



Gmsh

C. Geuzaine¹ and J.-F. Remacle²

¹Université de Liège ²Université catholique de Louvain

January 26, 2026

Some background



- I am a professor at the University of Liège in Belgium, where I lead a team of about 15 people in the Montefiore Institute (EECS Dept.), at the intersection of applied math, scientific computing and engineering physics
- My research interests include modeling, analysis, algorithm development, and simulation for problems arising in various areas of engineering and science
- Current applications: low- and high-frequency electromagnetics, geophysics, biomedical problems
- We write quite a lot of codes, some released as open source software:
<https://gmesh.info>, <https://getdp.info>, <https://onelab.info>



Some background

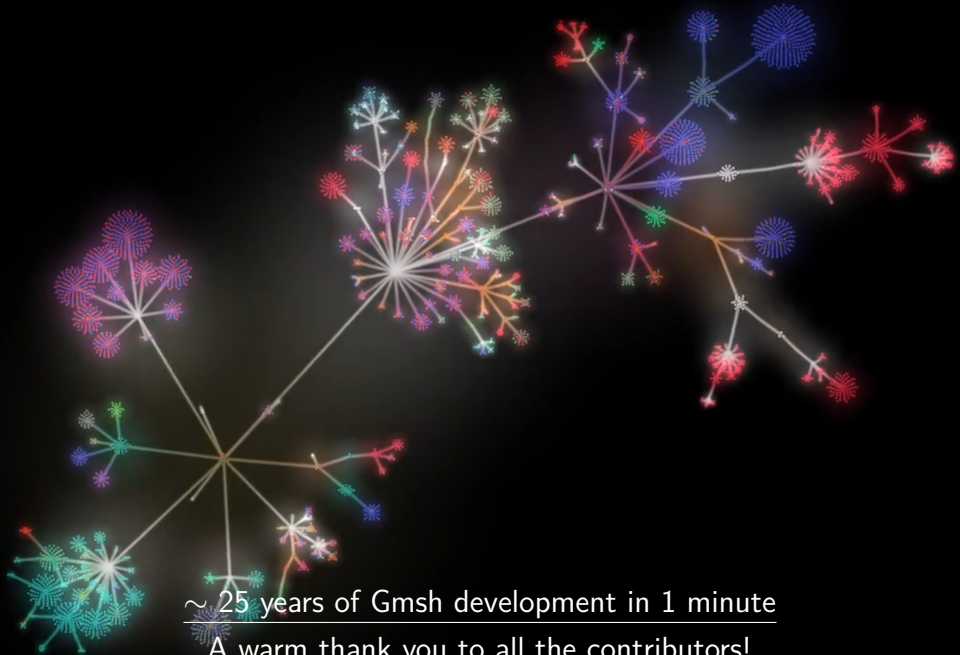
- I am a professor at the Université catholique de Louvain in Belgium, where I lead a team of a dozen researchers in the Institute of Mechanics, Materials and Civil Engineering
- My main research topics are mesh generation and computational mechanics
- I have been co-operating with Christophe for more than 20 years, a fruitful collaboration that has led to the creation of Gmsh

General overview of Gmsh

What is Gmsh?



- Gmsh (<https://gmsh.info>) is an open source 3D finite element mesh generator with a built-in CAD engine and post-processor
- Includes a graphical user interface (GUI) and can drive any simulation code through ONELAB
- Today, Gmsh represents about 400k lines of C++ code
 - still same 2 core developers; about 100 with ≥ 1 commit
 - about 3,000 registered users on the development site <https://gitlab.onelab.info>
 - about 20,000 downloads per month (70% Windows)
 - about 1,000 citations per year – the Gmsh paper is cited about 10,000 times
 - Gmsh has become one of the most popular open source finite element mesh generators worldwide



~ 25 years of Gmsh development in 1 minute
A warm thank you to all the contributors!

A little bit of history

- Gmsh was started in 1996, as a side project
- 1998: First public release
- 2003: Open Sourced under GNU GPL
- 2006: OpenCASCADE integration (Gmsh 2)
- 2009: IJNME paper and switch to CMake
- 2012: Curvilinear meshing and quad meshing
- 2013: Homology and ONELAB solver interface
- 2015: Multi-Threaded 1D and 2D meshing (coarse-grained)
- 2017: Boolean operations and switch to Git (Gmsh 3)
- 2018: C++, C, Python and Julia API (Gmsh 4)
- 2019: Multi-Threaded 3D meshing (fine-grained), STL remeshing
- 2021: GmshFEM, Quasi-structured quad meshing
- 2023: GmshDDM, Fortran API

Strategic choices

- Design goals: fast, light and user-friendly
 - Written in simple C++
 - GUIs: FLTK (desktop), UIKit (iOS), Android
 - OpenGL graphics
 - Highly portable (OSes & compilers)
 - Easy to distribute & install: zero dependencies on installation
- Handling of numerous third party libraries
 - Build system based on CMake – everything is optional
 - Some libs integrated and redistributed directly in gmsh/contrib (HXT, BAMG, Concorde, ...)
- Funding
 - Hobby until 2006, then industry, Wallonia, Belgium & EU

