the (1,1)-entry grouptable.

filename = "myfilename.new".

The following operators have to be used in this order.

`setgenerators(group,elemlist,relationlist);`

Example:

```
setgenerators(K4,{s1K4,s2K4},
  {s1K4^2=id,s2K4^2=id,s1K4@s2K4=s2K4@s1K4});
```

setelements(group,relationlist);

The group elements except the neutral element are given as product of the defined generators. The neutral element is always called `id`.

Example:

```
setelements(K4,
     {s1K4=s1K4,s2K4=s2K4,rK4=s1K4@s2K4});
```

setgrouptable(group,grouptable);

installs the group table.

Example:

```
tab:=
 mat((grouptable,     id,    s1K4, s2K4, rK4),
     (id        ,     id,    s1K4, s2K4, rK4),
     (s1K4      ,    s1K4,     id,  rK4,s2K4),
     (s2K4      ,    s2K4,    rK4,   id,s1K4),
     (rK4       ,     rK4,   s2K4, s1K4,  id));

setgrouptable(K4,tab);
```

Rsetrepresentation(representation,type);

is used to define the real irreducible representations of the group. The variable `type` is either *realtype* or *complextype* which indicates the type of the real irreducible representation.

Example:

```
eins:=mat((1));
mineins:=mat((-1));
rep3:={K4,s1K4=eins,s2K4=mineins};
Rsetrepresentation(rep3,realtype);
```

Csetrepresentation(representation);

This defines the complex irreducible representations.

setavailable(group);

terminates the installation of the group203. It checks some properties of the irreducible representations and makes the group available for the operators in Sections 2 and 3.

storegroup(group,filename);

writes the information concerning the group to the file with name *filename*.

loadgroups(filename);

loads a user defined group from the file *filename* into the system.

20.63 TRI: TeX REDUCE Interface

This package provides facilities written in REDUCE-Lisp for typesetting REDUCE formulas using TEX. The TEX-REDUCE-Interface incorporates three levels of TEXoutput: without line breaking, with line breaking, and with line breaking plus indentation.

Author: Werner Antweiler

Further documentation is available at https://reduce-algebra.sourceforge.io/extra-docs/tri.pdf.

20.64 TRIGD: Trigonometrical Functions with Degree Arguments

This module provides facilities for the numerical evaluation and algebraic simplification of expressions involving trigonometrical functions with arguments given in degrees rather than in radians. The degree-valued inverse functions are also provided.

Author: Alan Barnes

20.64.1 Introduction

This module provides facilities for the numerical evaluation and algebraic simplification of expressions involving trigonometrical functions with arguments given in degrees rather than in radians. The degree-valued inverse functions are also provided.

Any user at all familiar with the normal trig functions in REDUCE should have no trouble in using the facilities of this module. The names of the degree-based functions are those of the normal trig functions with the letter d appended, for example `sind`, `cosd` and `tand` denote the sine, cosine and tangent repectively and their corresponding inverse functions are `asind`, `acosd` and `atand`. The secant, cosecant and cotangent functions and their inverses are also supported and, indeed, are treated more as *first class* objects than their corresponding radian-based functions which are often converted to expressions involving sine and cosine by some of the standard REDUCE simplifications rules.

Below I give a brief description of the facilities available together with a few examples of their use. More examples and the output that they should produce may be found in the test files `trigd-num.tst` and `trigd-simp.tst` and their corresponding log files with extension `.rlg` which may be found in the directory `packages/misc` of the REDUCE distribution along with the source code of the module.

These degree-based functions are probably best regarded as functions defined for *real* values only, but complex arguments are supported for completeness. The numerical evaluation routines are fairly comprehensive for both real and complex arguments. However, few simplifications occur for trigd functions with complex arguments.

The range of the principal values returned by the inverse functions is consistent with those of the corresponding radian-valued functions. More precisely, for `asind`, `atand` and `acscd` the (closure of the) range is $[-90, 90]$ whilst for `acosd`, `acotd` and `asecd` the (closure of the) range is $[0, 180]$. In addition the operator `atan2d` is the degree valued version of the two argument inverse tangent

function which returns an angle in the half-open interval $(-180, 180]$ in the correct quadrant depending on the signs of its two arguments. For $x > 0$, `atan2d(y, x)` returns the same numerical value as `atand(y/x)`. If $x = 0$ then ± 90 is returned depending on the sign of y.

It might be thought that the facilities provided in this module could be easily provided by defining suitable rule lists to convert between the radian and degree-based versions of the trig functions. For example:

```
1: operator sind, asind$
2:  d2r_list := {sind(~x) =>
        sin(x*pi/180), asind(~x) => 180*asin(x)/pi}$
3:  r2d_list := {sin(~x) =>
       sind(180x/pi), asin(~x) => pi*sind(x)/180}$
4:  sind(x+360) where d2r_list$
5:  ws where r2d_list;
         sind(x)
6:  sind(360) where d2r_list;
       0
```

However, this approach *seldom works* — try it! The result produced by step 4 defeats the current rule[51] used to simplify expressions of the form $\sin(x + 2\pi)$ although it does manage step 6. The rule list approach is more reliable if differentiation, integration or numerical evaluation of expressions involving `sind` etc. is required. However it is not particularly convenient even if the rules and operator declarations are stored in a file so that they may be loaded at will.

This module aims to overcome these deficiences by providing the degree-based trig functions as *first class* objects of the system just like their radian-based cousins. The aim is to provide facilities for numerical evaluation, symbolic simplification and differentiation totally analgous to those for the the basic trig functions and their inverses. It is hoped that the module will be of value to students and teachers at secondary school level as well as being sufficiently powerful and flexible to be of genuine utility in fields where angles measured in degrees (and arc minutes and seconds) are in common usage. For more advanced situations (involving integration, complex arguments and values etc.), users are urged to use the standard trig functions already provided by the system.

20.64.2 Simplification

As in other parts of REDUCE, basic simplification of expressions involving the `trigd` functions takes place automatically (bracketted terms are multiplied out, like terms are gathered together, zero terms removed from sums and so on). The

[51] These rules may be improved in the next version of REDUCE.

system *knows* and automatically applies the basic properties of the functions to simplify the input. For example `sind(0)` is replaced by 0 and `sind(-X)` by `sind(X)`. If the switch `rounded` is `off` all arithmetic is exact and transcendental functions such as `sind` are not evaluated numerically even if their arguments are purely numerical.

The built-in simplification rules are totally analogous to those of the standard trig functions namely:

- Replacement of a function application by its value if a simple analytical value is known. For example `cosd(60) => 1/2` and `acscd(1) => 90`. Currently the only argument values where simplification takes place correspond to angles that are integral multiples of $15°$.
- Use of the odd and even properties of the trig. functions so that for example `sind(-x) => -sind(x)`, `cosd(-x) => cosd(x)` and `acosd(-x) => 180 - acosd(x)`.
- Argument shifts by integral multiples of $180°$ so that any residual numerical argument lies in the range $-90° \ldots 90°$. Thus

```
sind(x+540) => -sind(x),
cosd(x+350) => cosd(x-10).
```

- Removal of argument shifts of $\pm 90°$ so that for example `sind(x-90) => -cosd(x)` and `cotd(x+90) => -tand(x)`.
- Replacement of `tand(x)` by `sind(x)/cosd(x)` and e.g., `secd(x)` by `1/cosd(x)` and the like, but *only when the final result is simpler than the original*.
- Basic properties relating a function and it inverse so that for example `sind(asind(x)) => x`.
- A few basic rules for `atan2d` when the signs of its arguments can be determined. For example `atan2d(y, 0)` is replaced by ± 90 depending on the sign of y.

Extra rules can be added by the user for example addition formulae, double angle rules and tangent half-angle formulae as and when required as described in chapter 11.

Rules are provided for the symbolic differentiation of all the trig functions and their inverses. These rules are sufficient fot the power series of the trig functions and their inverses to be found using either the TPS or TAYLOR packages in the standard way.

20.64.3 Numerical Evaluation

When the switch `rounded` is `on` and the arguments of the operators evaluate to numbers, then the floating point value of the expression is calculated to the currently specified `precision` in the normal way. The *bigfloat* capabilities are the same as for the standard trig functions.

If these functions are supplied with complex numerical arguments, numerical evaluation will *NOT* be performed when the switch `rounded` is `on`, but the switch `complex` is `off` — the input expression will be returned basically unaltered. Similarly inputs such as `asind(2)` or `asecd(0.5)` are not evaluated numerically. The values of these expressions are, of course, complex.

If the switch `complex` is also `on` , numerical evaluation is performed. For example:

```
1: load_package trigd$

2:  on rounded;

3:  asecd(2);

          60.0

4:  asecd(0.5);

       asecd(0.5)

5: on complex;

    *** Domain mode rounded changed to complex-rounded

6:  asecd(0.5);

     75.4561292902*i
```

The function `atan2d` (like `atan2`) is only defined if *BOTH* its arguments are real. If they are also numerical, it will be evaluated whenever `rounded` is `on`. Attempting to evaluate it with complex numerical arguments will cause either the unaltered expression to be returned or an error to be raised when the switch `complex` is `off` or `on` respectively.

Note the sine of an angle specified in degrees, minutes and seconds *cannot* be calculated by calling `sind` directly with a dms list (i.e. as a list of length 3).

Instead one must first convert the dms values to degrees using a call to `dms2deg` and then call `sind` on the result. Applied directly to a list (of any length) any `trigd` function wil be applied to each member of the list separately just like most other REDUCE operators. Here is an example illustrating tese points:

```
1: load_package trigd$

2: on rounded;

3: sind dms2deg {60, 45, 30};

0.872567064923

4: sind {60,45, 30};

{0.866025403784,0.707106781187,0.5}

5: off rounded;

6: sind{60, 45, 30};

  sqrt(3)   sqrt(2)   1
{---------,---------,---}
     2         2      2
```

Of course the results will be formatted much more attractively on a terminal supporting nice graphics.

20.64.4 Bugs, Restrictions and Planned Extensions

The behaviour of the numerical evaluation routines for inverse trig functions with complex arguments at branch points could be improved; these values are *undefined* and attempting to evaluate such a function at one of its branch points *ought* to raise an error, however sometimes the input expression will be returned unaltered. It is hoped to improve this behaviour in due course.

Currently there are no facilities analogous to those provided in the module `TRIGSIMP` for the standard trig. functions. There users have a wide range of standard simplification formulae available for use and can control which are to be used depending on the requirements of their particular application: whether to eliminate `sin` in favour of `cos` or vice-versa or to get rid of both in favour of `tan` of half-angles; or whether to use the trigonometrical addition formulae in order to transform trig functions whose arguments are sums into a form where the

arguments are single terms or whether to perform the inverse transformations. It is hoped to make the `TRIGSIMP` faciliites available for use with the `TRIGD` functions in the near future.

Integration is not directly supported although the approach using rule-lists to convert the `TRIGD` functions to standard trig ones should work well. Introducing direct supp<ort for integration will not therefore be a priority.

For the standard sine function there is a rule for imaginary arguments namely: `sin(i*x) => i*sinh(x)`. The corresponding rule for the degree version is `sind(i*x) => i*sinh(x*pi/180)`. However, currently such rules are *NOT* implemented by the system. They may be implemented in future, but it is not a high priority as it is felt that the radian-based trig functions are best suited for such symbolic calculations.

There are *NO D* versions of the hyperbolic functions — that would be a *step too far*! And should the new functions be called `sinhd` and so on? Or perhaps `sindh`[52] etc?

[52] One is perhaps reminded here of the (in)famous bilingual pun: *peccavi* attributed to Charles James Napier — apparently no relation to his logarithmic namesake – see Wikipedia for details!

20.65 TRIGINT: Weierstraß Substitution in REDUCE

Author: Neil Langmead

This package was written when the author was a placement student at ZIB Berlin.

20.65.1 Introduction

This package is an implementation of a new algorithm proposed by D. J. Jeffrey and A. D. Rich [JR94] to remove "spurious" discontinuities from integrals. Their paper focuses on the Weierstraß substitution, $u = \tan(x/2)$, currently used in conjunction with the Risch algorithm in most computer algebra systems to evaluate trigonometric integrals. Expressions obtained using this substitution sometimes contain discontinuities, which limit the domain over which the expression is correct. The algorithm presented finds a better expression, in the sense that it is continous on wider intervals whilst still being an anti derivative of the integrand.

20.65.1.1 Example

Consider the following problem:

$$\int \frac{3}{5-4\cos(x)}\,dx$$

REDUCE computes an anti derivative to the given function using the Weierstraß substitution $u = \tan(\frac{x}{2})$, and then the Risch algorithm is used, returning:

$$\frac{2\arctan(3\tan(\frac{x}{2}))}{3},$$

which is discontinuous at all odd multiples of π. Yet our original function is continuous everywhere on the real line, and so by the Fundamental Theorem of Calculus, any anti-derivative should also be everywhere continuous. The problem arises from the substitution used to transform the given trigonometric function to a rational function: often, the substituted function is discontinuous, and spurious discontinuities are introduced as a result.

Jeffery and Richs' algorithm returns the following to the given problem:

$$\int \frac{3}{5-4\cos(x)}\,dx = 2\arctan\left(3\tan\left(\frac{x}{2}\right)\right) + 2\pi\left\lfloor\frac{x-\pi}{2\pi}\right\rfloor$$

which differs from (2) by the constant 2π, and this is the correct way of removing the discontinuity.

20.65.2 Statement of the Algorithm

We define a Weierstraß substitution to be one that uses a function $u = \Phi(x)$ appearing in the following table:

Choice	$\Phi(x)$	$\sin(x)$	$\cos(x)$	dx	b	p
(a)	$\tan(x/2)$	$\frac{2u}{1+u^2}$	$\frac{1-u^2}{1+u^2}$	$\frac{2du}{1+u^2}$	π	2π
(b)	$\tan(\frac{x}{2}+\frac{\pi}{4})$	$\frac{u^2-1}{u^2+1}$	$\frac{2u}{u^2+1}$	$\frac{2du}{1+u^2}$	$\frac{\pi}{2}$	2π
(c)	$\cot(x/2)$	$\frac{2u}{1+u^2}$	$\frac{u^2-1}{1+u^2}$	$\frac{-2du}{1+u^2}$	0	2π
(d)	$\tan(x)$	$\frac{u}{\sqrt{1+u^2}}$	$\frac{1}{\sqrt{1+u^2}}$	$\frac{du}{1+u^2}$	$\frac{\pi}{2}$	π

Table 20.26: Functions $u = \Phi$ used in the Weierstraß Alg. and their corresponding substitutions

There are of course, other trigonometric substitutions, used by REDUCE, such as $\sin$ and $\cos$, but since these are never singular, they cannot lead to problems with discontinuities.

Given an integrable function $f(\sin x, \cos x)$ whose indefinite integral is required, select one of the substitutions listed in the table. The choice is based on the following heuristics: choice (a) is used for integrands not containing $\sin x$, choice (b) for integrands not containing $\cos x$; (c) is useful in cases when (a) gives an integral that cannot be evaluated by REDUCE, and (d) is good for conditions described in Gradshteyn and Ryzhik (1979, sect 2.50). The integral is then transformed using the entries in the table,; for example, with choice (c), we have:

$$\int f(\sin x, \cos x)\, dx = \int f\left(\frac{2u}{1+u^2}, \frac{u^2-1}{1+u^2}\right) \frac{-2\, du}{1+u^2}.$$

The integral in u is now evaluated using the standard routines of the system, then u is substituted for. Call the result $\hat{g}(x)$. Next we calculate

$$K = \lim_{x \to b^-} \hat{g}(x) - \lim_{x \to b^+} \hat{g}(x),$$

where the point b is given in the table. the corrected integral is then

$$g(x) = \int f(\sin x, \cos x)\, dx = \hat{g}(x) + K \left\lfloor \frac{x-b}{p} \right\rfloor,$$

where the period p is taken from the table, and $\lfloor x \rfloor$ is the floor function.

20.65.3 REDUCE implementation

The name of the function used in REDUCE to implement these ideas is trigint, which has the following syntax:

```
trigint(⟨exp⟩, ⟨var⟩)
```

where ⟨*exp*⟩ is the expression to be integrated, and ⟨*var*⟩ is the variable of integration.

If trigint is used to calculate the integrals of trigonometric functions for which no substitution is necessary, then non standard results may occur. For example, if we calculate `trigint(cos(x), x)`, we obtain

$$\frac{2\tan\frac{x}{2}}{\tan^2\frac{x}{2}+1}$$

which, by using simple trigonometric identities, simplifies to:

$$\frac{2\tan\frac{x}{2}}{\tan^2\frac{x}{2}+1} \to \frac{2\tan\frac{x}{2}}{\sec^2\frac{x}{2}} \to 2\sin\frac{x}{2}\cos\frac{x}{2} \to \sin\left(2\frac{x}{2}\right) \to \sin x,$$

which is the answer we would normally expect. In the absence of a normal form for trigonometric functions though, both answers are equally valid, although most would prefer the simpler answer $\sin x$. Thus, some interesting trigonometric identities could be derived from the program if one so wished.

20.65.3.1 Examples

Using our example in (1), we compute the corrected result, and show a few other examples as well:

```
REDUCE Development Version,  4-Nov-96 ...

1: trigint(3/(5-4*cos(x)),x);

                 x                - pi + x
2*(atan(3*tan(---)) + floor(------------)*pi)
                 2                  2*pi

2: trigint(3/(5+4*sin(x)),x);

                 pi + 2*x
2*(atan(3*tan(----------))
                    4
```

```
                - pi + 2*x
    + floor(-------------)*pi)
                  4*pi
```

```
3: trigint(15/(cos(x)*(5-4*cos(x))),x);
```

```
                x                    - pi + x
8*atan(3*tan(---)) + 8*floor(-----------)*pi
                2                     2*pi

                x                        x
 - 3*log(tan(---) - 1) + 3*log(tan(---) + 1)
                2                        2
```

20.65.4 Definite Integration

The corrected expressions can now be used to calculate some definite integrals, provided the region of integration lies between adjacent singularities. For example, using our earlier function, we can use the corrected primitive to calculate

$$\int_0^{4\pi} \frac{1}{2+\cos x}\,dx \tag{20.100}$$

trigint returns the answer below to give an indefinite integral, $F(x)$:

```
                          x
                      tan(---)
                          2                    - pi + x
 2*sqrt(3)*(atan(-----------) + floor(-----------)*pi)
                     sqrt(3)                  2*pi
-------------------------------------------------------
                           3
```

And now, we can apply the Fundamental Theorem of Calculus to give

$$\int_0^{4\pi} \frac{1}{2+\cos x}\,dx = F(4\pi) - F(0) \tag{20.101}$$

```
sub(x=4*pi,F)-sub(x=0,F);
```

```
   4*sqrt(3)*pi
-----------------
        3
```

and this is the correct value of the definite integral. Note that although the expression in (*) is continuous, the functions value at the points $x = \pi, 3\pi$ etc. must be intepreted as a limit, and these values cannot substituted directly into the formula given in (*). Hence care should be taken to ensure that the definite integral is well defined, and that singularities are dealt with appropriately. For more details of this in REDUCE, please see the documentation for the *cwi* addition to the DEFINT package.

20.65.5 Tracing the *trigint* function

The package includes a facility to trace in some detail the inner workings of the `trigint` program. Messages are given at key points of the algorithm, together with the results obtained. These messages are displayed whenever the switch `tracetrig` is on, which is done in REDUCE with the command `on tracetrig;` This switch is off by default. In particular, the messages inform the user which substitution is being tried, and the result of that substitution. The error message

```
***** system cannot integrate after subs
```

means that REDUCE has tried all four of the Weierstraß substitutions, and the system's standard integrator is unable to integrate after the substitution has been completed.

20.65.6 Bugs, comments, suggestions

This program was written whilst the author was a placement student at ZIB Berlin.

20.66 V3TOOLS: Computations with Polynomials of Scalar Vector Products

A *scalar vector product* of 3-component vectors is a generalization of the scalar triple product in which the vector product becomes a repeated vector product of an arbitrary number of vectors.

Author: Thomas Wolf

20.66.1 Purpose

Procedures in this package perform computationally expensive and non-trivial computations with polynomials of scalar vector products of 3-component vectors. The individual procedures can be grouped into the following categories:

- initialization of the package,
- generation of scalar vector expressions according to a multiple weighting scheme where each vector has multiple weights,
- conversions of scalar vector expressions between different representations,
- the computation of Poisson brackets between vector expressions,
- a computation determining whether a given scalar vector expression is functionally dependent on a list of other scalar vector expressions.

A possible application of these procedures is the classification of integrable 'vectorial' Hamiltonians, the study whether obtained expressions are functionally independent of known expressions and the compactification of results for publication.

These routines were designed and implemented to support the classification of integrable Hamiltonians. Examples are chosen from this application. More details are given in [SW06].

20.66.2 Notation

The chosen conventions aim to display large scalar vector expressions as compactly as possible. All vectors have 3 components and are represented by a single letter, like A or B. In this manual we use capital letters in mathematical formulas and lower case letters for REDUCE input and output. REDUCE is not case sensitive but output (normally) displays lower case letters. Vector components have an extra digit 1, 2 or 3, like $A2$, $B3$. Products of vectors are represented by a single identifier where the following convention is used: $ABCD\ldots$ stands for $(A\ ,\ B\times(C\times(D\times$

$\ldots)))$ where $(\ ,\ )$ denotes the symmetric scalar product and $\times$ the skew-symmetric vector product. For example, we have $AB = (A, B)$, which is the normal scalar product $A \cdot B$, and $ABC = (A, B \times C)$, which is the normal scalar triple product $A \cdot B \times C$. In the following we distinguish between three forms of scalar vector expressions:

- **the component form** involving only the components $A1, A2, A3, B1, \ldots$,
- **the standard vector form** involving only scalar products $AB\,(= (A, B))$ and triple products $ABC\,(= (A, B \times C))$,
- **the extended vector form** involving any product $ABCD\ldots$.

Because of the commutativity of the scalar product $(A, B) = (B, A)$ there seem to be two identifiers AB and BA suited to represent the same product. Similarly for triple products we have the relation $ABC = BCA = CAB = -ACB = -CBA = -BAC$. In order to have an unambiguous notation such that vanishing rational expressions simplify to zero we adopt the convention that for scalar products and triple products only identifiers are used in which letters are sorted alphabetically. For example, the program uses AB, AV, ABV, BUV but not BA, VA, BVA, VBU. In order to simplify manual input one can assign expressions using identifiers, like BA, BUA, and afterwards convert them into proper notation using the procedure `e2s` (see below or the beginning of the file `packages/crack/v3tools.tst`).

In order to use non-vectorial parameters, like $ALPHA$ or $BETA2$ every non-vectorial parameter is preceded by `!&`, e.g. one would input `!&alpha` or `!&beta2`.

20.66.3 Initialization

Scalar products and scalar triple products of 3-component vectors satisfy identities, for example, the four vectors A, B, U, V satisfy

$$0 = AB\ BUV - ABU\ BV + ABV\ BU - AUV\ BB.$$

Therefore, scalar vector expressions, i.e. polynomials of scalar vector products, usually do not have a unique form. They can be re-written using these identities, for example, in order to minimize the number of terms in the polynomial or to bring the expression into a canonical form. Sometimes computations have to be done modulo these identities, more precisely, modulo a Gröbner basis of these identities. The computation of the identities and their Gröbner basis has to be done only once with the procedure `vinit` which initializes the global variables `v_`, `gbase_`, `heads_`. The procedure `vinit` takes as argument a list of all vectors that occur in the computation, for example, `vinit({u,v,w,z})`. In the default

case of vectors a, b, u, v the global variables v_, gbase_, heads_ are already assigned and vinit does not have to be called initially. But if some of the vectors satisfy a special relation, like $AB = 0$, then this relation should be assigned (like ab:=0;) and vinit should be run afterwards. Here is an overview of the three global variables:

v_ : a list of vectors involved in all the computations. The global variable v_ is needed in the procedure poisson_v. The default value is v_={a,b,u,v}.

gbase_ : a list of polynomials which are a Gröbner basis for all identities between scalar products AB and triple products ABC of any vectors in the list v_.

heads_ : a list of leading terms of the polynomials in gbase_.

20.66.4 Main Routines

Which routines work only for homogeneous vectorial expressions and which for any vectorial expressions?

vinit : initializes all three global variables vl_, gbase_, heads_ for a given list of vectors. This procedure is not necessary if the list of vectors is A, B, U, V. The procedure is necessary if some of the vectors satisfy extra conditions, like $AB = 0$. Example:

```
vinit({a,b,c,d});
```

e2s : converts a vectorial polynomial from extended vector form into standard vector form by replacing products $ABCD\ldots$ through scalar and triple products. Example:

```
e2s(buuav);  ⟶  abv*uu - auv*bu
```

It is also useful to convert from standard vector form into standard vector form if one wants to sort factors in scalar and triple products lexicographically. This is useful for working with this package on expressions that have been generated outside this package and that do not obey the lexicographical ordering of factors. Example:

```
e2s(avb*uu + uva*bu);  ⟶  - abv*uu + auv*bu
```

v2c : converts an expression from any vector form (extended or standard) into component form. Example:

```
v2c(abu);  ⟶

a1*b2*u3 - a1*b3*u2 - a2*b1*u3 + a2*b3*u1 +
a3*b1*u2 - a3*b2*u1
```

c2sl : converts an expression from component form into standard vector form. The resulting vector expression is returned partitioned as a list of sublists. Each sublist contains expressions with the same multiplicity of each vector (the same homogeneity). The first expression of each list is the vector equivalent of the corresponding input terms in component form. All other expressions in each sublist are identically vanishing vector expressions, i.e. vector identities with the same multiplicity as the first expression in each sublist. That means that the second, third, ... expression in each sublist multiplied with a constant could be added to the first expression of each sublist in order to change its form but not its value. Example:

```
c2sl(a1*b2*u3 - a1*b3*u2 - a2*b1*u3 +
a2*b3*u1 + a3*b1*u2 - a3*b2*u1 +
a1*b1*b2*u3*v1 - a1*b1*b2*u1*v3 + a1*b1*b3*u1*v2 -
a1*b1*b3*u2*v1 - a1*b2**2*u2*v3 + a1*b2**2*u3*v2 -
a1*b3**2*u2*v3 + a1*b3**2*u3*v2 + a2*b1**2*u1*v3 -
a2*b1**2*u3*v1 + a2*b1*b2*u2*v3 - a2*b1*b2*u3*v2 +
a2*b2*b3*u1*v2 - a2*b2*b3*u2*v1 + a2*b3**2*u1*v3 -
a2*b3**2*u3*v1 - a3*b1**2*u1*v2 + a3*b1**2*u2*v1 +
a3*b1*b3*u2*v3 - a3*b1*b3*u3*v2 - a3*b2**2*u1*v2 +
a3*b2**2*u2*v1 - a3*b2*b3*u1*v3 + a3*b2*b3*u3*v1);
```

$\longrightarrow$

```
{{abu},{abu*bv - abv*bu,
        ab*buv - abu*bv + abv*bu - auv*bb}}
```

The input expression is equal to the sum of the first expressions of all sublists, i.e. equal to $ABU + ABUBV - ABVBU$. The second expression of the second sublist is an identity, i.e. $0 = ABBUV - ABUBV + ABVBU - AUVBB$, where each vector occurs in each term equally often as in each term of the first expression in the second sublist.

c2s : is equivalent to `c2sl` with the difference that all expressions of all sublists are added up with all identity expressions obtaining coefficients `!&c1`, `!&c2`,.... Example:

```
c2s( same input as for c2sl above );  ⟶

abu + abu*bv - abv*bu +
      !&c1*(ab*buv - abu*bv + abv*bu - auv*bb)
```

s2s : generates and uses vector identities to reduce the length of an expression in standard vector form. Example:

```
s2s( - abu*vv + abv*uv + av*buv);  ⟶  auv*bv
```

s2e : like `s2s` but also uses identities involving products $ABCD\ldots$ and thus transforms standard vector form into extended vector form. Example:

```
s2e(abv*uu - auv*bu);  ⟶  buuav
```

In its current form it only uses products that involve all the vectors involved in each term of the standard vector form. In the above example it would not try to use products with 4 factors but only with 5 factors, like `buuav`.

genpro : generation of all scalar and triple products for a given vector list. Example:

```
genpro({a,b,u,v});  ⟶
{aa,ab,au,av,bb,bu,bv,uu,uv,vv,abu,abv,auv,buv}
```

genpro_wg : similar to `genpro` with the difference that each vector is accompanied by a list of weights and the resulting scalar and triple products also come with a list of weights. Example:

```
genpro_wg({{a,1,0,0},{b,0,1,0},{u,0,0,1},
           {v,1,0,1}});

⟶

{{aa,2,0,0},{ab,1,1,0},{au,1,0,1},{av,2,0,1},
 {bb,0,2,0},{bu,0,1,1},{bv,1,1,1},{uu,0,0,2},
 {uv,1,0,2},{vv,2,0,2},{abu,1,1,1},{abv,2,1,1},
 {auv,2,0,2},{buv,1,1,2}}
```

poisson_c : computes the poisson bracket of two arbitrary expressions. This procedure needs as input the Poisson structure matrix which we call `struc_cons` below. This is encoded as a list of lists, specifying all non-vanishing Poisson brackets between any two dynamical variables, here `u1,u2,u3,v1,v2,v3`. For example, the first sublist `{u1,u2,u3}` of `struc_cons` below, encodes the Poisson bracket relation $\{U1, U2\} = -\{U2, U1\} = U3$. Poisson brackets associated to other Lie-algebras are appended to the file `packages/crack/v3tools.red`. The structure matrix below encodes the Lie-Poisson bracket $e(3)$ for `!&kap=0`, $so(4)$ for `!&kap=1` and $so(3,1)$ for `!&kap=-1`. `poisson_c` needs *all* of its arguments in component form. Example:

```
ham:=v2c(ab*uu-2*au*bu+buv)$
fi :=v2c(bu*(2*auv-!&kap*uu+vv))$
!&kap:=-a1**2-a2**2-a3**2$
struc_cons:={{u1,u2, u3}, {u2,u3, u1}, {u3,u1, u2},
             {u1,v2, v3}, {u2,v3, v1}, {u3,v1, v2},
```

```
                {u1,v3,-v2}, {u2,v1,-v3}, {u3,v2,-v1},
                {v1,v2, !&kap*u3}, {v2,v3, !&kap*u1},
                {v3,v1, !&kap*u2}}$
```

```
poisson_c(ham,fi,struc_cons);  ⟶  0
```

The example shows that `fi` is a first integral of `ham` under the assumption `!&kap = - aa`. Note that `!&kap` could not have been expressed in terms of components of vectors before assigning `fi` as then the argument to `v2c` would not be a purely vectorial expression.

poisson_v : computes the Poisson bracket for two vector expressions. It uses the global variable `gbase_`. When `poisson_v` is called the first time it computes the Poisson bracket between any scalar and triple product once using the procedure `poisson_c` and stores these elementary Poisson brackets under the global operator `poi_`. Therefore, the third argument to `poisson_v` (the Poisson structure matrix) has to be given in component form. Example: Compare the following with the example for the call of `poisson_c`.

```
ham:=ab*uu-2*au*bu+buv$
fi :=bu*(2*auv-!&kap*uu+vv)$
!&kap:=-a1**2-a2**2-a3**2$
struc_cons:={{u1,u2, u3}, {u2,u3, u1}, {u3,u1, u2},
             {u1,v2, v3}, {u2,v3, v1}, {u3,v1, v2},
             {u1,v3,-v2}, {u2,v1,-v3}, {u3,v2,-v1},
             {v1,v2, !&kap*u3}, {v2,v3, !&kap*u1},
             {v3,v1, !&kap*u2}}$
!&kap:=-aa$

poisson_v(ham,fi,struc_cons);  ⟶  0
```

When calling `poisson_v` a second time the computation proceeds much faster as the Poisson brackets between scalar and triple products had been stored with the operator `poi_`. Note that at first `!&kap` is expressed in component form to have `struc_cons` in component form but afterwards `!&kap` is expressed in vector form to have `ham, fi` in vector form at the time of calling `poisson_v`.

gfi : generates a vector expression with unknown coefficients according to a specific list of weights $\{w_1, w_2, \ldots\}$. The number of weights is arbitrary but fixed. For example, the vector V could be given weights $\{w_1, w_2, w_3\} = \{1, 0, 1\}$ which will be input as `{v,1,0,1}` in the second argument to `gfi`. In the following example vectors A, B, U, V are each given 3 weights $\{w_1, w_2, w_3\}$ and the most general polynomial is generated where the sums of all $\{w_1, w_2, w_3\}$ values of all vectors in each term are equal to $\{2, 1, 3\}$. Example:

```
gfi({2,1,3},
    {{a,1,0,0},{b,0,1,0},{u,0,0,1},{v,1,0,1}},
    {aa, ab, bb, bu, uv, -aa*uu + vv,
     ab*uu - 2*au*bu + buv},
    heads_);
```

$\longrightarrow$

```
{&r1*abu*uv + &r2*abv*uu + &r3*bv*uv +
 &r4*auv*bu + &r5*bu*vv +
       2
 &r6*au *bu + &r7*ab*au*uu,
 &r7,&r6,&r5,&r4,&r3,&r2,&r1}
```

The meaning of the parameters:

1. The first parameter of `gfi` is a list of the total weights of each term in the generated polynomial.
2. The second parameter is the list of vectors to be used for generating the expression, each followed by its list of weight values, like `{a,1,0,0}`.
3. The third parameter is a list of homogeneous vector expressions. The polynomial returned by `gfi` must not include any expression that is functionally dependent on the expressions in this list. In other words, from the most general polynomial with proper homogeneity that is generated at first within `gfi` terms are dropped so that it is not possible to choose in the remaining polynomial the undetermined coefficients `!&r1`, `!&r2`, ... such that an expression results which is functionally dependent only on the expressions of the third parameter list. In this example we want to formulate an ansatz for a first integral that is functionally independent of the expressions `aa`, `ab`, `bb` (because A, B are assumed to be constant vectors and U, V to be dynamical vectors), `bu` (a known first integral), `uv`, `-aa*uu + vv` (two Casimirs, i.e. first integrals) and `ab*uu - 2*au*bu + buv` which is the Hamiltonian. The third parameter prevents the generation of the term `bu*aa*uu` which also has weights $\{2,1,3\}$ (as `b` has weights $\{0,1,0\}$, `u` has weights $\{0,0,1\}$ and `a` has weights $\{1,0,0\}$). The term `bu*aa*uu` is dropped because the resulting polynomial would otherwise contain the functionally dependent expression `bu*(-aa*uu+vv)`.
4. The fourth parameter is the list of leading terms of the Gröbner basis of identities between the vectors. The resulting polynomial contains only terms which are not a multiple of any one of the terms in this list. By excluding terms which are not multiples of leading terms of identities in the Gröbner basis one defines a canonical form which vanishes only if all terms vanish.

