Towards (very) large scale finite element mesh generation with Gmsh

Christophe Geuzaine

Université de Liège

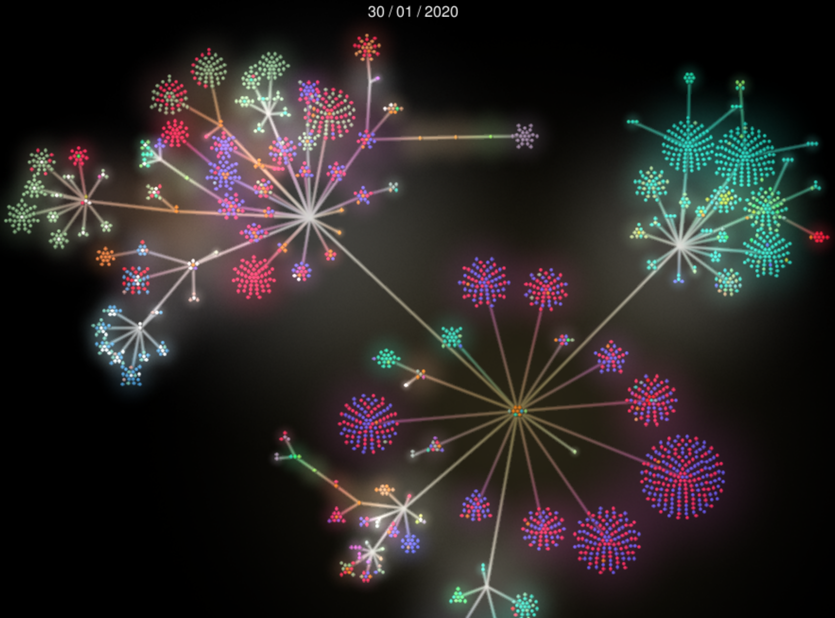
ELEMENT workshop, October 20, 2020

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
- Our 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: `http://gmsh.info`, `http://getdp.info`, `http://onelab.info`

What is Gmsh?

- Gmsh (http://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 500k lines of C++ code
  - still same 2 core developers (Jean-Francois Remacle from UCLouvain and myself); about 100 with ≥ 1 commit
  - about 1,000 people on mailing lists
  - about 10,000 downloads per month (70% Windows)
  - about 500 citations per year – the main Gmsh paper is cited about 4,500 times
  - Gmsh has probably become one of the most popular (open source) finite element mesh generators?



$\sim$ 20 years of Gmsh development in 1 minute

A warm thank you to all the contributors!

A little bit of history

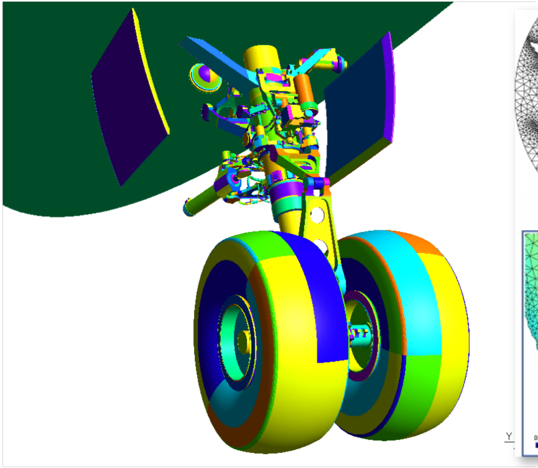
- Gmsh was started in 1996, as a side project
- 1998: First public release
- 2003: Open Sourced under GNU GPL
- 2006: OpenCASCADE integration (Gmsh 2)
- 2009: Gmsh paper and switch to CMake
- 2012: First curved meshing and quad meshing developments
- 2013: Homology and ONELAB solver interface
- 2015: 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), robust STL remeshing

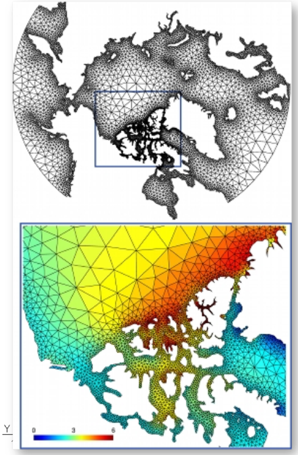
Basic concepts

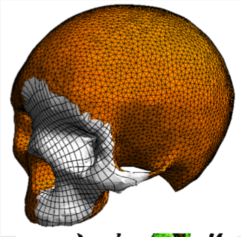
- Gmsh is based around four modules: Geometry, Mesh, Solver and Post-processing
- Gmsh can be used at 3 levels
  - Through the GUI
  - Through the dedicated .geo language
  - Through the C++, C, Python and Julia API
- Main characteristics
  - All algorithms are written in terms of abstract model entities, 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, ...)