Strategic choices

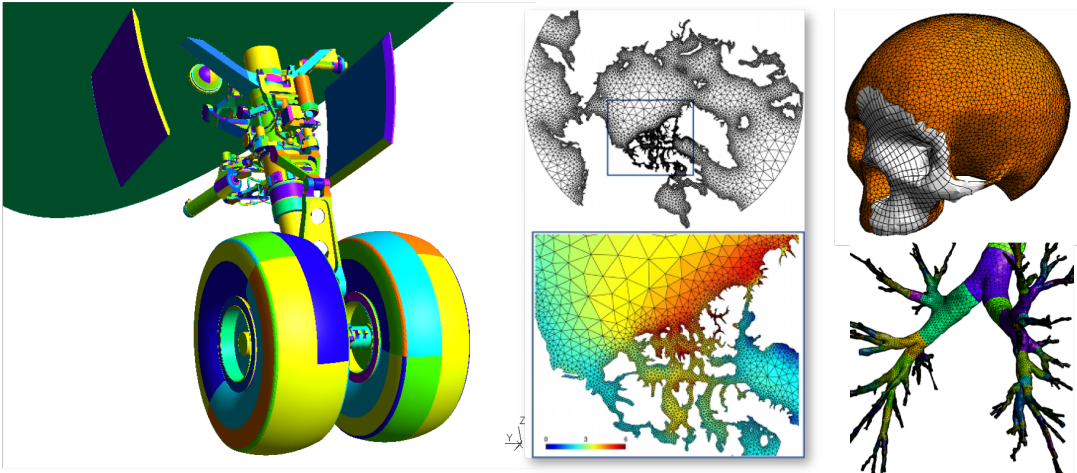
- Community infrastructure
 - Our own (using GitLab) to enable public/private parts (<https://gitlab.onelab.info/gmsh/gmsh>)
 - Continuous integration and delivery (CI/CD) of Gmsh app and Gmsh SDK on Windows, Linux and macOS
 - Web site (<https://gmsh.info>) with documentation, tutorials, etc.
 - Scientific aspects of algorithms detailed in journal papers
- Licensing
 - Gmsh is distributed under the GNU General Public License v2 or later, with exceptions to allow for easier linking with external libraries
 - We double-license to enable embedding in commercial codes

Basic concepts

- Gmsh is based around four modules: Geometry, Mesh, Solver and Post-processing
- Gmsh can be used at 3 levels
 - Through the GUI
 - Through the dedicated `.geo` scripting language
 - Through the C++, C, Python, Julia and Fortran API
- Main characteristics
 - All algorithms are written in terms of abstract model entities, using a Boundary REPresentation (BREP) approach
 - Gmsh never translates from one CAD format to another; it directly accesses each CAD kernel API (OpenCASCADE, Built-in, ...)

Basic concepts

The goal is to deal with very different underlying data representations in a transparent manner



Geometry module

Under the hood, 4 types of model entities are defined:

1. Model points G_i^0 that are topological entities of dimension 0
2. Model curves G_i^1 that are topological entities of dimension 1
3. Model surfaces G_i^2 that are topological entities of dimension 2
4. Model volumes G_i^3 that are topological entities of dimension 3

Geometry module

- Model entities are topological entities, i.e., they only deal with adjacencies in the model; a bi-directional data structure represents the graph of adjacencies

$$G_i^0 \rightleftharpoons G_i^1 \rightleftharpoons G_i^2 \rightleftharpoons G_i^3$$

- Any model is able to build its list of adjacencies of any dimension using local operations
- The BRep is extended with non-manifold features: adjacent entities, and *embedded* (internal) entities
- Model entities can be either CAD entities (e.g. from the built-in or OpenCASCADE kernel) or *discrete* entities (defined by a mesh, e.g. STL)

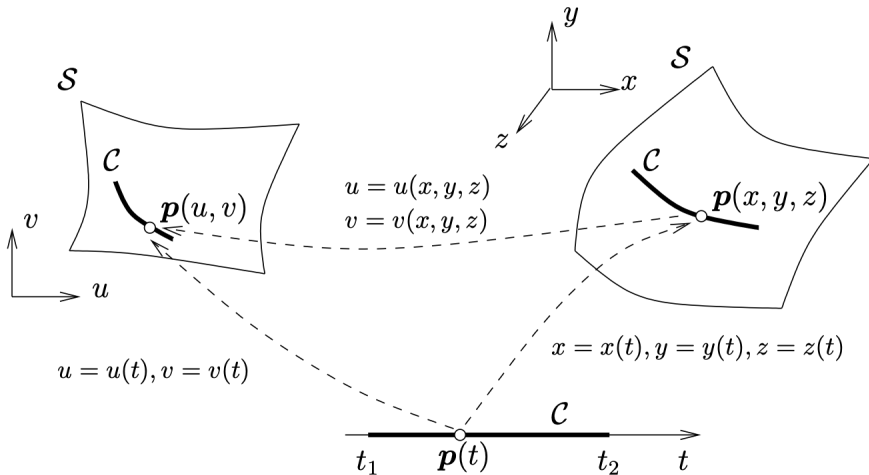
Geometry module

The geometry of a CAD model entity depends on the solid modeler kernel for its underlying representation. Solid modelers usually provide a parametrization of the shapes, i.e., a mapping:

$$\mathbf{p} \in R^d \mapsto \mathbf{x} \in R^3$$

1. The geometry of a model point G_i^0 is simply its 3-D location $\mathbf{x}_i = (x_i, y_i, z_i)$
2. The geometry of a model curve G_i^1 is its underlying curve \mathcal{C}_i with its parametrization $\mathbf{p}(t) \in \mathcal{C}_i, t \in [t_1, t_2]$
3. The geometry of a model surface G_i^2 is its underlying surface \mathcal{S}_i with its parametrization $\mathbf{p}(u, v) \in \mathcal{S}_i$
4. The geometry associated to a model volume is R^3

Geometry module

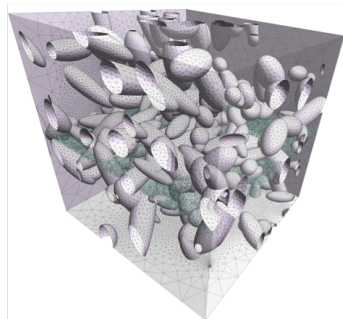
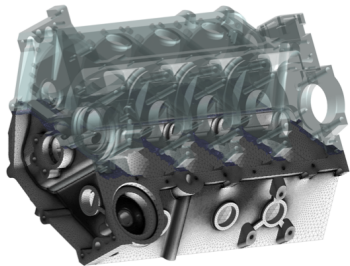


