Gmsh: general overview and recent developments

C. Geuzaine[1] and J.-F. Remacle[2]

[1]Université de Liège [2]Université catholique de Louvain

Workshop NEMESIS, Montpellier, 25 January 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

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

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://gmsh.info, https://getdp.info, https://onelab.info

Some background

UCLouvain

- 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

Some background

UCLouvain

- 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

LIÈGE université

UCLouvain

General overview of Gmsh

First model and mesh

Ingredients for unstructured triangulations

Surface meshing

Unstructured quad and hex meshing

Gmsh-based solvers

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

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

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, ...)

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

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

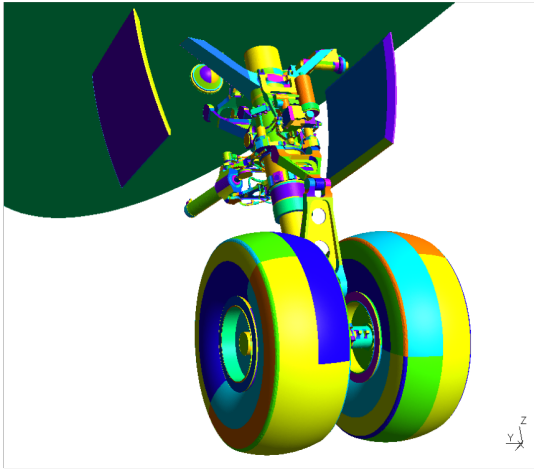
- Gmsh is based around four modules: Geometry, Mesh, Solver and Post-processing
- Gmsh can be used at 3 levels
  - Through the GUI
  - Through the dedicated `.geo` scripting language
  - Through the C++, C, Python, Julia and Fortran API

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, ...)

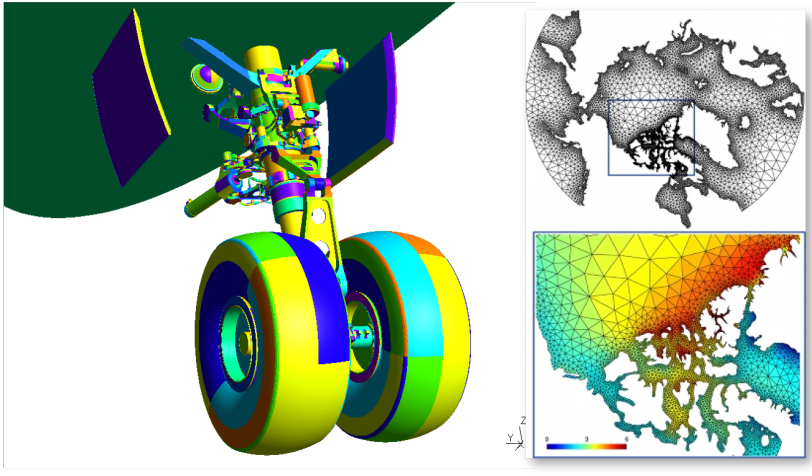
Basic concepts

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



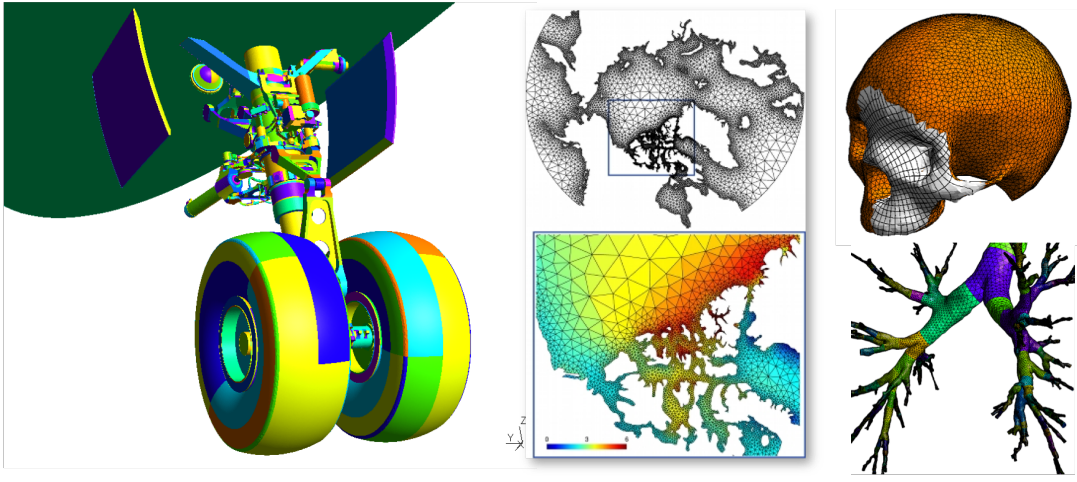
Basic concepts

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



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

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

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)

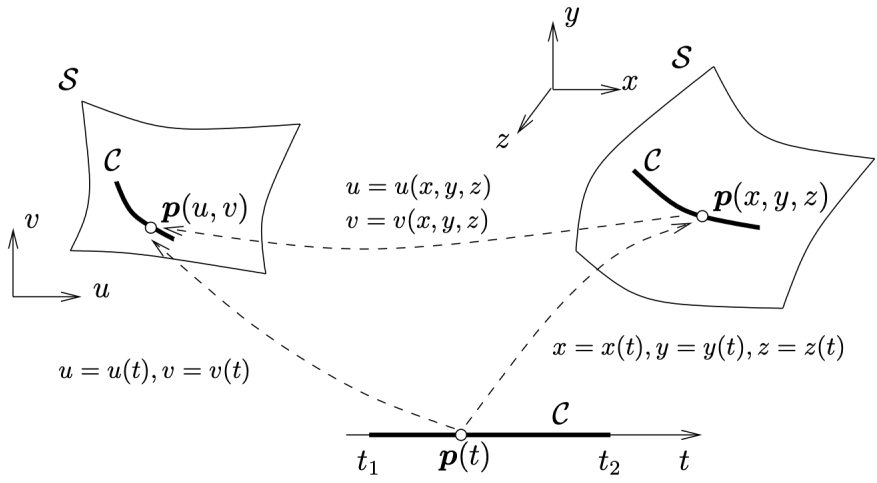
Geometry module

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

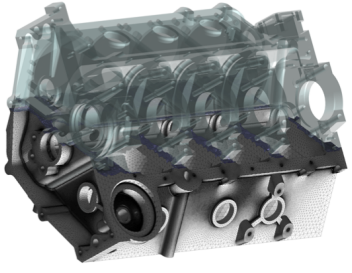


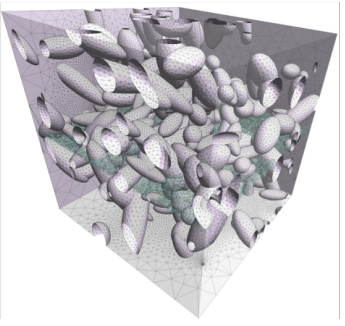
Point $\mathbf{p}$ located on a curve $\mathcal{C}$ that is itself embedded in a surface $\mathcal{S}$

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

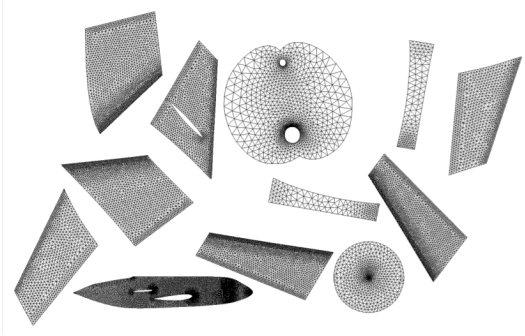


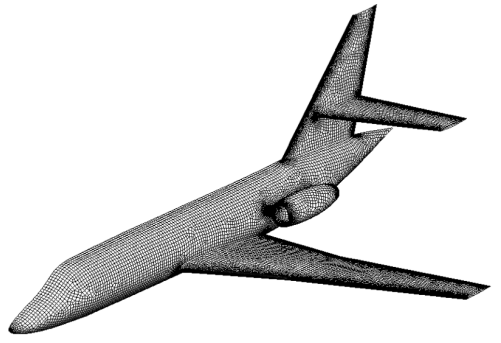


Geometry module

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

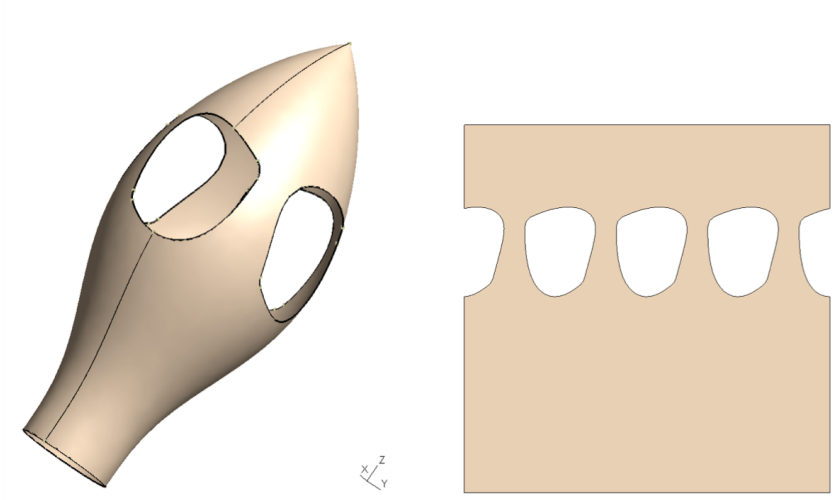
Mesh module

- 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

Mesh module

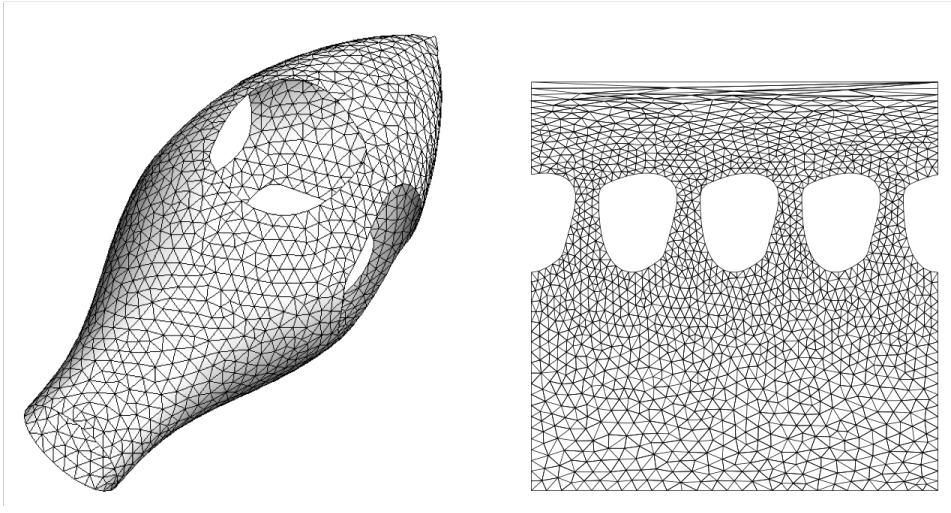
- A (conformal) finite element mesh of a model is a tessellation of its geometry by geometrical elements of various shapes (lines, triangles, quadrangles, tetrahedra, prisms, hexahedra, pyramids), arranged in such a way that if two of them intersect, they do so along a face, an edge or a node, and not 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