The result of `gfi` is a list. Its first element is the most general polynomial that has the required properties. The list of undetermined coefficients `!&r...` is appended.

`fnc_dep` : investigates whether a given vectorial expression (the first parameter) is functionally dependent on the elements of a list (the second parameter). The third parameter is the list of occurring vectors with their weights. In the following example it is to be checked whether the first integral `finew` is new or merely the known one `fiold` in disguised form. Example:

```
ham:=ab*uu - 1/2*au*bu + buv$
fiold:=bu**2*((bu*aa-ab*au)**2*uu +
              2*(bu*aa-ab*au)*(bu*auv-buv*au) -
              vv*abu**2 - (bu*av-bv*au)**2 -
              uu*vv*ab**2+aa*uu*vv*bb)$
finew:=( - 16*aa**2*bu**4*uu + 96*aa*ab**2*bb*uu**3 -
96*aa*ab*au*bb*bu*uu**2 + 32*aa*ab*au*bu**3*uu +
32*aa*abu*bu**3*uv - 32*aa*abv*bu**3*uu +
24*aa*au**2*bb*bu **2*uu - 16*aa*bu**4*vv +
56*ab**4*uu**3 - 84*ab**3*au*bu*uu**2 +
168*ab**2*abu*bv*uu**2 - 168*ab**2*abv*bu*uu**2 +
26*ab**2*au**2*bu**2*uu + 360*ab**2*auv*bb*uu**2 +
168*ab**2*bb*uu**2*vv - 184*ab**2*bb*uu*uv**2 -
168*ab**2*bu**2*uu*vv + 336*ab**2*bu*bv*uu*uv -
168*ab**2*bv**2*uu**2 + 264*ab*abu*au*bb*uu*uv -
32*ab* abu*au*bu**2*uv - 168*ab*abu*au*bu*bv*uu +
192*ab*abu*av*bb*uu**2 - 456*ab*abv*au*bb*uu**2 +
200*ab*abv*au*bu**2*uu - 7*ab*au**3*bu**3 -
180*ab*au*bb*bu*uu*vv + 44*ab*au*bb*bu*uv**2 +
192*ab*au*bb*bv*uu*uv + 116*ab*au*bu**3*vv -
168*ab*au* bu**2*bv*uv + 84*ab*au*bu*bv**2*uu +
192*ab*av*bb*bu*uu*uv - 192*ab*av*bb*bv*uu **2 -
264*abu*au**2*bb*bv*uu + 42*abu*au**2*bu**2*bv -
96*abu*au*av*bb*bu*uu + 96*abu*bb*bu*uv*vv -
136*abu*bb*bv*uu*vv + 24*abu*bb*bv*uv**2 -
56*abu*bu**2*bv* vv + 112*abu*bu*bv**2*uv -
56*abu*bv**3*uu + 360*abv*au**2*bb*bu*uu -
42*abv*au **2*bu**3 + 40*abv*bb*bu*uu*vv -
24*abv*bb*bu*uv**2 + 56*abv*bu**3*vv -
112*abv* bu**2*bv*uv + 56*abv*bu*bv**2*uu -
264*au**2*auv*bb**2*uu + 42*au**2*auv*bb*bu** 2 +
96*au**2*bb**2*uu*vv - 40*au**2*bb*bu**2*vv -
96*au**2*bb*bv**2*uu + 16*au** 2*bu**2*bv**2 -
192*au*av*bb**2*uu*uv + 192*au*av*bb*bu*bv*uu -
32*au*av*bu**3* bv - 136*auv*bb**2*uu*vv +
24*auv*bb**2*uv**2 + 40*auv*bb*bu**2*vv +
112*auv*bb* bu*bv*uv - 56*auv*bb*bv**2*uu +
96*av**2*bb**2*uu**2 - 96*av**2*bb*bu**2*uu +
16*av**2*bu**4 - 96*bb**2*uu*vv**2 +
96*bb**2*uv**2*vv + 96*bb*bu**2*vv**2 -
```

```
192*bb*bu*bv*uv*vv + 96*bb*bv**2*uu*vv)/8$

if fnc_dep(finew,{aa,ab,bb,bu,uv,uu*aa-vv,ham,fiold},
           {{A,1,0,0},{B,0,1,0},{U,0,0,1},{V,1,0,1}})
then write "This is a known first integral."
else write "This is a new first integral!"$

⟶

The expression in question is functionally dependent
on the list of expressions {p_1,p_2,...} in the
following way:
             2              3               2
10*p_2*p_3*p_5 *p_7+7*p_2*p_7 +12*p_3*p_6*p_7 -2*p_8

This is a known first integral.
```

20.66.5 Complete list of global variables

Type	Name	Purpose
algebraic	`v_`, `gbase_`, `heads_`	needed for `poisson_v()`, `gfi()`, `fnc_dep()`
	`!&c1`, `!&c2`, ...	constant coefficients introduced by `c2s()`
symbolic	`fino_`, `wgths_`	used internally in `gen()`
	`!&r1`, `!&r2`, ...	constant coefficients introduced by `gfi()`
	`tr_vec`	global tracing flag
operator	`poi_`	stores Poisson brackets between scalar and triple products

20.66.6 Requirements

```
load_package v3tools;
```

20.67 WITH: Local Switch Settings

Author: Francis Wright

The operator `with` allows an expression to be evaluated and its value displayed subject to switch settings that apply only locally during the evaluation and display of this expression. Its syntax is

expression `with` *on/off switches*`,` *on/off switches*`,` ...

where *on/off* is either `on` or `off`, and *switches* is a single switch name or a comma-separated sequence of switch names (as for the `on` and `off` commands). It is intended primarily for interactive use and provides a convenient way to experiment with the effects of different switches. Messages about changes of domain mode are suppressed.

Here are some examples, assuming default switch settings:

```
(a+b)^2/2 with off exp, on div;

 1         2
---*(a + b)
 2

pi with on rounded;

3.14159265359

fix(sqrt 10) with on rounded;

3
```

The `with` operator has precedence immediately above `:=`, so `with` binds tighter than `:=` but looser than almost every other infix operator. Therefore,

```
x := pi + e with on rounded;
```

parses as

```
x := ((pi + e) with on rounded);
```

Hence,

```
x := pi + e with on rounded;
```

```
5.85987448205

x;

 103088002085129
-----------------
 17592186044416

x with on rounded;

5.85987448205
```

In the above example, x is assigned a floating-point number, but it is displayed as a rational number when the default switch settings are in effect.

The keywords `on` and `off` can appear as many times as desired in the right operand of `with`. If a switch already has the setting specified in the right operand then it is not changed, but if it has the opposite setting then it is changed before the left operand is evaluated, and displayed if required, and changed back afterwards. Repeated identical switch settings are ignored but conflicting settings cause an error. The order of switch settings is preserved.

Not all switches work sensibly locally, but for example

```
int(sin x, x) with on trint;
```

turns on integration tracing temporarily.

20.68 WU: Wu Algorithm for Polynomial Systems

This is a simple implementation of the Wu algorithm implemented in REDUCE working directly from [Wt87].

Author: Russell Bradford

Its purpose was to aid my understanding of the algorithm, so the code is simple, and has a lot of tracing included. This is a working implementation, but there is magnificent scope for improvement and optimisation. Things like using intelligent sorts on polynomial lists, and avoiding the re-computation of various data spring easily to mind. Also, an attempt at factorization of the input polynomials at each pass might have beneficial results. Of course, exploitation of the natural parallel structure is a must!

All bug fixes and improvements are welcomed.

The interface:

```
wu({x^2+y^2+z^2-r^2, x*y+z^2-1, x*y*z-x^2-y^2-z+1},
   {x,y,z});
```

calls `wu` with the named polynomials, and with the variable ordering $\mathtt{x} > \mathtt{y} > \mathtt{z}$. In this example, `r` is a parameter.

The result is

```
    2    3    2
{{{r  + z  - z  - 1,

    2  2    2       2    4    2  2    2
   r *y  + r *z + r  - y  - y *z  + z  - z - 2,

          2
   x*y + z  - 1},

  y},

    2  4       2  2    2    7    6       5    4    3
 {{r *z  - 2*r *z  + r  + z  - z  - 2*z  + z  + z

       2
    + z  - 1,

    2   2    3    2
```

```
 y *(r  + z  - z  - 1),

         2
 x*y + z  - 1},

    2    3    2
y*(r  + z  - z  - 1)}}
```

namely, a list of pairs of characteristic sets and initials for the characteristic sets.

Thus, the first pair above has the characteristic set

$$r^2 + z^3 - z^2 - 1, r^2y^2 + r^2z + r^2 - y^4 - y^2z^2 + z^2 - z - 2, xy + z^2 - 1$$

and initial y.

According to Wu's theorem, the set of roots of the original polynomials is the union of the sets of roots of the characteristic sets, with the additional constraints that the corresponding initial is non-zero. Thus, for the first pair above, we find the roots of $\{r^2 + z^3 - z^2 - 1, \dots\}$ under the constraint that $y \neq 0$. These roots, together with the roots of the other characteristic set (under the constraint of $y(r^2 + z^3 - z^2 - 1) \neq 0$), comprise all the roots of the original set.

Additional information about the working of the algorithm can be gained by

```
on trwu;
```

This prints out details of the choice of basic sets, and the computation of characteristic sets.

The second argument (the list of variables) may be omitted, when all the variables in the input polynomials are implied with some random ordering.

20.69 XCOLOR: Color Factor in some Field Theories

This package calculates the color factor in non-abelian gauge field theories using an algorithm due to Cvitanovich.

Documentation for this package is in plain text.

Author: A. Kryukov

Program "xCOLOR" is intended for calculation the colour factor in non-abelian gauge field theories. It is realized Cvitanovich algorithm [Cvi76]. In comparison to the program "COLOR" [KR88] many improvements were made. The package was written in symbolic mode. This version is more than 10 times faster than the one in [KR88].

After load the program by the following command `load xcolor;`
user can be able to use the next additional commands and operators.

Command SUdim.

Format: `SUdim <any expression>;`
Set the order of SU group.
The default value is 3, i.e. SU(3).

Command SpTT.

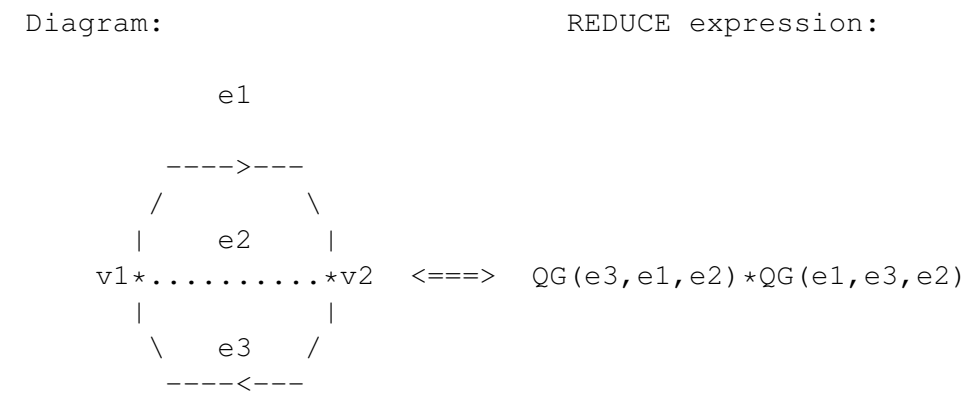
Format: `SpTT <any expression>;`
Set the normalization coefficient A: Sp(TiTj) = A*Delta(i,j). Default value is 1/2.

Operator QG.

Format: `QG(inQuark,outQuark,Gluon)`
Describe the quark-gluon vertex. Parameters may be any identifiers. First and second of then must be in- and out- quarks correspondently. Third one is a gluon.

Operator G3.

Format: `G3(Gluon1,Gluon2,Gluon3)`
Describe the three-gluon vertex. Parameters may be any identifiers. The order of gluons must be clock.
In terms of QG and G3 operators you input diagram in "color" space as a product of these operators. For example.



For more detail see [KR88].

20.70 XIDEAL: Gröbner Bases for Exterior Algebra

XIDEAL constructs Gröbner bases for solving the left ideal membership problem: Gröbner left ideal bases or GLIBs. For graded ideals, where each form is homogeneous in degree, the distinction between left and right ideals vanishes. Furthermore, if the generating forms are all homogeneous, then the Gröbner bases for the non-graded and graded ideals are identical. In this case, XIDEAL is able to save time by truncating the Gröbner basis at some maximum degree if desired.

Author: David Hartley

20.70.1 Description

The method of Gröbner bases in commutative polynomial rings introduced by Buchberger (e.g. [Buc85]) is a well-known and very important tool in polynomial ideal theory, for example in solving the ideal membership problem. XIDEAL extends the method to exterior algebras using algorithms from [HT93] and [Ape92].

There are two main departures from the commutative polynomial case. First, owing to the non-commutative product in exterior algebras, ideals are no longer automatically two-sided, and it is necessary to distinguish between left and right ideals. Secondly, because there are zero divisors, confluent reduction relations are no longer sufficient to solve the ideal membership problem: a unique normal form for every polynomial does not guarantee that all elements in the ideal reduce to zero. This leads to two possible definitions of Gröbner bases as pointed out by Stokes [Sto90].

XIDEAL constructs Gröbner bases for solving the left ideal membership problem: Gröbner left ideal bases or GLIBs. For graded ideals, where each form is homogeneous in degree, the distinction between left and right ideals vanishes. Furthermore, if the generating forms are all homogeneous, then the Gröbner bases for the non-graded and graded ideals are identical. In this case, XIDEAL is able to save time by truncating the Gröbner basis at some maximum degree if desired.

XIDEAL uses the exterior calculus package EXCALC of E. Schrüfer [Sch85a] to provide the exterior algebra definitions. EXCALC is loaded automatically with XIDEAL. The exterior variables may be specified explicitly, or extracted automatically from the input polynomials. All distinct exterior variables in the input are assumed to be linearly independent – if a dimension has been fixed (using the EXCALC `spacedim` or `coframe` statements), then input containing distinct exterior variables with degrees totaling more than this number will generate an error.

20.70.2 Declarations

xorder

`xorder` sets the term ordering for all other calculations. The syntax is

```
xorder k
```

where `k` is one of `lex`, `gradlex` or `deglex`. The lexicographical ordering `lex` is based on the prevailing EXCALC kernel ordering for the exterior variables. The EXCALC kernel ordering can be changed with the REDUCE `korder` or EXCALC `forder` declarations. The graded lexicographical ordering `gradlex` puts terms with more factors first (irrespective of their exterior degrees) and sorts terms of the same grading lexicographically. The degree lexicographical ordering `deglex` takes account of the exterior degree of the variables, putting highest degree first and then sorting terms of the same degree lexicographically. The default ordering is `deglex`.

xvars

It is possible to consider scalar and 0-form variables in exterior polynomials in two ways: as variables or as coefficient parameters. This difference is crucial for Gröbner basis calculations. By default, all scalar variables which have been declared as 0-forms are treated as exterior variables, along with any EXCALC kernels of degree 0. This division can be changed with the `xvars` declaration. The syntax is

```
xvars U,V,W,...
```

where the arguments are either kernels or lists of kernels. All variables specified in the `xvars` declaration are treated as exterior variables in subsequent XIDEAL calculations with exterior polynomials, and any other scalars are treated as parameters. This is true whether or not the variables have been declared as 0-forms. The declaration

```
xvars {}
```

causes all degree 0 variables to be treated as parameters, and

```
xvars nil
```

restores the default. Of course, p-form kernels with $p \neq 0$ are always considered as exterior variables. The order of the variables in an `xvars` declaration has no effect on the REDUCE kernel ordering or XIDEAL term ordering.

20.70.3 Operators

xideal

`xideal` calculates a Gröbner left ideal basis in an exterior algebra. The syntax is

`xideal`(⟨*S:list of forms*⟩[,⟨*V:list of kernels*⟩][,⟨*R:integer*⟩]) :list of forms.

`xideal` calculates a Gröbner basis for the left ideal generated by $\langle S \rangle$ using the current term ordering. The resulting list can be used for subsequent reductions with `xmod` as long as the term ordering is not changed. Which 0-form variables are to be regarded as exterior variables can be specified in an optional argument $\langle V \rangle$ (just like an `xvars` declaration). The order of variables in $\langle V \rangle$ has no effect on the term ordering. If the set of generators $\langle S \rangle$ is graded, an optional parameter $\langle R \rangle$ can be given, and `xideal` produces a truncated basis suitable for reducing exterior forms of degree less than or equal to $\langle R \rangle$ in the left ideal. This can save time and space with large problems, but the result cannot be used for exterior forms of degree greater than $\langle R \rangle$. The forms returned by `xideal` are sorted in increasing order. See also the switches `trxideal` and `xfullreduction`.

xmodideal

`xmodideal` reduces exterior forms to their (unique) normal forms modulo a left ideal. The syntax is

`xmodideal`(⟨*F:form*⟩, ⟨*S:list of forms*⟩):form
or
`xmodideal`(⟨*F:list of forms*⟩, ⟨*S:list of forms*⟩) :list of forms.

An alternative infix syntax is also available:

$\langle F \rangle$ xmodideal $\langle S \rangle$.

`xmodideal`($\langle F \rangle$,$\langle S \rangle$) first calculates a Gröbner basis for the left ideal generated by $\langle S \rangle$, and then reduces $\langle F \rangle$. $\langle F \rangle$ may be either a single exterior form, or a list of forms, and $\langle S \rangle$ is a list of forms. If $\langle F \rangle$ is a list of forms, each element is reduced, and any which vanish are deleted from the result. If the set of generators $\langle S \rangle$ is graded, then a truncated Gröbner basis is calculated using the degree of $\langle F \rangle$ (or the maximal degree in $\langle F \rangle$). See also `trxmod`.

xmod

`xmod` reduces exterior forms to their (not necessarily unique) normal forms modulo a set of exterior polynomials. The syntax is

`xmod`(⟨*F:form*⟩, ⟨*S:list of forms*⟩):form
or
`xmod`(⟨*F:list of forms*⟩, ⟨*S:list of forms*⟩):list of forms.

An alternative infix syntax is also available:

⟨*F*⟩ `xmod` ⟨*S*⟩ .

`xmod`(⟨*F*⟩,⟨*S*⟩) reduces ⟨*F*⟩ with respect to the set of exterior polynomials ⟨*S*⟩, which is not necessarily a Gröbner basis. ⟨*F*⟩ may be either a single exterior form, or a list of forms, and ⟨*S*⟩ is a list of forms. This operator can be used in conjunction with `xideal` to produce the same effect as `xmodideal`: for a single homogeneous form ⟨*F*⟩ and a set of exterior forms ⟨*S*⟩, ⟨*F*⟩ `xmodideal` ⟨*S*⟩ is equivalent to ⟨*F*⟩ `xmod` `xideal`(⟨*S*⟩,`exdegree` ⟨*F*⟩). See also `trxmod`.

xauto

`xauto` autoreduces a set of exterior forms. The syntax is

`xauto`(⟨*S:list of forms*⟩):list of forms.

`xauto` ⟨*S*⟩ returns a set of exterior polynomials which generate the same left ideal, but which are in normal form with respect to each other. For linear expressions, this is equivalent to finding the reduced row echelon form of the coefficient matrix.

excoeffs

The operator `excoeffs`, with syntax

`excoeffs`(⟨*F:form*⟩):list of expressions

returns the coefficients from an exterior form as a list. The coefficients are always scalars, but which degree 0 variables count as coefficient parameters is controlled by the command `xvars`.

exvars

The operator `exvars`, with syntax

`exvars`(⟨*F:form*⟩):list of kernels

returns the exterior powers from an exterior form as a list. All non-scalar variables are returned, but which degree 0 variables count as coefficient parameters is controlled by the command `xvars`.

20.70.4 Switches

xfullreduce

`on xfullreduce` allows `xideal` and `xmodideal` to calculate reduced, monic Gröbner bases, which speeds up subsequent reductions, and guarantees a unique form for the Gröbner basis. `off xfullreduce` turns of this feature, which may speed up calculation of some Gröbner basis. `xfullreduce` is `on` by default.

trxideal

`on trxideal` produces a trace of the calculations done by `xideal` and `xmodideal`, showing the basis polynomials and the results of the critical element calculations. This can generate profuse amounts of output. `trxideal` is `off` by default.

trxmod

`on trxmod` produces a trace of reductions to normal form during calculations by XIDEAL operators. `trxmod` is `off` by default.

20.70.5 Examples

Suppose XIDEAL has been loaded, the switches are at their default settings, and the following exterior variables have been declared:

```
pform x=0,y=0,z=0,t=0,f(i)=1,h=0,hx=0,ht=0;
```

In a commutative polynomial ring, a single polynomial is its own Gröbner basis. This is no longer true for exterior algebras because of the presence of zero divisors, and can lead to some surprising reductions:

```
xideal {d x^d y - d z^d t};

     {d t^d z + d x^d y,

      d x^d y^d z,

      d t^d x^d y}

f(3)^f(4)^f(5)^f(6)
  xmodideal {f(1)^f(2) + f(3)^f(4) + f(5)^f(6)};

     0
```

The heat equation, $h_{xx} = h_t$ can be represented by the following exterior differential system.

```
S := {d h - ht*d t - hx*d x,
      d ht^d t + d hx^d x,
      d hx^d t - ht*d x^d t};
```

`xmodideal` can be used to check that the exterior differential system is closed under exterior differentiation.

```
d S xmodideal S;

     {}
```

`xvars`, or a second argument to `xideal` can be used to change the division between exterior variables of degree 0 and parameters.

```
xideal {a*d x+y*d y};

          d x*a + d y*y
        {---------------}
                a

xvars {a};
xideal {a*d x+y*d y};

        {d x*a + d y*y,d x^d y}

xideal({a*d x+y*d y},{a,y});

        {d x*a + d y*y,
```

```
          d x^d y*y}

xvars {};       % all 0-forms are coefficients
excoeffs(d u - (a*p - q)*d y);

        {1, - a*p + q}

exvars(d u - (a*p - q)*d y);

        {d u,d y}

xvars {p,q};    % p,q are no longer coefficients
excoeffs(d u - (a*p - q)*d y);

        { - a,1,1}

exvars(d u - (a*p - q)*d y);

        {d y*p,d y*q,d u}

xvars nil;
```

Non-graded left and right ideals are no longer the same:

```
d t^(d z+d x^d y) xmodideal {d z+d x^d y};

     0

(d z+d x^d y)^d t xmodideal {d z+d x^d y};

      - 2*d t^d z
```

Any form containing a 0-form term generates the whole ideal:

```
xideal {1 + f(1) + f(1)^f(2) + f(2)^f(3)^f(4)};

     {1}
```

20.71 ZEILBERG: Indefinite and Definite Summation

This package is a careful implementation of the Gosper and Zeilberger algorithms for indefinite and definite summation of hypergeometric terms, respectively. Extensions of these algorithms are also included that are valid for ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials that are rational-linear in their arguments.

Authors: Gregor Stölting and Wolfram Koepf

20.71.1 Introduction

This package is a careful implementation of the Gosper[53] and Zeilberger algorithms for indefinite, and definite summation of hypergeometric terms, respectively. Further, extensions of these algorithms given by the first author are covered. An expression a_k is called a *hypergeometric term* (or *closed form*), if a_k/a_{k-1} is a rational function with respect to k. Typical hypergeometric terms are ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials (Pochhammer symbols) that are integer-linear in their arguments.

The extensions of Gosper's and Zeilberger's algorithm mentioned in particular are valid for ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials that are rational-linear in their arguments.

20.71.2 Gosper Algorithm

The Gosper algorithm [Gos78] is a *decision procedure*, that decides by algebraic calculations whether or not a given hypergeometric term a_k has a hypergeometric term antidifference g_k, i.e. $g_k - g_{k-1} = a_k$ with rational g_k/g_{k-1}, and returns g_k if the procedure is successful, in which case we call a_k *Gosper-summable*. Otherwise *no hypergeometric term antidifference exists*. Therefore if the Gosper algorithm does not return a closed form solution, it has *proved* that no such solution exists, an information that may be quite useful and important. The Gosper algorithm is the discrete analogue of the Risch algorithm for integration in terms of elementary functions.

Any antidifference is uniquely determined up to a constant, and is denoted by

$$g_k = \sum_k a_k \ .$$

Finding g_k given a_k is called *indefinite summation*. The antidifference operator Σ is the inverse of the downward difference operator $\nabla a_k = a_k - a_{k-1}$. There is

[53] The SUM package contains also a partial implementation of the Gosper algorithm.

an analogous summation theory corresponding to the upward difference operator $\Delta a_k = a_{k+1} - a_k$.

In case, an antidifference g_k of a_k is known, any sum

$$\sum_{k=m}^{n} a_k = g_n - g_{m-1}$$

can be easily calculated by an evaluation of g at the boundary points like in the integration case. Note, however, that the sum

$$\sum_{k=0}^{n} \binom{n}{k} \tag{20.102}$$

e. g. is not of this type since the summand $\binom{n}{k}$ depends on the upper boundary point n explicitly. This is an example of a definite sum that we consider in the next section.

Our package supports the input of powers (`a^k`), factorials (`factorial(k)`), Γ function terms (`gamma(a)`), binomial coefficients (`Binomial(n,k)`), shifted factorials (`Pochhammer(a,k)` $= a(a+1)\cdots(a+k-1) = \Gamma(a+k)/\Gamma(a)$), and partially products (`prod(f,k,k1,k2)`). It takes care of the necessary simplifications, and therefore provides you with the solution of the decision problem as long as the memory or time requirements are not too high for the computer used.

20.71.3 Zeilberger Algorithm

The (fast) Zeilberger algorithm [Zei90]–[Zei91] deals with the *definite summation* of hypergeometric terms. Zeilberger's paradigm is to find (and return) a linear homogeneous recurrence equation with polynomial coefficients (called *holonomic equation*) for an *infinite sum*

$$s(n) = \sum_{k=-\infty}^{\infty} f(n,k) \ ,$$

the summation to be understood over all integers k, if $f(n,k)$ is a hypergeometric term with respect to both k and n. The existence of a holonomic recurrence equation for $s(n)$ is then generally guaranteed.

If one is lucky, and the resulting recurrence equation is of first order

$$p(n)\, s(n-1) + q(n)\, s(n) = 0 \qquad (p, q \text{ polynomials}) \ ,$$

$s(n)$ turns out to be a hypergeometric term, and a closed form solution can be easily established using a suitable initial value, and is represented by a ratio of Pochhammer or Γ function terms if the polynomials p, and q can be factored.