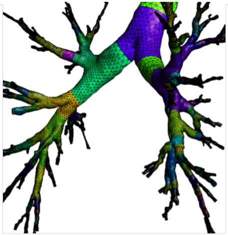
Basic concepts

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









Recent developments

- Application Programming Interface (API)
- Multi-threaded meshing
- Robust STL remeshing based on parametrizations

Application Programming Interface

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

- Allow to do everything that can be done in .geo files
  - ... and then much more!
- Be robust, in particular to wrong input data (i.e. “never crash”)
- Be efficient; but still allow to do simple things, simply
- Be maintainable over the long run

Application Programming Interface

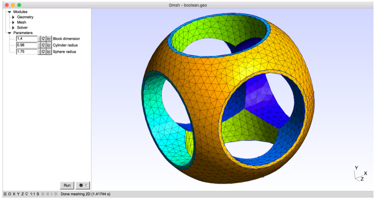
To achieve these goals the Gmsh API

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

Application Programming Interface

A simple example written using the Python API:

```python
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()
gsmh.model.mesh.generate(3)
gmsh.fltk.run()
gmsh.finalize()
```

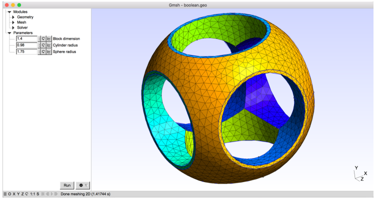


gmsh/demos/api/boolean.py

Application Programming Interface

... or using the C++ API:

```cpp
#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/demos/api/boolean.cpp

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/demos/api/poisson.py`)
- Create post-processing views
- Run the GUI, or build custom GUIs, e.g. for domain-specific codes (see `gmsh/demos/api/custom_gui.py`) or co-post-processing via ONELAB

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 and Julia modules

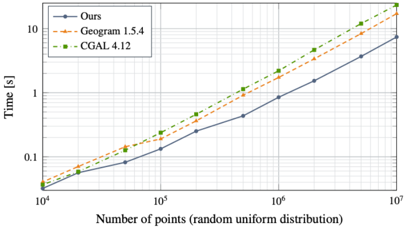
Multi-threaded meshing

Meshing is multi-threaded 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 is multi-threaded using a fine-grained approach.

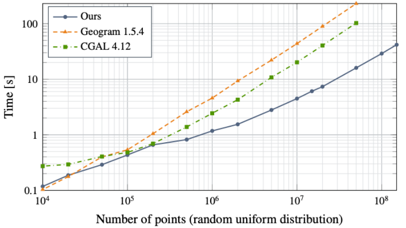
You need to recompile Gmsh with `-DENABLE_OPENMP=1` to enable this; then e.g.
`gmsh file.geo -3 -nt 8 -algo hxt`

Multi-threaded meshing



(a) 4-core Intel® Core™ i7-6700HQ CPU.

# vertices	10^4	10^5	10^6	10^7
Ours	0.032	0.13	0.85	7.40
Geogram	0.041	0.19	1.73	17.11
CGAL	0.037	0.24	2.20	23.37



(b) 64-core Intel® Xeon Phi™ 7210 CPU.

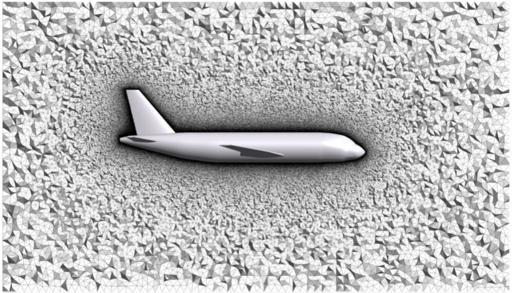
# vertices	10^4	10^5	10^6	10^7	10^8
Ours	0.11	0.43	1.17	4.48	28.95
Geogram	0.10	0.54	4.58	43.70	/