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*

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:

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

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

Mesh module

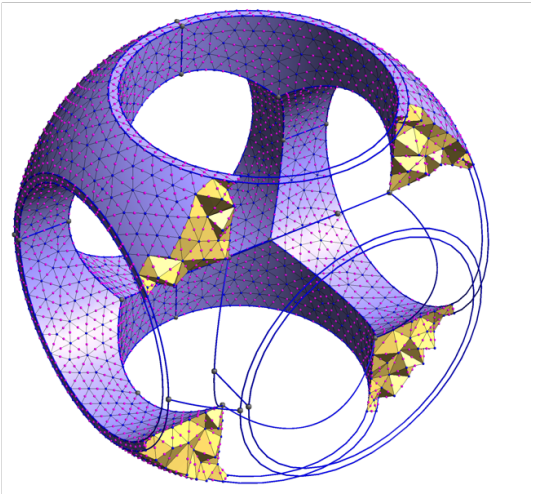
- Mesh data is made of *elements* (points, lines, triangles, quadrangles, tetrahedra, hexahedra, ...) defined by an ordered list of their *nodes*
- Elements and nodes are stored (*classified*) in the model entity they discretize:
  - A model point will thus contain a mesh element of type point, as well 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)

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)

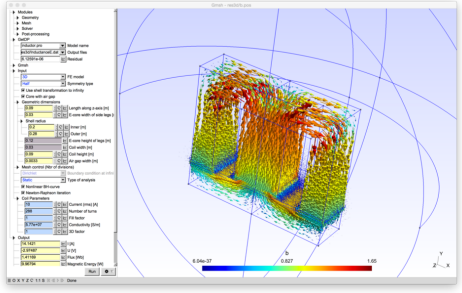
Mesh module

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

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

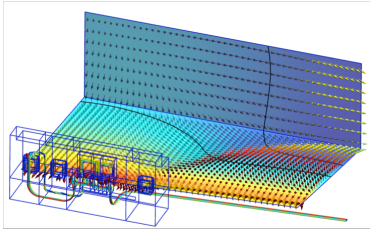
Post-processing module

- Post-processing data is made of *views*
- A view stores both display *options* and *data* (unless the view is an *alias* of another view)

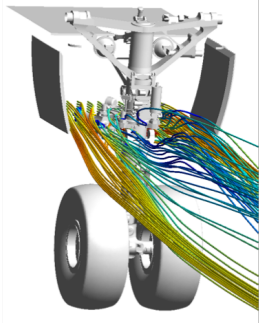
Post-processing module

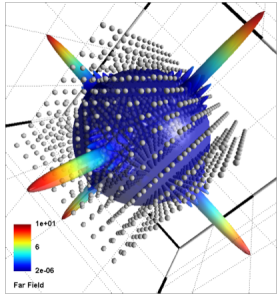
- Post-processing data is made of *views*
- A view stores both display *options* and *data* (unless the view is an *alias* of another view)
- View data can contain several *steps* (e.g. to store time series) and can be either linked to one or more models (*mesh-based* data, as stored in `.msh` or `.med` 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

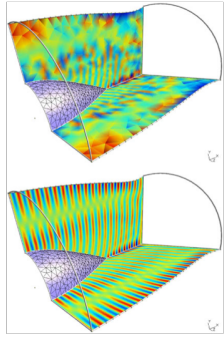
Post-processing module



- 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

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!

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`)

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`)

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`)
- Create post-processing views

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`)
- Create post-processing views
- Run the graphical user-interface

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`)
- 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` or `gmsh/examples/api/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

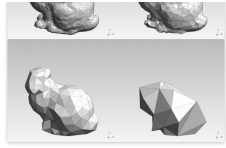
Download

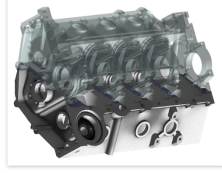
Gmsh is distributed under the terms of the GNU General Public License (GPL):

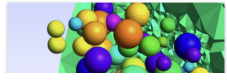
- **Current stable release (version 4.14.1, 2 September 2025):**
  - **Download Gmsh for Windows, Linux, macOS (x86) or macOS (ARM)** *
  - **Download the source code**
  - **Download the Software Development Kit (SDK) for Windows, Linux, macOS (x86) or macOS (ARM)** *
  - **Download both Gmsh and the 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
```

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
```

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};
```

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)
cl = gmsh.model.geo.addCurveLoop([l1, l2, l3, l4])
gmsh.model.geo.addPlaneSurface([cl])
gmsh.model.geo.synchronize() # sync CAD kernel data to model
gmsh.fltk.run() # launch the GUI
gmsh.finalize()
```

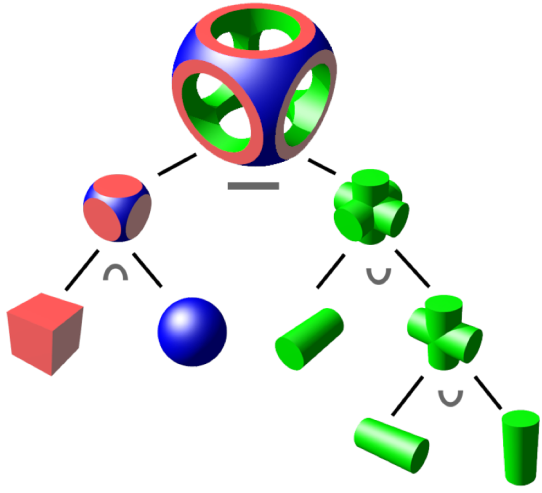
Same in Python using the Gmsh API

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)
cl = gmsh.model.geo.addCurveLoop([l1, l2, l3, l4])
gmsh.model.geo.addPlaneSurface([cl])
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)