Zeilberger's algorithm does not guarantee to find the holonomic equation of lowest order, but often it does.

If the resulting recurrence equation has order larger than one, this information can be used for identification purposes: Any other expression satisfying the same recurrence equation, and the same initial values, represents the same function.

Note that a *definite sum* $\sum_{k=m_1}^{m_2} f(n,k)$ is an infinite sum if $f(n,k) = 0$ for $k < m_1$ and $k > m_2$. This is often the case, an example of which is the sum (20.102) considered above, for which the hypergeometric recurrence equation $2s(n-1) - s(n) = 0$ is generated by Zeilberger's algorithm, leading to the closed form solution $s(n) = 2^n$.

Definite summation is trivial if the corresponding indefinite sum is Gosper-summable analogously to the fact that definite integration is trivial as soon as an elementary antiderivative is known. If this is not the case, the situation is much more difficult, and it is therefore quite remarkable and non-obvious that Zeilberger's method is just a clever application of Gosper's algorithm.

Our implementation is mainly based on [Koo93] and [Koe94b]. More examples can be found in [PS95], [Str93], [Wil90], and [Wil93] many of which are contained in the test file `zeilberg.tst`.

20.71.4 REDUCE operator GOSPER

The `gosper` operator is an implementation of the Gosper algorithm.

- `gosper(a,k)` determines a closed form antidifference. If it does not return a closed form solution, then a closed form solution does not exist.
- `gosper(a,k,m,n)` determines

$$\sum_{k=m}^{n} a_k$$

  using Gosper's algorithm. This is only successful if Gosper's algorithm applies.

Example:

```
2: gosper((-1)^(k+1)*(4*k+1)*factorial(2*k)/
   (factorial(k)*4^k*(2*k-1)*factorial(k+1)),k);

                k
      - ( - 1)  *factorial(2*k)
-----------------------------------
```

```
  2*k
2    *factorial(k + 1)*factorial(k)
```

This solves a problem given in SIAM Review ([OK94], Problem 94–2) where it was asked to determine the infinite sum

$$S = \lim_{n\to\infty} S_n\,, \qquad S_n = \sum_{k=1}^{n} \frac{(-1)^{k+1}(4k+1)(2k-1)!!}{2^k(2k-1)(k+1)!}\,,$$

($(2k-1)!! = 1\cdot 3\cdots(2k-1) = \frac{(2k)!}{2^k\,k!}$). The above calculation shows that the summand is Gosper-summable, and the limit $S = 1$ is easily established using Stirling's formula.

The implementation solves further deep and difficult problems some examples of which are:

```
3:  gosper(sub(n=n+1,binomial(n,k)^2/binomial(2*n,n))-
    binomial(n,k)^2/binomial(2*n,n),k);

                      2
((binomial(n + 1,k)  *binomial(2*n,n)

                                                    2
   - binomial(2*(n + 1),n + 1)*binomial(n,k)  )

                                  2
 *(2*k - 3*n - 1)*(k - n - 1)  )/((

      2*(2*(n + 1) - k)*(2*n + 1)*k

                             2
       - (3*n + 1)*(n + 1)  )

   *binomial(2*(n + 1),n + 1)*binomial(2*n,n))

4: gosper(binomial(k,n),k);

 (k + 1)*binomial(k,n)
-----------------------
         n + 1

5: gosper((-25+15*k+18*k^2-2*k^3-k^4)/
   (-23+479*k+613*k^2+137*k^3+53*k^4+5*k^5+k^6),k);

          2
    - (2*k  - 15*k + 8)*k
-----------------------------
      3      2
 23*(k  + 4*k  + 27*k + 23)
```

The Gosper algorithm is not capable to give antidifferences depending on the harmonic numbers

$$H_k := \sum_{j=1}^{k} \frac{1}{j} \,,$$

e. g. $\sum_k H_k = (k+1)(H_{k+1}-1)$, but, is able to give a proof, instead, for the fact that H_k does not possess a closed form evaluation:

```
6: gosper(1/k,k);

***** Gosper algorithm: no closed form solution exists
```

The following code gives the solution to a summation problem proposed in Gosper's original paper [Gos78]. Let

$$f_k = \prod_{j=1}^{k} (a + b\,j + c\,j^2) \qquad \text{and} \qquad g_k = \prod_{j=1}^{k} (e + b\,j + c\,j^2) \,.$$

Then a closed form solution for

$$\sum_k \frac{f_{k-1}}{g_k}$$

is found by the definitions

```
7: operator ff,gg$

8: let {ff(~k+~m) => ff(k+m-1)*(c*(k+m)^2+b*(k+m)+a)
        when (fixp(m) and m>0),
   ff(~k+~m) => ff(k+m+1)/(c*(k+m+1)^2+b*(k+m+1)+a)
        when (fixp(m) and m<0)}$

9: let {gg(~k+~m) => gg(k+m-1)*(c*(k+m)^2+b*(k+m)+e)
        when (fixp(m) and m>0),
   gg(~k+~m) => gg(k+m+1)/(c*(k+m+1)^2+b*(k+m+1)+e)
        when (fixp(m) and m<0)}$
```

and the calculation

```
10: gosper(ff(k-1)/gg(k),k);

     ff(k)
---------------
 (a - e)*gg(k)

11: clear ff,gg$
```

Similarly closed form solutions of $\sum_k \frac{f_{k-m}}{g_k}$ for positive integers m can be obtained, as well as of $\sum_k \frac{f_{k-1}}{g_k}$ for

$$f_k = \prod_{j=1}^{k}(a + b\,j + c\,j^2 + d\,j^3) \qquad \text{and} \qquad g_k = \prod_{j=1}^{k}(e + b\,j + c\,j^2 + d\,j^3)$$

and for analogous expressions of higher degree polynomials.

20.71.5 REDUCE operator EXTENDED_GOSPER

The `extended_gosper` operator is an implementation of an extended version of Gosper's algorithm given by Koepf [Koe94b].

- `extended_gosper(a,k)` determines an antidifference g_k of a_k whenever there is a number m such that $h_k - h_{k-m} = a_k$, and h_k is an *m-fold hypergeometric term*, i.e.

  $$h_k/h_{k-m} \quad \text{is a rational function with respect to } k.$$

  If it does not return a solution, then such a solution does not exist.

- `extended_gosper(a,k,m)` determines an *m-fold antidifference* h_k of a_k, i.e. $h_k - h_{k-m} = a_k$, if it is an m-fold hypergeometric term.

Examples:

```
12: extended_gosper(binomial(k/2,n),k);

                    k                            k - 1
 (k + 2)*binomial(---,n) + (k + 1)*binomial(-------,n)
                    2                              2
-------------------------------------------------------
                        2*(n + 1)

13: extended_gosper(k*factorial(k/7),k,7);

                    k
(k + 7)*factorial(---)
                    7
```

20.71.6 REDUCE operator SUMRECURSION

The `sumrecursion` operator is an implementation of the (fast) Zeilberger algorithm.

- `sumrecursion(f,k,n)` determines a holonomic recurrence equation for

$$\texttt{summ(n)} = \sum_{k=-\infty}^{\infty} f(n,k)$$

with respect to n, applying `extended_sumrecursion` if necessary, see § 20.71.7. The resulting expression equals zero.

- `sumrecursion(f,k,n,j)` searches for a holonomic recurrence equation of order j. This operator does not use `extended_sumrecursion` automatically. Note that if j is too large, the recurrence equation may not be unique, and only one particular solution is returned.

A simple example deals with Equation (20.102)[54]

```
14: sumrecursion(binomial(n,k),k,n);

2*summ(n - 1) - summ(n)
```

The whole *hypergeometric database* of the *Vandermonde*, *Gauß*, *Kummer*, *Saalschütz*, *Dixon*, *Clausen* and *Dougall identities* (see [Wil93]), and many more identities (see e. g. [Koe94b]), can be obtained using `sumrecursion`. As examples, we consider the difficult cases of Clausen and Dougall:

```
15: summand:=factorial(a+k-1)*factorial(b+k-1)/
      (factorial(k)*factorial(-1/2+a+b+k))
     *factorial(a+n-k-1)*factorial(b+n-k-1)/
      (factorial(n-k)*factorial(-1/2+a+b+n-k))$

16: sumrecursion(summand,k,n);

(2*a + 2*b + 2*n - 1)*(2*a + 2*b + n - 1)*summ(n)*n

 - 2*(2*a + n - 1)*(a + b + n - 1)*(2*b + n - 1)*summ(n - 1)

17: summand:=pochhammer(d,k)*pochhammer(1+d/2,k)*
     pochhammer(d+b-a,k)*pochhammer(d+c-a,k)*
     pochhammer(1+a-b-c,k)*pochhammer(n+a,k)*
     pochhammer(-n,k)/(factorial(k)*pochhammer(d/2,k)*
     pochhammer(1+a-b,k)*pochhammer(1+a-c,k)*
     pochhammer(b+c+d-a,k)*pochhammer(1+d-a-n,k)*
     pochhammer(1+d+n,k))$

18: sumrecursion(summand,k,n);
```

[54] Note that with REDUCE Version 3.5 we use the global operator `summ` instead of `sum` to denote the sum.

```
 - (n - 1 + d + c + b - a)*(n - 1 - d + a)

*(b - n - a)*(c - n - a)*summ(n) -

(d - n + c + b - 2*a)*(n - 1 + b)*(n - 1 + c)

*(d + n)*summ(n - 1)
```

corresponding to the statements

$$ {}_4F_3\left(\left.\begin{matrix} a\,,b\,,1/2-a-b-n\,,-n \\ 1/2+a+b\,,1-a-n\,,1-b-n \end{matrix}\right|1\right)=\frac{(2a)_n\,(a+b)_n\,(2b)_n}{(2a+2b)_n\,(a)_n\,(b)_n} $$

and

$$ {}_7F_6\left(\left.\begin{matrix} d\,,1+d/2\,,d+b-a\,,d+c-a\,,1+a-b-c\,,n+a\,,-n \\ d/2\,,1+a-b\,,1+a-c\,,b+c+d-a\,,1+d-a-n\,,1+d+n \end{matrix}\right|1\right) $$

$$ =\frac{(d+1)_n\,(b)_n\,(c)_n\,(1+2\,a-b-c-d)_n}{(a-d)_n\,(1+a-b)_n\,(1+a-c)_n\,(b+c+d-a)_n} $$

(compare next section), respectively.

Other applications of the Zeilberger algorithm are connected with the verification of identities. To prove the identity

$$ \sum_{k=0}^{n}\binom{n}{k}^3=\sum_{k=0}^{n}\binom{n}{k}^2\binom{2k}{n}\,, $$

e. g., we may prove that both sums satisfy the same recurrence equation

```
19: sumrecursion(binomial(n,k)^3,k,n);


          2                             2
8*(n - 1)  *summ(n - 2) - summ(n)*n

        2
 + (7*n  - 7*n + 2)*summ(n - 1)

20: sumrecursion(binomial(n,k)^2*binomial(2*k,n),k,n);


          2                             2
8*(n - 1)  *summ(n - 2) - summ(n)*n
```

```
        2
 + (7*n  - 7*n + 2)*summ(n - 1)
```

and finally check the initial conditions:

```
21: sub(n=0,k=0,binomial(n,k)^3);

1

22: sub(n=0,k=0,binomial(n,k)^2*binomial(2*k,n));

1

23: sub(n=1,k=0,binomial(n,k)^3)
    +sub(n=1,k=1,binomial(n,k)^3);

2

24: sub(n=1,k=0,binomial(n,k)^2*binomial(2*k,n))+
    sub(n=1,k=1,binomial(n,k)^2*binomial(2*k,n));

2
```

20.71.7 REDUCE operator EXTENDED_SUMRECURSION

The `extended_sumrecursion` operator is an implementation of an extension of the (fast) Zeilberger algorithm given by Koepf [Koe94b].

- `extended_sumrecursion(f,k,n,m,l)` determines a holonomic recurrence equation for `summ(n)` $= \sum_{k=-\infty}^{\infty} f(n,k)$ with respect to n if $f(n,k)$ is an (m,l)-*fold hypergeometric term* with respect to (n,k), i.e.

  $$\frac{F(n,k)}{F(n-m,k)} \quad \text{and} \quad \frac{F(n,k)}{F(n,k-l)}$$

  are rational functions with respect to both n and k. The resulting expression equals zero.

- Internally, `sumrecursion(f,k,n)` calls (with suitable values m and l) `extended_sumrecursion(f,k,n,m,l)` and covers therefore the extended algorithm completely.

Examples:

```
25: extended_sumrecursion(binomial(n,k)*binomial(k/2,n),
                          k,n,1,2);

summ(n - 1) + 2*summ(n)
```

which can be obtained automatically by

```
26: sumrecursion(binomial(n,k)*binomial(k/2,n),k,n);

summ(n - 1) + 2*summ(n)
```

Similarly, we get

```
27: extended_sumrecursion(binomial(n/2,k),k,n,2,1);

2*summ(n - 2) - summ(n)

28: sumrecursion(binomial(n/2,k),k,n);

2*summ(n - 2) - summ(n)

29: sumrecursion(hyperterm({a,b,a+1/2-b,1+2*a/3,-n},
    {2*a+1-2*b,2*b,2/3*a,1+a+n/2},4,k)/(factorial(n)*2^(-n)/
    factorial(n/2))/
     hyperterm({a+1,1},{a-b+1,b+1/2},1,n/2),k,n);

summ(n - 2) - summ(n)
```

In the last example, the progam chooses $m = 2$, and $l = 1$ to derive the resulting recurrence equation (see [Koe94b], Table 3, (1.3)).

20.71.8 REDUCE operator `HYPERRECURSION`

Sums to which the Zeilberger algorithm applies, in general are special cases of the *generalized hypergeometric function*

$$ {}_pF_q\left(\begin{array}{cccc} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{array}\,\middle|\, x\right) := \sum_{k=0}^{\infty} \frac{(a_1)_k \cdot (a_2)_k \cdots (a_p)_k}{(b_1)_k \cdot (b_2)_k \cdots (b_q)_k \, k!} x^k $$

with upper parameters $\{a_1, a_2, \ldots, a_p\}$, and lower parameters $\{b_1, b_2, \ldots, b_q\}$. If a recursion for a generalized hypergeometric function is to be established, you can use the following REDUCE operator:

- `hyperrecursion(upper,lower,x,n)` determines a holonomic recurrence equation with respect to n for ${}_pF_q\left(\begin{array}{cccc} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{array}\,\middle|\, x\right)$,

where `upper=` $\{a_1, a_2, \ldots, a_p\}$ is the list of upper parameters, and `lower=` $\{b_1, b_2, \ldots, b_q\}$ is the list of lower parameters depending on n. If Zeilberger's algorithm does not apply, `extended_sumrecursion` of § 20.71.7 is used.

- `hyperrecursion(upper,lower,x,n,j)` ($j \in \mathbb{N}$) searches only for a holonomic recurrence equation of order j. This operator does not use `extended_sumrecursion` automatically.

Therefore

```
30: hyperrecursion({-n,b},{c},1,n);

(b - c - n + 1)*summ(n - 1) + (c + n - 1)*summ(n)
```

establishes the Vandermonde identity

$$ {}_2F_1\left(\begin{matrix} -n\,, & b \\ & c \end{matrix} \middle| 1 \right) = \frac{(c-b)_n}{(c)_n}\,, $$

whereas

```
31: hyperrecursion({d,1+d/2,d+b-a,d+c-a,1+a-b-c,n+a,-n},
              {d/2,1+a-b,1+a-c,b+c+d-a,1+d-a-n,1+d+n},1,n);

 - (n - 1 + d + c + b - a)*(n - 1 - d + a)

*(b - n - a)*(c - n - a)*summ(n) -

(d - n + c + b - 2*a)*(n - 1 + b)*(n - 1 + c)

*(d + n)*summ(n - 1)
```

proves Dougall's identity, again.

If a hypergeometric expression is given in hypergeometric notation, then the use of `hyperrecursion` is more natural than the use of `sumrecursion`.

Moreover you may use the REDUCE operator

- `hyperterm(upper,lower,x,k)` that yields the hypergeometric term

$$ \frac{(a_1)_k \cdot (a_2)_k \cdots (a_p)_k}{(b_1)_k \cdot (b_2)_k \cdots (b_q)_k\, k!} x^k $$

with upper parameters `upper=` $\{a_1, a_2, \ldots, a_p\}$, and lower parameters `lower=` $\{b_1, b_2, \ldots, b_q\}$

in connection with hypergeometric terms.

The operator `sumrecursion` can also be used to obtain three-term recurrence equations for systems of orthogonal polynomials with the aid of known hypergeometric representations. By ([NUS91], (2.7.11a)), the discrete Krawtchouk polynomials $k_n^{(p)}(x, N)$ have the hypergeometric representation

$$k_n^{(p)}(x,N) = (-1)^n\, p^n \binom{N}{n}\, {}_2F_1\left(\begin{matrix} -n\,, & -x \\ & -N \end{matrix} \,\middle|\, \frac{1}{p} \right) ,$$

and therefore we declare

```
32: krawtchoukterm:=
    (-1)^n*p^n*binomial(NN,n)*hyperterm({-n,-x},{-NN},1/p,k)$
```

and get the three three-term recurrence equations

```
33: sumrecursion(krawtchoukterm,k,n);

((2*p - 1)*n - nn*p - 2*p + x + 1)*summ(n - 1)

 - (n - nn - 2)*(p - 1)*summ(n - 2)*p - summ(n)*n

34: sumrecursion(krawtchoukterm,k,x);

(2*(x - 1)*p + n - nn*p - x + 1)*summ(x - 1)

 - ((x - 1) - nn)*summ(x)*p - (p - 1)*(x - 1)*summ(x - 2)

35: sumrecursion(krawtchoukterm,k,NN);

(x + 1 + n + (p - 2)*nn)*summ(nn - 1) - (

   (x + 1 - nn)*summ(nn - 2)

    - (n - nn)*(p - 1)*summ(nn))
```

with respect to the parameters n, x, and N respectively.

20.71.9 REDUCE operator HYPERSUM

With the operator `hypersum`, hypergeometric sums are directly evaluated in closed form whenever the extended Zeilberger algorithm leads to a recurrence equation containing only two terms:

- `hypersum(upper,lower,x,n)` determines a closed form representa-

tion for ${}_pF_q\left(\begin{matrix} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{matrix} \middle| x\right)$, where `upper`$= \{a_1, a_2, \ldots, a_p\}$ is the list of upper parameters, and `lower`$= \{b_1, b_2, \ldots, b_q\}$ is the list of lower parameters depending on n. The result is given as a hypergeometric term with respect to n.

If the result is a list of length m, we call it *m-fold symmetric*, which is to be interpreted as follows: Its j^{th} part is the solution valid for all n of the form $n = mk + j - 1$ $(k \in \mathbb{N}_0)$. In particular, if the resulting list contains two terms, then the first part is the solution for even n, and the second part is the solution for odd n.

Examples [Koe94b]:

```
36: hypersum({a,1+a/2,c,d,-n},{a/2,1+a-c,1+a-d,1+a+n},1,n);

 pochhammer(a - c - d + 1,n)*pochhammer(a + 1,n)
---------------------------------------------------
 pochhammer(a - c + 1,n)*pochhammer(a - d + 1,n)

37: hypersum({a,1+a/2,d,-n},{a/2,1+a-d,1+a+n},-1,n);

  pochhammer(a + 1,n)
-------------------------
 pochhammer(a - d + 1,n)
```

Note that the operator `togamma` converts expressions given in factorial-Γ-binomial-Pochhammer notation into a pure Γ function representation:

```
38: togamma(ws);

 gamma(a - d + 1)*gamma(a + n + 1)
-----------------------------------
 gamma(a - d + n + 1)*gamma(a + 1)
```

Here are some m-fold symmetric results:

```
39: hypersum({-n,-n,-n},{1,1},1,n);

          n/2              2   n                1   n
  ( - 27)    *pochhammer(---,---)*pochhammer(---,---)
                          3   2                3   2
{-------------------------------------------------------,
                                n  2
                  factorial(---)
                                2
 0}
```

```
40: hypersum({-n,n+3*a,a},{3*a/2,(3*a+1)/2},3/4,n);

                     2   n                1   n
        pochhammer(---,---)*pochhammer(---,---)
                     3   3                3   3
{-------------------------------------------------,
              3*a + 2   n                3*a + 1   n
  pochhammer(---------,---)*pochhammer(---------,---)
                 3      3                   3      3
 0,

 0}
```

These results correspond to the formulas (compare [Koe94b])

$$ {}_3F_2\left(\left.\begin{matrix}-n\,,-n\,,-n\\ 1\,,1\end{matrix}\right|1\right)=\begin{cases}0 & \text{if } n \text{ odd}\\ \dfrac{(1/3)_{n/2}\,(2/3)_{n/2}}{(n/2)!^2}\,(-27)^{n/2} & \text{otherwise}\end{cases} $$

and

$$ {}_3F_2\left(\left.\begin{matrix}-n\,,n+3a\,,a\\ 3a/2\,,(3a+1)/2\end{matrix}\right|\frac{3}{4}\right) =\begin{cases}0 & \text{if } n\neq 0 \pmod 3\\ \dfrac{(1/3)_{n/3}\,(2/3)_{n/3}}{(a+1/3)_{n/3}\,(a+2/3)_{n/3}} & \text{otherwise}\end{cases} $$

20.71.10 REDUCE operator SUMTOHYPER

With the operator `sumtohyper`, sums given in factorial-Γ-binomial-Pochhammer notation are converted into hypergeometric notation.

`sumtohyper(f,k)` determines the hypergeometric representation of $\sum_{k=-\infty}^{\infty} f_k$, i.e. its output is `c*hypergeometric(upper,lower,x)`, corresponding to the representation

$$ \sum_{k=-\infty}^{\infty} f_k = c\cdot\ {}_pF_q\left(\left.\begin{matrix}a_1, & a_2, & \cdots, & a_p\\ b_1, & b_2, & \cdots, & b_q\end{matrix}\right|x\right)\ , $$

where `upper=` $\{a_1,a_2,\ldots,a_p\}$ and `lower=` $\{b_1,b_2,\ldots,b_q\}$ are the lists of upper and lower parameters.
Examples:

```
41: sumtohyper(binomial(n,k)^3,k);
```

```
hypergeometric({ - n, - n, - n},{1,1},-1)

42: sumtohyper(binomial(n,k)/2^n
               -sub(n=n-1,binomial(n,k)/2^n),k);

                  - n + 2           - n
  - hypergeom({----------, - n},{------},-1)
                    2                2
------------------------------------------
                       n
                      2
```

20.71.11 Simplification Operators

For the decision that an expression a_k is a hypergeometric term, it is necessary to find out whether or not a_k/a_{k-1} is a rational function with respect to k. For the purpose to decide whether or not an expression involving powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols is a hypergeometric term, the following simplification operators can be used:

- `simplify_gamma(f)` simplifies an expression `f` involving only rational, powers and Γ function terms according to a recursive application of the simplification rule $\Gamma\,(a+1) = a\,\Gamma\,(a)$ to the expression tree. Since all Γ arguments with integer difference are transformed, this gives a decision procedure for rationality for integer-linear Γ term product ratios.
- `simplify_combinatorial(f)` simplifies an expression `f` involving powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols by converting factorials, binomial coefficients, and Pochhammer symbols into Γ function terms, and applying `simplify_gamma` to its result. If the output is not rational, it is given in terms of Γ functions. If you prefer factorials you may use
- `gammatofactorial` (rule) converting Γ function terms into factorials using $\Gamma\,(x) \to (x-1)!$.
- `simplify_gamma2(f)` uses the duplication formula of the Γ function to simplify f.
- `simplify_gamman(f,n)` uses the multiplication formula of the Γ function to simplify f.

The use of `simplify_combinatorial(f)` is a safe way to decide the rationality for any ratio of products of powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols.

Example:

```
43: simplify_combinatorial(sub(k=k+1,krawtchoukterm)/
                           krawtchoukterm);

  (k - n)*(k - x)
--------------------
 (k - nn)*(k + 1)*p
```

From this calculation, we see again that the upper parameters of the hypergeometric representation of the Krawtchouk polynomials are given by $\{-n, -x\}$, its lower parameter is $\{-N\}$, and the argument of the hypergeometric function is $1/p$.

Other examples are

```
44: simplify_combinatorial(binomial(n,k)/binomial(2*n,k-1));

  gamma( - (k - 2*n - 2))*gamma(n + 1)
---------------------------------------
 gamma( - (k - n - 1))*gamma(2*n + 1)*k

45: ws where gammatofactorial;

 factorial( - k + 2*n + 1)*factorial(n)
----------------------------------------
  factorial( - k + n)*factorial(2*n)*k

46: simplify_gamma2(gamma(2*n)/gamma(n));

  2*n        2*n + 1
 2   *gamma(---------)
                2
-----------------------
      2*sqrt(pi)

47: simplify_gamman(gamma(3*n)/gamma(n),3);

  3*n        3*n + 2          3*n + 1
 3   *gamma(---------)*gamma(---------)
                3                3
----------------------------------------
              2*sqrt(3)*pi
```

20.71.12 Tracing

If you set

```
48: on zb_trace;
```

tracing is enabled, and you get intermediate results, see [Koe94b].

Example for the Gosper algorithm:

```
49: gosper(pochhammer(k-n,n),k);

                  k - 1
a(k)/a(k-1):= -----------
              k - n - 1

Gosper algorithm applicable

p:= 1

q:= k - 1

r:= k - n - 1

degreebound := 0

       1
f:= -------
     n + 1

Gosper algorithm successful

 pochhammer(k - n,n)*k
-----------------------
         n + 1
```

Example for the Zeilberger algorithm:

```
50: sumrecursion(binomial(n,k)^2,k,n);

                        2
                       n
F(n,k)/F(n-1,k):= ----------
                           2
                  (k - n)

                                2
                  (k - n - 1)
F(n,k)/F(n,k-1):= --------------
                         2
                        k

Zeilberger algorithm applicable
```

```
applying Zeilberger algorithm for order:= 1

                  2                                          2    2
p:= zb_sigma(1)*k  - 2*zb_sigma(1)*k*n + zb_sigma(1)*n  + n

     2                   2
q:= k  - 2*k*n - 2*k + n  + 2*n + 1

     2
r:= k

degreebound := 1

     2*k - 3*n + 2
f:= ---------------
           n

           2        2        2               3      2
      - 4*k *n + 2*k  + 8*k*n  - 4*k*n - 3*n  + 2*n
p:= ----------------------------------------------
                            n

Zeilberger algorithm successful

4*summ(n - 1)*n - 2*summ(n - 1) - summ(n)*n

51: off zb_trace;
```

20.71.13 Global Variables and Switches

The following global variables and switches can be used in connection with the ZEILBERG package:

- `zb_trace`, switch; default setting `off`. Turns tracing on and off.
- `zb_direction`, variable; settings: `down`, `up`; default setting `down`.

  In the case of the Gosper algorithm, either a downward or a forward antidifference is calculated, i.e., `gosper` finds g_k with either

  $$a_k = g_k - g_{k-1} \qquad \text{or} \qquad a_k = g_{k+1} - g_k,$$

  respectively.

  In the case of the Zeilberger algorithm, either a downward or an upward recurrence equation is returned. Example:

```
52: zb_direction:=up$

53: sumrecursion(binomial(n,k)^2,k,n);

summ(n + 1)*n + summ(n + 1) - 4*summ(n)*n - 2*summ(n)

54: zb_direction:=down$
```

- `zb_order`, variable; settings: any nonnegative integer; default setting `5`. Gives the maximal order for the recurrence equation that `sumrecursion` searches for.
- `zb_factor`, switch; default setting `on`. If `off`, the factorization of the output usually producing nicer results is suppressed.
- `zb_proof`, switch; default setting `off`. If `on`, then several intermediate results are stored in global variables:
- `gosper_representation`, variable; default setting `nil`.

  If a `gosper` command is issued, and if the Gosper algorithm is applicable, then the variable `gosper_representation` is set to the list of polynomials (with respect to k) `{p,q,r,f}` corresponding to the representation

$$\frac{a_k}{a_{k-1}} = \frac{p_k}{p_{k-1}}\,\frac{q_k}{r_k}\,, \qquad g_k = \frac{q_{k+1}}{p_k}\,f_k\,a_k\,,$$

  see [Gos78]. Examples:

```
55: on zb_proof;

56: gosper(k*factorial(k),k);

(k + 1)*factorial(k)

57: gosper_representation;

{k,k,1,1}

58: gosper(
       1/(k+1)*binomial(2*k,k)/(n-k+1)
         *binomial(2*n-2*k,n-k),k);

((2*k - n + 1)*(2*k + 1)

 *binomial( - 2*(k - n), - (k - n))

 *binomial(2*k,k))/((k + 1)*(n + 2)*(n + 1))
```

```
59: gosper_representation;

{1,

 (2*k - 1)*(k - n - 2),

 (2*k - 2*n - 1)*(k + 1),

   - (2*k - n + 1)
 ------------------}
  (n + 2)*(n + 1)
```

- `zeilberger_representation`, variable; default setting `nil`.

  If a `sumrecursion` command is issued, and if the Zeilberger algorithm is successful, then the variable `zeilberger_representation` is set to the final Gosper representation used, see [Koo93].

- `zb_f`, internal operator, do not use.

- `zb_sigma`, internal operator, do not use.

20.71.14 Messages

The following messages may occur:

- `*****`
  `Gosper algorithm: no closed form solution exists`

  Example input:

  `gosper(factorial(k),k).`

- `***** Gosper algorithm not applicable`

  Example input:

  `gosper(factorial(k/2),k).`

  The term ratio a_k/a_{k-1} is not rational.

- `***** illegal number of arguments`

  Example input:

  `gosper(k).`

- `***** Zeilberger algorithm fails. Enlarge zb_order`

  Example input:

  `sumrecursion(binomial(n,k)*binomial(6*k,n),k,n)`

  For this example a setting `zb_order:=6` is needed.

- `***** Zeilberger algorithm not applicable`

  Example input:

  `sumrecursion(binomial(n/2,k),k,n)`

  One of the term ratios $f(n,k)/f(n-1,k)$ or $f(n,k)/f(n,k-1)$ is not rational.

- `***** SOLVE given inconsistent equations`

  You can ignore this message that occurs with Version 3.5.

20.72 ZTRANS: Z-Transform Package

This package is an implementation of the Z-transform of a sequence. This is the discrete analogue of the Laplace Transform.

Authors: Wolfram Koepf and Lisa Temme

20.72.1 Z-Transform

The Z-Transform of a sequence $\{f_n\}$ is the discrete analogue of the Laplace Transform, and

$$\mathcal{Z}\{f_n\} = F(z) = \sum_{n=0}^{\infty} f_n z^{-n} \ .$$

This series converges in the region outside the circle $|z| = |z_0| = \limsup_{n\to\infty} \sqrt[n]{|f_n|}$.