CGAL	0.27	0.48	2.44	20.15	/

[C. Marot et al., IJNME, 2019]

Multi-threaded meshing

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

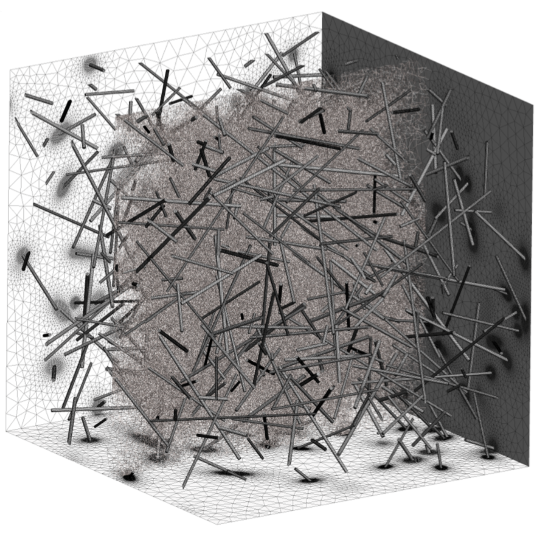


Aircraft

# 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

Multi-threaded meshing



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

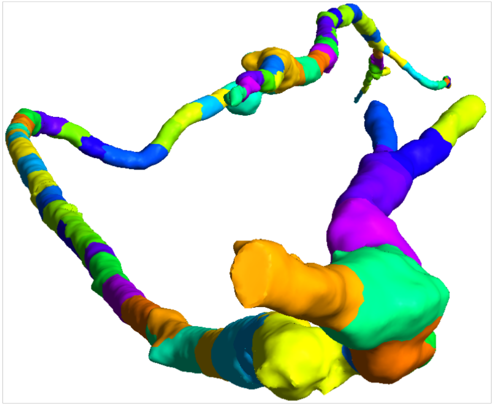
AMD EPYC 2x 64-core

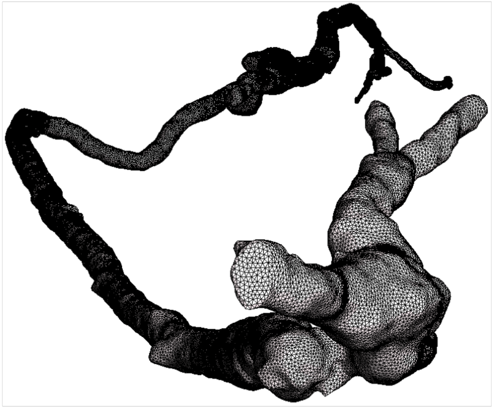
Robust STL remeshing

New pipeline to remesh discrete surfaces (represented by triangulations):

- Automatic construction of a set of parametrizations that form an atlas of the model
- Each parametrization is guaranteed to be one-to-one, amenable to meshing using existing algorithms
- New nodes are guaranteed to be on the input triangulation (“no modelling”)
- Optional pre-processing (i.e. edge detection) to color sub-patches if sharp features need to be preserved

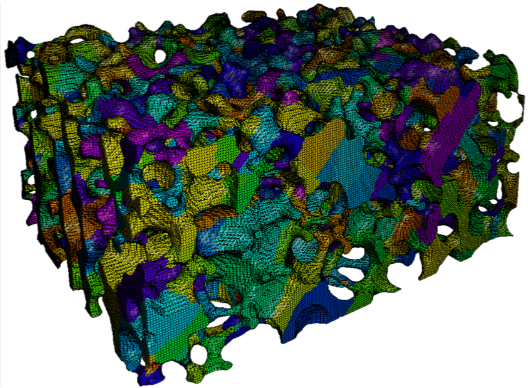
Robust STL remeshing

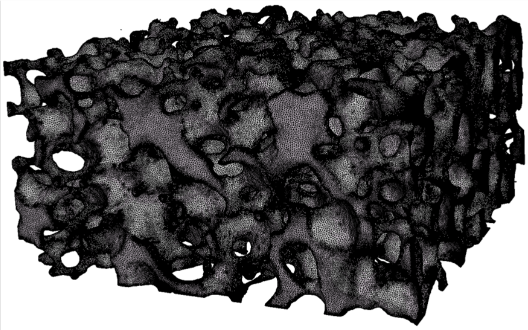




CT scan of an artery: automatic atlas creation – each patch is provably parametrizable by solving a linear PDE, using mean value coordinates

Robust 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

Towards (very) large scale mesh generation

PARSEC (PRACE 6iP 2020-2022):
Parallel Adaptive Refinement for Simulations on Exascale Computers

Partners:

- Barcelona Supercomputing Center
- KTH Royal Institute of Technology
- Université of Liège
- Cenaero

Aim: Sharing best practices and collaboratively modernize the AMR implementation of three leading edge community codes (Alya, Nek5000, Argo), for the exploitation of future (pre-)Exacale machines

Towards (very) large scale mesh generation

Meshing, refinement, coarsening (including for high-order meshes)

Tools: Alya SFC partitioner, ParMETIS, MAdLib, Gmsh

Strategies:

- Classical “freeze and move interfaces” strategy (Gmsh kernel in Alya and MAdLib; Gmsh parallel partition topology as BRep)
- Proof-of-concept “single step” using coarse partition and remeshing of discrete interfaces
- Particular focus on high-order meshes in Gmsh-MAdLib coupling

Looking forward to sharing results/code with ExCALIBUR!

Thank you

cgeuzaine@uliege.be