Point \mathbf{p} located on a curve \mathcal{C} that is itself embedded in a surface \mathcal{S}

Geometry module

Operations on CAD model entities are performed directly within their respective CAD kernels:

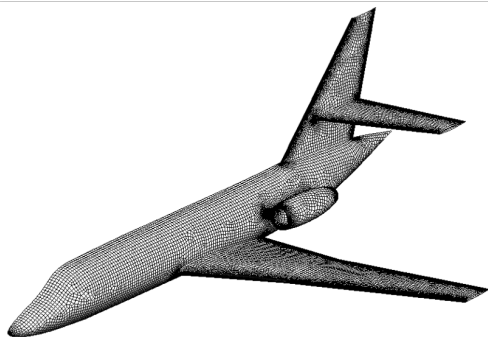
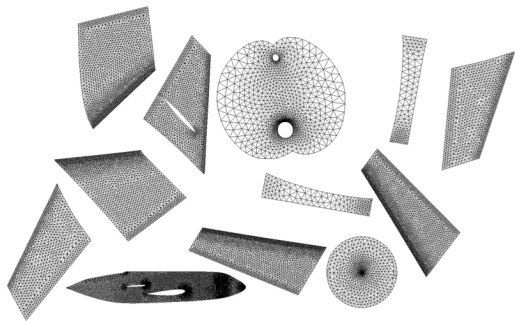
- There is no common internal geometrical representation
- Rather, Gmsh directly performs the operations (translation, rotation, intersection, union, fragments, ...) on the native geometrical representation using each CAD kernel's own API



Geometry module

Discrete model entities are defined by a mesh (e.g. STL):

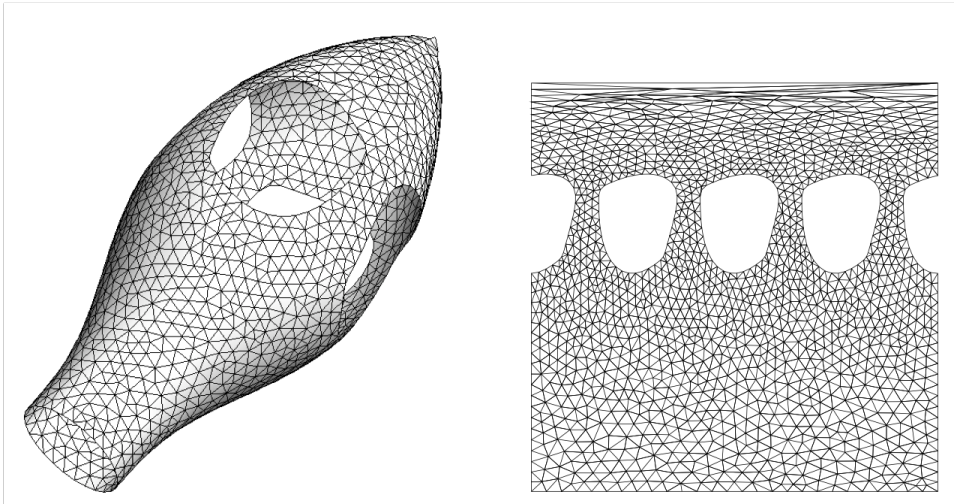
- They can be equipped with a geometry through a *reparametrization* procedure
- The parametrization is then used for meshing, in exactly the same way as for CAD entities



Mesh module

- A (conformal) finite element mesh of a model is a tessellation of its geometry by geometrical elements of various shapes (lines, triangles, quadrangles, tetrahedra, prisms, hexahedra, pyramids), arranged in such a way that if two of them intersect, they do so along a face, an edge or a node, and not otherwise
- Gmsh implements several meshing algorithms with specific characteristics
 - 1D, 2D and 3D
 - Structured, unstructured and hybrid
 - Isotropic and anisotropic
 - Straight-sided and curved
 - From standard CAD data or from STL through reparametrization
- Built-in interfaces to external mesh generators (BAMG [F. Hecht, 1998], MMG3D [C. Dobrzynski et al., 2012], Netgen [J. Schoeberl, 1997])

Mesh module



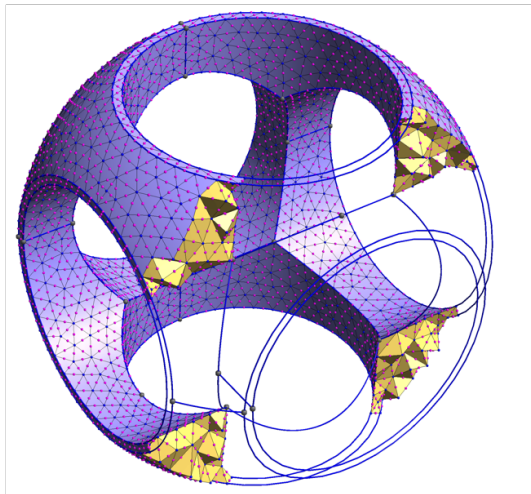
Typical CAD kernel idiosyncrasies: seam edges and degenerated edges

Mesh module

- Mesh data is made of *elements* (points, lines, triangles, quadrangles, tetrahedra, hexahedra, ...) defined by an ordered list of their *nodes*
- Elements and nodes are stored (*classified*) in the model entity they discretize:
 - A model point will thus contain a mesh element of type point, as well as a mesh node
 - A model curve will contain line elements as well as its interior nodes, while its boundary nodes will be stored in the bounding model points
 - A model surface will contain triangular and/or quadrangular elements and all the nodes not classified on its boundary or on its embedded entities (curves and points)
 - A model volume will contain tetrahedra, hexahedra, etc. and all the nodes not classified on its boundary or on its embedded entities (surfaces, curves and points)