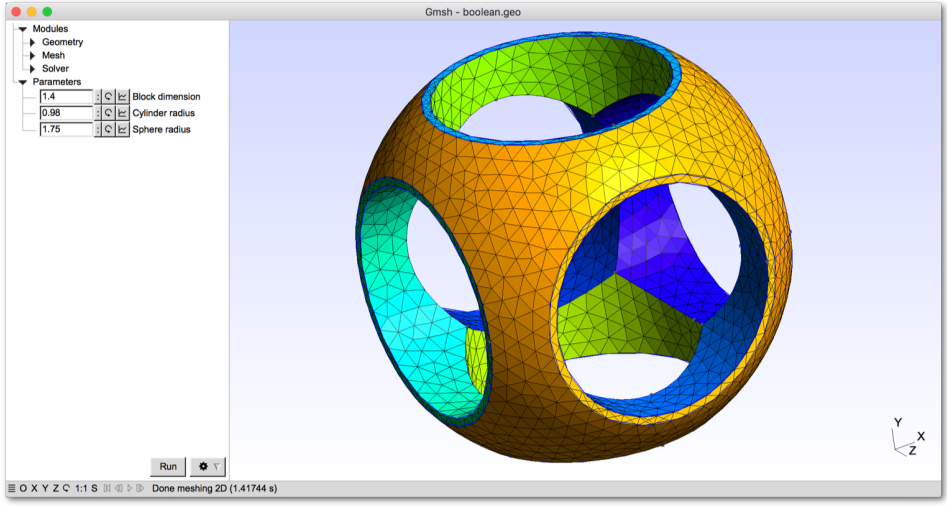


https://en.wikipedia.org/wiki/Constructive_solid_geometry

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

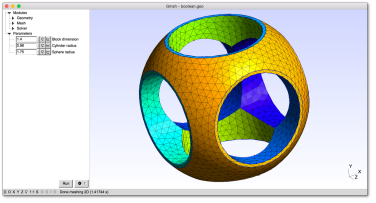


gmsh/examples/boolean/boolean.geo

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()
```

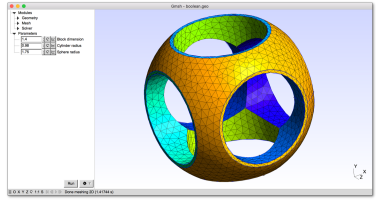


gmsh/examples/api/boolean.py

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/examples/api/boolean.cpp

Ingredients for unstructured meshing

Triangulations

A *simplex* is a generalization of the notion of a triangle or tetrahedron to arbitrary dimensions

A triangulation $T(S)$ of the n points $S = \{p_1, \ldots, p_n\} \in \mathbb{R}^d$ is a set of non overlapping simplices that covers exactly the convex hull $\Omega(S)$ of the point set, and leaves no point p_i isolated

Triangulations

A *simplex* is a generalization of the notion of a triangle or tetrahedron to arbitrary dimensions

A triangulation $T(S)$ of the n points $S = \{p_1, \ldots, p_n\} \in \mathbb{R}^d$ is a set of non overlapping simplices that covers exactly the convex hull $\Omega(S)$ of the point set, and leaves no point p_i isolated

Points p_j are *in general position* when they do not fall on subvarieties of lower degree than necessary; in the plane two points should not be coincident, three points should not fall on a line, four points should not fall on a circle

Triangulations

There exist a finite but combinatorial number of triangulations (Catalan numbers) for a given set of points

In dimension 2, the number of triangles is constant for every triangulation of the same set of points; this is not true in 3D and in higher dimensions

The *Delaunay triangulation* is a special triangulation that exists and is unique if points are in general position

Triangulations

There exist a finite but combinatorial number of triangulations (Catalan numbers) for a given set of points

In dimension 2, the number of triangles is constant for every triangulation of the same set of points; this is not true in 3D and in higher dimensions

The *Delaunay triangulation* is a special triangulation that exists and is unique if points are in general position

There exist algorithms to generate the Delaunay triangulation in $\mathcal{O}(n \log(n))$ complexity! Yet, the constant grows rapidly with the dimension d

Delaunay triangulation

The Delaunay triangulation DT(S) of a point set S has the fundamental geometrical property that the circumsphere of any simplex is empty

If the empty sphere condition is verified for all simplices, the triangulation $T(S)$ is said to be a Delaunay triangulation

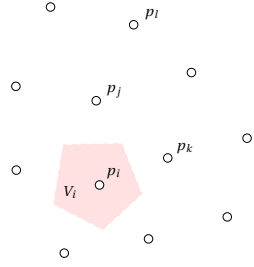
In dimension 2, DT(S) has interesting properties

The Voronoï diagram

Consider a finite set $S = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^2$ of n distinct points in the plane. The *Voronoi cell* V_i of $p_i \in S$ is the set of points x that are closer to p_i than to any other points of the set:

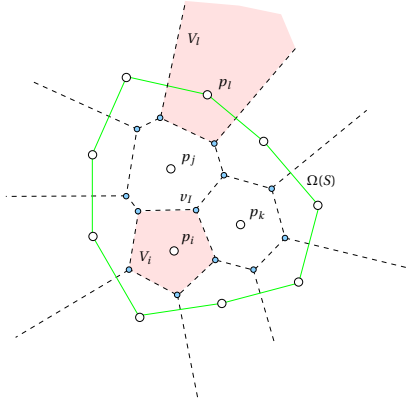
$$V_i = \left\{ x \in \mathbb{R}^2 \mid \|x - p_i\| < \|x - p_j\|, \ \forall 1 \leq i \leq n, i \neq j \right\}$$

where $\|x - y\|$ is the euclidean distance between x and y



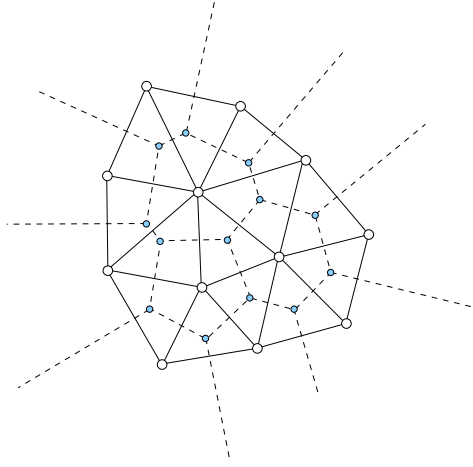
The Voronoï diagram

The Voronoi diagram $V(S)$ is the unique subdivision of the plane into n cells. It is the union of all Voronoi cells V_p:



The Delaunay triangulation

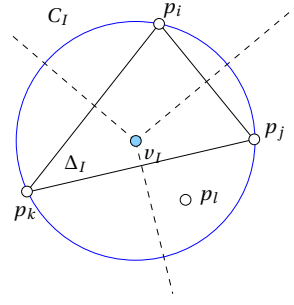
The Delaunay triangulation $\mathrm{DT}(S)$ is the geometric dual of the Voronoï diagram



The empty circle property

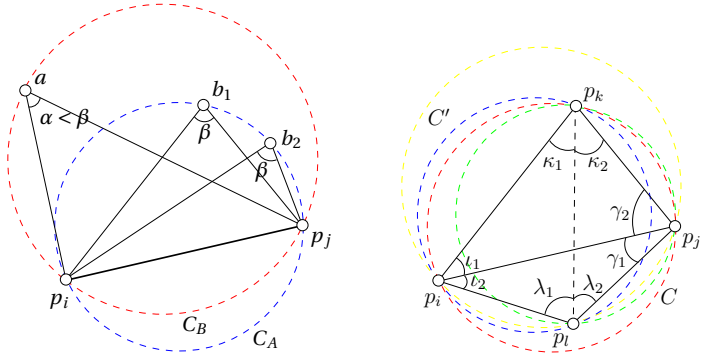
The circumcircle of any triangle in the Delaunay triangulation is empty i.e. it contains no point of S

- Consider the Delaunay triangle $\Delta_I = p_i p_j p_k$. Assume now that point $p_l \in C_I$ where C_I is the circumcircle of Δ_I
- By definition, the triple point v_I is at equal distance to p_i, p_j and p_k and no other points of S are closer to v_I than those three points
- Then, if a point like p_l exist in S, v_I is not a triple point and triangle Δ_I cannot be a Delaunay triangle



The MaxMin property

The Delaunay triangulation $\mathrm{DT}(S)$ is angle-optimal: it maximizes the minimum angle among all possible triangulations



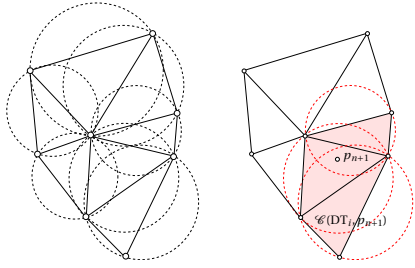
Thales theorem (left) and MaxMin property illustrated (right)

Bowyer-Watson algorithm