SYNTAX: `ztrans`(f_n, n, z) where f_n is an expression, and n,z are identifiers.

20.72.2 Inverse Z-Transform

The calculation of the Laurent coefficients of a regular function results in the following inverse formula for the Z-Transform:
If $F(z)$ is a regular function in the region $|z| > \rho$ then $\exists$ a sequence $\{f_n\}$ with $\mathcal{Z}\{f_n\} = F(z)$ given by

$$f_n = \frac{1}{2\pi i} \oint F(z) z^{n-1} dz$$

SYNTAX: `invztrans(`$F(z)$`, z, n)` where $F(z)$ is an expression, and z,n are identifiers.

20.72.3 Input for the Z-Transform

This package can compute the Z-Transforms of the following list of functions f_n, and certain combinations thereof.

1	$e^{\alpha n}$	$\frac{1}{(n+k)}$
$\frac{1}{n!}$	$\frac{1}{(2n)!}$	$\frac{1}{(2n+1)!}$
$\frac{\sin(\beta n)}{n!}$	$\sin(\alpha n+\phi)$	$e^{\alpha n}\sin(\beta n)$
$\frac{\cos(\beta n)}{n!}$	$\cos(\alpha n+\phi)$	$e^{\alpha n}\cos(\beta n)$
$\frac{\sin(\beta(n+1))}{n+1}$	$\sinh(\alpha n+\phi)$	$\frac{\cos(\beta(n+1))}{n+1}$
$\cosh(\alpha n+\phi)$	$\binom{n+k}{m}$	

Other Combinations

Linearity	$\mathcal{Z}\{af_n+bg_n\}=a\mathcal{Z}\{f_n\}+b\mathcal{Z}\{g_n\}$
Multiplication by n	$\mathcal{Z}\{n^k\cdot f_n\}=-z\frac{d}{dz}\left(\mathcal{Z}\{n^{k-1}\cdot f_n,n,z\}\right)$
Multiplication by λ^n	$\mathcal{Z}\{\lambda^n\cdot f_n\}=F\left(\frac{z}{\lambda}\right)$
Shift Equation	$\mathcal{Z}\{f_{n+k}\}=z^k\left(F(z)-\sum_{j=0}^{k-1}f_jz^{-j}\right)$
Symbolic Sums	$\mathcal{Z}\left\{\sum_{k=0}^{n}f_k\right\}=\frac{z}{z-1}\cdot\mathcal{Z}\{f_n\}$
	$\mathcal{Z}\left\{\sum_{k=p}^{n+q}f_k\right\}$ combination of the above

where $k,\lambda\in\mathbf{N}\setminus\{0\}$; and a,b are variables or fractions; and $p,q\in\mathbf{Z}$ or are functions of n; and α,β and ϕ are angles in radians.

20.72.4 Input for the Inverse Z-Transform

This package can compute the Inverse Z-Transforms of any rational function, whose denominator can be factored over **Q**, in addition to the following list of $F(z)$.

$$\sin\left(\frac{\sin(\beta)}{z}\right)e^{\left(\frac{\cos(\beta)}{z}\right)} \qquad \cos\left(\frac{\sin(\beta)}{z}\right)e^{\left(\frac{\cos(\beta)}{z}\right)}$$

$$\sqrt{\frac{z}{A}}\sin\left(\sqrt{\frac{z}{A}}\right) \qquad \cos\left(\sqrt{\frac{z}{A}}\right)$$

$$\sqrt{\frac{z}{A}}\sinh\left(\sqrt{\frac{z}{A}}\right) \qquad \cosh\left(\sqrt{\frac{z}{A}}\right)$$

$$z\log\left(\frac{z}{\sqrt{z^2-Az+B}}\right) \qquad z\log\left(\frac{\sqrt{z^2+Az+B}}{z}\right)$$

$$\arctan\left(\frac{\sin(\beta)}{z+\cos(\beta)}\right)$$

where $k, \lambda \in \mathbf{N} \setminus \{0\}$ and A, B are fractions or variables ($B > 0$) and α, β, and ϕ are angles in radians.

20.72.5 Application of the Z-Transform

Solution of difference equations

In the same way that a Laplace Transform can be used to solve differential equations, so Z-Transforms can be used to solve difference equations.
Given a linear difference equation of k-th order

$$f_{n+k} + a_1 f_{n+k-1} + \ldots + a_k f_n = g_n \tag{20.103}$$

with initial conditions $f_0 = h_0$, $f_1 = h_1$, ..., $f_{k-1} = h_{k-1}$ (where h_j are given), it is possible to solve it in the following way. If the coefficients $a_1, \ldots, a_k$ are constants, then the Z-Transform of (20.103) can be calculated using the shift equation, and results in a solvable linear equation for $\mathcal{Z}\{f_n\}$. Application of the Inverse Z-Transform then results in the solution of (20.103).
If the coefficients $a_1, \ldots, a_k$ are polynomials in n then the Z-Transform of (20.103) constitutes a differential equation for $\mathcal{Z}\{f_n\}$. If this differential equation can be solved then the Inverse Z-Transform once again yields the solution of (20.103). Some examples of these methods of solution can be found in §20.72.6.

20.72.6 EXAMPLES

Here are some examples for the Z-Transform

```
1: ztrans((-1)^n*n^2,n,z);

     z*( - z + 1)
---------------------
  3       2
```

```
 z  + 3*z  + 3*z + 1

2: ztrans(cos(n*omega*t),n,z);

   z*(cos(omega*t) - z)
--------------------------
                      2
 2*cos(omega*t)*z - z  - 1

3: ztrans(cos(b*(n+2))/(n+2),n,z);

                                    z
z*( - cos(b) + log(------------------------------)*z)
                                       2
                    sqrt( - 2*cos(b)*z + z  + 1)

4: ztrans(n*cos(b*n)/factorial(n),n,z);

  cos(b)/z
(e

        sin(b)                   sin(b)
 *(cos(--------)*cos(b) - sin(--------)*sin(b)))/z
          z                        z

5: ztrans(sum(1/factorial(k),k,0,n),n,z);

  1/z
 e   *z
--------
 z - 1

6: operator f$

7: ztrans((1+n)^2*f(n),n,z);

                              2
df(ztrans(f(n),n,z),z,2)*z  - df(ztrans(f(n),n,z),z)*z
+ ztrans(f(n),n,z)
```

Here are some examples for the Inverse Z-Transform

```
8: invztrans((z^2-2*z)/(z^2-4*z+1),z,n);
```

```
                n       n                  n
 (sqrt(3) - 2) *( - 1)  + (sqrt(3) + 2)
-----------------------------------------
                    2

9: invztrans(z/((z-a)*(z-b)),z,n);

  n    n
 a  - b
---------
  a - b

10: invztrans(z/((z-a)*(z-b)*(z-c)),z,n);

  n      n      n      n      n      n
 a *b - a *c - b *a + b *c + c *a - c *b
-----------------------------------------
  2      2        2      2    2        2
 a *b - a *c - a*b  + a*c  + b *c - b*c

11: invztrans(z*log(z/(z-a)),z,n);

  n
 a *a
-------
 n + 1

12: invztrans(e^(1/(a*z)),z,n);

        1
-----------------
  n
 a *factorial(n)

13: invztrans(z*(z-cosh(a))/(z^2-2*z*cosh(a)+1),z,n);

cosh(a*n)
```

Examples: Solutions of Difference Equations

I (See [BS81], p. 651, Example 1).
Consider the homogeneous linear difference equation

$$f_{n+5} - 2f_{n+3} + 2f_{n+2} - 3f_{n+1} + 2f_n = 0$$

with initial conditions $f_0 = 0$, $f_1 = 0$, $f_2 = 9$, $f_3 = -2$, $f_4 = 23$. The Z-Transform of the left hand side can be written as $F(z) = P(z)/Q(z)$ where $P(z) = 9z^3 - 2z^2 + 5z$ and $Q(z) = z^5 - 2z^3 + 2z^2 - 3z + 2 = (z-1)^2(z+2)(z^2+1)$, which can be inverted to give

$$f_n = 2n + (-2)^n - \cos\frac{\pi}{2}n\ .$$

The following REDUCE session shows how the present package can be used to solve the above problem.

```
1: operator f;

2: f(0):=0$ f(1):=0$ f(2):=9$ f(3):=-2$ f(4):=23$


7: equation :=
   ztrans(f(n+5)-2*f(n+3)+2*f(n+2)-3*f(n+1)+2*f(n),n,z);

                                  5
equation := ztrans(f(n),n,z)*z

                                     3
            - 2*ztrans(f(n),n,z)*z

                                    2
            + 2*ztrans(f(n),n,z)*z

            - 3*ztrans(f(n),n,z)*z

                                       3      2
            + 2*ztrans(f(n),n,z) - 9*z  + 2*z

            - 5*z

8: ztransresult:=solve(equation,ztrans(f(n),n,z));

ztransresult :=

                              2
                       z*(9*z  - 2*z + 5)
{ztrans(f(n),n,z)=-----------------------------}
                    5      3      2
                   z  - 2*z  + 2*z  - 3*z + 2
```

```
9: result:=invztrans(part(first(ztransresult),2),z,n);

result :=

     n        n             n  n     n
  - i *( - 1)  + 2*( - 1) *2  - i  + 4*n
----------------------------------------
                    2
```

II (See [BS81], p. 651, Example 2).
Consider the inhomogeneous difference equation:

$$f_{n+2} - 4f_{n+1} + 3f_n = 1$$

with initial conditions $f_0 = 0$, $f_1 = 1$. Giving

$$\begin{aligned} F(z) &= \mathcal{Z}\{1\}\left(\frac{1}{z^2-4z+3} + \frac{z}{z^2-4z+3}\right) \\ &= \frac{z}{z-1}\left(\frac{1}{z^2-4z+3} + \frac{z}{z^2-4z+3}\right). \end{aligned}$$

The Inverse Z-Transform results in the solution

$$f_n = \frac{1}{2}\left(\frac{3^{n+1}-1}{2} - (n+1)\right).$$

The following REDUCE session shows how the present package can be used to solve the above problem.

```
10: clear(f)$ operator f$ f(0):=0$ f(1):=1$


14: equation:=ztrans(f(n+2)-4*f(n+1)+3*f(n)-1,n,z);

                                    3
equation := (ztrans(f(n),n,z)*z

                                       2
              - 5*ztrans(f(n),n,z)*z

              + 7*ztrans(f(n),n,z)*z

                                     2
              - 3*ztrans(f(n),n,z) - z )/(z - 1)
```

```
15: ztransresult:=solve(equation,ztrans(f(n),n,z));

ztransresult :=

                                2
                               z
{ztrans(f(n),n,z)=---------------------}
                     3       2
                    z  - 5*z  + 7*z - 3

16: result:=invztrans(part(first(ztransresult),2),z,n);

               n
           3*3  - 2*n - 3
result := ----------------
                  4
```

III Consider the following difference equation, which has a differential equation for $\mathcal{Z}\{f_n\}$.

$$(n+1)\cdot f_{n+1} - f_n = 0$$

with initial conditions $f_0 = 1$, $f_1 = 1$. It can be solved in REDUCE using the present package in the following way.

```
17: clear(f)$ operator f$ f(0):=1$ f(1):=1$


21: equation:=ztrans((n+1)*f(n+1)-f(n),n,z);

equation :=

                                  2
 - (df(ztrans(f(n),n,z),z)*z  + ztrans(f(n),n,z))

22: operator tmp;

23: equation:=sub(ztrans(f(n),n,z)=tmp(z),equation);

                                   2
equation :=  - (df(tmp(z),z)*z  + tmp(z))

24: load_package(odesolve);

25: ztransresult:=odesolve(equation,tmp(z),z);
```

```
                          1/z
ztransresult := {tmp(z)=e   *arbconst(1)}

39: preresult :=
      invztrans(part(first(ztransresult),2),z,n);

              arbconst(1)
preresult := --------------
              factorial(n)

40: solve({sub(n=0,preresult)=f(0),
           sub(n=1,preresult)=f(1)},
          arbconst(1));

{arbconst(1)=1}

41: result:=preresult where ws;

                1
result := --------------
           factorial(n)
```

Chapter 21

Symbolic Mode

At the system level, REDUCE is based on a version of the programming language Lisp known as *Standard Lisp* which is described in [MHGG80]. We shall assume in this section that the reader is familiar with the material in that paper. This also assumes implicitly that the reader has a reasonable knowledge about Lisp in general, say at the level of the LISP 1.5 Programmer's Manual ([MAE+62]) or any of the books mentioned at the end of this section. Persons unfamiliar with this material will have some difficulty understanding this section.

Although REDUCE is designed primarily for algebraic calculations, its source language is general enough to allow for a full range of Lisp-like symbolic calculations. To achieve this generality, however, it is necessary to provide the user with two modes of evaluation, namely an algebraic mode and a symbolic mode. To enter symbolic mode, the user types `symbolic;` (or `lisp;`) and to return to algebraic mode one types `algebraic;`. Evaluations proceed differently in each mode so the user is advised to check what mode he is in if a puzzling error arises. He can find his mode by typing

```
eval_mode;
```

The current mode will then be printed as `algebraic` or `symbolic`.

Expression evaluation may proceed in either mode at any level of a calculation, provided the results are passed from mode to mode in a compatible manner. One simply prefixes the relevant expression by the appropriate mode. If the mode name prefixes an expression at the top level, it will then be handled as if the global system mode had been changed for the scope of that particular calculation.

For example, if the current mode is `algebraic`, then the commands

```
symbolic car '(a);
x+y;
```

will cause the first expression to be evaluated and printed in symbolic mode and the second in algebraic mode. Only the second evaluation will thus affect the expression workspace. On the other hand, the statement

```
x + symbolic car '(12);
```

will result in the algebraic value `X+12`.

The use of `symbolic` (and equivalently `algebraic`) in this manner is the same as any operator. That means that parentheses could be omitted in the above examples since the meaning is obvious. In other cases, parentheses must be used, as in

```
symbolic(x := 'a);
```

Omitting the parentheses, as in

```
symbolic x := a;
```

would be wrong, since it would parse as

```
symbolic(x) := a;
```

For convenience, it is assumed that any operator whose *first* argument is quoted is being evaluated in symbolic mode, regardless of the mode in effect at that time. Thus, the first example above could be equally well written:

```
car '(a);
```

Except where explicit limitations have been made, most REDUCE algebraic constructions carry over into symbolic mode. However, there are some differences. First, expression evaluation now becomes Lisp evaluation. Secondly, assignment statements are handled differently, as we shall discuss shortly. Thirdly, local variables and array elements are initialized to `nil` rather than `0`. (In fact, any variables not explicitly declared `integer` are also initialized to `nil` in algebraic mode, but the algebraic evaluator recognizes `nil` as `0`.) Finally, function definitions follow the conventions of Standard Lisp.

To begin with, we mention a few extensions to our basic syntax which are designed primarily if not exclusively for symbolic mode.

21.1 Symbolic Infix Operators

There are three binary infix operators in REDUCE intended for use in symbolic mode, namely `.` (`cons`), `eq` and `memq`. The precedence of these operators was given in another section.

21.2 Symbolic Expressions

These consist of scalar variables and operators and follow the normal rules of the Lisp meta language.

Examples:

```
x
car u . reverse v
simp (u+v^2)
```

21.3 Quoted Expressions

Because symbolic evaluation requires that each variable or expression has a value, it is necessary to add to REDUCE the concept of a quoted expression by analogy with the Lisp `quote` function. This is provided by the single quote mark `'`. For example,

`'a` represents the Lisp S-expression `(quote a)`
`'(a b c)` represents the Lisp S-expression `(quote (a b c))`

Note, however, that strings are constants and therefore evaluate to themselves in symbolic mode. Thus, to print the string `"A String"`, one would write

```
prin2 "A String";
```

Within a quoted expression, identifier syntax rules are those of REDUCE. Thus `(a !. b)` is the list consisting of the three elements `a`, `.`, and `b`, whereas `(a . b)` is the dotted pair of `a` and `b`.

21.4 Lambda Expressions

`lambda` expressions provide the means for constructing Lisp `lambda` expressions in symbolic mode. They may not be used in algebraic mode.

Syntax:

⟨`lambda` *expression*⟩ ⟶ `lambda` ⟨*varlist*⟩⟨*terminator*⟩ ⟨*statement*⟩

where

⟨*varlist*⟩ ⟶ `(`⟨*variable*⟩`, ... ,` ⟨*variable*⟩`)`

e.g.,

```
lambda (x,y); car x . cdr y;
```

is equivalent to the Lisp `lambda` expression

```
(lambda (x y) (cons (car x) (cdr y)))
```

The parentheses may be omitted in specifying the variable list if desired.

`lambda` expressions may be used in symbolic mode in place of prefix operators, or as an argument of the reserved word `function`.

In those cases where a `lambda` expression is used to introduce local variables to avoid recomputation, a `where` statement can also be used. For example, the expression

```
(lambda (x,y); list(car x,cdr x,car y,cdr y))
    (reverse u,reverse v)
```

can also be written

```
{car x,cdr x,car y,cdr y}
    where x=reverse u,y=reverse v
```

Where possible, `where` syntax is preferred to `lambda` syntax, since it is more natural.

21.5 Symbolic Assignment Statements

In symbolic mode, if the left side of an assignment statement is a variable, a `setq` of the right-hand side to that variable occurs. If the left-hand side is an expression, it must be of the form of an array element, otherwise an error will result. For example, `x:=y` translates into `(setq x y)` whereas `a(3) := 3` will be valid if `a` has been previously declared a single dimensioned array of at least four elements.

21.6 FOR EACH Statement

The `for each` form of the `for` statement, designed for iteration down a list, is more general in symbolic mode. Its syntax is:

`for each` ⟨*id:identifier*⟩ (`in` | `on`) ⟨*lst:list*⟩ (`do` | `collect` | `join` | `product` | `sum`)⟨*exprn:S-expr*⟩

As in algebraic mode, if the keyword `in` is used, iteration is on each element of the list. With `on`, iteration is on the whole list remaining at each point in the iteration. As a result, we have the following equivalence between each form of `for each` and the various mapping functions in Lisp:

	do	collect	join
in	mapc	mapcar	mapcan
on	map	maplist	mapcon

Example: To list each element of the list `(a b c)`:

```
for each x in '(a b c) collect list x;
```

21.7 Symbolic Procedures

All the functions described in the Standard Lisp Report are available to users in symbolic mode. Additional functions may also be defined as symbolic procedures. For example, to define the Lisp function `assoc`, the following could be used:

```
symbolic procedure assoc(u,v);
   if null v then nil
    else if u = caar v then car v
    else assoc(u, cdr v);
```

If the default mode were symbolic, then `symbolic` could be omitted in the above definition. `macro`s may be defined by prefixing the keyword `procedure` by the word `macro`. (In fact, ordinary functions may be defined with the keyword `expr` prefixing `procedure` as was used in the Standard Lisp Report.) For example, we could define a `macro conscons` by

```
symbolic macro procedure conscons l;
   expand(cdr l,'cons);
```

Another form of macro, the `smacro` is also available. These are described in the Standard Lisp Report. The Report also defines a function type `fexpr`. However,

its use is discouraged since it is hard to implement efficiently, and most uses can be replaced by macros. At the present time, there are no `fexprs` in the core REDUCE system.

21.8 Standard Lisp Equivalent of REDUCE Input

A user can obtain the Standard Lisp equivalent of his REDUCE input by turning on the switch `defn` (for definition). The system then prints the Lisp translation of his input but does not evaluate it. Normal operation is resumed when `defn` is turned off.

21.9 Communicating with Algebraic Mode

One of the principal motivations for a user of the algebraic facilities of REDUCE to learn about symbolic mode is that it gives one access to a wider range of techniques than is possible in algebraic mode alone. For example, if a user wishes to use parts of the system defined in the basic system source code, or refine their algebraic code definitions to make them more efficient, then it is necessary to understand the source language in fairly complete detail. Moreover, it is also necessary to know a little more about the way REDUCE operates internally. Basically, REDUCE considers expressions in two forms: prefix form, which follow the normal Lisp rules of function composition, and so-called canonical form, which uses a completely different syntax.

Once these details are understood, the most critical problem faced by a user is how to make expressions and procedures communicate between symbolic and algebraic mode. The purpose of this section is to teach a user the basic principles for this.

If one wants to evaluate an expression in algebraic mode, and then use that expression in symbolic mode calculations, or vice versa, the easiest way to do this is to assign a variable to that expression whose value is easily obtainable in both modes. To facilitate this, a declaration `share` is available. `share` takes a list of identifiers as argument, and marks these variables as having recognizable values in both modes. The declaration may be used in either mode.

E.g.,

```
share x,y;
```

says that `x` and `y` will receive values to be used in both modes.

If a `share` declaration is made for a variable with a previously assigned algebraic value, that value is also made available in symbolic mode.

21.9.1 Passing Algebraic Mode Values to Symbolic Mode

If one wishes to work with parts of an algebraic mode expression in symbolic mode, one simply makes an assignment of a shared variable to the relevant expression in algebraic mode. For example, if one wishes to work with `(a+b)^2`, one would say, in algebraic mode:

```
x := (a+b)^2;
```

assuming that `x` was declared shared as above. If we now change to symbolic mode and say

```
x;
```

its value will be printed as a prefix form with the syntax:

`(*sq` ⟨*standard quotient*⟩ `t)`

This particular format reflects the fact that the algebraic mode processor currently likes to transfer prefix forms from command to command, but doesn't like to reconvert standard forms (which represent polynomials) and standard quotients back to a true Lisp prefix form for the expression (which would result in excessive computation). So `*sq` is used to tell the algebraic processor that it is dealing with a prefix form which is really a standard quotient and the second argument (`t` or `nil`) tells it whether it needs further processing (essentially, an *already simplified* flag).

So to get the true standard quotient form in symbolic mode, one needs `cadr` of the variable. E.g.,

```
z := cadr x;
```

would store in `Z` the standard quotient form for `(a+b)^2`.

Once you have this expression, you can now manipulate it as you wish. To facilitate this, a standard set of selectors and constructors are available for getting at parts of the form. Those presently defined are as follows:

REDUCE Selectors

`denr`	denominator of standard quotient
`lc`	leading coefficient of polynomial
`ldeg`	leading degree of polynomial
`lpow`	leading power of polynomial
`lt`	leading term of polynomial
`mvar`	main variable of polynomial
`numr`	numerator (of standard quotient)
`pdeg`	degree of a power
`red`	reductum of polynomial
`tc`	coefficient of a term
`tdeg`	degree of a term
`tpow`	power of a term

REDUCE Constructors

- `.+` add a term to a polynomial
- `./` divide (two polynomials to get quotient)
- `.*` multiply power by coefficient to produce term
- `.^` raise a variable to a power

For example, to find the numerator of the standard quotient above, one could say:

```
numr z;
```

or to find the leading term of the numerator:

```
lt numr z;
```

Conversion between various data structures is facilitated by the use of a set of functions defined for this purpose. Those currently implemented include:

- `!*a2f` convert an algebraic expression to a standard form. If result is rational, an error results;
- `!*a2k` converts an algebraic expression to a kernel. If this is not possible, an error results;
- `!*f2a` converts a standard form to an algebraic expression;
- `!*f2q` convert a standard form to a standard quotient;
- `!*k2f` convert a kernel to a standard form;
- `!*k2q` convert a kernel to a standard quotient;
- `!*kk2f` convert a non-unique kernel to a standard form;
- `!*kk2q` convert a non-unique kernel to a standard quotient;
- `!*p2f` convert a standard power to a standard form;
- `!*n2f` convert a number to a standard form;
- `!*p2q` convert a standard power to a standard quotient;
- `!*q2f` convert a standard quotient to a standard form. If the quotient denominator is not 1, an error results;
- `!*q2k` convert a standard quotient to a kernel. If this is not possible, an error results;
- `!*t2f` convert a standard term to a standard form
- `!*t2q` convert a standard term to a standard quotient.

21.9.2 Passing Symbolic Mode Values to Algebraic Mode

In order to pass the value of a shared variable from symbolic mode to algebraic mode, the only thing to do is make sure that the value in symbolic mode is a prefix expression. E.g., one uses `(expt (plus a b) 2)` for `(a+b)^2`, or the format (`*sq` ⟨*standard quotient*⟩ `t`) as described above. However, if you have been working with parts of a standard form they will probably not be in this form. In that case, you can do the following:

1. If it is a standard quotient, call `prepsq` on it. This takes a standard quotient as argument, and returns a prefix expression. Alternatively, you can call `mk!*sq` on it, which returns a prefix form like (`*sq` ⟨*standard quotient*⟩ `t`) and avoids translation of the expression into a true prefix form.

2. If it is a standard form, call `prepf` on it. This takes a standard form as argument, and returns the equivalent prefix expression. Alternatively, you can convert it to a standard quotient and then call `mk!*sq`.

3. If it is a part of a standard form, you must usually first build up a standard form out of it, and then go to step 2. The conversion functions described earlier may be used for this purpose. For example,

   (a) If `z` is an expression which is a term, `!*t2f z` is a standard form.

   (b) If `z` is a standard power, `!*p2f z` is a standard form.

   (c) If `z` is a variable, you can pass it direct to algebraic mode.

For example, to pass the leading term of `(a+b)^2` back to algebraic mode, one could say:

```
y:= mk!*sq !*t2q lt numr z;
```

where `y` has been declared shared as above. If you now go back to algebraic mode, you can work with `y` in the usual way.

21.9.3 Complete Example

The following is the complete code for doing the above steps. The end result will be that the square of the leading term of $(a+b)^2$ is calculated.

```
share x,y;                      % declare X and Y
                                % as shared
x := (a+b)^2;                   % store (a+b)^2 in X
symbolic;                       % transfer to symbolic mode
```

```
z := cadr x;                    % store a true standard
                                % quotient in Z
lt numr z;                      % print the leading term
                                % of the numerator of Z
y := mk!*sq !*t2q lt numr z;    % store the prefix form of
                                % this leading term in Y
algebraic;                      % return to algebraic mode
y^2;                            % evaluate square of the
                                % leading term of (a+b)^2
```

21.9.4 Defining Procedures for Intermode Communication

If one wishes to define a procedure in symbolic mode for use as an operator in algebraic mode, it is necessary to declare this fact to the system by using the declaration `operator` in symbolic mode. Thus

```
symbolic operator leadterm;
```

would declare the procedure `leadterm` as an algebraic operator. This declaration *must* be made in symbolic mode as the effect in algebraic mode is different. The value of such a procedure must be a prefix form.

The algebraic processor will pass arguments to such procedures in prefix form. Therefore if you want to work with the arguments as standard quotients you must first convert them to that form by using the function `SIMP!*`. This function takes a prefix form as argument and returns the evaluated standard quotient.

For example, if you want to define a procedure `leadterm` which gives the leading term of an algebraic expression, one could do this as follows:

```
% Declare leadterm as a symbolic mode procedure to
% be used in algebraic mode.
symbolic operator leadterm;

% Define leadterm.
symbolic procedure leadterm u;
   mk!*sq !*t2q lt numr simp!* u;
```

Note that this operator has a different effect than the operator `lterm`. In the latter case, the calculation is done with respect to the second argument of the operator. In the example here, we simply extract the leading term with respect to the system's choice of main variable.

Finally, if you wish to use the algebraic evaluator on an argument in a symbolic mode definition, the function `reval` can be used. The one argument of `reval`

must be the prefix form of an expression. `reval` returns the evaluated expression as a true Lisp prefix form.

21.10 Rlisp '88

Rlisp '88 is a superset of the Rlisp that has been traditionally used for the support of REDUCE. It is fully documented in the book [Mar93]. Rlisp '88 adds to the traditional Rlisp the following facilities:

1. more general versions of the looping constructs `for`, `repeat` and `while`;
2. support for a backquote construct;
3. support for active comments;
4. support for vectors of the form name[index];
5. support for simple structures;
6. support for records.

In addition, "–" is a letter in Rlisp '88. In other words, `A-B` is an identifier, not the difference of the identifiers `A` and `B`. If the latter construct is required, it is necessary to put spaces around the - character. For compatibility between the two versions of Rlisp, we recommend this convention be used in all symbolic mode programs.

To use Rlisp '88, type `on rlisp88;`. This switches to symbolic mode with the Rlisp '88 syntax and extensions. While in this environment, it is impossible to switch to algebraic mode, or prefix expressions by "algebraic". However, symbolic mode programs written in Rlisp '88 may be run in algebraic mode provided the rlisp88 package has been loaded. We also expect that many of the extensions defined in Rlisp '88 will migrate to the basic Rlisp over time. To return to traditional Rlisp or to switch to algebraic mode, say "`off rlisp88;`".

21.11 References

There are a number of useful books which can give you further information about LISP. Here is a selection: [All78, MAE+62, Tou84, WH81].

Chapter 22

Calculations in High Energy Physics

A set of REDUCE commands is provided for users interested in symbolic calculations in high energy physics. Several extensions to our basic syntax are necessary, however, to allow for the different data structures encountered.

22.1 High Energy Physics Operators

We begin by introducing three new operators required in these calculations.

22.1.1 . (Cons) Operator

Syntax:

```
(exprn1:vector_expression)
          . (exprn2:vector_expression):algebraic.
```

The binary `.` operator, which is normally used to denote the addition of an element to the front of a list, can also be used in algebraic mode to denote the scalar product of two Lorentz four-vectors. For this to happen, the second argument must be recognizable as a vector expression at the time of evaluation. With this meaning, this operator is often referred to as the *dot* operator. In the present system, the index handling routines all assume that Lorentz four-vectors are used, but these routines could be rewritten to handle other cases.

Components of vectors can be represented by including representations of unit vectors in the system. Thus if `eo` represents the unit vector `(1,0,0,0)`, `(p.eo)` represents the zeroth component of the four-vector P. Our metric and notation fol-

lows Bjorken and Drell [JDB65]. Similarly, an arbitrary component `p` may be represented by `(p.u)`. If contraction over components of vectors is required, then the declaration `index` must be used. Thus

```
index u;
```

declares `u` as an index, and the simplification of

```
p.u * q.u
```

would result in

```
P.Q
```

The metric tensor $g^{\mu\nu}$ may be represented by `(u.v)`. If contraction over `u` and `v` is required, then they should be declared as indices.

Errors occur if indices are not properly matched in expressions.

If a user later wishes to remove the index property from specific vectors, he can do it with the declaration `remind`. Thus `remind v1,...,vn;` removes the index flags from the variables `V1` through `Vn`. However, these variables remain vectors in the system.