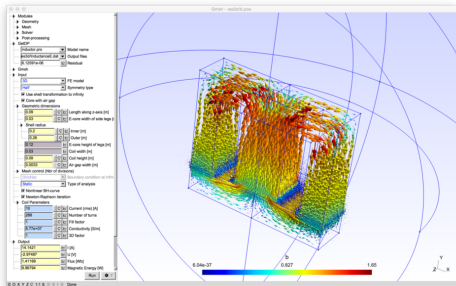
Mesh module

This mesh data structure allows to easily and efficiently handle the creation, modification and destruction of conformal finite element meshes



Solver module

- Gmsh implements a ONELAB (<https://onelab.info>) server to pilot external solvers, called “clients”
- Example client: GetDP finite element solver (<https://getdp.info>)
 - The ONELAB interface allows to call such clients and have them share parameters and modeling information
 - Parameters are directly controllable from the GUI



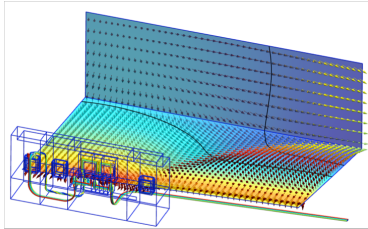
Solver module

- The implementation is based on a client-server model, with a server-side database and local or remote clients communicating in-memory or through TCP/IP sockets
 - Contrary to most solver interfaces, the ONELAB server has no a priori knowledge about any specifics (input file format, syntax, ...) of the clients
 - This is made possible by having any simulation preceded by an analysis phase, during which the clients are asked to upload their parameter set to the server
 - The issues of completeness and consistency of the parameter sets are completely dealt with on the client side: the role of ONELAB is limited to data centralization, modification and re-dispatching

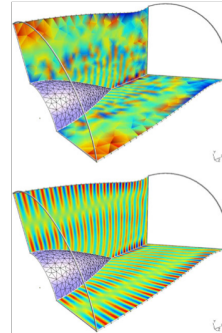
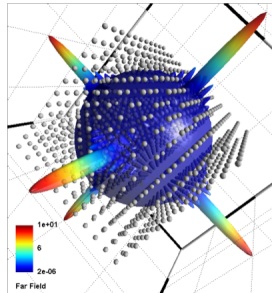
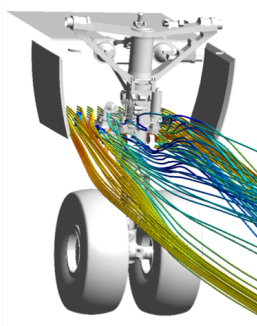
Post-processing module

- Post-processing data is made of *views*
- A view stores both display *options* and *data* (unless the view is an *alias* of another view)
- View data can contain several *steps* (e.g. to store time series) and can be either linked to one or more models (*mesh-based* data, as stored in `.msh` or `.med` files) or independent from any model (*list-based* data, as stored in parsed `.pos` files)
- Data is interpolated through arbitrary polynomial interpolation schemes; automatic mesh refinement is used for adaptive visualization of high-order views
- Various *plugins* exist to create and modify views

Post-processing module



- Cuts, iso-curves and vectors
- Elevation maps
- Streamlines
- Adaptive high-order visualization



Application Programming Interface

Gmsh 4 introduced a stable Application Programming Interface (API) for C++, C, Python, Julia and Fortran, with the following design goals:

- Allow to do everything that can be done in .geo scripts
 - ... and then much more!
- Be robust, in particular to wrong input data
- Be efficient; but still allow to do simple things, simply
- Be maintainable over the long run

Application Programming Interface

To achieve these goals the Gmsh API

- is purely functional
- only uses basic types from the target language (C++, C, Python, Julia and Fortran)
- is automatically generated from a master API description file
- is documented

Application Programming Interface

In addition to CAD creation and meshing, the API can be used to

- Access mesh data (`getNodes`, `getElements`)
- Generate interpolation (`getBasisFunctions`) and integration (`getJacobians`) data to build Finite Element and related solvers (see e.g. [gmsh/examples/api/poisson.py](https://gmsh.org/doc/10.4.0/python_examples/poisson.py))
- Create post-processing views
- Run the graphical user-interface
- Build custom graphical user-interfaces, e.g. for domain-specific codes (see [gmsh/examples/api/prepro.py](https://gmsh.org/doc/10.4.0/python_examples/prepro.py) or [gmsh/examples/api/custom_gui.py](https://gmsh.org/doc/10.4.0/python_examples/custom_gui.py)) or co-post-processing via ONELAB

Application Programming Interface

We publish a binary Software Development Toolkit (SDK):

- Continuously delivered (for each commit in master), like the Gmsh app
- Contains the dynamic Gmsh library together with the corresponding C++/C header files, and Python, Julia and Fortran modules

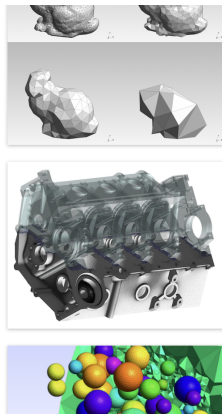
Download

Gmsh is distributed under the terms of the [GNU General Public License \(GPL\)](#):

- **Current stable release (version 4.14.1, 2 September 2025):**
 - Download Gmsh for [Windows](#), [Linux](#), [macOS \(x86\)](#) or [macOS \(ARM\)](#) *
 - Download the [source code](#)
 - Download the Software Development Kit (SDK) for [Windows](#), [Linux](#), [macOS \(x86\)](#) or [macOS \(ARM\)](#) *
 - Download both Gmsh and the SDK with pip: `'pip install --upgrade gmsh'`

Make sure to read the [tutorial](#) and the [FAQ](#) before sending questions or bug reports.