Let DT_n be the Delaunay triangulation of a point set $S_n = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$ that are in general position

The Bowyer-Watson algorithm is an incremental process allowing the insertion of a given point $p_{n+1} \in \Omega(S_n)$ into DT_n and to build the Delaunay triangulation DT_{n+1} of $S_{n+1} = \{p_1, \ldots, p_n, p_{n+1}\}$

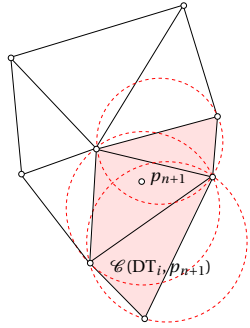
$$\mathsf{DT}_{n+1} = \mathsf{DT}_n - \mathcal{C}(\mathsf{DT}_n, p_{n+1}) + \mathcal{B}(\mathsf{DT}_n, p_{n+1}). \tag{1}$$



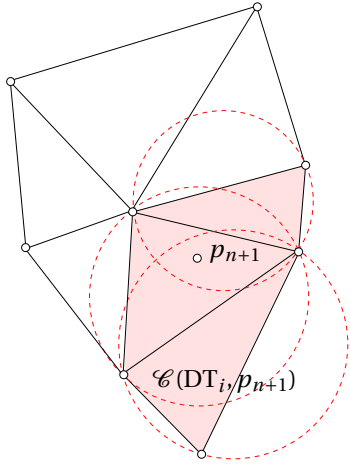
Bowyer-Watson algorithm

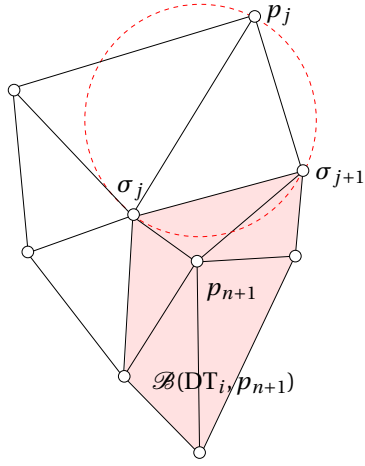
The Delaunay cavity $\mathcal{C}(T_n, p_{n+1})$ is the set of m triangles $\Delta_1, \ldots, \Delta_m \in \mathrm{DT}_n$ for which their circumcircle contains p_{n+1}

The Delaunay cavity contains the set of triangles that cannot belong to T_{n+1}: the region covered by those invalid triangles should be emptied and re-triangulated in a Delaunay fashion



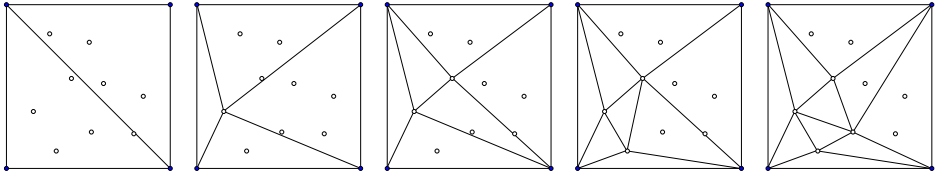
Bowyer-Watson algorithm

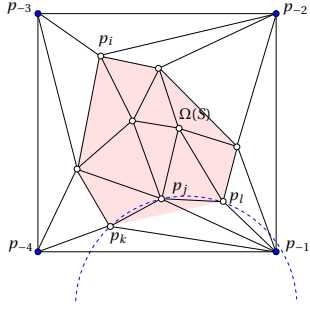




Bowyer-Watson algorithm

Super triangles:

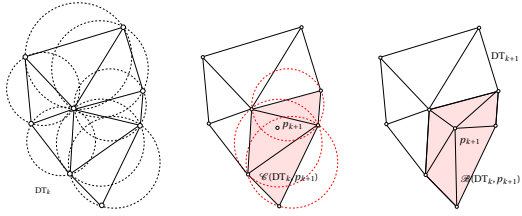




DT of n points in $n\log(n)$ complexity

- Use Bowyer-Watson algorithm

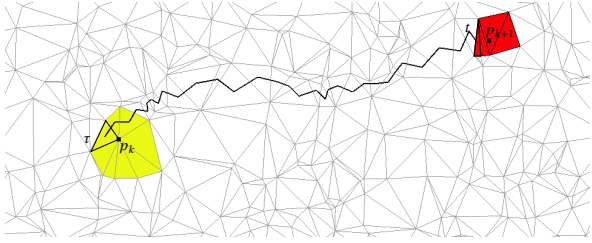
$$\mathsf{DT}_{k+1} = \mathsf{DT}_k - \mathcal{C}(\mathsf{DT}_k, p_{k+1}) + \mathcal{B}(\mathsf{DT}_k, p_{k+1})$$



DT of n points in $n\log(n)$ complexity

- Use Bowyer-Watson algorithm
- Sort the points [N. Amenta, S. Choi, and G. Rote. *Incremental constructions con BRIO*, 2003]: the Biased Randomized Insertion Order can e.g. use a space-filling curve like a Hilbert curve

Without sort: $\mathcal{O}(n^{1/d})$ "walking" steps per insertion $\rightarrow$ overall (best) complexity of $\mathcal{O}(n^{1+\frac{1}{d}})$



DT of n points in $n\log(n)$ complexity

- Use Bowyer-Watson algorithm
- Sort the points [N. Amenta, S. Choi, and G. Rote. *Incremental constructions con BRIO*, 2003]: the Biased Randomized Insertion Order can e.g. use a space-filling curve like a Hilbert curve

With sort along Hilbert curve: constant number of steps

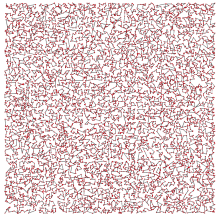
n	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6
	2D				3D			
N_{walk}	23	73	230	727	17	38	85	186
t(sec)	$3.6\ 10^{-3}$	$9.1\ 10^{-2}$	3.98	187	$1.2\ 10^{-2}$	$1.8\ 10^{-1}$	3.42	73
	2D (BRIO)				3D (BRIO)			
N_{walk}	2.3	2.4	2.5	2.5	2.9	3.0	3.1	3.1
t(sec)	$2\ 10^{-3}$	$1.5\ 10^{-2}$	$1.5\ 10^{-1}$	1.47	$9.0\ 10^{-3}$	$7.5\ 10^{-2}$	$7.8\ 10^{-1}$	7.81

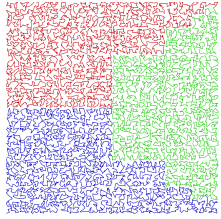
Sorting cost is $\mathcal{O}(n\log(n)) \rightarrow$ overall (best) complexity $\mathcal{O}(n\log(n))$

DT of n points in $n\log(n)$ complexity

- Use Bowyer-Watson algorithm
- Sort the points [N. Amenta, S. Choi, and G. Rote. *Incremental constructions con BRIO*, 2003]: the Biased Randomized Insertion Order can e.g. use a space-filling curve like a Hilbert curve
- Multithreading: distribute the Hilbert curve in M threads

$$\mathsf{DT}_{k+1} = \mathsf{DT}_k + \sum_{i=0}^{M-1} \left[-\mathcal{C}(\mathsf{DT}_k, p_{k+i\frac{n}{M}}) + \mathcal{B}(\mathsf{DT}_k, p_{k+i\frac{n}{M}}) \right].$$





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

Multithreaded meshing in Gmsh

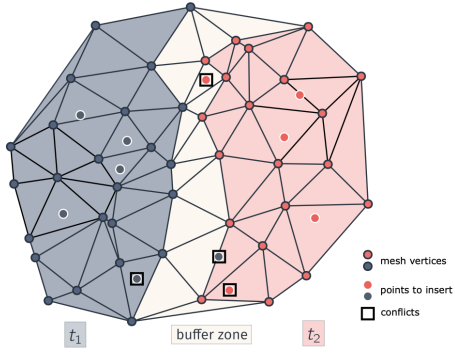
The meshing pipeline is multithreaded using OpenMP:

- 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:

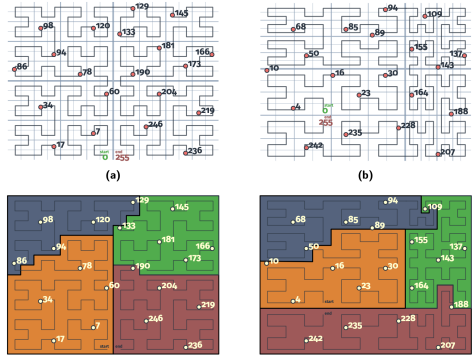
```
gmsh file.geo -3 -nt 8 -algo hxt
```

Multithreaded meshing in Gmsh



- Points are partitioned such that each point belongs to a single thread
- A triangle can only be modified by a thread that owns all of its three nodes
- Triangles that cannot be modified by any thread form a buffer zone

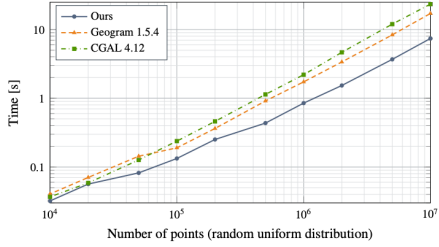
Multithreaded meshing in Gmsh



Modification of the partitions to insert the points for which insertion failed because the point cavity spans multiple partitions:

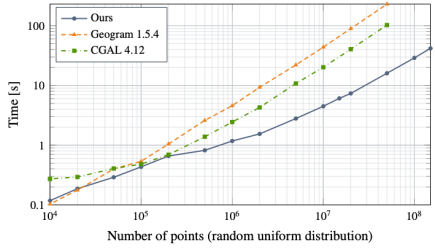
- Circular shift to move the zero index around the Hilbert/Moore curve
- Coordinates below a random threshold are linearly compressed, whereas coordinates above the threshold are linearly expanded

Multithreaded meshing in Gmsh



# 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

(a) 4-core Intel® Core™ i7-6700HQ 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	/

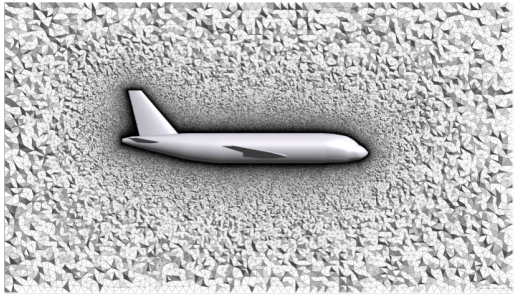
(b) 64-core Intel® Xeon Phi™ 7210 CPU.

[C. Marot et al., IJNME 2019]