22.1.2 G Operator for Gamma Matrices

Syntax:

```
g(id:identifier[,exprn:vector_expression])
        :gamma_matrix_expression.
```

`g` is an n-ary operator used to denote a product of γ matrices contracted with Lorentz four-vectors. Gamma matrices are associated with fermion lines in a Feynman diagram. If more than one such line occurs, then a different set of γ matrices (operating in independent spin spaces) is required to represent each line. To facilitate this, the first argument of `g` is a line identification identifier (not a number) used to distinguish different lines.

Thus

```
g(l1,p) * g(l2,q)
```

denotes the product of $\gamma \cdot p$ associated with a fermion line identified as `l1`, and $\gamma \cdot q$ associated with another line identified as `l2` and where p and q are Lorentz four-vectors. A product of γ matrices associated with the same line may be written

in a contracted form.

Thus

```
g(l1,p1,p2,...,p3) = g(l1,p1)*g(l1,p2)*...*g(l1,p3) .
```

The vector `a` is reserved in arguments of `g` to denote the special γ matrix γ^5. Thus

`g(l,a)`	$=\gamma^5$	associated with the line `l`
`g(l,p,a)`	$=\gamma\cdot p\times\gamma^5$	associated with the line `l`.

γ^μ (associated with the line `l`) may be written as `g(l,u)`, with `u` flagged as an index if contraction over `u` is required.

The notation of Bjorken and Drell is assumed in all operations involving γ matrices.

22.1.3 EPS Operator

Syntax:

```
eps(exprn1:vector_expression,...,exprn4:vector_exp)
   :vector_exp.
```

The operator `eps` has four arguments, and is used only to denote the completely antisymmetric tensor of order 4 and its contraction with Lorentz four-vectors. Thus

$$\epsilon_{ijkl}=\begin{cases}+1 & \text{if } i,j,k,l \text{ is an even permutation of 0,1,2,3}\\ -1 & \text{if } i,j,k,l \text{ is an odd permutation of 0,1,2,3}\\ 0 & \text{otherwise}\end{cases}$$

A contraction of the form $\epsilon_{ij\mu\nu}p_\mu q_\nu$ may be written as `eps(i,j,p,q)`, with `i` and `j` flagged as indices, and so on.

22.2 Vector Variables

Apart from the line identification identifier in the `g` operator, all other arguments of the operators in this section are vectors. Variables used as such must be declared so by the type declaration `vector`, for example:

```
vector  p1,p2;
```

declares `p1` and `p2` to be vectors. Variables declared as indices or given a mass are automatically declared vector by these declarations.

22.3 Additional Expression Types

Two additional expression types are necessary for high energy calculations, namely

22.3.1 Vector Expressions

These follow the normal rules of vector combination. Thus the product of a scalar or numerical expression and a vector expression is a vector, as are the sum and difference of vector expressions. If these rules are not followed, error messages are printed. Furthermore, if the system finds an undeclared variable where it expects a vector variable, it will ask the user in interactive mode whether to make that variable a vector or not. In batch mode, the declaration will be made automatically and the user informed of this by a message.

Examples:

Assuming `p` and `q` have been declared vectors, the following are vector expressions

```
p
2*q/3
2*x*y*p - p.q*q/(3*q.q)
```

whereas `p*q` and `p/q` are not.

22.3.2 Dirac Expressions

These denote those expressions which involve γ matrices. A γ matrix is implicitly a 4×4 matrix, and so the product, sum and difference of such expressions, or the product of a scalar and Dirac expression is again a Dirac expression. There are no Dirac variables in the system, so whenever a scalar variable appears in a Dirac expression without an associated γ matrix expression, an implicit unit 4 by 4 matrix is assumed. For example, `g(l,p) + m` denotes `g(l,p) + m*`⟨*unit 4 by 4 matrix*⟩. Multiplication of Dirac expressions, as for matrix expressions, is of course non-commutative.

22.4 Trace Calculations

When a Dirac expression is evaluated, the system computes one quarter of the trace of each γ matrix product in the expansion of the expression. One quarter of each trace is taken in order to avoid confusion between the trace of the scalar `m`, say, and `m` representing `m *` ⟨*unit 4 by 4 matrix*⟩. Contraction over indices

occurring in such expressions is also performed. If an unmatched index is found in such an expression, an error occurs.

The algorithms used for trace calculations are the best available at the time this system was produced. For example, in addition to the algorithm developed by Chisholm for contracting indices in products of traces, REDUCE uses the elegant algorithm of Kahane for contracting indices in γ matrix products. These algorithms are described in [Chi63] and [Kah68].

It is possible to prevent the trace calculation over any line identifier by the declaration `nospur`. For example,

```
nospur l1,l2;
```

will mean that no traces are taken of γ matrix terms involving the line numbers `l1` and `l2`. However, in some calculations involving more than one line, a catastrophic error

```
NOSPUR on more than one line not implemented
```

can occur (for the reason stated!) If you encounter this error, please let us know!

A trace of a γ matrix expression involving a line identifier which has been declared `nospur` may be later taken by making the declaration `spur`.

See also the CVIT package for an alternative mechanism (section D.2).

22.5 Mass Declarations

It is often necessary to put a particle "on the mass shell" in a calculation. This can, of course, be accomplished with a `let` command such as

```
let p.p= m^2;
```

but an alternative method is provided by two commands `mass` and `mshell`. `mass` takes a list of equations of the form:

⟨*vector variable*⟩ = ⟨*scalar variable*⟩

for example,

```
mass p1=m, q1=mu;
```

The only effect of this command is to associate the relevant scalar variable as a mass with the corresponding vector. If we now say

`mshell` ⟨*vector variable*⟩`, ..., `⟨*vector variable*⟩⟨*terminator*⟩

and a mass has been associated with these arguments, a substitution of the form

⟨*vector variable*⟩ `.` ⟨*vector variable*⟩ `=` ⟨*mass*⟩`^2`

is set up. An error results if the variable has no preassigned mass.

22.6 Example

We give here as an example of a simple calculation in high energy physics the computation of the Compton scattering cross-section as given in Bjorken and Drell Eqs. (7.72) through (7.74). We wish to compute the trace of

$$\frac{\alpha^2}{2}\left(\frac{k'}{k}\right)^2\left(\frac{\gamma\cdot p_f+m}{2m}\right)\left(\frac{\gamma\cdot e'\gamma\cdot e\gamma\cdot k_i}{2k.p_i}+\frac{\gamma\cdot e\gamma\cdot e'\gamma\cdot k_f}{2k'\cdot p_i}\right)$$
$$\left(\frac{\gamma\cdot p_i+m}{2m}\right)\left(\frac{\gamma\cdot k_i\gamma\cdot e\gamma\cdot e'}{2k.p_i}+\frac{\gamma\cdot k_f\gamma\cdot e'\gamma\cdot e}{2k'\cdot p_i}\right)$$

where k_i and k_f are the four-momenta of incoming and outgoing photons (with polarization vectors e and e' and laboratory energies k and k' respectively) and p_i, p_f are incident and final electron four-momenta.

Omitting therefore an overall factor $\frac{\alpha^2}{2m^2}\left(\frac{k'}{k}\right)^2$ we need to find one quarter of the trace of

$$(\gamma\cdot p_f+m)\left(\frac{\gamma\cdot e'\gamma\cdot e\gamma\cdot k_i}{2k.p_i}+\frac{\gamma\cdot e\gamma\cdot e'\gamma\cdot k_f}{2k'.p_i}\right)\times$$
$$(\gamma\cdot p_i+m)\left(\frac{\gamma\cdot k_i\gamma\cdot e\gamma\cdot e'}{2k.p_i}+\frac{\gamma\cdot k_f\gamma\cdot e'\gamma\cdot e}{2k'.p_i}\right)$$

A straightforward REDUCE program for this, with appropriate substitutions (using `p1` for p_i, `pf` for p_f, `ki` for k_i and `kf` for k_f) is

```
on div; % this gives output in same form
        % as Bjorken and Drell.
mass ki= 0, kf= 0, p1= m, pf= m; vector e,ep;
% if e is used as a vector, it loses its scalar
%      identity as the base of natural logarithms.
```

```
mshell ki,kf,p1,pf;
let p1.e= 0, p1.ep= 0, p1.pf= m^2+ki.kf, p1.ki= m*k,
    p1.kf= m*kp, pf.e= -kf.e, pf.ep= ki.ep,
    pf.ki= m*kp, pf.kf= m*k, ki.e= 0, ki.kf= m*(k-kp),
    kf.ep= 0, e.e= -1, ep.ep=-1;
operator gp;
for all p let gp(p)= g(l,p)+m;
comment this is just to save us a lot of writing;
gp(pf)*(g(l,ep,e,ki)/(2*ki.p1) + g(l,e,ep,kf)/
    (2*kf.p1))
  * gp(p1)*(g(l,ki,e,ep)/(2*ki.p1) + g(l,kf,ep,e)/
    (2*kf.p1))$
write "The Compton cxn is ",ws;
```

(We use `p1` instead of `pi` in the above to avoid confusion with the reserved variable `pi`).

This program will print the following result

```
                          2    1       -1    1    -1
The Compton cxn is 2*e.ep  + ---*k*kp   + ---*k  *kp - 1
                             2            2
```

22.7 Extensions to More Than Four Dimensions

In our discussion so far, we have assumed that we are working in the normal four dimensions of QED calculations. However, in most cases, the programs will also work in an arbitrary number of dimensions. The command

`vecdim` ⟨*expression*⟩⟨*terminator*⟩

sets the appropriate dimension. The dimension can be symbolic as well as numerical. Users should note however, that the `eps` operator and the γ_5 symbol (`a`) are not properly defined in other than four dimensions and will lead to an error if used.

22.8 The CVIT algorithm

An alternative algorithm for computing traces of products of gamma matrices is available, based on treating of gamma-matrices as 3-j symbols (details may be found in [IKRT89, Ken82]).

This alternative algorithm is used when the switch `cvit` is set to on. With `cvit`

off, calculations of Diracs matrices traces are performed using standard REDUCE facilities.

For more information see section D.2.

Chapter 23

REDUCE and Rlisp Utilities

REDUCE and its associated support language system Rlisp include a number of utilities which have proved useful for program development over the years. The following are supported in most of the implementations of REDUCE currently available.

23.1 The Standard Lisp Compiler

Many versions of REDUCE include a Standard Lisp compiler that is automatically loaded on demand. You should check your system specific user guide to make sure you have such a compiler. To make the compiler active, the switch `comp` should be turned on. Any further definitions input after this will be compiled automatically. If the compiler used is a derivative version of the original Griss-Hearn compiler ([GH79]), there are other switches that might also be used in this regard. However, these additional switches are not supported in all compilers. They are as follows:

`plap` If on, causes the printing of the portable macros produced by the compiler;

`pgwd` If on, causes the printing of the actual assembly language instructions generated from the macros;

`pwrds` If on, causes a statistic message of the form
⟨*function*⟩ `COMPILED,` ⟨*words*⟩ `WORDS,` ⟨*words*⟩ `LEFT`
to be printed. The first number is the number of words of binary program space the compiled function took, and the second number the number of words left unused in binary program space.

23.2 Fast Loading Code Generation Program

In most versions of REDUCE, it is possible to take any set of Lisp, Rlisp or REDUCE commands and build a fast loading version of them. In Rlisp or REDUCE, one does the following:

```
faslout <filename>;
<commands or IN statements>
faslend;
```

To load such a file, one uses the command `load`, e.g. `load foo;` or `load foo,bah;`

This process produces a fast-loading version of the original file. In some implementations, this means another file is created with the same name but a different extension. For example, in PSL-based systems, the extension is `b` (for binary). In CSL-based systems, however, this process adds the fast-loading code to a single file in which all such code is stored. Particular functions are provided by CSL for managing this file, and described in the CSL user documentation.

In doing this build, as with the production of a Standard Lisp form of such statements, it is important to remember that some of the commands must be instantiated during the building process. For example, macros must be expanded, and some property list operations must happen. The REDUCE sources should be consulted for further details on this.

To avoid excessive printout, input statements should be followed by a `$` instead of the semicolon. With `load` however, the input doesn't print out regardless of which terminator is used with the command.

If you subsequently change the source files used in producing a fast loading file, don't forget to repeat the above process in order to update the fast loading file correspondingly. Remember also that the text which is read in during the creation of the fast load file, in the compiling process described above, is *not* stored in your REDUCE environment, but only translated and output. If you want to use the file just created, you must then use `load` to load the output of the fast-loading file generation program.

When the file to be loaded contains a complete package for a given application, `load_package` rather than `load` should be used. The syntax is the same. However, `load_package` does some additional bookkeeping such as recording that this package has now been loaded, that is required for the correct operation of the system.

23.3 The Standard Lisp Cross Reference Program

`cref` is a Standard Lisp program for processing a set of Standard LISP function definitions to produce:

1. A "summary" showing:
   (a) A list of files processed;
   (b) A list of "entry points" (functions which are not called or are only called by themselves);
   (c) A list of undefined functions (functions called but not defined in this set of functions);
   (d) A list of variables that were used non-locally but not declared `global` or `fluid` before their use;
   (e) A list of variables that were declared `global` but not used as `fluid`s, i.e., bound in a function;
   (f) A list of `fluid` variables that were not bound in a function so that one might consider declaring them `global`s;
   (g) A list of all `global` variables present;
   (h) A list of all `fluid` variables present;
   (i) A list of all functions present.
2. A "global variable usage" table, showing for each non-local variable:
   (a) Functions in which it is used as a declared `fluid` or `global`;
   (b) Functions in which it is used but not declared;
   (c) Functions in which it is bound;
   (d) Functions in which it is changed by `setq`.
3. A "function usage" table showing for each function:
   (a) Where it is defined;
   (b) Functions which call this function;
   (c) Functions called by it;
   (d) Non-local variables used.

The program will also check that functions are called with the correct number of arguments, and print a diagnostic message otherwise.

The output is alphabetized on the first seven characters of each function name.

23.3.1 Restrictions

Algebraic procedures in REDUCE are treated as if they were symbolic, so that algebraic constructs will actually appear as calls to symbolic functions, such as `aeval`.

23.3.2 Usage

To invoke the cross reference program, the switch `cref` is used. `on cref` causes the cref program to load and the cross-referencing process to begin. After all the required definitions are loaded, `off cref` will cause the cross-reference listing to be produced. For example, if you wish to cross-reference all functions in the file `tst.red`, and produce the cross-reference listing in the file `tst.crf`, the following sequence can be used:

```
out "tst.crf";
on cref;
in "tst.red"$
off cref;
shut "tst.crf";
```

To process more than one file, more `in` statements may be added before the call of `off cref`, or the `in` statement changed to include a list of files.

23.3.3 Options

Functions with the flag `nolist` will not be examined or output. Initially, all Standard Lisp functions are so flagged. (In fact, they are kept on a list `nolist!*`, so if you wish to see references to *all* functions, then `cref` should be first loaded with the command `load cref`, and this variable then set to `nil`).

It should also be remembered that any macros with the property list flag `expand`, or, if the switch `force` is on, without the property list flag `noexpand`, will be expanded before the definition is seen by the cross-reference program, so this flag can also be used to select those macros you require expanded and those you do not.

23.4 Prettyprinting REDUCE Expressions

REDUCE includes a module for printing REDUCE syntax in a standard format. This module is activated by the switch `pret`, which is normally off.

Since the system converts algebraic input into an equivalent symbolic form, the

printing program tries to interpret this as an algebraic expression before printing it. In most cases, this can be done successfully. However, there will be occasional instances where results are printed in symbolic mode form that bears little resemblance to the original input, even though it is formally equivalent.

If you want to prettyprint a whole file, say `off output,msg;` and (hopefully) only clean output will result. Unlike `defn`, input is also evaluated with `pret` on.

23.5 Prettyprinting Standard Lisp S-Expressions

REDUCE includes a module for printing S-expressions in a standard format. The Standard Lisp function for this purpose is `prettyprint` which takes a Lisp expression and prints the formatted equivalent.

Users can also have their REDUCE input printed in this form by use of the switch `defn`. This is in fact a convenient way to convert REDUCE (or Rlisp) syntax into Lisp. `off msg;` will prevent warning messages from being printed.

NOTE: When `defn` is on, input is not evaluated.

Chapter 24

Maintaining REDUCE

Since January 1, 2009 REDUCE is Open Source Software. It is hosted at

https://sourceforge.net/projects/reduce-algebra/

We mention here three ways in which REDUCE is maintained. The first is the collection of queries, observations and bug-reports. All users are encouraged to subscribe to the mailing list that SourceForge provides so that they will receive information about updates and concerns. Also on SourceForge there is a bug tracker and a discussion forum. The expectation is that the maintainers and keen users of REDUCE will monitor those and try to respond to issues. However these resources are not there to seek answers to Maths homework problems – they are intended specifically for issues to do with the use and support of REDUCE.

The second level of support is provided by the fact that all the sources of REDUCE are available, so any user who is having difficulty either with a bug or understanding system behaviour can consult the code to see if (for instance) comments in it clarify something that was unclear from the regular documentation.

The source files for REDUCE are available on SourceForge in the Subversion repository, which provides the command for using a Subversion client to fetch the most up to date copy of everything. From time to time there may be one-file archives of a snapshot of the sources placed in the download area (Files tab) on SourceForge, and eventually some of these may be marked as "stable" releases, but at present it is recommended that developers use a copy from the Subversion repository.

The files fetched there come with a directory called "trunk" that holds the main current REDUCE, and one called "branches" that is reserved for future experimental versions. All the files that we have for creating help files and manuals should also be present in the files you fetch.

The packages that make up the source for the algebraic capabilities of REDUCE are in the "packages" sub-directory, and often there are test files for a package present there and especially for contributed packages there will be documentation in the form of a LaTeX file. Although REDUCE is coded in its own language many people in the past have found that it does not take too long to start to get used to it.

In various cases even fairly "ordinary end users" may wish to fetch the source version of REDUCE and compile it all for themselves. This may either be because they need the benefit of a bug-fix only recently checked into the Subversion repository or because no pre-compiled binary is available for the particular computer and operating system they use. This latter is to some extent unavoidable since REDUCE can run on both 32 and 64-bit Windows, the various MacOSX options (e.g. Intel and Powerpc), many different distributions of Linux, some BSD variants and Solaris (at least). It is not practically feasible for us to provide a constant stream of up to date ready-built binaries for all these.

There are instructions for compiling REDUCE present at the top of the trunk source tree. Usually the hardest issue seems to be ensuring that your computer has an adequate set of development tools and libraries installled before you start, but once that is sorted out the hope is that the compilation of REDUCE should proceed uneventfully if sometimes tediously.

In a typical Open Source way the hope is that some of those who build REDUCE from source or explore the source (out of general interest or to pursue an understanding of some bug or detail) will transform themselves into contributors or developers, which moves on to the third level of support.

At this third level any user can contribute proposals for bug fixes or extensions to REDUCE or its documentation. It might be valuable to collect a library of additional user-contributed examples illustrating the use of the system too. To do this first ensure that you have a fully up to date copy of the sources from Subversion, and then depending on just what sort of change is being proposed provide the updates to the developers via the SourceForge bug tracker or other route. In time we may give more concrete guidance about the format of changes that will be easiest to handle. It is obviously important that proposed changes have been properly tested and that they are accompanied with a clear explanation of why they are of benefit. A specific concern here is that in the past fixes to a bug in one part of REDUCE have had bad effects on some other applications and packages, so some degree of caution is called for. Anybody who develops a significant whole new package for REDUCE is encouraged to make the developers aware so that it can be considered for inclusion.

So the short form explanation about Support and Maintenance is that it is mainly focussed around the SourceForge system. If discussions about bugs, requirements or issues are conducted there then all users and potential users of REDUCE will be able to benefit from reviewing them, and the Sourceforge mailing lists, tracker,

forums and wiki will grow to be both a static repository of answers to common questions, an active set of locations to get new issues looked at and a focus for guiding future development.

Appendix A

Reserved Identifiers

We list here all identifiers that are normally reserved in REDUCE including names of commands, operators and switches initially in the system. Excluded are words that are reserved in specific implementations of the system.

A.1 Commands

```
<<...>> algebraic anticom antisymmetric array bye clear
clear_dummy_base clear_dummy_names clearrules comment
coframe complex_conjugates cont decompose defid define
define_spaces defindex defpoly depend dfp_commute
display displayframe dummy_base dummy_name ed editdef
end even factor fdomain for forall forder foreach frame
generic_function getcsystem global_sign go goto if in
in_tex index indexrange index_symmetries infix input
integer keep killing_vector korder let linear lisp
listargp load load_package make_variables mass match
mathstyle matrix mshell nodepend noncom nonzero nospur
nosum notrealvalued noxpnd odd off on operator order out
pause pform plotreset plotshow precedence print_indexed
print_noindexed print_precision procedure putcsystem
putgrass quit real realvalued rem_dummy_indices
rem_spaces rem_tensor remanticom remember remfac
remforder remgrass remind remindex remnoncom renosum
remsym remvector resetreduce retry return riemannconx
rtr rtrout rtrst saveas scalar selfconjugate setmod
setring share show_dummy_names showtime shut signature
spacedim sparse spur symbolic symmetric symtree tensor
trrl trrlid tvector unitmat unrtr unrtrst unset untrrl
```

untrrlid vec vecdim vector weight write wtlevel vstart xorder xpnd xvars

A.2 Boolean Operators

abaglistp baglistp bagp checkproplist evenp fixp freeof grassp matrixp numberp oddp ordp primep realvaluedp setp sparsematp squarep symmetricp taylorseriesp

A.3 Infix Operators

:= :- ::- = == >= >< > <= < => + - -> -> * / // ^ ** *** . .+ .* .: ./ .= .. # \ _= _| |_ with where setq or and cons cross difference divide dot eq equal expt geq greaterp intersect intersection leq lessp member memq minus mod neq plus poly_quotient quotient recip set_eq setdiff subset subset_eq times tpmat union vmod xmod xmodideal

A.4 Numerical Operators

abs acos acosd acosh acot acotd acoth acsc acscd acsch AGM_function Airy_Ai Airy_Aiprime Airy_Bi Airy_Biprime arg argd arccd arccn arccs arcdc arcdn arcds arcnd arcnc arcns arcsc arcsd arcsn asec asecd asech asin asind asinh atan atand atanh atan2 atan2d Bernoulli BesselI BesselJ BesselK BesselY Beta ceiling cos cosd cosh cot cotd coth csc cscd csch deg2rad deg2dms dms2rad dms2deg EllipticE EllipticE!' EllipticF EllipticK EllipticK!' ellipticctheta1 ellipticctheta2 ellipticctheta3 ellipticctheta4 Euler exp factorial fix floor Gamma Hankel1 Hankel2 hypot ibeta igamma jacobiam JacobiE jacobicn jacobidn jacobisn JacobiZeta KummerM KummerU legendre_symbol Lerch_Phi log logb log10 Lommel1 Lommel2 m_gamma nextprime norm Pochhammer Polygamma psi rad2deg rad2dms round sec secd sech weierstrass_sigma weierstrass_sigma0 weierstrass_sigma1 weierstrass_sigma2 weierstrass_sigma3 sin sind sinh sqrt StruveH StruveL tan tand tanh weierstrass

weierstrass1 weierstrassZeta weierstrassZeta1
WhittakerM WhittakerU Zeta

A.5 Prefix Operators

@ add_columns add_rows add_to_columns add_to_rows
adjoint_cdiffop affine_monomial_curve affine_points
alatomp alg_to_symb algnlist algsort alkernp
allsymmetrybases analytic_spread annihilator append
appendn arbcomplex arbint arbrat arglength
array_to_list asfirst asflist aslast asrest assgrad
assist assisthelp asslist augment_columns
availablegroups avec baglmat band_matrix belast
BernoulliP Bernstein_base bettiNumbers bibasis
bibasis_print_statistics Binomial bounds block_matrix
blowup canonical canonicaldecomposition cde cde_grading
cf cf_continuents cf_convergent cf_convergents cf_euler
cf_expression cf_remove_constant cf_remove_fractions
cf_transform cf_unit_denominators cf_unit_numerators
cfrac change_termorder change_termorder1 char_matrix
char_poly character charactern charactertable
chebyshev_df chebyshev_eval chebyshev_fit chebyshev_int
ChebyshevT ChebyshevU Chebyshev_base_T Chebyshev_base_U
cholesky Ci clearbag clearcaliprintterms clearflag
clearfunctions clearop clearprop Clebsch_Gordan codim
coeff coeff_matrix coeff2 coeffn coercemat cofactor
column_dim combinations combnum comm companion conj
CONTFRAC continued_fraction conv_cdiff2superfun
conv_superfun2cdiff coordinates copy_into cresys
crossvect Csetrepresentation curl cyclicpermlist
dd_groebner defint deflineint deg degree
degsfromresolution delete delete_all deleteunits
dellastdigit delpair delsq den depatom depth depvarp
der_deg_ordering det detidnum df df_odd dfp diagonal
diagonalize diff diffset dilog dim dimzerop directsum
displayflag displayprop div divpol dlineint dotgrad
dummy_indices dpgcd dsolve dvint dvolint easydim
easyindepset easyprimarydecomposition Ei eliminate
ell_function elmult eps eqhull Erf eta1 eta2 eta3
euler_df EulerP eval2 evalb evalproc excoeffs exdegree
expand_cases expand_td explicit ext extend
extended_gosper extended_sumrecursion

extendedgroebfactor extendedgroebfactor1 extractlist
extremum exvars factorize Fibonacci FibonacciP
find_companion first firstroot followline fourier_cos
fourier_sin FPS frequency frobenius funcvar g gbasis gcd
gcdnl GDIMENSION GegenbauerP Gegenbauer_base generators
get_columns get_rows getdegrees getecart getelmat
getring getkbase getleadterms getroot getrules gfnewt
gfroot ghostfactor gindependent_sets glexconvert
gnuplot gosper grad GradedBettinumbers gram_schmidt
grassparity greduce greduce_orders groebfactor groebner
groebnert groebnerf groebner_walk groepostproc
groesolve gsort gsplit gspoly gvars gzerodim!?
hankel_transform hconcmat hermat HermiteP Hermite_base
hermitian_tp hessian hilbert hilbertpolynomial
HilbertSeries homstbasis hypergeometric hyperrecursion
hypersum hyperterm hypexpand hypreduce I_setting
i_solve ideal_of_minors ideal_of_pfaffians ideal2list
ideal2mat gb intersection idealpower idealprod
idealquotient idealsum (CALI) idealsum (IDEALS) impart
implicit implicit_taylor indepvarsets ineq_solve infsum
initialize_equations initmat insert insert_keep_order
int integrate_equation interpol interreduce intersect
invbase inverse_taylor invlap invlex invtorder
invztrans irreduciblerepnr irreduciblereptable
isolatedprimes isprime iszeroradical jacobian JacobiP
jet_dim jet_fiber_dim jordan jordan_block
jordansymbolic K_transform kernlist korderlist
kronecker_product LaguerreP Laguerre_base LAPLACE
laplace_transform last lattice_delta lattice_e1
lattice_e2 lattice_e3 lattice_g lattice_g2 lattice_g3
lattice_generators lattice_invariants lattice_roots
lazystbasis leadterm LegendreP Legendre_base lcm lcof
left_factor left_factors length lhs lieclass
liendimcom1 lineint lineint linelength list
list_to_array list_to_ids listbag listgroebfactor
loadgroups log_sum lowestdeg lpdofac lpdofacx
lpdofactorize lpdofactorizex lpdogp lpdogdp lpdoord
lpdoptl lpdos lpdoset lpdosym lpdosym2dp lpdoweyl
lpower lterm lu_decom m_solve m_roots mainvar
make_identity map make_partic_tens mat mat2list
matappend mateigen matextc matextr mathomogenize
matintersect matjac matqquot matquot matrix_augment
matrix_stack matstabquot matsubr matsubc matsum max
MeijerG merge_list min minimal_generators minor minors

minvect mk_cdiffop mk_ids_belong_space
mk_ids_belong_anyspace mk_superfun mkalllinodd
mkdepth_one mkgam mkid mkidm mkidnew mklist mkrandtabl
mkset mkvarlist1 monom modequalp modulequotient
monomial_base Motzkin mpvect mult_columns mult_rows
multi_coeff nc_cleanup nc_compact nc_divide
nc_factorize nc_factorize_ALL nc_groebner nc_preduce
nc_setup nearestroot nm noether noexpand_td nome
nome2!K nome2!K!' nome2mod nome2mod!' nordp normalform
nullspace num num_fit num_int num_min num_odesolve
num_solve num_to_perm nzdp odesolve one_of ov_limit
pade pair part partial periodic periodic2rational
perm_to_num permutations pf pfaffian pivot plot
plus_or_minus poleorder position precision precp
preduce preducet preimage prgen primarydecomposition
printgroup proc prod proj_monomial_curve proj_points
print_conditions prsys PS pschangevar pscompose pscopy
psdepvar pseudo_divide pseudo_inverse pseudo_quotient
pseudo_remainder psexplim psexpansionpt psfunction
psordlim psorder psreverse pssum pstaylor psterm
pstruncate put_equations_used putbag putflag putprop
pvar_df quasi_period_factors r_solve radical rand
random random_linear_form random_matrix random_new_seed
randomlist randpoly rank ratint rational2periodic
ratjordan ratpreimage realroots rederr redexpr reduct
remainder remove remove_columns remove_rows reimpart
repart repfirst represt residue resolve rest restaslist
result resultant reverse rhs left_factor left_factors
rlrootno root_of root_of_unity root_val rootacc
rootprec roots roots_at_prec root_val row_dim
rows_pivot Rsetrepresentation saturation save_cde_state
savemat scalefactors scalvect schouten_bracket second
select selectvars sequences set setavailable
setcaliprintterms setcalitrace setdegrees setelements
setelmat setgbasis setgenerators setgrouptable setideal
setmodule setring setrules show show_epsilons
show_spaces showproc showrules Si sieve sign SimpleDE
simplex simplify_gamma simplify_gamma2 simplify_gamman
simplify_combinatorial simpsys singular_locus
SixJSymbol smithex smithex_int solve SolidHarmonicY
sortlist sortnumlist spadd_columns spadd_rows
spadd_to_COLUMNS spadd_to_ROWS spaugment_columns
spband_matrix spblock_matrix spchar_matrix spchar_poly
spcholesky spcoeff_matrix spcol_dim spcompanion

spcopy_into spdiagonal spextend spfind_companion
spget_columns spget_rows spgram_schmidt
SphericalHarmonicY sphermitian_tp sphessian spjacobian
spjordan_block split_field split splitext_list
splitext_opequ splitplusminus splitterms
splitvars_opequ splu_decom spmake_identity
spmatrix_augment spmatrix_stack spminor spmult_columns
spmult_rows sppivot sppseudo_inverse spremove_columns
spremove_rows sprow_dim sprows_pivot spstack_rows
spsub_matrix spsvd spswap_columns spswap_entries
spswap_rows stack_rows Stirling1 Stirling2 storegroup
structr struveh_transform sub sub_matrix submat
submodulep substitute sum summ sumvect sumtohyper
sumrecursion super_vectorfield suppress svd svec
swap_columns swap_entries swap_rows switches switchorg
sym symb_to_alg symdiff symmetrize symmetrybasis
symmetrybasispart symbolic_power syzygies tangentcone
taylor taylorcoefflist taylorcombine taylororiginal
taylorrevert taylortemplate taylortostandard testbool
theta1d theta2d theta3d theta4d third ThreeJSymbol
toeplitz torder torder_compile totaldeg tp trace
triang_adjoint trigexpand trigfactorize triggcd trigint
trigonometric_base trigreduce trigsimp union
vandermonde vardf varname varopt vconcmat vdf vint
volint volintegral vtaylor WeightedHilbertSeries
wholespace_dim wu xauto xmod xmodideal xideal
Y_transform zeroprimarydecomposition zeroprimes
zeroradical zerosolve ztrans

A.6 Reserved Variables

!__FILE__ !__LINE__ all_graded_der all_parametric_der
all_parametric_odd all_principal_der all_principal_odd
assumptions cali!=basering cali!=degrees cali!=monset
card_no catalan coords dep_var e euler_gamma eval_mode
fort_width fps_search_depth gltb glterms gmodule
golden_ratio gosper_representation gorders groebmonfac
greduce_result groebprotfile groebresmax
groebrestriction gvarslast hfactors high_pow i
indep_var infinity invtempbasis jacobian k!* khinchin
laline!* lie_list lie_class liemat lientrans low_pow mm
nc_factor_time negative nil nn no_glaisher odd_var

pclass pi positive principal_der rates
repprincparam_der repprincparam_odd requirements
root_multiplicities rootacc!# rootsreal rootscomplex
set_distribution_rule species t taylorprintterms to_cn
to_dn to_sn total_order volintorder wholespace zb_f
zb_direction zb_order zb_sigma
zeilberger_representation

A.7 Switches

adjprec algint allbranch allfac allowdfint anticom
arbvars balanced_mod bcsimp bezout bfspace cf_taylor
checkord combineexpt combinelogs commutedf comp complex
CONTRACT cramer cref CVIT defn demo detectunits dfint
dfprint dispjacobian DISTRIBUTE div echo errcont
evallhseqp exdelt exp expanddf expandlogs ezgcd factor
factorprimes factorunits failhard fastsimplex fort
fortupper fullroots gcd gltbasis glterms
groebfullreduction groebopt groebprot groebstat
hardzerotest heugcd horner ifactor imaginary int intstr
latex latex lexefgb lcm lhyp list listargs lmon
lower_matrix ltrig mcd modular msg multiplicities
multiroot nat nero nocommutedf noconvert Noetherian
nointsubst nolnr nosplit nosum not_negative
odesolve_basis odesolve_check odesolve_expand
odesolve_explicit odesolve_fast odesolve_full
odesolve_implicit odesolve_noint odesolve_verbose
onespace only_integer output overview period plotkeep
precise precise_complex pret pri psprintorder rat
ratarg rational rationalize ratpri ratroot red_total
revpri rlisp88 rootmsg roundall roundbf rounded
savestructr semantic simpnoncomdf solvesingular
symmetric taylorautocombine taylorautoexpand
taylorkeeporiginal taylorprintorder time tr_lie tra
tracefps traceratint tracetrig trcompact trdefint trfac
trgroeb trgroeb1 trgroebr trgroebs trigform trint trint
trode trplot trpm sym!-assoc trroot trsolve trsum
trxideal trxmod trwu upper_matrix varopt latex
xfullreduce zb_factor zb_proof zb_trace

A.8 Other Reserved Ids

```
bag begin do then expr fexpr function input lambda lisp
listproc macro matrixproc product repeat smacro sum then
until when while ws
```

Appendix B

Bibliography

[AG98] Beatrice Amrhein and Oliver Gloor. The fractal walk. In Bruno Buchberger and Franz Winkler, editors, *Gröbner Bases and Applications*, volume 251 of *London Mathematical Society Lecture Note Series*, pages 305–322. Cambridge University Press, Feb 1998.