- Development version:
 - Download the latest automatic Gmsh snapshot for [Windows](#), [Linux](#), [macOS \(x86\)](#) or [macOS \(ARM\)](#) *
 - Download the latest automatic [source code](#) snapshot
 - Download the latest automatic SDK snapshot for [Windows](#), [Linux](#), [macOS \(x86\)](#) or [macOS \(ARM\)](#) *
 - Access the Git repository: `'git clone https://gitlab.onelab.info/gmsh/gmsh.git'`
 - Download the latest automatic snapshot of both Gmsh and the SDK with pip: `'pip install -i https://gmsh.info/python-packages-dev --force-reinstall --no-cache-dir gmsh'` (on Linux systems without X windows, use `python-packages-dev-nox` instead of `python-packages-dev`)
- All versions: [binaries](#) and [sources](#)



Download

To download the Gmsh SDK:

- Simplest way:

```
pip install --upgrade gmsh
```

- For the latest development version:

```
pip install -i https://gmsh.info/python-packages-dev  
--force-reinstall --no-cache-dir gmsh
```

- All other options: go to <https://gmsh.info>

First model and mesh

First .geo script

Save this script as a text file `file.geo`:

```
lc = 0.1; // target mesh size at points
Point(1) = {0, 0, 0, lc};
Point(2) = {1, 0, 0, lc};
Point(3) = {1, 1, 0, lc};
Point(4) = {0, 1, 0, lc};
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};
Curve Loop(1) = {1, 2, 3, 4};
Plane Surface(1) = {1};
```

- Run the script interactively with `gmsh file.geo`
- Or launch the Gmsh app and open the script with the File/Open menu
- Or create a mesh in batch mode with `gmsh file.geo -2`

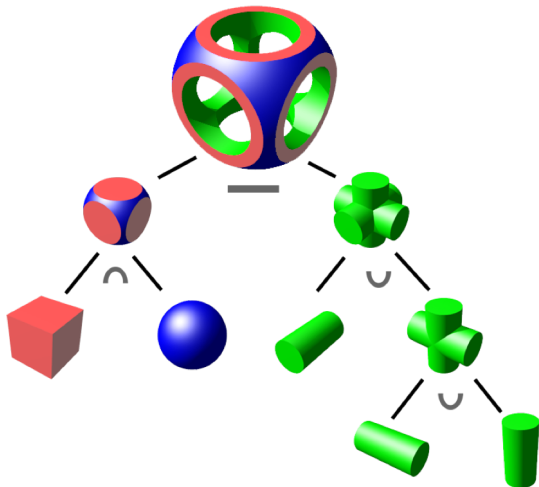
Same in Python using the Gmsh API

Save this as a Python script file .py:

```
import gmsh
gmsh.initialize()
lc = 0.1 # mesh size at points
p1 = gmsh.model.geo.addPoint(0, 0, 0, lc)
p2 = gmsh.model.geo.addPoint(1, 0, 0, lc)
p3 = gmsh.model.geo.addPoint(1, 1, 0, lc)
p4 = gmsh.model.geo.addPoint(0, 1, 0, lc)
l1 = gmsh.model.geo.addLine(p1, p2)
l2 = gmsh.model.geo.addLine(p2, p3)
l3 = gmsh.model.geo.addLine(p3, p4)
l4 = gmsh.model.geo.addLine(p4, p1)
c1 = gmsh.model.geo.addCurveLoop([l1, l2, l3, l4])
gmsh.model.geo.addPlaneSurface([c1])
gmsh.model.geo.synchronize() # sync CAD kernel data to model
gmsh.fltk.run() # launch the GUI
gmsh.finalize()
```

Run with `python3 file.py`

Constructive Solid Geometry (CSG)



https://en.wikipedia.org/wiki/Constructive_solid_geometry

CSG with a .geo script

```

SetFactory("OpenCASCADE"); // use OpenCASCADE CAD kernel

R = DefineNumber [ 1.4 , Min 0.1, Max 2, Step 0.01,
                  Name "Parameters/Box dimension" ];
Rs = DefineNumber [ R*.7 , Min 0.1, Max 2, Step 0.01,
                  Name "Parameters/Cylinder radius" ];
Rt = DefineNumber [ R*1.25, Min 0.1, Max 2, Step 0.01,
                  Name "Parameters/Sphere radius" ];

Box(1) = {-R,-R,-R, 2*R,2*R,2*R}; // explicit entity tag

Sphere(2) = {0,0,0, Rt};

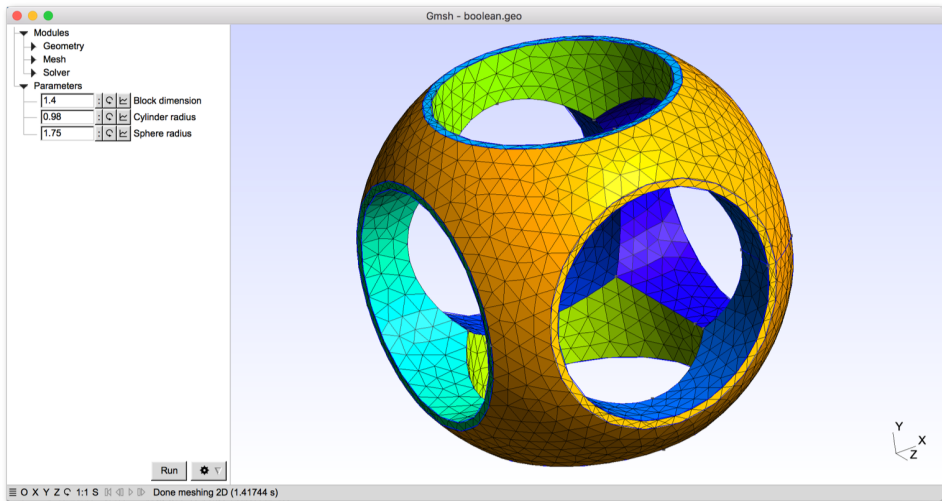
BooleanIntersection(3) = { Volume{1}; Delete; }{ Volume{2}; Delete; };
                        // delete object and tool

Cylinder(4) = {-2*R,0,0, 4*R,0,0, Rs};
Cylinder(5) = {0,-2*R,0, 0,4*R,0, Rs};
Cylinder(6) = {0,0,-2*R, 0,0,4*R, Rs};

BooleanUnion(7) = { Volume{4}; Delete; }{ Volume{5,6}; Delete; };
BooleanDifference(8) = { Volume{3}; Delete; }{ Volume{7}; Delete; };