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

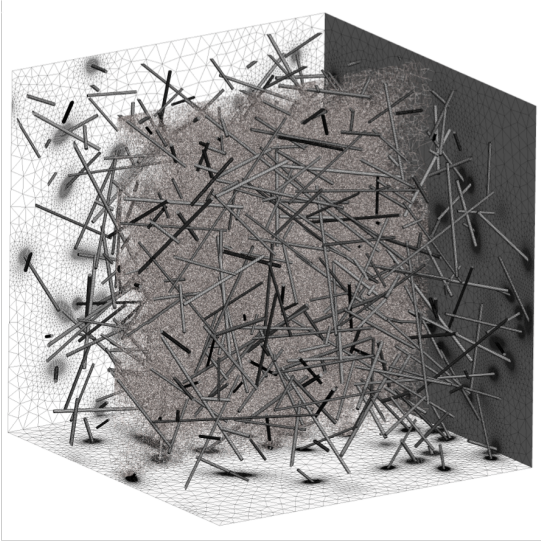


Aircraft

# 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

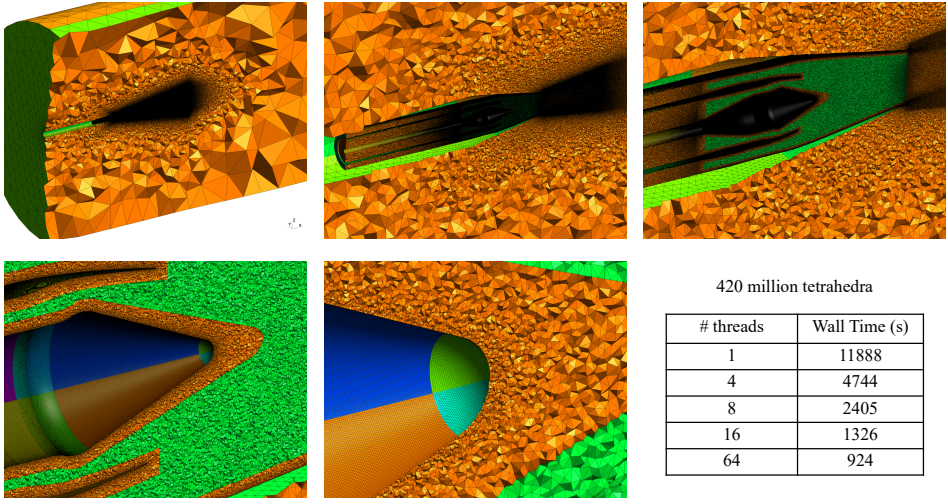
# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	325 611 841	3.1	492.1	497.2
2	325 786 170	2.9	329.7	334.3
4	325 691 796	2.8	229.5	233.9
8	325 211 989	2.7	154.6	158.7
16	324 897 471	2.8	96.8	100.9
32	325 221 244	2.7	71.7	75.8
64	324 701 883	2.8	55.8	60.1
127	324 190 447	2.9	47.6	52.0

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

More ingredients: boundary recovery

Remember the super triangles?

- For conformal finite element mesh generation Gmsh starts from the boundary mesh, i.e. boundary segments in 2D and boundary triangles in 3D

More ingredients: boundary recovery

Remember the super triangles?

- For conformal finite element mesh generation Gmsh starts from the boundary mesh, i.e. boundary segments in 2D and boundary triangles in 3D
- An “empty mesh” is created first, triangulating the boundary points

More ingredients: boundary recovery

Remember the super triangles?

- For conformal finite element mesh generation Gmsh starts from the boundary mesh, i.e. boundary segments in 2D and boundary triangles in 3D
- An "empty mesh" is created first, triangulating the boundary points
- The boundary segments/triangles are then "recovered" so that they are edges/faces of the interior triangles/tetrahedra
  - In the 3D case Gmsh uses Tetgen's boundary recovery code [H. Si. *Tetgen, a Delaunay-based quality tetrahedral mesh generator.*, 2015]

More ingredients: boundary recovery

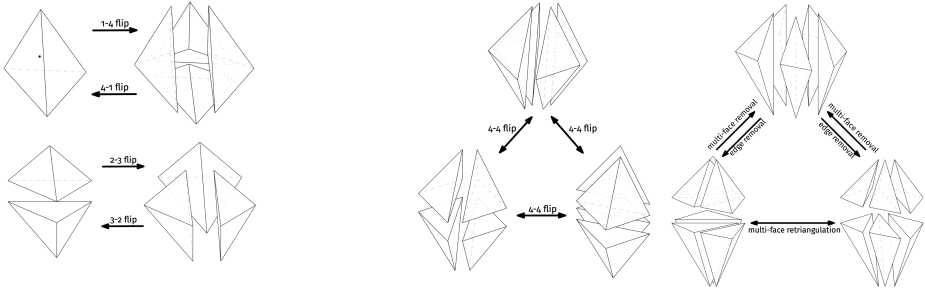
Remember the super triangles?

- For conformal finite element mesh generation Gmsh starts from the boundary mesh, i.e. boundary segments in 2D and boundary triangles in 3D
- An "empty mesh" is created first, triangulating the boundary points
- The boundary segments/triangles are then "recovered" so that they are edges/faces of the interior triangles/tetrahedra
  - In the 3D case Gmsh uses Tetgen's boundary recovery code [H. Si. *Tetgen, a Delaunay-based quality tetrahedral mesh generator.*, 2015]
- Then the mesh is refined with (multi-threaded) point insertion

More ingredients: mesh improvement

Once a mesh with the desired local mesh size is obtained by point insertion, a final improvement step is performed to

- locally eliminate badly shaped tetrahedra (e.g. slivers)
- optimize the quality of the mesh by means of specific topological operations and vertex relocations (flipping, edge removal, smoothing, small polyhedron reconnection, ...)



Surface meshing

Surface meshing

- When mesh generation procedures have access to parametrizations of surfaces, one can generate a planar mesh in the parametric domain and map it in 3D —it is an *indirect* approach

Surface meshing

- When mesh generation procedures have access to parametrizations of surfaces, one can generate a planar mesh in the parametric domain and map it in 3D —it is an *indirect* approach
- In Gmsh, surface meshes are generated in the parameter plane (u, v) and standard "off the shelf" anisotropic 2D meshers are used for generating surface meshes

Surface meshing

- When mesh generation procedures have access to parametrizations of surfaces, one can generate a planar mesh in the parametric domain and map it in 3D —it is an *indirect* approach
- In Gmsh, surface meshes are generated in the parameter plane (u, v) and standard "off the shelf" anisotropic 2D meshers are used for generating surface meshes
- Ensuring that a planar mesh is valid is trivial: all triangles should be positively oriented

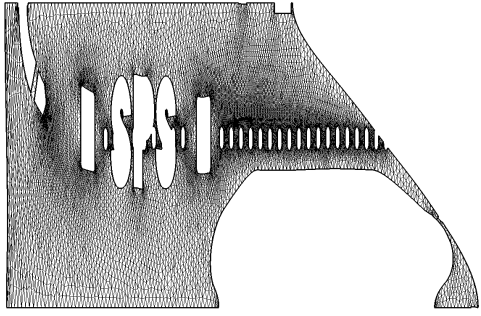
Surface meshing

- When mesh generation procedures have access to parametrizations of surfaces, one can generate a planar mesh in the parametric domain and map it in 3D —it is an *indirect* approach
- In Gmsh, surface meshes are generated in the parameter plane (u, v) and standard "off the shelf" anisotropic 2D meshers are used for generating surface meshes
- Ensuring that a planar mesh is valid is trivial: all triangles should be positively oriented
- If the surface parametrization $\mathbf{x}(u, v) \in \mathbb{R}^3$ is regular, then the mapping of the (u, v) mesh onto the surface is itself valid because the composition of two regular mappings is regular

Surface meshing