[AGK96a] Beatrice Amrhein, Oliver Gloor, and Wolfgang Kuechlin. How fast does the walk run? In *5th Rhine Workshop on Computer Algebra, volume PR 801/96, pages 8.1–8.9. Institut Franco–Allemand de Recherches de Saint–Louis*, Jan 1996.

[AGK96b] Beatrice Amrhein, Oliver Gloor, and Wolfgang Kuechlin. Walking faster. In J. Calmet and C. Limongelli, editors, *DISCO 1996: Design and Implementation of Symbolic Computation Systems*, volume 1128 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 1996.

[All78] J. R. Allen. *The Anatomy of LISP*. McGraw-Hill, New York, 1978.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[AM90] V. S. Adamchik and O. I. Marichev. The algorithm for calculating integrals of hypergeometric type functions and its realization in REDUCE system. In S. Watanabe and Morio Nagata, editors, *Proceedings of the 1990 International Symposium on Symbolic and Algebraic Computation*, pages 212–224. ACM, Addison-Wesley, 1990.

[Ape92] Joachim Apel. *A relationship between Gröbner bases of ideals and vector modules of G-algebras*, volume 131.2 of *Contemporary Mathematics*, pages 195–204. AMS, 1992.

[AS72] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, 1972.

[ASW89] Werner Antweiler, Andreas Strotmann, and Volker Winkelmann. A TEX-REDUCE-interface. *SIGSAM Bulletin*, 23(2):26–33, February 1989.

[ASY74] E A Arais, V P Shapeev, and N N Yanenko. Computer realization of cartan's exterior calculus,. *Soviet Math Dokl*, 15:203–205, 1974.

[Bar67a] D. Barton. A scheme for manipulative algebra on a computer. *Computer Journal*, 9(4):340–344, Feb 1967.

[Bar67b] D. Barton. A scheme for manipulative algebra on a computer. *Astronomical Journal*, 72:1281–1287, 1967.

[BB89] A. V. Bocharov and M. L. Bronstein. *Efficiently Implementing Two Methods of the Geometrical Theory of Differential Equations: An Experience in Algorithm and Software Design*, pages 143–166. Number 16 in Acta Applicandae Mathematicae. Kluwer, 1989.

[BC82] Gregory Butler and John J. Cannon. Computing in permutation and matrix groups. I. Normal closure, commutator subgroups, series. *Math. Comp.*, 39:663–670, 1982.

[BC94] A. Burnel and H. Caprasse. Computing the BRST operator used in quantization of gauge theories. *International Journal of Modern Physics C*, 5(6):1035–1047, December 1994.

[BCD+99] A. V. Bocharov, V. N. Chetverikov, S. V. Duzhin, N. G. Khor'kova, A. V. Samokhin, Yu. N. Torkhov, and A. M. Verbovetsky:. *Symmetries and Conservation Laws for Differential Equations of Mathematical Physics*, volume 182 of *Translations of Math. Monographs*. AMS, 1999.

[BCG+91] Robert L Byrant, S S Chern, Robert B Gardner, Hubert L Goldschmidt, and P A Griffiths. *Computer realization of Cartan's exterior calculus*, volume 18 of *Mathematical Sciences Research Institute Publications*. Springer-Verlag New York, 1991.

[BCRT93] Anna Maria Bigatti, Pasqualina Conti, Lorenzo Robbiano, and Carlo Traverso. A "Divide and conquer" algorithm for Hilbert-Poincaré series, multiplicity and dimension of monomial ideals. In G. Cohen, T. Mora, and O. Moreno, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes. AAECC 1993*, volume 673 of *Lecture Notes in Computer Science*, pages 76–88, Berlin, Heidelberg, 1993. Springer-Verlag.

[BF72] D. Barton and J. P. Fitch. The application of symbolic algebra system to physics. *Reports on Progress in Physics*, 35(1):235–314, Jan 1972.

[BGDW95] P. A. Broadbery, T. Gómez-Díaz, and S. M. Watt. On the Implementation of Dynamic Evaluation. In A. Levelt, editor, *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, ISSAC '95, pages 77–84, New York, NY, USA, 1995. ACM Press.

[BGK86] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Groebner bases. *Journal of Symbolic Computation*, 2(1):83–98, March 1986.

[BGM96] George A. Baker and Peter Graves-Morris. *Padé Approximants*, volume 13 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2nd edition, 1996.

[BH10] D. E. Baldwin and W. Hereman. A symbolic algorithm for computing recursion operators of nonlinear partial differential equations. *International Journal of Computer Mathematics*, 87(5):1094–1119, 2010.

[BHPS86] R. J. Bradford, A. C. Hearn, J. A. Padget, and E. Schrüfer. Enlarging the reduce domain of computation. In *SYMSAC '86: Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*, pages 100–106, New York, NY, USA, 1986. ACM.

[BL85] Gregory Butler and Clement W. H. Lam. A General Backtrack Algorithm for the Isomorphism Problem of Combinatorial Objects. *Journal of Symbolic Computation*, 1(4):363–381, 1985.

[BO78] Carl M. Bender and Steven A. Orszag. *Advanced Mathematical Methods for Scientists and Engineers*. McGraw-Hill, 1978.

[Bou72] S. R. Bourne. Literal expressions for the co-ordinates of the moon. I. The First Degree Terms. *Celestial Mechanics*, 6(2):167–186, Sep 1972.

[Bra93] Russell Bradford. Algebraic simplification of multiple-valued functions. In John Fitch, editor, *DISCO 1992: Design and Implementation of Symbolic Computation Systems*, volume 721 of *Lecture Notes in Computer Science*, pages 13–21, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Bro96] E. W. Brown. *An Introductory Treatise on the Lunar Theory*. Cambridge University Press, 1896.

[Bro97] Manuel Bronstein. *Symbolic Integration I: Transcendental Functions*. Springer-Verlag, Heidelberg, 1997.

[BS81] Ilja N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Thun und Frankfurt (Main), 1981.

[BS92] Dave Bayer and Mark Stillman. Computation of Hilbert functions. *Journal of Symbolic Computation*, 14(1):31–50, 1992.

[Buc85] Bruno Buchberger. *Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory*, pages 184–232. Reidel, Dordrecht, 01 1985.

[Buc88] Bruno Buchberger. Applications of Gröbner bases in non-linear computational geometry. In R. Janssen, editor, *Mathematical Aspects of Scientific Software*, volume 296 of *Lecture Notes in Computer Science*, pages 52–88. Springer, New York, 1988.

[But82] Gregory Butler. Computing in permutation and matrix groups ii: Backtrack algorithm. *Mathematics of Computation*, 39(160):671–680, 1982.

[BW92] A. Brand and T. Wolf. The computer algebra package CRACK for investigating PDEs. In *Proc. of ERCIM, Partial Differential Equations and Group Theory, Bonn*, page 24, 1992.

[BW95] A. Brand and T. Wolf. Investigating DEs with CRACK and related programs. *SIGSAM Bulletin*, Special Issue:1–8, June 1995.

[BWK93] Thomas Becker, Volker Weispfenning, and Heinz Kredel. *Gröbner bases, a computational approach to commutative algebra*. Springer - Verlag (Graduate Texts in Mathematics 141), 1993.

[Cap97] H. Caprasse. Brst charge and poisson algebras. *Discrete Mathematics and Theoretical Computer Science. DMTCS [electronic only]*, 1(2):115–127, 1997.

[Car88] B.C. Carlson. A table of elliptic integrals of the third kind. 51 (183), pp. 267–280, s1–s5. *Mathematics of Computation*, 51 (183):267–280, 1988.

[Car99] B.C. Carlson. Toward symbolic integration of elliptic integrals. *J. Symbolic Computation*, 28(6):739–753, 1999.

[CAT] Cathode (computer algebra tools for handling ordinary differential equations). OBSOLETE: http://www-lmc.imag.fr/CATHODE/, http://www-lmc.imag.fr/CATHODE2/.

[CF00] B.C. Carlson and J. FitzSimmons. Reduction theorems for elliptic integrands with the square root of two quadratic factors. *J. Computational Applied Mathematics*, 118(1-2):71–85, 2000.

[CGG+91] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual*. Springer, 1991.

[Chi63] J. S. R. Chisholm. Relativistic scalar products of γ matrices. *Il Nuovo Cimento (1955-1965)*, 30(1):426–428, Oct 1963.

[CHW91] B. Champagne, W. Hereman, and P. Winternitz. The computer calculation of Lie point symmetries of large systems of differential equations. *Computer Physics Communications*, 66(2–3):319–340, 1991.

[CJ92] Robert M. Corless and David J. Jeffrey. Well ... It Isn't Quite That Simple. *SIGSAM Bull.*, 26(3):2–6, Aug 1992.

[CKM97] S. Collart, M. Kalkbrenner, and D. Mall. Converting Bases with the Gröbner Walk. *Journal of Symbolic Computation*, 24(3-4):465–469, Sep 1997.

[CLO92] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms: An Introduction of Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag, 1992.

[CLV20] M. Casati, P. Lorenzoni, and R. Vitolo. Three computational approaches to weakly nonlocal poisson brackets. *Studies in Applied Mathematics*, 2020.

[CN81] B.C. Carlson and E.M. Notis. Algorithm 577: Algorithm for incomplete elliptic intergrals [s21]. *ACM Transactions Mathematical Software*, 7(3):398–403, 1981.

[Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In H. Brakhage, editor, *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[Cvi76] Predrag Cvitanović. Group theory for Feynman diagrams in non-Abelian gauge theories. *Phys. Rev. D*, 14:1536, Sep 1976.

[Dan63] George Bernard Dantzig. Linear programming and extensions. Technical report, RAND Corporation, 1963.

[Dav81] James Harold Davenport. *On the Integration of Algebraic Functions*, volume 102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[Del86] C. Delaunay. *Théorie du Mouvement de la Lune*. Extraits des Mém. Acad. Sci. Mallet-Bachelier, Paris, 186.

[DF94] James Davenport and Christèle Faure. The "unknown" in computer algebra. *Programming and Computer Software*, 20:1–5, 01 1994.

[DGV96] D. Duval and Laureano González-Vega. Dynamic evaluation and real closure. *Mathematics and Computers in Simulation*, 42:551–560, 11 1996.

[DLS90] Ladislav Drska, Richard Liska, and Milan Sinor. Two practical packages for computational physics-GCPM, RLFI. *Computer Physics Communications*, 61(1-2):225–230, November 1990.

[DN83] B. A. Dubrovin and S. P. Novikov. Hamiltonian formalism of one-dimensional systems of hydrodynamic type and the bogolyubov-whitham averaging method. *Dokl. Akad. Nauk SSSR*, 27(3):665–669, 1983. Translated by J. R. Schulenberger.

[DN84] B. A. Dubrovin and S. P. Novikov. Poisson brackets of hydrodynamic type. *Dokl. Akad. Nauk SSSR*, 30(3):651–654, Jan 1984.

[DP85] James Davenport and Julian Padget. HEUGCD: how elementary upperbounds generate cheaper data. In *Proc. EUROCAL 1985, Lecture Notes in Computer Science*, volume 204, pages 18–28. Springer-Verlag, 1985.

[DR94a] Dominique Duval and Jean-Claude Reynaud. Sketches and computation I: basic definitions and static evaluation. *Mathematical Structures in Computer Science*, 4(2):185–238, 1994.

[DR94b] Dominique Duval and Jean-Claude Reynaud. Sketches and computation II: dynamic evaluation and applications. *Mathematical Structures in Computer Science*, 4(2):239–271, 1994.

[DS96] Andreas Dolzmann and Thomas Sturm. Redlog user manual. Technical Report MIP-9616, FMI, Universität Passau, D-94030 Passau, Germany, October 1996. Edition 1.0 for Version 1.0.

[DS97a] Andreas Dolzmann and Thomas Sturm. REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

[DS97b] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, August 1997.

[DS99] Andreas Dolzmann and Thomas Sturm. *Redlog User Manual*. FMI, Universität Passau, D-94030 Passau, Germany, April 1999. Edition 2.0 for Version 2.0.

[DST93] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and Algorithms for Algebraic Computation*. Academic Press, 2nd edition, 1993.

[Dub96] Boris Dubrovin. Geometry of 2D topological field theories. In Mauro Francaviglia and Silvio Greco, editors, *Integrable Systems and Quantum Groups: Lectures given at the 1st Session of the Centro Internazionale Matematico Estivo (C.I.M.E.) held in Montecatini Terme, Italy, June 14–22, 1993*, volume 1620 of *Lecture Notes in Mathematics*, pages 120–348. Springer, Berlin Heidelberg, 1996.

[Eas87] James W. Eastwood. Orthovec: A REDUCE program for 3-D vector analysis in orthogonal curvilinear coordinates. *Computer Physics Communications*, 47(1):139–147, October 1987.

[Eas91] James W. Eastwood. ORTHOVEC: version 2 of the REDUCE program for 3-D vector analysis in orthogonal curvilinear coordinates. *Computer Physics Communications*, 64(1):121–122, April 1991.

[Edm57] A. R. Edmonds. *Angular Momentum in Quantum Mechanics*. Princeton University Press, 1957.

[Eis95] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*, volume 150 of *Graduate Texts in Math.* Springer-Verlag, Berlin and New York, 1995.

[Eul48] L. Euler. *Introductio in analysin infinitorum, Vol. 1*, chapter 18. Lausanne: Marcum-Michaelem Bousquet, 1748.

[Fat74] Richard J. Fateman. On the multiplication of poisson series. *Celestial Mechanics*, 10(2):243–247, Oct 1974.

[Fat87] Richard J. Fateman. TEX output from macsyma-like systems. *ACM SIGSAM Bulletin*, 21(4):1–5, 1987.

[FGLM93] J. C. Faugère, P. Gianni, D. Lazard, and T .Mora. Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering. *Journal of Symbolic Computation*, 16(4):329–344, Oct 1993.

[FGMN97] E. V. Ferapontov, C .A. P. Galvao, O. Mokhov, and Y. Nutku. Bi-hamiltonian structure in 2-d field theory. *Communications in Mathematical Physics*, 186(3):649–669, Jan 1997.

[FH74] John A. Fox and Anthony C. Hearn. Analytic computation of some integrals in fourth order quantum electrodynamics. *Journal of Computational Physics*, 14(3):301–317, March 1974.

[Fil92] Sandra Fillebrown. Faster computation of bernoulli numbers. *Journal of Algorithms*, 13(3):431–445, September 1992.

[Fit75] J. P. Fitch. Syllabus for algebraic manipulation lectures in cambridge. *SIGSAM Bulletin*, 32:15, 1975.

[Fit83] J. P. Fitch. CAMAL User's Manual. Technical report, University of Cambridge Computer Laboratory, 1983. 2nd edition.

[FJ03] J. C. Faugère and A. Joux. Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases. In *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Springer, Berlin, Heidelberg, 2003.

[FK89] W. I. Fushchich and V. V. Kornyak. Computer algebra application for determining lie and lie–bäcklund symmetries of differential equations. *Journal of Symbolic Computation*, 7:611–619, 1989.

[FPV14] E. V. Ferapontov, M .V. Pavlov, and R.F. Vitolo. Projective-geometric aspects of homogeneous third-order hamiltonian operators. *Journal of Geometry and Physics*, 85:16–28, 2014.

[FPV16] E. V. Ferapontov, M. V. Pavlov, and R. F. Vitolo. Towards the classification of homogeneous third-order hamiltonian operators. *International Mathematics Research Notices*, 2016(22):6829–6855, Jan 2016.

[Gas95] G. Gasper. Lecture notes for an introductory minicourse on q-series. Mini-course lecture notes, 1995.

[GB98] V. P. Gerdt and Yu. A. Blinkov. Involutive bases of polynomial ideals. *Math. Comp. Simul.*, 45(5-6):519–541, March 1998.

[GCL92] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[GD96] Teresa Gómez-Díaz. Examples of using dynamic constructible closure. *Mathematics and Computers in Simulation*, 42(4–6):375–383, 1996.

[gde] Geometry of differential equations web site.

[Ger05] Vladimir P. Gerdt. Involutive algorithms for computing groebner bases. In Svetlana Cojocaru, Gerhard Pfister, and Victor Ufnarovski, editors, *Computational Commutative and Non-Commutative Algebraic Geometry*, volume 196 of *NATO Science Series, III: Computer and Systems Sciences*, pages 199–225, 2005. Proceedings of the NATO Advanced Research Workshop "Computational commutative and non-commutative algebraic geometry" (Chishinau, June 6-11, 2004).

[Get02] Ezra Getzler. A darboux theorem for hamiltonian operators in the formal calculus of variations. *Duke Mathematical Journal*, 111, Mar 2002.

[GH79] Martin L. Griss and Anthony C. Hearn. Portable LISP compiler. *Software - Practice and Experience*, 11(6):541–605, June 1979.

[GK82] Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms*, volume 1 of *Progress in Computer Science and Applied Logic (PCS)*. Birkhäuser Boston, 2nd edition, 1982.

[GM88] Rüdiger Gebauer and H. Michael Möller. On an installation of Buchberger's algorithm. *Journal of Symbolic Computation*, 6(2 and 3):275–286, 1988.

[GMM+81] V G Ganzha, S V Meleshko, F A Murzin, V P Shapeev, and N N Yanenko. Computer realization of an algorithm for investigating the compatibility of systems of partial differential equations,. *Soviet Math Dokl*, 24:638–640, 1981.

[GMN+91] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "";one sugar cube, please" or selection strategies in the buchberger algorithm. In Watt [Wat91], pages 49–54.

[Gos78] R. W. Gosper, Jr. Decision procedure for indefinite hypergeometric summation. *Proc. Natl. Acad. Sci. USA*, 75(1):40–42, January 1978.

[GR90] G. Gasper and M. Rahman. *Basic Hypergeometric Series*. Number 35 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, London and New York, 1990.

[Grä93] Hans-Gert Gräbe. Two remarks on independent sets. *Journal of Algebraic Combinatorics*, 2:137–145, 1993.

[Grä94a] Hans-Gert Gräbe. On factorized gröbner bases. In J. Fleischer, J. Grabmeier, F. W. Hehl, and W. Küchlin, editors, *Computer*

Algebra in Science and Engineering. World Scientific, 1994. Proceedings of the Conference, Bielefeld, Germany, 28–31 August 1994.

[Grä94b] Hans-Gert Gräbe. The tangent cone algorithm and homogenization. *Journal of Pure and Applied Algebra*, 97(3):303–312, Dec 1994.

[Grä95a] Hans-Gert Gräbe. Algorithms in local algebra. *Journal of Symbolic Computation*, 19(6):545–577, Jun 1995.

[Grä95b] Hans-Gert Gräbe. Triangular systems and factorized gröbner bases. In Gérard Cohen, Marc Giusti, and Teo Mora, editors, *Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, Lecture Notes in Computer Science, pages 248–261, Berlin, Heidelberg, 1995. Springer.

[Grä97] Hans-Gert Gräbe. Minimal primary decomposition and factorized gröbner bases. *Applicable Algebra in Engineering, Communication and Computing*, 8(4):265–278, 1997.

[Gru96] Dominik Gruntz. *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, ETH Zürich, 1996.

[GTZ88] Patrizia Gianni, Barry Trager, and Gail Zacharias. Gröbner bases and primary decomposition of polynomial ideals. *Journal of Symbolic Computation*, 6(2-3):149–167, Oct 1988.

[GZ08a] V. P. Gerdt and M. V. Zinin. Involutive method for computing Gröbner bases over $\mathbb{F}_2$. *Programming and Computer Software*, 34(4):191–203, Jul 2008.

[GZ08b] Vladimir P. Gerdt and Mikhail V. Zinin. A pommaret division algorithm for computing grobner bases in boolean rings. In *Proceedings of the Twenty-first International Symposium on Symbolic and Algebraic Computation*, ISSAC '08, pages 95–102, New York, NY, USA, 2008. ACM.

[GZB10] V. P. Gerdt, M. V. Zinin, and Yu. A. Blinkov. Programming and computing software. *Program. Comput. Softw.*, 36(2):117–123, mar 2010.

[Han75] E. R. Hansen. *A table of series and products*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[Har12] G. H. Hardy. Properties of logarithmico-exponential functions. *Proceedings of the London Mathematical Society*, s2-10(1):54–90, 1912.

[Har79] Steven J. Harrington. A new symbolic integration system in REDUCE. *The Computer Journal*, 22(2):127–131, 1979.

[Har89] David Harper. Vector33: a REDUCE program for vector algebra and calculus in orthogonal curvilinear coordinates. *Computer Physics Communications*, 54(2 and 3):295–305, June and July 1989.

[HCGJ92] D. E. G. Hare, R. M. Corless, G. H. Gonnet, and D. J. Jeffrey. On Lambert's W Function. Preprint, University of Waterloo, 1992.

[HCJE93] H. Hong, G. E. Collins, J. R. Johnson, and M. J. Encarnacion. QEPCAD interactive version 12, Sep 1993. Kindly communicated to us by Hoon Hong.

[Hea68] Anthony C. Hearn. REDUCE: A user-oriented interactive system for algebraic simplification. In M. Klerer and J. Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 79–90, New York, 1968. Academic Press.

[Hea69] Anthony C. Hearn. The problem of substitution. In R.G. Tobey, editor, *Proc. of the 1968 Summer Institute on Symbolic Mathematical Computation*, pages 3–19, Cambridge, Mass, 1969. IBM Boston Prog. Center.

[Hea71] Anthony C. Hearn. REDUCE 2: A system and language for algebraic manipulation. In S.R. Petrick, editor, *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 128–133. ACM, New York, 1971.

[Hea79] Anthony C. Hearn. Non-modular computation of polynomial GCDs using trial division. In *Symbolic and Algebraic Computation (EUROSAM '79, An International Symposium on Symbolic and Algebraic Manipulation, Marseille, France, June 1979)*, volume 72 of *Lecture Notes in Computer Science*, pages 227–239. Springer Berlin / Heidelberg, 1979.

[Hea95] Anthony C. Hearn. REDUCE User's Manual, Version 3.6. Report CP 78, RAND, July 1995.

[Her95] W. Hereman. *Symbolic Software for Lie Symmetry Analysis*, volume 3, chapter 13. CRC Press, Boca Raton, Florida, 1995.
Systems described in this paper are among others:
DELiA (Alexei Bocharov et.al.) Pascal
DIFFGROB2 (Liz Mansfield) Maple
DIMSYM (James Sherring and Geoff Prince) REDUCE
HSYM (Vladimir Gerdt) Reduce
LIE (V. Eliseev, R.N. Fedorova and V.V. Kornyak) Reduce

LIE (Alan Head) muMath
Lie (Gerd Baumann) Mathematica
LIEDF/INFSYM (Peter Gragert and Paul Kersten) Reduce
Liesymm (John Carminati, John Devitt and Greg Fee) Maple
MathSym (Scott Herod) Mathematica
NUSY (Clara Nucci) Reduce
PDELIE (Peter Vafeades) Macsyma
SPDE (Fritz Schwarz) Reduce and Axiom
SYM_DE (Stanly Steinberg) Macsyma
Symmgroup.c (Dominique Berube and Marc de Montigny) Mathematica
STANDARD FORM (Gregory Reid and Alan Wittkopf) Maple
SYMCAL (Gregory Reid) Macsyma and Maple
SYMMGRP.MAX (Benoit Champagne, Willy Hereman and Pavel Winternitz) Macsyma
LIE package (Khai Vu) Maple
Toolbox for symmetries (Mark Hickman) Maple
Lie symmetries (Jeffrey Ondich and Nick Coult) Mathematica.

[Hil99] Dietmar Hillebrand. Triangulierung nulldimensionaler Ideale - Implementierung und Vergleich zweier Algorithmen - in German . Diplomarbeit im Studiengang Mathematik der Universität Dortmund. Betreuer: Prof. Dr. H. M. Möller. Technical report, Universität Dortmund, 1999.

[HT91] David Hartley and Robin W. Tucker. A constructive implementation of the Cartan-Kähler theory of exterior differential systems. *Journal of Symbolic Computation*, 12(6):655–667, December 1991.

[HT93] David Hartley and Philip Tuckey. A direct characterisation of gröbner bases in clifford and grassmann algebras. Preprint MPI-Ph/93–96, Max-Planck-Institut für Physik, 1993.

[IK96] V. A. Ilyin and A. P. Kryukov. ATENSOR - REDUCE program for tensor simplification. *Computer Physics Communications*, 96(1):36–52, July 1996.

[IKRT89] V. A. Ilyin, A. P. Kryukov, A. Ya. Rodioniov, and A. Yu. Taranov. Fast algorithm for calculation of Dirac's gamma-matrices traces. *SIGSAM Bulletin*, 23(4):15–24, Oct 1989.

[IVV04] Sergei Igonin, A. Verboretsky, and Raffaele Vitolo. Variational multivectors and brackets in the geometry of jet spaces. In R.O. Popovych A.G. Nikitin, V.M. Boyko and I.A. Yehorchenko, editors, *V Int. Conf. on on Symmetry in Nonlinear Mathematical Physics, Kyiv 2003*, volume 50, pages 1335–1342, Oct 2004.

[Jan29] M. Janet. *Leçons sur les systèmes d'équations aux dérivées*. Gauthier-Villars, Paris, 1929.

[JDB65] Sidney D. Drell James D. Bjorken. *Relativistic Quantum Mechanics*. International Series In Pure and Applied Physics. McGraw-Hill, New York, 1965.

[Jef70] W. H. Jefferys. A fortran-based list processor for poisson series. *Celestial Mechanics*, 2(4):474–480, Dec 1970.

[JR94] D. J. Jeffrey and A. D. Rich. The evaluation of trigonometric integrals avoiding spurious discontinuities. *ACM Trans. Math. Softw.*, 20(1):124–135, March 1994.

[JT80] W.B. Jones and W.J. Thron. *Continued Fractions: Analytic Theory and Applications*, volume 11 of *Encyclopedia of mathematics and its applications*. Addison-Wesley Publishing Company, 1980.

[Kah68] Joseph Kahane. Algorithm for reducing contracted products of γ matrices. *Journal of Mathematical Physics*, 9(10):1732–1738, 1968.

[Kah87] W. Kahan. Branch cuts for complex elementary functions or much ado about nothing's sign bit. In A. Iserles and M.J.D. Powell, editors, *The State of the Art in Numerical Analysis : Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis held at the University of Birmingham, 14-18 April 1986*, Oxford, 1987. Clarendon Press.