```

CSG with a .geo script



`gmsch/examples/boolean/boolean.geo`

CSG with the Python API

Same example, but using the Python API:

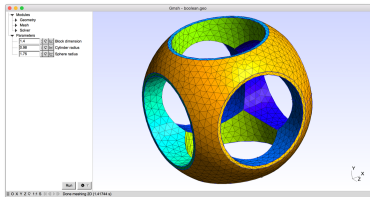
```
import gmsh

gmsh.initialize()
gmsh.model.add("boolean")

R = 1.4; Rs = R*.7; Rt = R*1.25

gmsh.model.occ.addBox(-R,-R,-R, 2*R,2*R,2*R, 1)
gmsh.model.occ.addSphere(0,0,0,Rt, 2)
gmsh.model.occ.intersect([(3, 1)], [(3, 2)], 3)
gmsh.model.occ.addCylinder(-2*R,0,0, 4*R,0,0, Rs, 4)
gmsh.model.occ.addCylinder(0,-2*R,0, 0,4*R,0, Rs, 5)
gmsh.model.occ.addCylinder(0,0,-2*R, 0,0,4*R, Rs, 6)
gmsh.model.occ.fuse([(3, 4)], (3, 5)], [(3, 6)], 7)
gmsh.model.occ.cut([(3, 3)], [(3, 7)], 8)

gmsh.model.occ.synchronize()
gmsh.model.mesh.generate(3)
gmsh.fltk.run()
gmsh.finalize()
```



[gmsh/exemples/api/boolean.py](https://gmsh.info/exemples/api/boolean.py)

CSG with the C++ API

... or using the C++ API:

```
#include <gmsh.h>

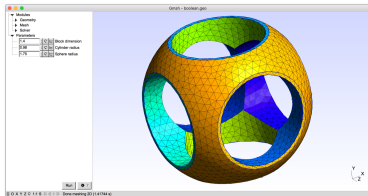
int main(int argc, char **argv)
{
    gmsh::initialize(argc, argv);
    gmsh::model::add("boolean");

    double R = 1.4, Rs = R*.7, Rt = R*1.25;

    std::vector<std::pair<int, int> > ov;
    std::vector<std::vector<std::pair<int, int> > > ovv;
    gmsh::model::occ::addBox(-R,-R,-R, 2*R,2*R,2*R, 1);
    gmsh::model::occ::addSphere(0,0,0,Rt, 2);
    gmsh::model::occ::intersect({{3, 1}}, {{3, 2}}, ov, ovv, 3);
    gmsh::model::occ::addCylinder(-2*R,0,0, 4*R,0,0, Rs, 4);
    gmsh::model::occ::addCylinder(0,-2*R,0, 0,4*R,0, Rs, 5);
    gmsh::model::occ::addCylinder(0,0,-2*R,0,0,4*R, Rs, 6);
    gmsh::model::occ::fuse({{3, 4}}, {{3, 5}}, {{3, 6}}, ov, ovv, 7);
    gmsh::model::occ::cut({{3, 3}}, {{3, 7}}, ov, ovv, 8);

    gmsh::model::occ::synchronize();

    gmsh::model::mesh::generate(3);
    gmsh::fltk::run();
    gmsh::finalize();
    return 0;
}
```



[gmsh/exemples/api/boolean.cpp](https://gmsh.info/exemples/api/boolean.cpp)

Multithreaded meshing in Gmsh

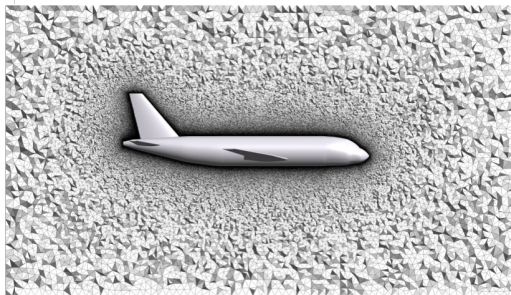
The meshing pipeline is multithreaded using OpenMP:

- 1D and 2D algorithms are multithreaded using coarse-grained approach, i.e. several curves/surfaces are meshed concurrently
- The new 3D Delaunay-based algorithm (HXT) is multi-threaded using a fine-grained approach based on Hilbert curve (more precisely a Moore curve) sort

You can specify the number of threads with the `General.NumThreads` option (set it to 0 to use the system value), or with the `-nt` command line switch:

```
gmsh file.geo -3 -nt 8 -algo hxt
```