In this example, the depicted trimmed surface has no irregular points and the mesh generation procedure is quite straightforward: the anisotropic frontal-Delaunay approach from Gmsh was used to produce the mesh, based on the metric tensor (full rank everywhere)

$$\mathcal{M} = J^T J = \begin{pmatrix} \|\partial_u \mathbf{x}\|^2 & \partial_u \mathbf{x} \cdot \partial_v \mathbf{x} \\ \partial_u \mathbf{x} \cdot \partial_v \mathbf{x} & \|\partial_v \mathbf{x}\|^2 \end{pmatrix}$$



Surface meshing: singularities

Surfaces with isolated irregular points are however very common in CAD systems: spheres, cones and other surfaces of revolution may contain one or two irregular/singular points

Mesh generation procedures are known to be prone to failure close to irregularities

Surface meshing: singularities

Surfaces with isolated irregular points are however very common in CAD systems: spheres, cones and other surfaces of revolution may contain one or two irregular/singular points

Mesh generation procedures are known to be prone to failure close to irregularities

Consider a sphere of radius R centered at the origin is parametrized as

$$\begin{aligned} x(u,v) &= R\,\sin u\,\cos v \\ y(u,v) &= R\,\sin u\,\sin v \\ z(u,v) &= R\,\cos u \end{aligned}$$

where $u \in [0, \pi]$ is the inclination and $v \in [0, 2\pi[$ is the azimuth

Surface meshing: singularities

At the poles, i.e. when $u = 0$ or $u = \pi$,

$$\partial_v \mathbf{x} = R(-\sin u \sin v, \sin u \cos v, 0) = (0, 0, 0)$$

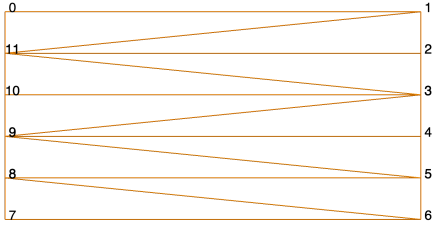
vanishes and this parametrization is irregular

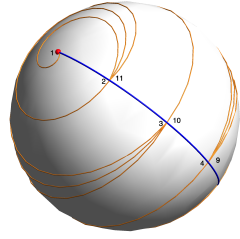
Surface meshing: singularities

At the poles, i.e. when $u = 0$ or $u = \pi$,

$$\partial_v \mathbf{x} = R(-\sin u \sin v, \sin u \cos v, 0) = (0, 0, 0)$$

vanishes and this parametrization is irregular



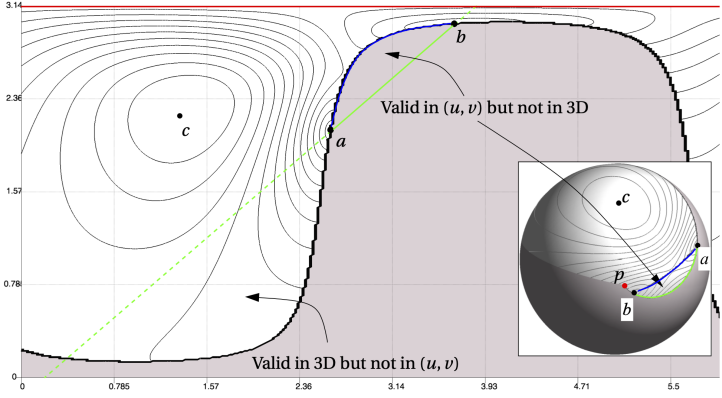


All the straight-sided 3D triangles above would be invalid (zero-area)

Surface meshing

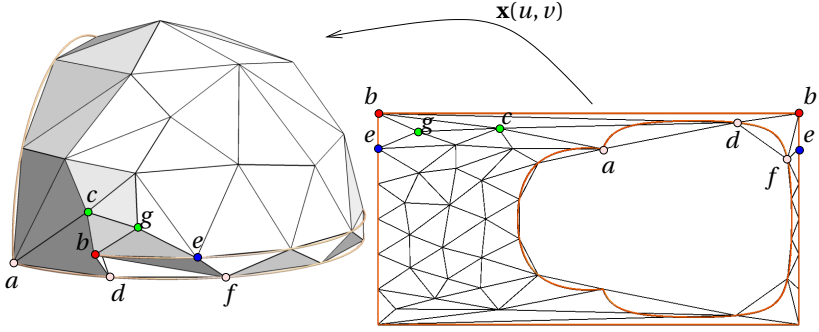
Consider an edge (a, b) with b close to the pole p:

- Plot of iso-values of quality of a triangle (a, b, c) with c positioned anywhere in the parameter plane
- Shaded zone corresponds to positions of c leading to invalid triangles in 3D
- Observe zones where valid 2D triangles are invalid in 3D, and conversely!



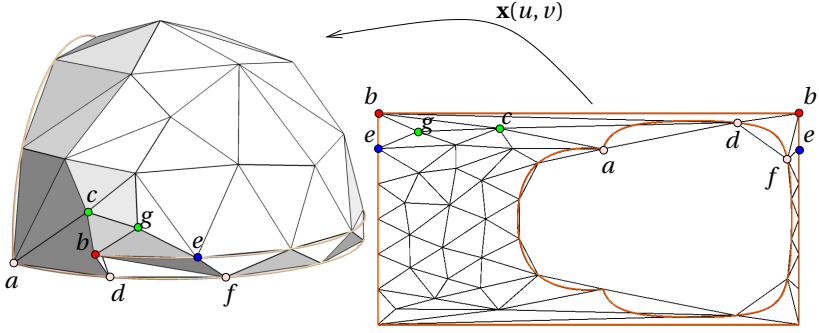
Surface meshing in Gmsh

The main issue here is *not* the fact that the metric tensor is of rank 1 at irregular points and very distorted around it



Surface meshing in Gmsh

The main issue here is *not* the fact that the metric tensor is of rank 1 at irregular points and very distorted around it



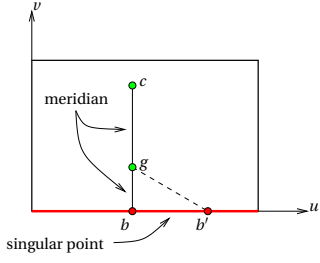
The issue is essentially related to triangles (e.g. (b, c, d), valid in 2D but not in 3D) and edges that have one node like b that corresponds to an irregular point of the parametrization

Surface meshing in Gmsh

Consider a surface of revolution with respect to the z-axis and suppose that the generating curve is $\mathbf{c}(v) = (f(v), 0, g(v)) \quad , \quad v \in [0, T]$
The parametrization of the surface is given by

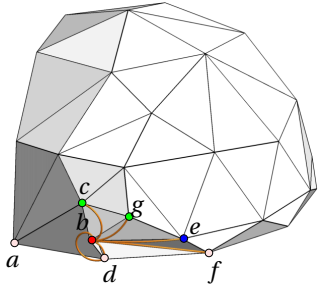
$$\mathbf{x}(u, v) = (f(v)\cos(u), f(v)\sin(u), g(v)) \quad , \quad (u, v) \in [0, 2\pi[\times[0, T]$$

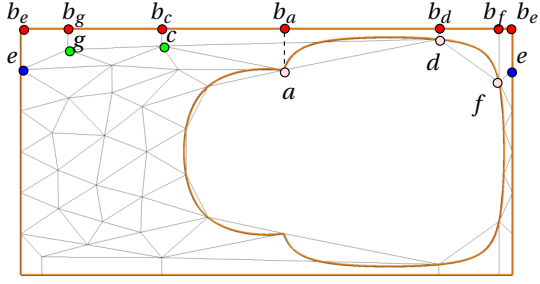
One interesting property of surfaces of revolution is that meridian curves $u = \text{cst}$ are geodesics.



Surface meshing in Gmsh

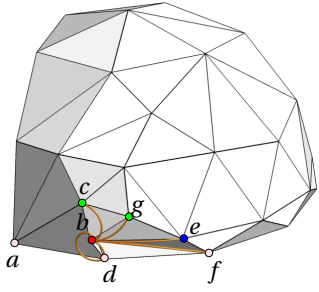
Solution: replace all the edges that are incident to irregular points by geodesics

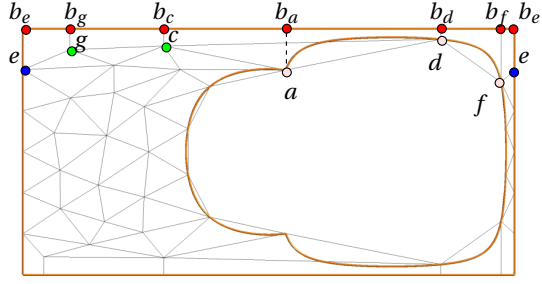




Surface meshing in Gmsh

Solution: replace all the edges that are incident to irregular points by geodesics

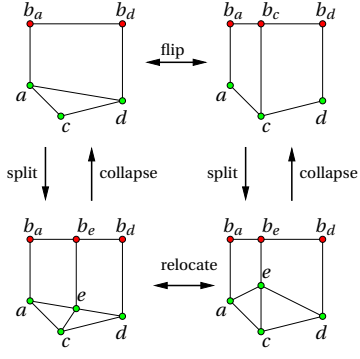




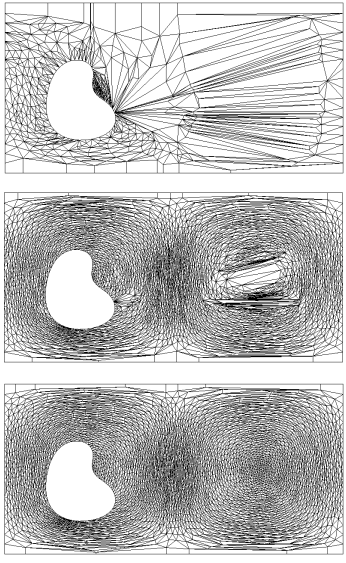
The edge flip algorithm in Gmsh 4 has been updated for this new representation for both Delaunay-based and local mesh adaptation algorithm (MeshAdapt)

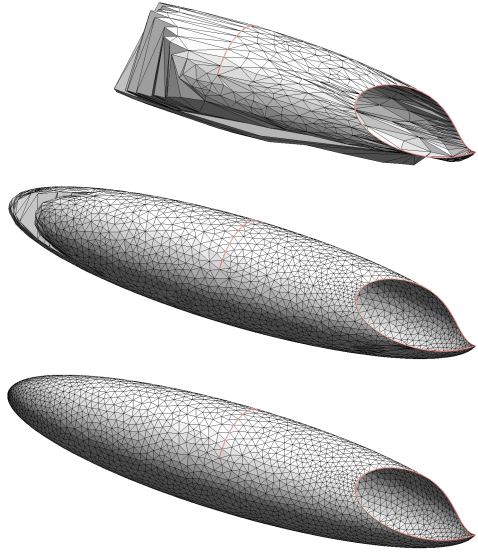
Gmsh's MeshAdapt algorithm

1. Each edge that is too long is split
2. Each edge that is too short is collapsed
3. Edge flips are performed in order to obtain a better configuration
4. Vertices are re-located optimally after steps 1, 2 and 3

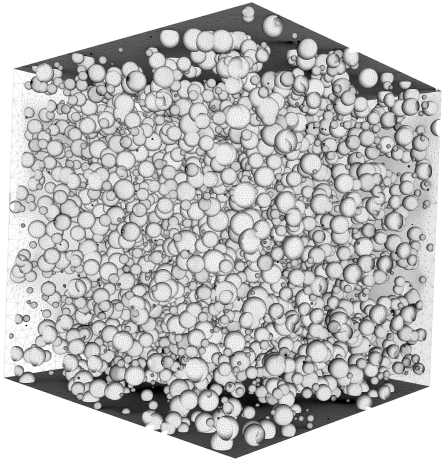


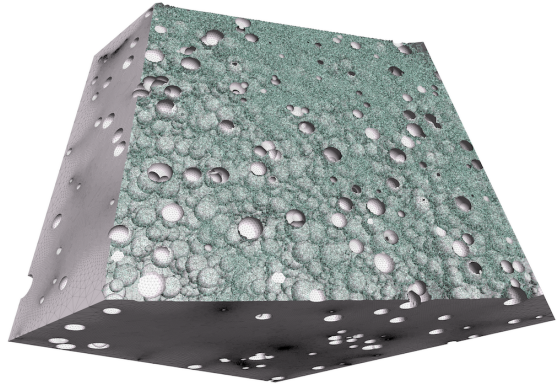
Gmsh's MeshAdapt algorithm





Gmsh's MeshAdapt algorithm





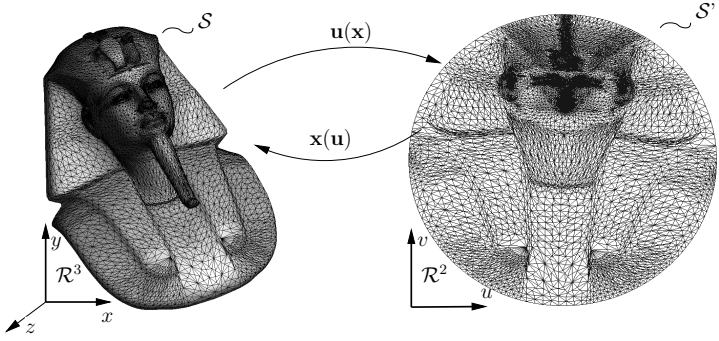
Gmsh's MeshAdapt algorithm

Gmsh's surface meshing pipeline defaults to MeshAdapt when other (e.g. Delaunay-based) algorithms fail