[Kam59] E. Kamke. *Differentialgleichungen, Lösungsmethoden und Lösungen, Band 1, Gewöhnliche Differentialgleichungen*, volume 1. Chelsea Publishing Company, New York, 1959.

[Kaz87] C. Kazasov. Laplace transformations in REDUCE 3. In *Proc. EUROCAL '87, Lecture Notes in Computer Science*, volume 378, pages 132–133. Springer-Verlag, 1987.

[Ken82] A. D. Kennedy. Diagrammatic methods for spinors in feynman diagrams. *Phys. Rev. D*, 26:1936–1955, Oct 1982.

[Khi64] Aleksandr J. Khinchin. *Continued Fractions*. University of Chicago Press, 1964.

[KKV04] P. H. M. Kersten, I. S. Krasil'shchik, and A. M. Verbovetsky. Hamiltonian operators and ℓ^*-covering. *Journal of Geometry and Physics*, 50(1–4):273–302, 2004.

[KKV06] P. H. M. Kersten, I. S. Krasil'shchik, and A. M. Verbovetsky. A geometric study of the dispersionless boussinesq type equation. *Acta Applicandae Mathematica*, 90:143–178, 2006. 1–2.

[KKVV09] P. Kersten, I. S. Krasil'shchik, A. Verboretsky, and Vito. *Hamiltonian Structures for General PDEs*, pages 187–198. Abel Symposia. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Publishing Company, 2nd edition, 1981.

[Knu84] Donald E. Knuth. *The TEXbook*. Computers & typesetting. Addison-Wesley, Reading, 1984.

[Koe92] Wolfram Koepf. Power series in computer algebra. *Journal of Symbolic Computation*, 13(6):581–603, June 1992.

[Koe93a] Wolfram Koepf. Algorithmic development of power series. In J. Calmet and J. A. Campbell, editors, *Artificial intelligence and symbolic mathematical computing*, volume 737 of *Lecture Notes in Computer Science*, pages 195–213, Berlin, Heidelberg, 1993. Springer-Verlag. International Conference AISMC-1, Karlsruhe, Germany, August 1992, Proceedings.

[Koe93b] Wolfram Koepf. Examples for the algorithmic calculation of formal puiseux, laurent and power series. *ACM SIGSAM Bulletin*, 27(1):20–32, Jan 1993.

[Koe94a] Wolfram Koepf. Algorithmic work with orthogonal polynomials and special functions. Preprint sc 94-5, Konrad-Zuse-Zentrum Berlin (ZIB), 1994.

[Koe94b] Wolfram Koepf. Algorithms for the indefinite and definite summation. Preprint SC 94-33, Konrad-Zuse-Zentrum für Informationstechnik Berlin, December 1994.

[Koe95a] Wolfram Koepf. Algorithms for m-fold hypergeometric summation. *Journal of Symbolic Computation*, 20(4):399–417, Oct 1995.

[Koe95b] Wolfram Koepf. REDUCE package for the indefinite and definite summation. *SIGSAM Bulletin*, 29(1):14–30, January 1995.

[Koo93] T. H. Koornwinder. On Zeilberger's algorithm and its q-analogue: a rigorous description. *J. of Comput. and Appl. Math.*, 48(1-2):91–111, October 1993.

[KR88] A. P. Kryukov and A. Ya. Rodionov. Program "COLOR" for computing the group-theoretic weight of Feynman diagrams in Non-Abelian gauge theories. *Computer Physics Communications*, 48(2):327–334, February 1988.

[Kre87] Heinz Kredel. Primary ideal decomposition. In James Davenport, editor, *EUROCAL '87, European Conference on Computer Algebra, Leipzig, GDR, June 2-5, 1987, Proceedings*, 1987.

[Kre88] Heinz Kredel. Admissible term orderings used in computer algebra systems. *SIGSAM Bulletin*, 22(1):28–31, January 1988.

[KS94] R. Koekoek and R. F. Swarttouw. The askey-scheme of hypergeometric orthogonal polynomials and its q-analogue. Report 94-05, Faculty of Technical Mathematics and Informatics, Technische Universiteit Delft, Delft, 1994.

[Kub] M. Kubitza. Private communication.

[Kup94] B. A. Kuperschmidt. *Geometric Hamiltonian forms for the Kadomtsev-Petviashvili and Zabolotskaya-Khokhlov equations*, pages 155–172. World Scientific, 1994.

[KV11] Joseph Krasil'shchik and Alexander Verbovetsky. Geometry of jet spaces and integrable systems. *Journal of Geometry and Physics*, 61:1633–1674, 2011.

[KVV12] Joseph Krasil'shchik, Alexander Verbovetsky, and Raffaele Vitolo. A unified approach to computation of integrable structures. *Acta Applicandae Mathematicae*, 120(1):199–218, Aug 2012.

[KVV18] Joseph Krasil'shchik, Alexander Verbovetsky, and Raffaele Vitolo. *The Symbolic Computation of Integrability Structures for Partial Differential Equations*. Texts & Monographs in Symbolic Computation. Springer International Publishing, 1 edition, 2018.

[KW88] Heinz Kredel and Volker Weispfenning. Computing dimension and independent sets for polynomial ideals. *Journal of Symbolic Computation*, 6(2-3):231–247, Oct 1988.

[Lam86] Leslie Lamport. *LaTeX - A Document Preparation System*. Addison-Wesley, Reading, 1986.

[Law89] Derek F. Lawden. *Elliptic Functions and Applications*. Springer-Verlag, 1989.

[Laz83] D. Lazard. Gröbner bases, gaussian elimination and resolution of systems of algebraic equations. In *Proceedings of EUROCAL '83*, volume 162 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 1983.

[LB68] Landolt-Boernstein. *Zahlenwerte und Funktionen aus Naturwissenschaften und Technik*. Springer, 1968.

[LD90] R. Liska and L. Drska. FIDE: A REDUCE package for automation of FInite difference method for solving pDE. In S. Watanabe and Morio Nagata, editors, *Proceedings of the 1990 International Symposium on Symbolic and Algebraic Computation*, pages 169–176. ACM, Addison-Wesley, 1990.

[Leo80] Jeffrey S. Leon. On an algorithm for finding a base and a strong generating set for a group given by generating permutations. *Mathematics of Computation*, 35(151):941–974, 1980.

[Leo84] Jeffrey S. Leon. Computing Automorphism Groups of Combinatorial Objects. In Michael D. Atkinson, editor, *Computational Group Theory: Proceedings of the London Mathematical Society Symposium on Computational Group Theory*. London Mathematical Society, Academic Press, 1984.

[Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions, i: Theory and algorithms. *Journal of S*, 12:533–583, 1991.

[Lie67] S. Lie. *Differentialgleichungen*. Chelsea Publishing Company, New York, 1967.

[Lie75] S. Lie. Sophus lie's 1880 transformation group paper. Translated by M. Ackerman, comments by R. Hermann, Mathematical Sciences Press, Brookline, 1975.

[Lin91] Stephen A. Linton. Double coset enumeration. *Journal of Symbolic Computation*, 12(4):415–426, 1991.

[Lis91] R. Liska. Numerical code generation for finite difference schemes solving. In R. Vichnevetsky and J.J.H. Miller, editors, *IMACS'91 13th World Congress on Computation and Applied Mathematics, July 22-26, 1991*, pages 92–93, Dublin, 1991. IMACS.

[LM94] Eugenio Roanes Lozano and Eugenio Roanes Macias. An implementatio of "turtle graphics" in maple v. In Tony Scott, editor, *Maple in Mathematics and the Sciences - Maple Technical Newsletter Special Issue*, pages 82–85. Birkhäuser, 12 1994.

[Loo82] R. Loos. *Computing in Algebraic Extensions*, pages 173–187. Springer Vienna, Vienna, 1982.

[Loo83] Rüdiger Loos. Computing rational zeros of integral polynomials by p-adic expansion. *SIAM J. Computing*, 12(2):286–293, 1983.

[Mac88] M. A. H. MacCallum. An ordinary differential equation solver for REDUCE. In *Proc. of ISSAC '88*, volume 358, pages 196–205. Springer-Verlag, 1988.

[Mac89] M. A. H. MacCallum. An ordinary differential equation solver for reduce. In P. Gianni, editor, *Symbolic and Algebraic Computation*, pages 196–205, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[Mac99] M. A. H. MacCallum. *On the Classification of the Real Four-Dimensional Lie Algebras*, pages 299–317. Springer New York, New York, NY, 1999.

[MAE+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. M.I.T. Press, Aug 1962.

[Man94] Y.-K. Man. *Algorithmic Solution of ODEs and Symbolic Summation Using Computer Algebra*. PhD thesis, School of Mathematical Sciences, Queen Mary and Westfield College, University of London, July 1994.

[Man96] Elizabeth L. Mansfield. The differential algebra package diffgrob2. *Mapletech*, 3(2):33–37, 1996.

[Mar93] Jed Marti. *RLISP '88 An Evolutionary Approach to Program Design and Reuse*, volume 42 of *World Scientific Series in Computer Science*. World Scientific, Singapore, 1993.

[Mar09] M. Marvan. Sufficient set of integrability conditions of an orthonomic system. *Foundations of Computational Mathematics*, 9:652–674, Jan 2009.

[McI85] Kevin McIsaac. Pattern matching algebraic identities. *SIGSAM Bulletin*, 19(2):4–13, May 1985.

[McK78] Brendan D. McKay. Computing automorphisms and canonical labellings of graphs. In D. A. Holton and Jennifer Seberry, editors, *Combinatorial Mathematics*, pages 223–232, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[Mel95] Herbert Melenk. Reduce symbolic mode primer. Technical report, Konrad-Zuse-Institut, 1995.

[MF93] E Mansfield and E D Fackerell. Differential gröbner bases and involutivity of systems of non-linear partial differential equations. Preprint 92/108, School of Mathematics, Physics, Computer Science, and Electronics, Macquarie University, Sydney, Australia, 1993.

[MHGG80] J. Marti, A. C. Hearn, M. L. Griss, and C. Griss. Standard Lisp report. *SIGSAM Bulletin*, 14(1):23–41, February 1980.

[Mis93] Bhubaneswar Mishra. *Algorithmic Algebra*. Monographs in Computer Science. Springer, New York, 1993.

[MM86] H. Michael Möller and Ferdinando Mora. New constructive methods in classical ideal theory. *Journal of Algebra*, 100(1):138–178, 1986.

[MM97] Yiu-Kwong Man and Malcolm A. H. MacCallum. A rational approach to the prelle-singer algorithm. *Journal of Symbolic Computation*, 24(1):31–43, Jul 1997.

[MMM91] M. G. Marinari, H. M. Möller, and T. Mora. Gröbner bases of ideals given by dual bases. In Watt [Wat91].

[MMN88] H. Melenk, H. M. Möller, and W. Neun. On Gröbner bases computation on a supercomputer using REDUCE. Preprint SC 88-2, Konrad-Zuse-Zentrum für Informationstechnik Berlin, January 1988.

[MN02] H. Melenk and W. Neun. Rr: Parallel symbolic algorithm support for psl based reduce. Technical report, ZIB, 2002.

[Möl93] H. Michael Möller. On decomposing systems of polynomial equations with finitely many solutions. *Applicable Algebra in Engineering, Communication and Computing*, 4(4):217–230, 1993.

[MPT89] Teo Mora, Gerhard Pfister, and Carlo Traverso. An Introduction to the Tangent Cone Algorithm. *Publications de l'Institut de recherche mathématiques de Rennes*, (4):133–171, 1989.

[MR88] Teo Mora and Lorenzo Robbiano. The gröbner fan of an ideal. *Journal of Symbolic Computation*, 6(2-3):183–208, Oct 1988.

[MY73] Joel Moses and David Y. Y. Yun. The ez gcd algorithm. In *ACM Annual Conference*, pages 159–166. Association for Computing Machinery, 08 1973.

[ND79] A. C. Norman and J. H. Davenport. Symbolic integration - the dust settles? In *Proc. EUROSAM 1979*, volume 72 of *Lecture Notes in Computer Science*, pages 398–407. Springer-Verlag, 1979.

[NM77] A. C. Norman and P. M. A. Moore. Implementing the new Risch integration algorithm. In *Proc. of the Fourth Colloquium on Advanced Comp. Methods in Theor. Phys., St. Maximin, France*, March 1977.

[NM81] Arthur C. Norman and Mary Ann Moore. Implementing a polynomial factorization and gcd package. In *SYMSAC '81: Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 109–116. Association for Computing Machinery, 08 1981.

[NNS05] F. Neyzi, Y. Nutku, and M. B. Sheftel. Multi-hamiltonian structure of plebanski's second heavenly equation. *Journal of Physics A*, 38(39):8473, 2005.

[Nuc92] C. M. Nucci. *Interactive REDUCE programs for calculating classical, non-classical, and approximate symmetries of differential equations*, pages 345–350. Elsevier, Amsterdam, 1992.

[Nuc96] Maria Clara Nucci. *Interactive REDUCE programs for calculating Lie point, non-classical, Lie-Bäcklund, and approximate symmetries of differential equations: Manual and floppy disk*, volume 3 of *CRC Handbook of Lie Group Analysis of Differential Equations*, pages 415–481. CRC Press, Boca Raton, Jan 1996.

[NUS91] Arnold F. Nikiforov, Vasilii B. Uvarov, and Sergei K. Suslov. *Classical Orthogonal Polynomials of a Discrete Variable*. Springer Verlag, Berlin–Heidelberg–New York, 1991.

[NV] A. C. Norman and R. Vitolo. Inside reduce. Part of the official REDUCE documentation included in the source code.

[OK94] A. W. Overhauser and Y. J. Kim. An infinite sum from a diffusion problem. *SIAM Review*, 36(1):107, 1994.

[Oli] P. Oliveri. ReLie, Reduce software and user guide. online.

[Olv86] Peter J. Olver. *Applications of Lie Groups to Differential Equations*, volume 107 of *Graduate Texts in Mathematics*. Springer Verlag, New York, 1986.

[Olv93] Peter J. Olver. *Applications of Lie Groups to Differential Equations*, volume 107 of *Graduate Texts in Mathematics*. Springer-Verlag New York, 2nd edition, 1993.

[PB90] Julian Padget and Alan Barnes. Univariate power series expansions in REDUCE. In S. Watanabe and Morio Nagata, editors, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 82–87. ACM, Addison-Wesley, 1990.

[PBM89] A. P. Prudnikov, Yu. A. Brychkov, and O. I. Marichev. *Integrals and Series, Volume 3: More Special Functions*. Gordon and Breach Science Publishers, Oct 1989. Transl. from the Russian by G. G. Gould.

[Pos96] G. F. Post. A manual for the package TOOLS 2.1. Memorandum 1331, Dept. Appl. Math., University of Twente, 1996.

[PR95] P. Paule and A. Riese. A mathematica q-analogue of zeilberger's algorithm based on an algebraically motivated approach to q-hypergeometric telescoping. In *Fields Proceedings of the Workshop 'Special Functions, q-Series and Related Topics*, pages 179–210, Toronto, Ontario, 12-23 June 1995. Fields Institute for Research in Mathematical Sciences at University College.

[PS83] M. J. Prelle and M. F. Singer. Elementary first integrals of differential equations. *Transactions of the American Mathematical Society*, 279:215–229, 1983.

[PS95] Peter Paule and Markus Schorn. A MATHEMATICA version of zeilberger's algorithm for proving binomial coefficient identities. *Journal of Symbolic Computation*, 20(5–6):673–698, Nov 1995.

[PV15] Maxim V. Pavlov and Raffaele. Vitolo. On the bi-hamiltonian geometry of wdvv equations. *Letters in Mathematical Physics*, 105(8):1135–163, Aug 2015.

[PZ96] F. Postel and P. Zimmermann. A Review of the ODE Solvers of AXIOM, DERIVE, MAPLE MATHEMATICA, MACSYMA, MUPAD and REDUCE. In *Proceedings of the 5th Rhine Workshop on Computer Algebra, April 1-3, 1996, Saint-Louis, France*, 1996. Specific references are to the version dated April 11, 1996.

[PZ97] Zoran Putnik and Budimac Zoran. Implementation of turtle graphics for teaching purposes. In *EDUGRAPHICS '97. Third International Conference on Graphics Education. COMPUGRAPHICS '97. Sixth International Conference on Computational Graphics and Visualization Techniques. Combined Proceedings*, 01 1997.

[RED] Obtaining reduce.

[Rei90] G. J. Reid. A triangularization algorithm which determines the lie symmetry algebra of any system of pdes. *Journal of Physics A*, 23(17):L853–L859, 1990.

[Rei91] Gregory J. Reid. Algorithms for reducing a system of pdes to standard form, determining the dimension of its solution space and calculating its taylor series solution. *European Journal of Applied Mathematics*, 2(4):293–318, 1991.

[Riq10] C. Riquier. *Les systèmes d'équations aux dérivées partielles*. Gauthier–Villars, Paris, 1910.

[RLW93] R. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36:450–462, Jan 1993.

[RLW01] Gregory J. Reid, Ping Lin, and Allan D. Wittkopf. Differential elimination–completion algorithms for dae and pdae. *Studies in Applied Mathematics*, 106(1):1–45, 2001.

[Roa] Kelly Roach. Difficulties with trigonometrics. Notes of a talk.

[Rob89] Lorenzo Robbiano. Computer and commutative algebra. In Teo Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 31–44, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[Roe92a] G. H. M. Roelofs. The INTEGRATOR package for REDUCE, Version 1.0. Memorandum 1100, Dept. Appl. Math., University of Twente, 1992.

[Roe92b] G. H. M. Roelofs. The SUPER VECTOR FIELD package for REDUCE, Version 1.0. Memorandum 1099, Dept. Appl. Math., University of Twente, 1992.

[RT89] A.Ya. Rodionov and A.Yu. Taranov. Combinatorial aspects of simplification of algebraic expressions. In James H. Davenport, editor, *Eurocal '87*, pages 192–201, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[Rut92] Elizabeth W. Rutman. Gröbner bases and primary decomposition of modules. *Journal of Symbolic Computation*, 14(5):483–503, Nov 1992.

[RWB96] Gregory J. Reid, Allan D. Wittkopf, and Alan Boulton. Reduction of systems of nonlinear partial differential equations to simplified involutive forms. *European Journal of Applied Mathematics*, 7(6):635–666, 1996.

[Sch85a] Eberhard Schrüfer. Excalc: A system for doing calculations in modern differential geometry, user's manual. Technical report, Rand Publication, The Rand Corporation, Santa Monica, 1985.

[Sch85b] F. Schwarz. Automatically determining symmetries of partial differential equations. *Computing*, 34(2):91–106, Jun 1985.

[Sch85c] F. Schwarz. Automatically determining symmetries of partial differential equations. *Computing*, 34(2):91–106, November 1985.

[Sch87] Fritz Schwarz. Computer algebra and differential equations of mathematical physics. Technical report, GMD internal report, 1987.

[Sch88] F. Schwarz. Symmetries of differential equations: From Sophus Lie to computer algebra. *SIAM Review*, 30(3):450–481, 1988.

[Sch92] Franziska Schoebel. The symbolic classification of real four-dimensional lie algebras. Technical Report Preprint 27/92, Naturwissenschaftlich-Theoretisches Zentrum, Universitaet Leipzig, Germany, 1992.

[Sch93] Carsten Schöbel. A classification of real finite-dimensional lie algebras with a low-dimensional derived algebra. *Reports on Mathematical Physics*, 33(1–2):175..186, Aug–Oct 1993.

[Sei95] W M Seiler. Applying axiom to partial differential equations. Internal Report 95-17, Universität Karlsruhe, Fakultät für Informatik, 1995.

[Sei10] Werner M. Seiler. *Involution: The Formal Theory of Differential Equations and its Applications in Computer Algebra*, volume 24 of *Algorithms and Computation in Mathematics*. Springer Verlag Berlin Heidelberg, 2010.

[Ser77] Jean-Pierre Serre. *Linear Representations of Finite Groups*, volume 42 of *Graduate Texts in Mathematics*. Springer, New York, NY, 1977.

[SF79] Eduard Stiefel and Albert Fässler. *Gruppentheoretische Methoden und ihre Anwendung*. Teubner, Stuttgart, 1979.

[Sim71a] Charles C. Sims. Computation with permutation groups. In S. R. Petrick, editor, *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*, SYMSAC '71, page 23–28, New York, NY, USA, 1971. Association for Computing Machinery.

[Sim71b] Charles C. Sims. Determining the conjugation classes of a permutation group. In G. Birckhoff and M. Hall Jr., editors, *Computer in Algebra and Number Theory*, volume 4 of *SIAM-AMS Proceedings*, pages 191–195. American Mathematical Society, 1971.

[SLT65] Pascual Sconzo, A. R. Leschack, and Robert G. Tobey. Symbolic computation of f and g series by computer. *The Astronomical Journal*, 70:269, May 1965.

[Spi59] Murray R. Spiegel. *Vector Analysis*. Schaum's Outline. McGraw-Hill Education (India) Pvt Limited, 1959.

[Spi79] Michael Spivak. *A comprehensive introduction to differential geometry*. Publish or Perish, Berkeley, 1979.

[Ste89] Hans Stephani. *Differential equations, Their solution using symmetries*. Cambridge University Press, 1989.

[Sto90] Timothy Stokes. Gröbner bases in exterior algebra. *Journal of Automated Reasoning*, 6(3):233–250, Sep 1990.

[Str93] Volker Strehl. Binomial sums and identities. *Maple Technical Newsletter*, 10:37–49, 1993.

[SV14] Giuseppe Saccomandi and Raffaele Vitolo. On the mathematical and geometrical structure of the determining equations for shear waves in nonlinear isotropic incompressible elastodynamics. *Journal of Mathematical Physics*, 55(8):081502, 2014.

[SW06] Vladimir V Sokolov and Thomas Wolf. Integrable quadratic classical hamiltonians on so(4) and so(3, 1). *Journal of Physics A: Mathematical and General*, 39(8):1915, feb 2006.

[Tar48] A. Tarski. A decision method for elementary algebra and geometry. Technical report, University of California, 1948. Second edn., rev. 1951.

[Tho37] Joseph Miller Thomas. *Differential systems*, volume 21. American Mathematical Soc., 1937.

[Tou84] David S. Touretzky. *LISP: A Gentle Introduction to Symbolic Computation*. Harper & Row, New York, 1984.

[Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In Richard D. Jenks, editor, *SYMSAC '76: Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 196–208, New York, NY, USA, 1976. ACM.

[Vit] R. Vitolo. CDE: a reduce package for computations in geometry of Differential Equations. Available at https://gdeq.org.

[Vit19] R. Vitolo. Computing with hamiltonian operators. *Comput. Phys. Commun.*, 244:228–245, 2019. ArXiv: https://arxiv.org/abs/1808.03902.

[Wah93] Hugo D Wahlquist. Monte carlo calculation of cartan characters: using the maximal-slicing, ricci-flat ideal as an example. In Nora Bretón, Riccardo Capovilla, and Tonatiuh Matos, editors, *Proceedings, 12th International School-Seminar on The Actual Problems of Microworld Physics. Vol.1: Gomel, Belarus, July 22 - September 28, 2013*, pages 168–174, 1993.

[War91] M. Warns. Software extensions of reduce for operator calculus in quantum theory. In *Proceedings of the IV international Conference on Computer Algebra in Physical Research, Dubna 1990*. WORLD SCIENTIFIC, 1991.

[Wat91] Stephen M. Watt, editor. *ISSAC '91. Proceedings of the 1991 international symposium on Symbolic and algebraic computation. Bonn, Germany, July 15–17, 1991*, New York, NY, USA, 1991. ACM.

[WB] Thomas Wolf and Andreas Brand. Demonstration of the reduce package crack for investigating partial differential equations. online.

[Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symb*, 5(1–2):3–27, Feb 1988.

[Wei92] Volker Weispfenning. Comprehensive Gröbner bases. *Journal of Symbolic Computation*, 14(1):1–29, July 1992.

[Wei94] Volker Weispfenning. Quantifier elimination for real algebra – the cubic case. In *Symbolic and Algebraic Computation*, ISSAC, pages 258–263. SIGSAM, ACM, 1994.

[Wei97] V. Weispfenning. Quantifier elimination for real algebra — the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, Jan 1997.

[WH81] Patrick Winston and Berthold Horn. *LISP*. Addison-Wesley, 1981.

[Wil90] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, Boston, 1990.

[Wil93] Herbert S. Wilf. Identities and their computer proofs. SPICE lecture notes 31. Available by anonymous ftp to ftp.cis.upenn.edu as file pub/wilf/lecnotes.ps., 1993. A series of lectures delivered at the U.S. National Security Agency.

[Wir80] M. C. Wirth. *On the Automation of Computational Physics*. PhD thesis, Lawrence Livermore National Lab., CA., jan 1980.

[Wol93] T. Wolf. An efficiency improved program liepde for determining lie-symmetries of pdes. In *Proc. of Modern Group Analysis: advanced analytical and computational methods in mathematical physics, Catania, Italy, October 1992*, pages 377–385. Kluwer Academic Publishers, 1993.

[Wol95] Thomas Wolf. APPLYSYM - a package for the application of Lie-symmetries, 1995. Software distributed together with the computer algebra system REDUCE.

[Wol00] Thomas Wolf. The symbolic integration of exact pdes. *Journal of Symbolic Computation*, 30(5):619–629, 2000.

[Wol02a] T. Wolf. A comparison of four approaches to the calculation of conservation laws. *Euro. Jnl of Applied Mathematics*, 13(2):129–152, 2002.

[Wol02b] Thomas Wolf. Size reduction and partial decoupling of systems of equations. *Journal of Symbolic Computation*, 33(3):367–383, 2002.

[Wri97] F. J. Wright. An Enhanced ODE Solver for REDUCE. *Programmirovanie*, 3:5–22, 1997.

[Wri99] F. J. Wright. Design and Implementation of ODESolve 1+ : An Enhanced REDUCE ODE Solver. Technical report, Queen Mary, University of London, May 1999.

[Wt87] Wu Wen-tsun. A zero structure theorem for polynomial-equations-solving and its applications. In *Proceedings of the European Conference on Computer Algebra*, EUROCAL '87, page 44, Berlin, Heidelberg, 1987. Springer-Verlag.

[WW69] E. T. Whittaker and G. N. Watson. *A Course in Modern Analysis*. Cambridge University Press, 1969.

[ZB96] A. Yu. Zharkov and Yu. A. Blinkov. Involution approach to investigating polynomial systems. *Mathematics and Computers in Simulation*, 42:323–332, 11 1996.

[Zei90] D. Zeilberger. A fast algorithm for proving terminating hypergeometric identities. *Discrete Math.*, 80(2):207–211, March 1990.

[Zei91] D. Zeilberger. The method of creative telescoping. *Journal of Symbolic Computation*, 11(3):195–204, 1991.

[Zwi92] D. Zwillinger. *Handbook of Differential Equations*. Academic Press, second edition, 1992.

Appendix C

Changes since Version 3.8

New packages assert bibasis breduce cde cdiff clprl gcref guardian lalr lessons libreduce listvecops logoturtle lpdo redfront reduce4 sstools utf8 with

Core package rlisp Support for namespaces (::)

Default value in switch statement

Support for utf8 characters

in_tex command

Core package poly Improvements for differentiation: new switches `expanddf`, `allowdfint` etc (from odesolve)

New operator `reimpart`

Core package alg New switch `precise_complex`

Improvements for switch `combineexpt` (exptchk.red)

New command `unset`

New operators `continued_fraction`, `totaldeg`

Improvements to the `conj` operator, added `selfconjugate` declaration.

Added `complex_conjugates` declaration to associate pairs of identifiers as mutual complex-conjugates.

Core Package mathpr New switch `unicode_in_off_nat` to have unicode characters displayed as such when nat is off.

Core Package solve New boolean operator `polyp`(p,var), to determine whether p is a pure polynomial in var, ie. the coefficients of p do not contain var.

Core Package matrix New keyword **matrixproc** for declaration of matrix-valued procedures, and listproc for declaration of list-valued procedures.

Operators now defined in the REDUCE core From changevar: `changevar`,

From polyrat: `divide`, `pseudo_divide`, `pseudo_div`, `pseudo_quotient`, `pseudo_remainder`,

From specfn: `Li`, `Si`, `Ci`, `Shi`, `Chi`, `Fresnel_S`, `Fresnel_C`, `gamma`, `igamma`, `psi`, `polygamma`, `beta`, `ibeta`, `euler`, `bernoulli`, `pochhammer`, `lerch_phi`, `polylog`, `zeta`, `besselj`, `bessely`, `besseli`, `besselk`, `hankel1`, `hankel2`, `kummerM`, `kummerU`, `struveh`, `struvel`, `lommel1`, `lommel2`, `whittakerm`, `whittakerw`, `Airy_Ai`, `Airy_Bi`, `Airy_AiPrime`, `Airy_biprime`, `binomial`, `solidharmonic`, `sphericalharmonic`, `fibonacci`,`fibonaccip`, `motzkin`, `hypergeometric`, `MeijerG`

From limit: `limits`, `limit!+`, `limit!-`,

From taylor and tps `taylor`, `implicit_taylor`, `inverse_taylor`, `taylororiginal`, `taylortemplate`, `taylorcoefflist`, `taylortostandard`, `taylorcombine`, `taylorseriesp`, `taylorrevert`, `ps`, `psexplim`, `psordlim`, `psterm`, `psorder`, `pssetorder`, `psdepvar`, `psexpansionpt`, `psfunction`, `pschangevar`, `psreverse`, `pscompose`, `pssum`, `pstaylor`, `pscopy`, `pstruncate`

From fps: `fps`, `simplede`, `infsum`

From compact: `compact`

From residue: `residue`, `poleorder`

From ineq and rsolve: `ineq_solve`, `r_solve`, `i_solve`,

From roots: `realroots`, `isolater`, `rlrootno`, `roots`, `roots_at_prec`, `root_val`, `nearestroot`, `firstroot`, `getroot`, `mkpoly`, `gfnewt`, `gfroot`,

From laplace: `laplace`, `invlap`,

From defint: `laplace_transform`, `hankel_transform`, `y_transform`, `k_transform`, `struveh_transform`, `fourier_sin`, `fourier_cos`,

From arnum: `defpoly`, `split_field`

From zeilberg: `extended_gosper`, `extended_sumrecursion`, `gosper`,

hyperrecursion, hypersum, hyperterm, sumtohyper, sumrecursion, simplify_gamma, simplify_gamma2, simplify_gamman, simplify_combinatorial,

From trigsimp: trigsimp, triggcd, trigfactorize

From ratint: ratint, log_sum

From odesolve: odesolve, dsolve, root_of_unity, plus_or_minus,

From gnuplot: New operator plot, new commands gnuplot, plotshow and plotreset.