Multithreaded meshing in Gmsh



Truck tire

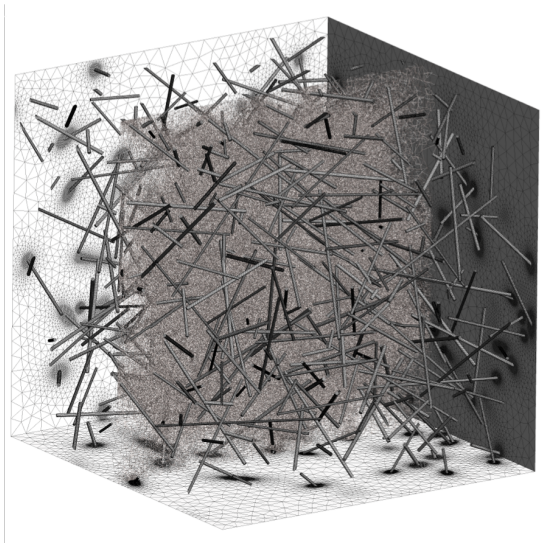
# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	123 640 429	75.9	259.7	364.7
2	123 593 913	74.5	166.8	267.1
4	123 625 696	74.2	107.4	203.6
8	123 452 318	74.2	95.5	190.0

Aircraft

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	672 209 630	45.2	1348.5	1418.3
2	671 432 038	42.1	1148.9	1211.5
8	665 826 109	39.6	714.8	774.8
64	664 587 093	38.7	322.3	380.9
127	663 921 974	38.1	255.0	313.3

AMD EPYC 2x 64-core

Multithreaded meshing in Gmsh



100 thin fibers

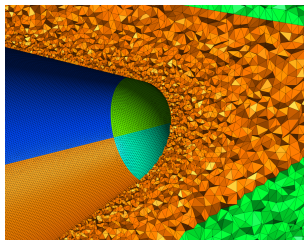
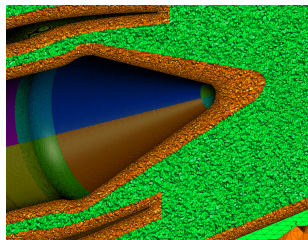
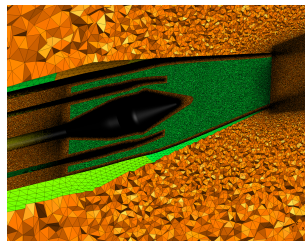
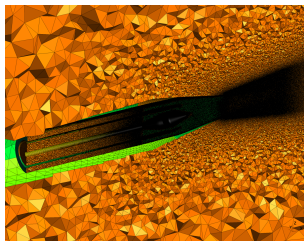
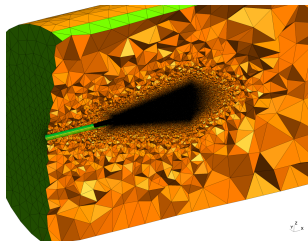
# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	325 611 841	3.1	492.1	497.2
2	325 786 170	2.9	329.7	334.3
4	325 691 796	2.8	229.5	233.9
8	325 211 989	2.7	154.6	158.7
16	324 897 471	2.8	96.8	100.9
32	325 221 244	2.7	71.7	75.8
64	324 701 883	2.8	55.8	60.1
127	324 190 447	2.9	47.6	52.0

500 thin fibers

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	723 208 595	18.9	1205.8	1234.4
2	723 098 577	16.0	780.3	804.8
4	722 664 991	86.6	567.1	659.8
8	722 329 174	15.8	349.1	370.1
16	723 093 143	15.6	216.2	236.5
32	722 013 476	15.6	149.7	169.8
64	721 572 235	15.9	119.7	140.4
127	721 591 846	15.9	114.2	135.2

AMD EPYC 2x 64-core

Multithreaded meshing in Gmsh

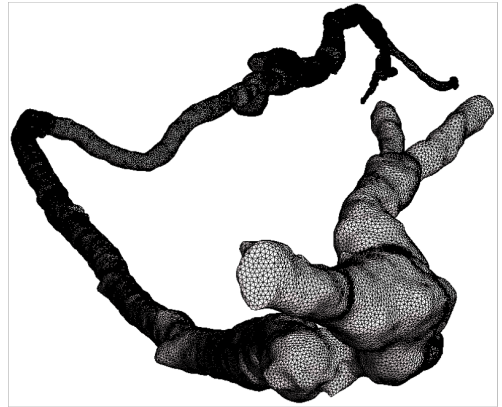
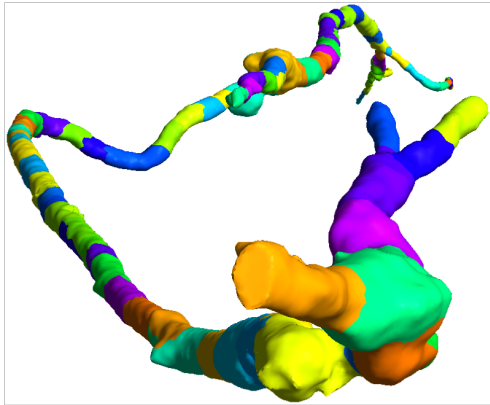


420 million tetrahedra

# threads	Wall Time (s)
1	11888
4	4744
8	2405
16	1326
64	924

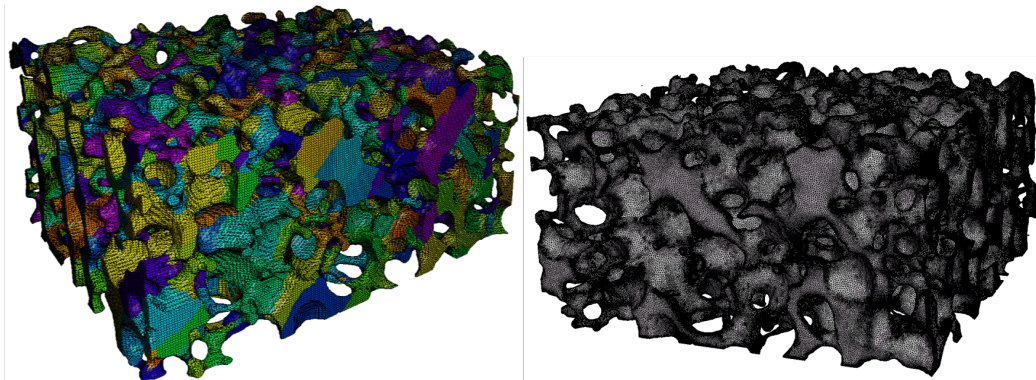
Adapted nozzle mesh, AMD EPYC Rome 64-core at 2.9 GHz