```
// Script to generate a CAD model with 5000 spherical inclusions in a cube
SetFactory("OpenCASCADE");

DefineConstant[
  rmin = {0.002, Name "Min radius"}
  rmax = {0.03, Name "Max radius"}
  n = {500, Name "Number of spheres"}
];

For i In {1:n}
  r = rmin + Rand(rmax - rmin);
  x = -0.5 + Rand(1);
  y = -0.5 + Rand(1);
  z = -0.5 + Rand(1);
  Sphere(i) = {x, y, z, r };
EndFor
Box(n + 1) = {-0.5, -0.5, -0.5, 1, 1, 1 };
BooleanDifference { Volume{n + 1}; Delete; }{ Volume {1:n}; Delete; }
```

Discrete surface meshing

In order to apply the indirect surface meshing approach to discrete surfaces, i.e. surfaces described by a triangulation T, we need to compute a parametrization of T



Computing discrete parametrizations

Assume T is a triangulation of a simply connected surface S

Finding a parametrization of T consists in assigning to every vertex $p_i(x_i, y_i, z_i)$ of the triangulation a pair of parametric coordinates $(u_i, v_i) \in S'$

Computing discrete parametrizations

Assume T is a triangulation of a simply connected surface S

Finding a parametrization of T consists in assigning to every vertex $p_i(x_i, y_i, z_i)$ of the triangulation a pair of parametric coordinates $(u_i, v_i) \in S'$

If every triangle (p_i, p_j, p_k), with $p_\bullet \in \mathbb{R}^3$ of the triangulation has a positive area in the $(u; v)$ plane, then the parametrization is injective

Computing discrete parametrizations

Assume T is a triangulation of a simply connected surface S

Finding a parametrization of T consists in assigning to every vertex $p_i(x_i, y_i, z_i)$ of the triangulation a pair of parametric coordinates $(u_i, v_i) \in S'$

If every triangle (p_i, p_j, p_k), with $p_\bullet \in \mathbb{R}^3$ of the triangulation has a positive area in the $(u; v)$ plane, then the parametrization is injective

Consider an internal vertex i of T and $J(i)$ the set of indices whose the corresponding nodes are connected to the node i (in other words, edge (i, j) exists $\forall j \in J(i)$)

Computing discrete parametrizations

The value of the parametric coordinates (u_i, v_i) at vertex i will be computed as a weighted average of the coordinates (u_j, v_j) of its neighboring vertices:

$$\sum_{j \in J(i)} \lambda_{ij}(u_i - u_j) = 0 \quad , \quad \sum_{j \in J(i)} \lambda_{ij}(v_i - v_j) = 0$$

where λ_{ij} are coefficients

This scheme is a called a *difference scheme* that involves only differences $(u_i - u_j)$, with $j \in J(i)$

Computing discrete parametrizations

The value of the parametric coordinates (u_i, v_i) at vertex i will be computed as a weighted average of the coordinates (u_j, v_j) of its neighboring vertices:

$$\sum_{j \in J(i)} \lambda_{ij}(u_i - u_j) = 0 \quad , \quad \sum_{j \in J(i)} \lambda_{ij}(v_i - v_j) = 0$$

where λ_{ij} are coefficients

This scheme is a called a *difference scheme* that involves only differences $(u_i - u_j)$, with $j \in J(i)$

If every λ_{ij} is positive, values of u_i and v_i are convex combinations of their surrounding values

Computing discrete parametrizations

The value of the parametric coordinates (u_i, v_i) at vertex i will be computed as a weighted average of the coordinates (u_j, v_j) of its neighboring vertices:

$$\sum_{j \in J(i)} \lambda_{ij}(u_i - u_j) = 0 \quad , \quad \sum_{j \in J(i)} \lambda_{ij}(v_i - v_j) = 0$$

where λ_{ij} are coefficients

This scheme is a called a *difference scheme* that involves only differences $(u_i - u_j)$, with $j \in J(i)$

If every λ_{ij} is positive, values of u_i and v_i are convex combinations of their surrounding values

From a geometrical point of view, it actually means that point (u_i, v_i) lies in the convex hull $\mathcal{H}_i$ of its neighboring vertices: it is easy to prove that the mapping provided by any such positive scheme is one-to-one

Discrete parametrizations using FE

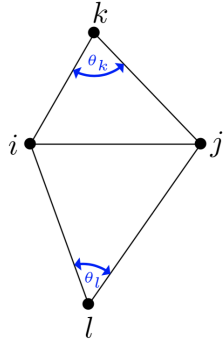
One can choose $\lambda_{ij}^{\text{Tutte}} = 1$, but the smoothness of the parametrization is important for meshing!

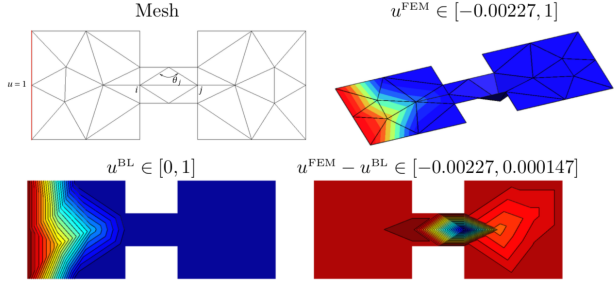
Discrete parametrizations using FE

One can choose $\lambda_{ij}^{\text{Tutte}} = 1$, but the smoothness of the parametrization is important for meshing!

Finite elements (Laplace) is smooth but *NOT provably positive*

$$\lambda_{ij}^{\text{FEM}} := \frac{1}{2}\left(\frac{\cos(\theta_k)}{\sin(\theta_k)} + \frac{\cos(\theta_l)}{\sin(\theta_l)}\right).$$



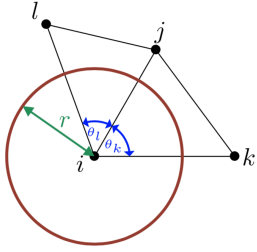


Discrete parametrizations using MVC

Mean value coordinates are provably positive:

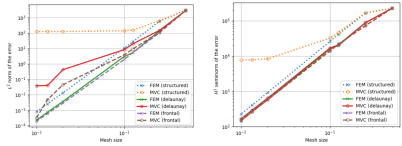
$$\lambda_{ij}^{\mathsf{MVC}} = \frac{\tan\left(\frac{\theta_k}{2}\right) + \tan\left(\frac{\theta_l}{2}\right)}{l_{ij}}$$