New switches tracefps, zb_factor, zb_proof, zb_trace

Constants now part of the core: catalan, euler_gamma, golden_ratio, khinchin.

Variables as part of the core: gosper_representation, zb_direction, zb_order, zeilberger_representation,

Consistent branch cuts for complex numerical functions.

Package specfn psi (digamma) function can now be calculated numerically for complex arguments.

Package specfn New functions: theta1d, theta2d, theta3d and theta4d — numerical evaluation derivatives of theta functions.

Package specfn Weierstrass, WeierstrassZeta, sigma, sigma1, sigma2, sigma3 and sigma4 — rules and numerical code added.

Package defint Added tracing output printing of which is controlled by the switch trdefint.

TeXmacs interface Print prompt numbers by setting the switch promptnumbers to on by default.

Package excalc New command killing_vector.

Package specfn arcsn, arccn, arcdn, arcns, arcnc, arcnd, arcsc, arcsd, arccs, arccd, arcds, arcdc — rules and numerical code added for inverse Jacobian elliptic funtions, real arguments and values only.

Package ellipfn New package comprising modules: `ellipfn`, `efjacobi`, `efellint`, `efjacinv`, `eftheta` and `efweier`. These are essentially copies of the modules `sfellip`, `sfellipi`, `sfellipinv`, `sftheta` and `sfweier` which have been removed from package `specfn`. Documentation moved into a new user contributed package in `ellipfn.tex`.

Appendix D

Description of Algorithms

D.1 Definite Integration

This section describes part of REDUCE's definite integration package that is able to calculate the definite integrals of many functions, including several special functions. There are other parts of this package, such as Stan Kameny's code for contour integration, that are not included here. The integration process described here is not the more normal approach of initially calculating the indefinite integral, but is instead the rather unusual idea of representing each function as a Meijer G-function (a formal definition of the Meijer G-function can be found in [PBM89]), and then calculating the integral by using the following Meijer G integration formula.

$$\int_0^\infty x^{\alpha-1} G_{uv}^{st}\left(\sigma x \,\middle|\, \begin{matrix}(c_u)\\(d_v)\end{matrix}\right) G_{pq}^{mn}\left(\omega x^{l/k} \,\middle|\, \begin{matrix}(a_p)\\(b_q)\end{matrix}\right) dx = k G_{kl}^{ij}\left(\xi \,\middle|\, \begin{matrix}(g_k)\\(h_l)\end{matrix}\right) \tag{D.1}$$

The resulting Meijer G-function is then retransformed, either directly or via a hypergeometric function simplification, to give the answer. A more detailed account of this theory can be found in [AM90].

D.1.1 Integration between zero and infinity

As an example, if one wishes to calculate the following integral

$$\int_0^\infty x^{-1} e^{-x} \sin(x)\, dx$$

then initially the correct Meijer G-functions are found, via a pattern matching

process, and are substituted into eq. D.1 to give

$$\sqrt{\pi}\int_0^\infty x^{-1}G_{01}^{10}\left(x\,\middle|\,\begin{matrix}\cdot\\0\end{matrix}\right)G_{02}^{10}\left(\frac{x^2}{4}\,\middle|\,\begin{matrix}\cdot\,\cdot\\\frac{1}{2}\;0\end{matrix}\right)dx$$

The cases for validity of the integral are then checked. If these are found to be satisfactory then the formula is calculated and we obtain the following Meijer G-function

$$G_{22}^{12}\left(1\,\middle|\,\begin{matrix}\frac{1}{2}\;1\\\frac{1}{2}\;0\end{matrix}\right)$$

This is reduced to the following hypergeometric function

$${}_2F_1(\frac{1}{2},1;\frac{3}{2};-1)$$

which is then calculated to give the correct answer of

$$\frac{\pi}{4}$$

The above formula (D.1) is also true for the integration of a single Meijer G-function by replacing the second Meijer G-function with a trivial Meijer G-function.

A list of numerous particular Meijer G-functions is available in [PBM89].

D.1.2 Integration over other ranges

Although the description so far has been limited to the computation of definite integrals between 0 and infinity, it can also be extended to calculate integrals between 0 and some specific upper bound, and by further extension, integrals between any two bounds. One approach is to use the Heaviside function, i.e.

$$\int_0^\infty x^2e^{-x}H(1-x)\,dx=\int_0^1 x^2e^{-x}dx$$

Another approach, again not involving the normal indefinite integration process, again uses Meijer G-functions, this time by means of the following formula

$$\int_0^y x^{\alpha-1}G_{pq}^{mn}\left(\sigma x\,\middle|\,\begin{matrix}(a_u)\\(b_v)\end{matrix}\right)dx=y^\alpha\,G_{p+1\;q+1}^{m\;n+1}\left(\sigma y\,\middle|\,\begin{matrix}(a_1..a_n,1-\alpha,a_{n+1}..a_p)\\(b_1..b_m,-\alpha,b_{m+1}..b_q)\end{matrix}\right)\tag{D.2}$$

For a more detailed look at the theory behind this see [AM90].

For example, if one wishes to calculate the following integral

$$\int_0^y \sin(2\sqrt{x})\,dx$$

then initially the correct Meijer G-function is found, by a pattern matching process, and is substituted into eq. D.2 to give

$$\int_0^y G_{02}^{10}\left(x \,\middle|\, \begin{matrix} \cdot & \cdot \\ \frac{1}{2} & 0 \end{matrix}\right) dx$$

which then in turn gives

$$y\, G_{13}^{11}\left(y \,\middle|\, \begin{matrix} 0 \\ \frac{1}{2}\ -1\ 0 \end{matrix}\right) dx$$

and returns the result

$$\frac{\sqrt{\pi}\, J_{3/2}(2\sqrt{y})\, y}{y^{1/4}}$$

D.1.2.1 Examples

$$\int_0^\infty e^{-x} dx$$

```
int(e^(-x),x,0,infinity);
```

```
1
```

$$\int_0^\infty x \sin(1/x)\,dx$$

```
int(x*sin(1/x),x,0,infinity);
```

```
            1
int(x*sin(---),x,0,infinity)
            x
```

$$\int_0^\infty x^2 \cos(x)\, e^{-2x} dx$$

```
int(x^2*cos(x)*e^(-2*x),x,0,infinity);

  4
-----
 125
```

$$\int_0^\infty x e^{-1/2x} H(1-x)\, dx = \int_0^1 x e^{-1/2x} dx$$

```
int(x*e^(-1/2x)*Heaviside(1-x),x,0,infinity);

 2*(2*sqrt(e) - 3)
------------------
      sqrt(e)
```

$$\int_0^1 x \log(1+x)\, dx$$

```
int(x*log(1+x),x,0,1);

 1
---
 4
```

$$\int_0^y \cos(2x)\, dx$$

```
int(cos(2x),x,y,2y);

 sin(4*y) - sin(2*y)
--------------------
          2
```

D.1.3 Integral Transforms

A useful application of the definite integration package is in the calculation of various integral transforms. The transforms available are as follows:

- Laplace transform
- Hankel transform
- Y-transform
- K-transform
- StruveH transform
- Fourier sine transform
- Fourier cosine transform

D.1.3.1 Laplace transform

The Laplace transform

$$f(s) = \mathcal{L}\{\mathrm{F}(t)\} = \int_0^\infty e^{-st} F(t)\, dt$$

can be calculated by using the `laplace_transform` operator.
This requires as parameters

- the function to be integrated
- the integration variable.

For example

$$\mathcal{L}\{e^{-at}\}$$

is entered as

```
laplace_transform(e^(-a*x),x);
```

and returns the result

$$\frac{1}{s+a}$$

D.1.3.2 Hankel transform

The Hankel transform

$$f(\omega) = \int_0^\infty F(t)\, J_\nu(2\sqrt{\omega t})\, dt$$

can be calculated by using the `hankel_transform` operator, e.g.,

```
hankel_transform(f(x),x);
```

This is used in the same way as the `laplace_transform` operator.

D.1.3.3 Y-transform

The Y-transform

$$f(\omega) = \int_0^\infty F(t)\, Y_\nu(2\sqrt{\omega t})\, dt$$

can be calculated by using the `Y_transform` operator, e.g.,

```
Y_transform(f(x),x);
```

This is used in the same way as the `laplace_transform` operator.

D.1.3.4 K-transform

The K-transform

$$f(\omega) = \int_0^\infty F(t)\, K_\nu(2\sqrt{\omega t})\, dt$$

can be calculated by using the `K_transform` operator, e.g.,

```
K_transform(f(x),x);
```

This is used in the same way as the `laplace_transform` operator.

D.1.3.5 StruveH transform

The StruveH transform

$$f(\omega) = \int_0^\infty F(t)\, StruveH(\nu, 2\sqrt{\omega t})\, dt$$

can be calculated by using the `struveh_transform` operator, e.g.,

```
struveh_transform(f(x),x);
```

This is used in the same way as the `laplace_transform` operator.

D.1.3.6 Fourier sine transform

The Fourier sine transform

$$f(s) = \int_0^\infty F(t) \sin(st)\, dt$$

can be calculated by using the `fourier_sin` operator, e.g.,

```
fourier_sin(f(x),x);
```

This is used in the same way as the `laplace_transform` operator.

D.1.3.7 Fourier cosine transform

The Fourier cosine transform

$$f(s) = \int_0^\infty F(t) \cos(st)\, dt$$

can be calculated by using the `fourier_cos` operator, e.g.,

```
fourier_cos(f(x),x);
```

This is used in the same way as the `laplace_transform` operator.

D.1.3.8 The print_conditions function

The required conditions for the validity of the transform integrals can be viewed using the following command:

```
print_conditions().
```

For example after calculating the following laplace transform

```
laplace_transform(x^k,x);
```

using the `print_conditions` operator would produce

```
                                1
repart(sum(ai) - sum(bj)) + ---*(q + 1 - p)
                                2

>(q - p)*repart(s) and

( - min(repart(bj))<repart(s))<1 - max(repart(ai))

 or arg(eta)=pi*delta or

( - min(repart(bj))<repart(s))<1 - max(repart(ai))

 or arg(eta)<pi*delta
```

where

$delta = s + t - \frac{u-v}{2}$
$eta = 1 - \alpha(v-u) - \mu - \rho$
$\mu = \sum_{j=1}^{q} b_j - \sum_{i=1}^{p} a_i + \frac{p-q}{2} + 1$
$\rho = \sum_{j=1}^{v} dj - \sum_{i=1}^{u} c_i + \frac{u-v}{2} + 1$
s, t, u, v, p, q, α as in eq. D.1

D.1.4 Extending the Tables

The relevant Meijer G representation for any function is found by a pattern-matching process which is carried out on a list of Meijer G-function definitions. This list, although extensive, can never hope to be complete and therefore the user may wish to add more definitions. Definitions can be added by adding the following lines:

```
defint_choose(f(~x),~var => f1(n,x);

symbolic putv(mellin!-transforms!*,n,'
              (() (m n p q) (ai) (bj) (C) (var)));
```

where $f(x)$ is the new function, $i = 1, \ldots, p$, $j = 1, \ldots, q$, C a constant, var =

variable, n = an indexing number, where all expression must be in internal prefix form.

For example when considering $\cos(x)$ we have the *Meijer G representation* –

$$\cos x = \sqrt{\pi}\, G^{10}_{02}\left(\frac{x^2}{4}\;\middle|\;\begin{matrix}\cdot & \cdot \\ 0 & \frac{1}{2}\end{matrix}\right) dx$$

i.e. $m = 1, n = 0, p = 0, q = 2, a_i = \{\}, b_j = \{0, 1/2\}, C = \sqrt{\pi}$. The corresponding *internal definite integration package representation* is

```
defint_choose(cos(~x),~var) => f1(3,x);
```

where 3 is the indexing number corresponding to the 3 in the following formula

```
symbolic putv(mellin!-transforms!*,3,'
              (() (1 0 0 2) () (nil (quotient 1 2))
              (sqrt pi) (quotient (expt x 2) 4)));
```

or the more interesting example of the Bessel function $J_n(x)$:

Meijer G representation –

$$J_n(x) G^{10}_{02}\left(\frac{x^2}{4}\;\middle|\;\begin{matrix}\cdot & \cdot \\ \frac{n}{2} & \frac{-n}{2}\end{matrix}\right) dx$$

Internal definite integration package representation –

```
defint_choose(besselj(~n,~x),~var) => f1(50,x,n);

symbolic putv(mellin!-transforms!*,50,'
        ((n) (1 0 0 2) () ((quotient n 2)
                           (minus (quotient n 2))) 1
                          (quotient (expt x 2) 4)));
```

D.1.5 Acknowledgements

Kerry Gaskell would like to thank Victor Adamchik whose implementation of the definite integration package for REDUCE is vital to this interface.

D.2 The CVIT package

This package provides an alternative method for computing traces of Dirac gamma matrices, based on an algorithm by Cvitanovich that treats gamma matrices as 3-j symbols.

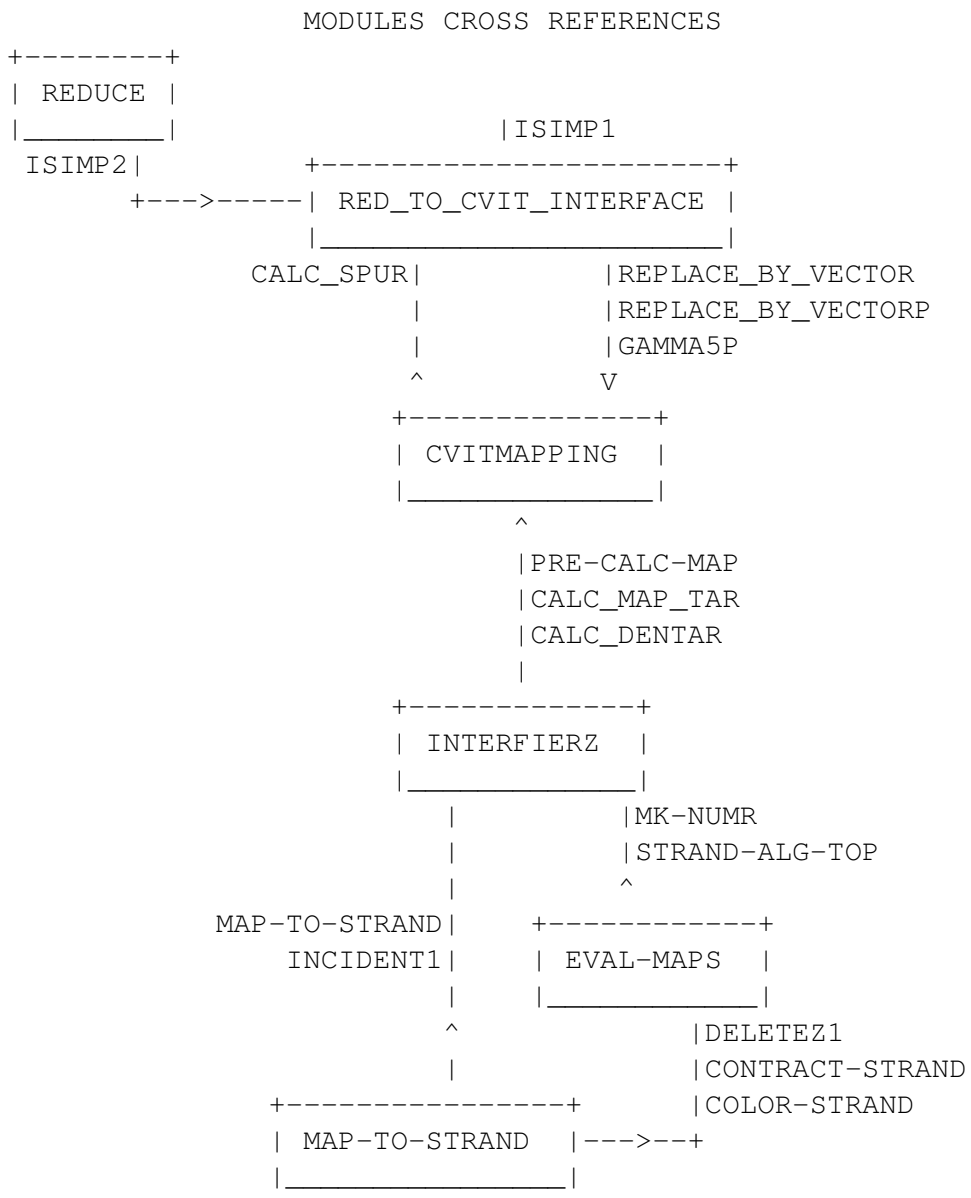
Authors: V.Ilyin, A.Kryukov, A.Rodionov, A.Taranov.

In modern high energy physics the calculation of Feynman diagrams are still very important. One of the difficulties of these calculations are trace calculations. So the calculation of traces of Dirac's γ-matrices were one of first task of computer algebra systems. All available algorithms are based on the fact that gamma-matrices constitute a basis of a Clifford algebra:

$$\{G_m, G_n\} = 2g_{mn}.$$

We present the implementation of an alternative algorithm based on treating of gamma-matrices as 3-j symbols (details may be found in [IKRT89, Ken82]).

The program consists of 5 modules described below.



Module RED_TO_CVIT_INTERFACE

Author: A.P.Kryukov
Purpose:interface REDUCE and CVIT package

RED_TO_CVIT_INTERFACE module is intended for connection of REDUCE with main module of CVIT package. The main idea is to preserve standard REDUCE syntax for high energy calculations. For realization of this we redefine SYMBOLIC PROCEDURE ISIMP1 from HEPhys module of REDUCE system.

After loading CVIT package user may use switch CVIT which is ON by default. If switch CVIT is OFF then calculations of Diracs matrices traces are performed using standard REDUCE facilities. If CVIT switch is ON then CVIT package will be active.

RED_TO_CVIT_INTERFACE module performs some primitive simplification and control input data independently. For example it remove $G_m G_m$, check parity of the number of Dirac matrices in each trace *etc*. There is one principal restriction concerning G5-matrix. There are no closed form for trace in non-integer dimension case when trace include G5-matrix. The next restriction is that if the space-time dimension is integer then it must be even (2,4,6,...). If these and other restrictions are violated then the user get corresponding error message. List of messages is included.

LIST OF IMPORTED FUNCTIONS

Function	From module
ISIMP2	HEPhys
CALC_SPUR	CVITMAPPING

LIST OF EXPORTED FUNCTION

Function	To module
ISIMP1	HEPhys (redefine)
REPLACE_BY_VECTOR	EVAL_MAP
REPLACE_BY_VECTORP	EVAL__MAP
GAMMA5P	CVITMAPPING, EVAL_MAP

Module CVITMAPPING

Author: A.Ya.Rodionov
Purpose: graphs reduction

CVITMAPPING module is intended for diagrams calculation according to Cvitanovic - Kennedy algorithm. The top function of this module CALC_SPUR is called from RED_TO_CVIT_INTERFACE interface module. The main idea of the algorithm consists in diagram simplification according to rules (1.9') and (1.14) from [1]. The input data - trace of Diracs gamma matrices (G-matrices) has a form of a list of identifiers lists with cyclic order. Some of identifiers may be

identical. In this case we assume summation over dummy indices. So trace Sp(GbGr).Sp(GwGbGcGwGcGr) is represented as list ((b r) (w b c w c r)).

The first step is to transform the input data to "map" structure and then to reduce the map to a "simple" one. This transformation is made by function TRANSFORM_MAP_ (top function). Transformation is made in three steps. At the first step the input data are transformed to the internal form - a map (by function PREPARE_MAP_). At the second step a map is subjected to Fierz transformations (1.14) (function MK_SIMPLE_MAP_). At this step of optimization can be maid (if switch CVITOP is on) by function MK_FIRZ_OP. In this case Fierzing starts with linked vertices with minimal distance (number of vertices) between them. After Fierz transformations map is further reduced by vertex simplification routine MK_SIMPLE_VERTEX using (1.9'). Vertices reduced to primitive ones, that is to vertices with three or less edges. This is the last (third) step in transformation from input to internal data.

The next step is optional. If switch CVITBTR is on factorisation of bubble (function FIND_BUBBLES1) and triangle (function FIND_TRIANGLES1) submaps is made. This factorisation is very efficient for "wheel" diagrams and unnecessary for "lattice" diagrams. Factorisation is made recursively by substituting composed edges for bubbles and composed vertices for triangles. So check (function SORT_ATLAS) must be done to test possibility of future marking procedure. If the check fails then a new attempt to reorganize atlas (so we call complicated structure witch consists of MAP, COEFFicient and DENOMinator) is made. This cause backtracking (but very seldom). Backtracking can be traced by turning on switch CVITRACE. FIND_BUBLTR is the top function of this program's branch.

Then atlases must be prepared (top function WORLD_FROM_ATLAS) for final algebraic calculations. The resulted object called "world" consists of edges names list (EDGELIST), their marking variants (VARIANTS) and WORLD1 structure. WORLD1 structure differs from WORLD structure in one point. It contains MAP2 structure instead of MAP structure. MAP2 is very complicated structure and consist of VARIANTS, marking plan and GSTRAND. (GSTRAND constructed by PRE!-CALC!-MAP_ from INTERFIERZ module.) By marking we understand marking of edges with numbers according to Cvitanovic - Kennedy algorithm.

The last step is performed by function CALC_WORLD. At this step algebraic calculations are done. Two functions CALC_MAP_TAR and CALC_DENTAR from INTERFIERZ module make algebraic expressions in the prefix form. This expressions are further simplified by function `REVAL`. This is the REDUCE system general function for algebraic expressions simplification. `REVAL` and `SIMP!*` are the only REDUCE functions used in this module.

There are also some functions for printing several internal structures:

PRINT_ATLAS, PRINT_VERTEX, PRINT_EDGE, PRINT_COEFF, PRINT_DENOM. This functions can be used for debugging.

If an error occur in module CVITMAPPING the error message “ERROR IN MAP CREATING ROUTINES” is displayed. Error has number 55. The switch CVITERROR allows to give full information about error: name of function where error occurs and names and values of function’s arguments. If CVITERROR switch is on and backtracking fails message about error in SORT_ATLAS function is printed. The result of computation however will be correct because in this case factorized structure is not used. This happens extremely seldom.

List of imported function

function	from module
REVAL	REDUCE
SIMP!*	REDUCE
CALC_MAP_TAR	INTERFIERZ
CALC_DENTAR	INTERFIERZ
PRE!-CALC!-MAP_	INTERFIERZ
GAMMA5P	RED_TO_CVIT_INTERFACE

List of exported function

function	to module
CALC_SPUR	REDUCE - CVIT interface

Data structure

```
WORLD     ::=  (EDGELIST,VARIANTS,WORLD1)
WORLD1    ::=  (MAP2,COEFF,DENOM)
MAP2      ::=  (MAPS,VARIANTS,PLAN)
MAPS      ::=  (EDGEPAIR . GSTRAND)
MAP1      ::=  (EDGEPAIR . MAP)
MAP       ::=  list of VERTICES (unordered)
EDGEPAIR  ::=  (OLDEDGELIST . NEWEDGELIST)
COEFF     ::=  list of WORLDS (unordered)
ATLAS     ::=  (MAP,COEFF,DENOM)
GSTRAND   ::=  (STRAND*,MAP,TADPOLES,DELTAS)
VERTEX    ::=  list of EDGEs (with cyclic order)
EDGE      ::=  (NAME,PROPERTY,TYPE)
NAME      ::=  ATOM
```

```
PROPERTY  ::=  (FIRSTPAIR . SECONDPAIR)
TYPE      ::=  T or NIL
-----------------------------------------------
*Define in module MAP!-TO!-STRAND.
```

Modules INTERFIERZ, EVAL_MAPS, AND MAP-TO-STRAND

Author: A.Taranov
Purpose: evaluate single Map

Module INTERFIERZ exports to module CVITMAPPING three functions: PRE-CALC-MAP_, CALC-MAP_TAR, CALC-DENTAR.

Function PRE-CALC-MAP_ is used for preliminary processing of a map. It returns a list of the form (STRAND NEWMAP TADEPOLES DELTAS) where STRAND is strand structure described in MAP-TO-STRAND module. NEWMAP is a map structure without “tadepoles” and “deltas”. “Tadepole” is a loop connected with map with only one line (edge). “Delta” is a single line disconnected from a map. TADEPOLES is a list of “tadepole” submaps. DELTAS is a list (CONS E1 E2) where E1 and E2 are

Function CALC_MAP_TAR takes a list of the same form as returned by PRE-CALC-MAP_, a-list, of the form (... edge . weight ...) and returns a prefix form of algebraic expression corresponding to the map numerator.

Function CALC-DENTAR returns a prefix form of algebraic expression corresponding to the map denominator.

Module EVAL-MAP exports to module INTERFIERZ functions MK-NUMR and STRAND-ALG-TOP.

Function MK-NUMR returns a prefix form for some combinatorial coefficient (Pohgammer symbol).

Function STRAND-ALG-TOP performs an actual computation of a prefix form of algebraic expression corresponding to the map numerator. This computation is based on a “strand” structure constructed from the “map” structure.

Module MAP-TO-STRAND exports functions MAP-TO-STRAND, INCIDENT1 to module INTERFIERZ and functions DELETEZ1, CONTRACT-STRAND, COLOR-STRAND to module EVAL-MAPS.

Function INCIDENT1 is a selector in “strand” structure. DELETEZ1 performs auxiliary optimization of “strand”. MAP-TO-STRAND transforms “map” to “strand” structure. The latter is describe in program module.

CONTRACT-STRAND do strand vertex simplifications of “strand” and

COLOR-STRAND finishes strand generation.

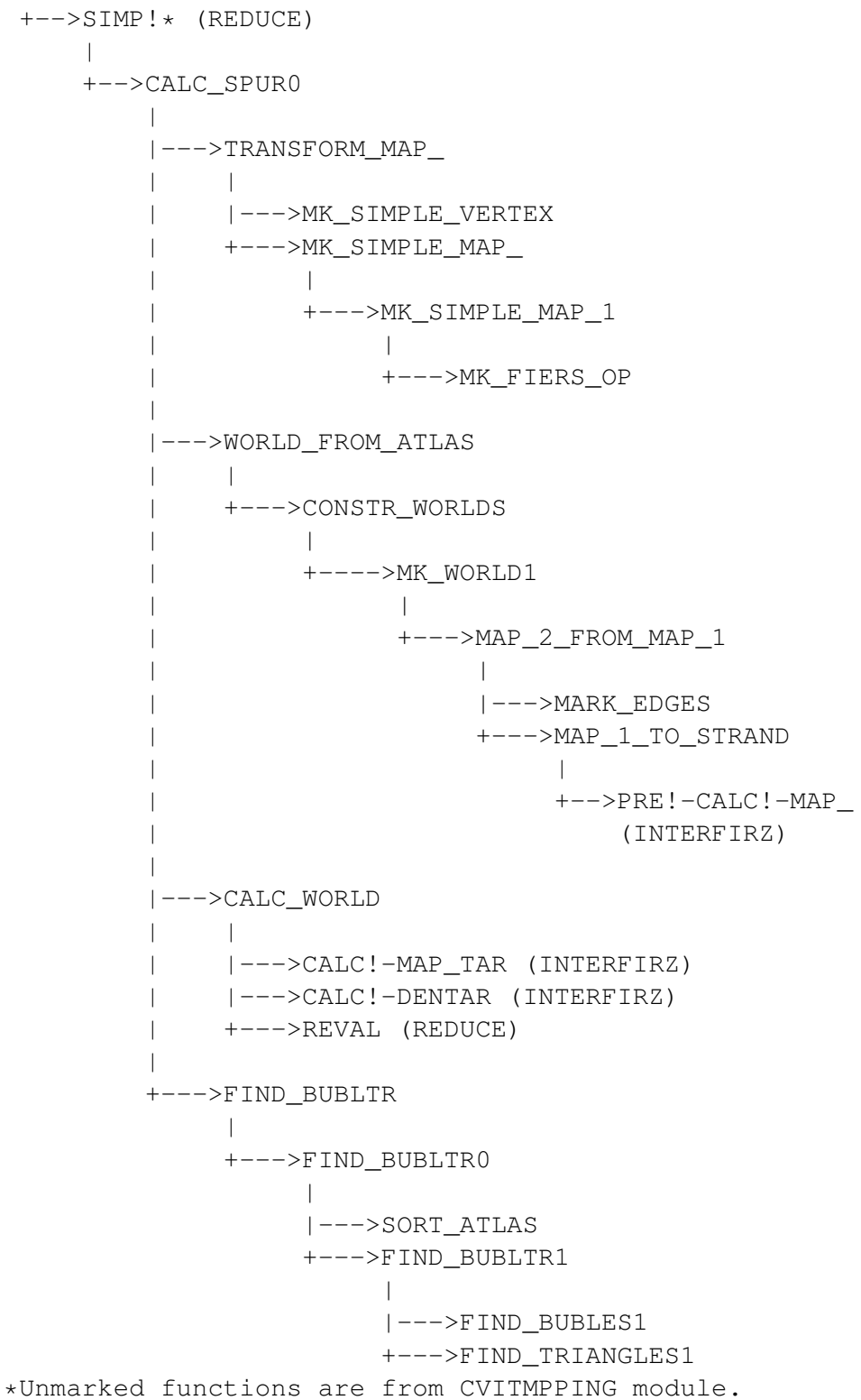
```
           Description of STRAND  data structure.
 STRAND ::=<LIST OF VERTEX>
 VERTEX ::=<NAME> . (<LIST OF ROAD> <LIST OF ROAD>)
 ROAD   ::=<ID> . NUMBER
 NAME   ::=NUMBER
```

LIST OF MESSAGES

- CALC_SPUR: <vecdim> IS NOT EVEN SPACE-TIME DIMENSION The dimension of space-time <vecdim> is integer but not even. Only even numeric dimensions are allowed.
- NOSPUR NOT YET IMPLEMENTED Attempt to calculate trace when NOSPUR switch is on. This facility is not implemented now.
- G5 INVALID FOR VECDIM NEQ 4 Attempt to calculate trace with gamma5-matrix for space-time dimension not equal to 4.
- CALC_SPUR: <expr> HAS NON-UNIT DENOMINATOR The <expr> has non-unit denominator.
- THREE INDICES HAVE NAME <name> There are three indices with equal names in evaluated expression.

```
             List of switches
-----------------------------------------------------
 switch      default    comment
-----------------------------------------------------
 CVIT        ON         If it is on then use Kennedy-
                        Cvitanovic algorithm else use
                        standard facilities.
 CVITOP      OFF        Fierz optimization switch
 CVITBTR     ON         Bubbles and triangles
                        factorisation switch
 CVITRACE    OFF        Backtracking tracing switch
-----------------------------------------------------
```

```
                Functions cross references*.

CALC_SPUR
 |
```



*Unmarked functions are from CVITMPPING module.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z