STL remeshing



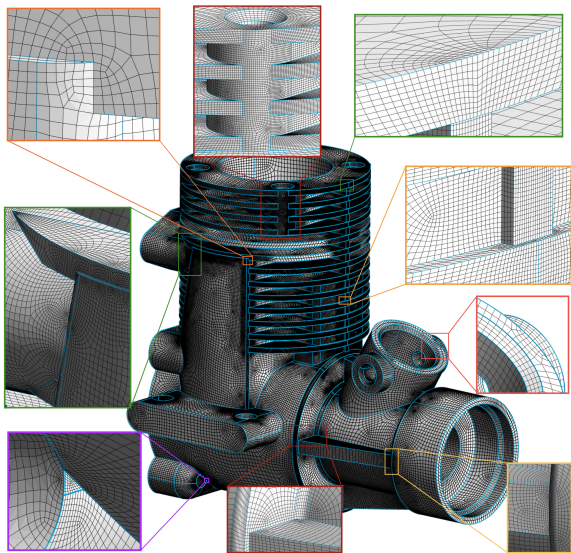
CT scan of an artery: automatic remeshing through reparametrization

STL remeshing



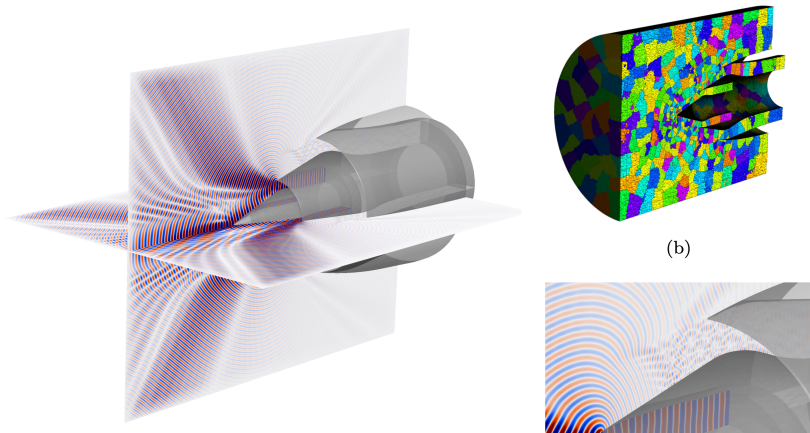
Remeshing of an X-ray tomography image of a silicon carbide foam by P. Duru, F. Muller and L. Selle (IMFT, ERC Advanced Grant SCIROCCO): 1,802 patches created for reparametrization

Quasi-structured quad meshing



- “Block” model: 533 surfaces, 1584 curves, 261.5k vertices, 261.6k quads
- Average SICN quality: 0.87 (minimum: 0.11)
- 58 sec. (initial unstructured quad mesh) + 33 sec. (quasi-structured improvement) on Intel Core i7 4 cores
- Quasi-structured improvement reduces the number of irregular from 14.4k to 3.6k

Gmsh{FEM,DDM} solver libraries

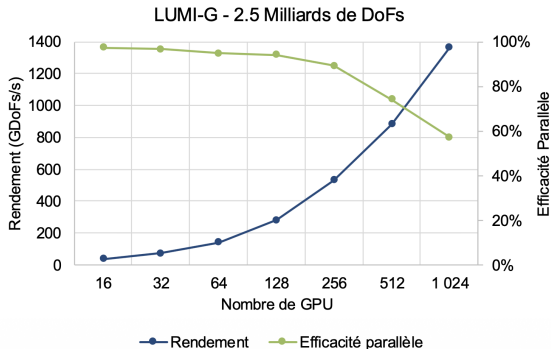
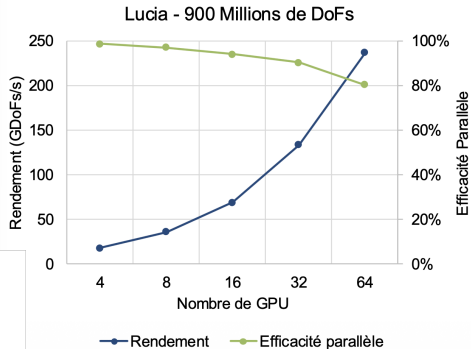


Cores (MPI×threads)	unknowns	nnz	peak memory	pre-pro	GMRES	It
4096×16	1.3B	96B	18.4Gb	1min	14min	555

Acoustic noise from turbofan engine exhaust (4096 partitions, LUMI)

GPU-accelerated Gmsh{DDM,DG}

- GmshDDM is being ported to GPU to speed up the iterative process
- We have also added multi-GPU support to our time-domain Discontinuous Galerkin code GmshDG for Maxwell:



Going further

Going further

Download Gmsh: <https://gmsh.info>

Gmsh reference manual: <https://gmsh.info/doc/texinfo/>

For fun, download the ONELAB app:

- [Google Play Store](#) (Android)
- [Apple AppStore](#) (iOS)

It contains a full-featured version of Gmsh + the finite element solver GetDP

... so you can impress your friends by solving finite element models on your smartphone!