They are not discretizing Laplace operator even at first order on structured meshes... but who cares, it discretizes a smooth PDE!



a. Structured. b. Delaunay. c. Frontal.

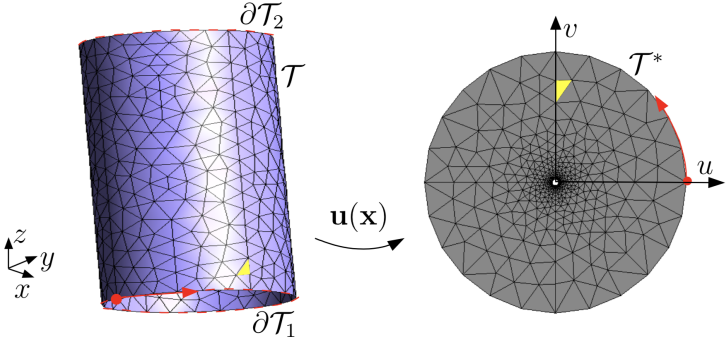
Figure 5: Types of meshes on a square.



a. L^2 norm. b. H^1 seminorm.

Beware of indistinguishable coordinates

$$\begin{cases} \Delta_\epsilon u = 0, \quad \Delta_\epsilon v = 0 & \text{in} \quad \mathcal{T}, \\ u = u_D, \quad v = v_D & \text{on} \quad \partial\mathcal{T}_1, \\ \partial_n u = 0, \quad \partial_n v = 0 & \text{on} \quad \partial\mathcal{T}\backslash\partial\mathcal{T}_1 \end{cases}, \tag{2}$$



Discrete surface meshing in Gmsh

Given a conforming "watertight" geometrical triangulation as input, Gmsh's discrete surface meshing pipeline consists in 3 steps:

1. Edge detection to define sub-patches if sharp features need to be preserved (optional)

Discrete surface meshing in Gmsh

Given a conforming "watertight" geometrical triangulation as input, Gmsh's discrete surface meshing pipeline consists in 3 steps:

1. Edge detection to define sub-patches if sharp features need to be preserved (optional)
2. Automatic construction of an atlas of parametrizations, by partitioning the geometrical triangulation until all computed parametrizations are valid
   - Each patch should have zero genus (no "handles"), isomorphic to a disk (potentially) with holes
   - Parametric coordinates should be distinguishable
   - Saving the model as a `.msh` file will contain the discrete curves and surfaces of the (parametrized) CAD

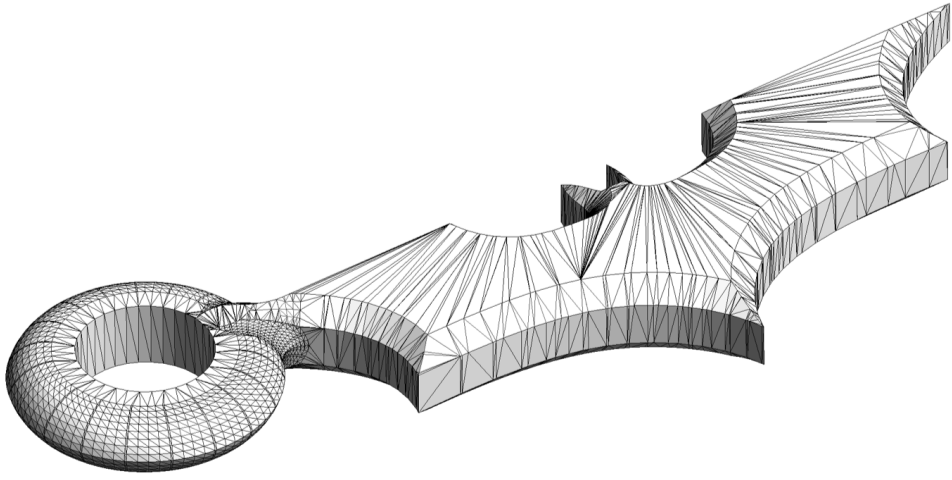
Discrete surface meshing in Gmsh

Given a conforming "watertight" geometrical triangulation as input, Gmsh's discrete surface meshing pipeline consists in 3 steps:

1. Edge detection to define sub-patches if sharp features need to be preserved (optional)
2. Automatic construction of an atlas of parametrizations, by partitioning the geometrical triangulation until all computed parametrizations are valid
   - Each patch should have zero genus (no "handles"), isomorphic to a disk (potentially) with holes
   - Parametric coordinates should be distinguishable
   - Saving the model as a `.msh` file will contain the discrete curves and surfaces of the (parametrized) CAD
3. Meshing of all charts of the atlas, possibly in parallel
   - All new mesh nodes are guaranteed to be on the input geometrical triangulation ("no modeling")

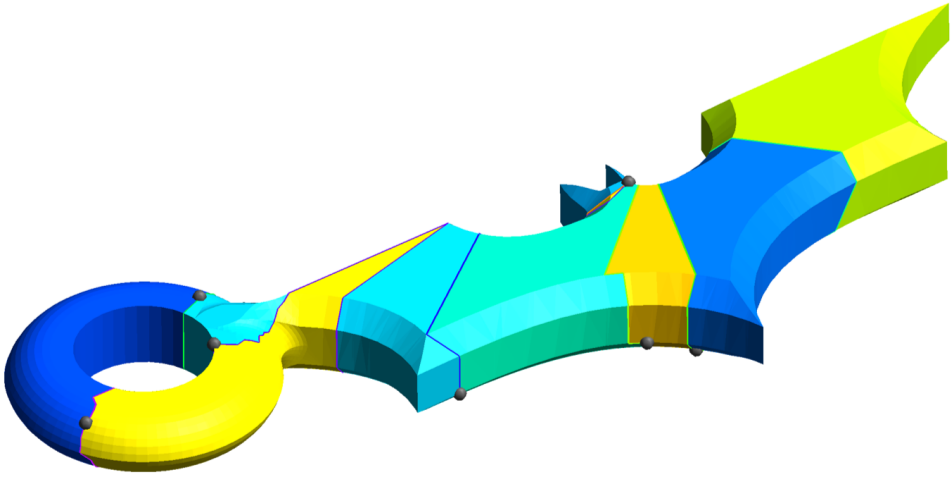
[P. A. Beaufort et al., JCP 2020]

Discrete surface meshing in Gmsh



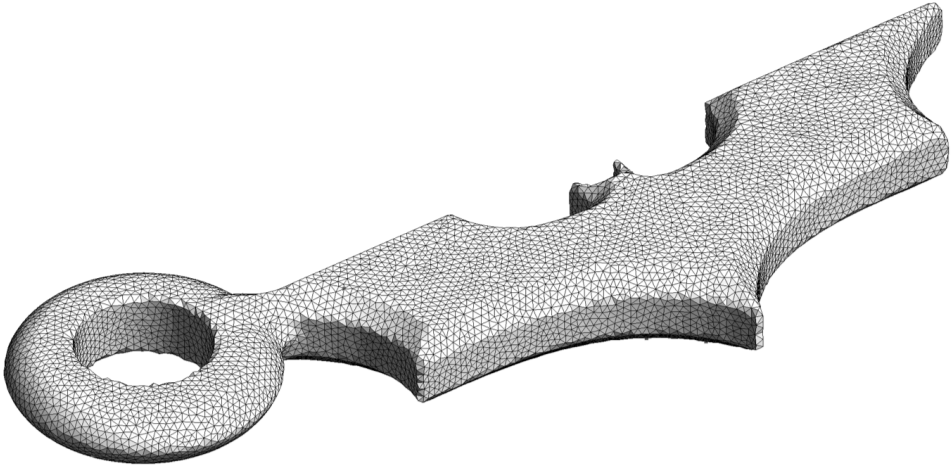
Batman STL mesh

Discrete surface meshing in Gmsh



Automatic atlas creation: each patch is provably parametrizable by solving a linear PDE, using mean value coordinates

Discrete surface meshing in Gmsh

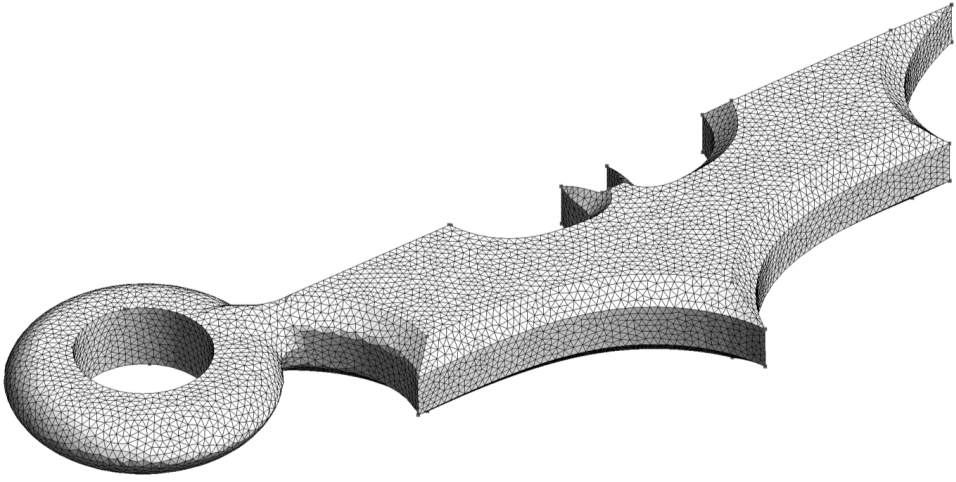


Remeshing

Discrete surface meshing in Gmsh

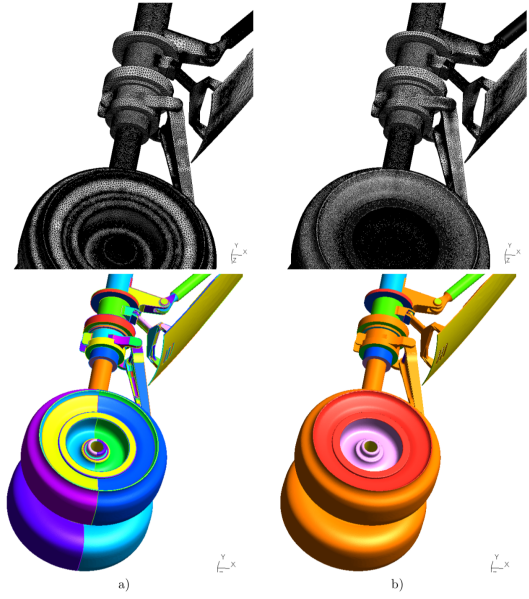
Automatic atlas creation, this time with feature edge detection

Discrete surface meshing in Gmsh

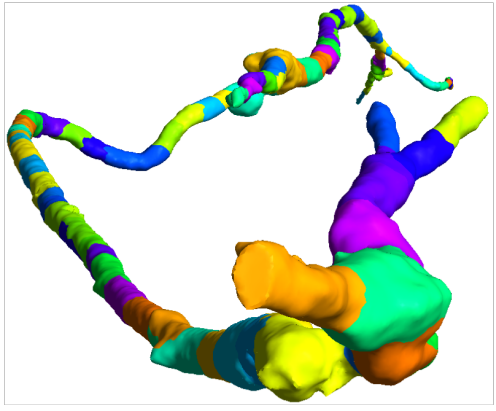


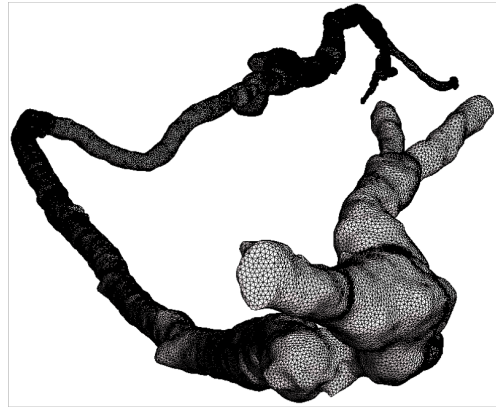
Remeshing with feature edge detection

Discrete surface meshing in Gmsh



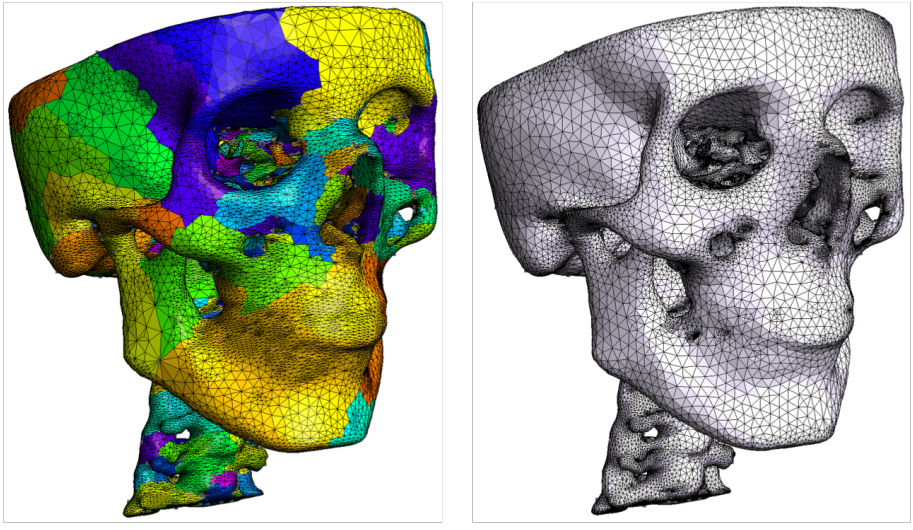
Discrete surface meshing in Gmsh





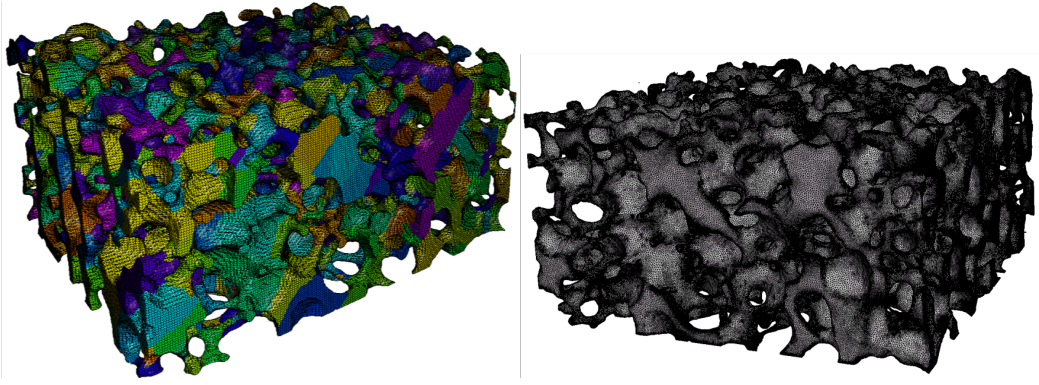
CT scan of an artery: 101 patches created, most because of the large aspect ratio

Discrete surface meshing in Gmsh



Remeshing of a skull: 715 patches created for reparametrization; mesh adapted to curvature

Discrete surface meshing in Gmsh

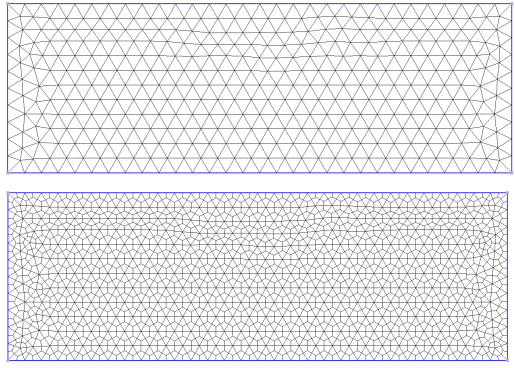


Remeshing of an X-ray tomography image of a silicon carbide foam by P. Duru, F. Muller and L. Selle (IMFT, ERC Advanced Grant SCIROCCO): 1,802 patches created for reparametrization

Unstructured quad meshing

Split Triangles – Full Quad

One triangle is divided in three quads, 20 lines of code, problem solved?



Matching

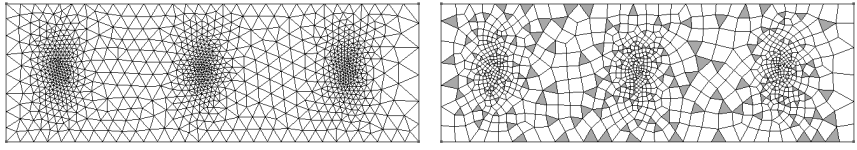
A quad q and its the four internal angles $\alpha_k, \quad k = 1, 2, 3, 4$. We define the quality $Q(q)$ of q as:

$$Q(q) = \max\left(1 - \frac{2}{\pi}\max_k\left(\left|\frac{\pi}{2} - \alpha_k\right|\right), 0\right). \tag{3}$$

Greedy *quad-dominant* algorithm [Frey & Borouchaki, *Adaptive triangular-quadrilateral mesh generation*, IJNME, 1998]

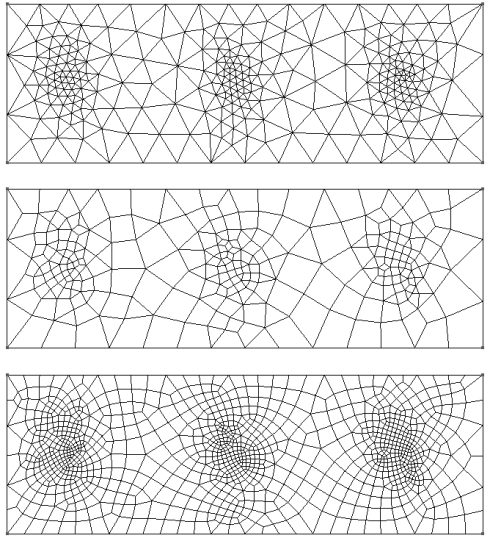
Rectangular domain of size 1×3 and a mesh size field defined by

$$h(x, y) = 0.1 + 0.08\sin(3x)\cos(6y)$$

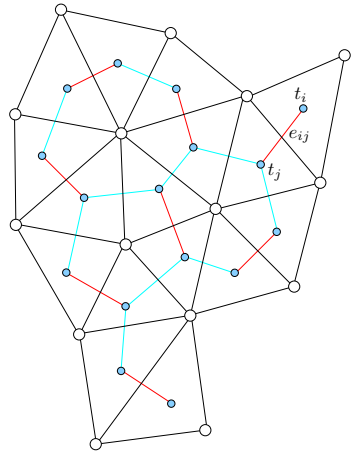


Full Quad

$2h(x, y)$ – Match – Split.



Perfect Matching



A mesh (in black) and its graph (in cyan and red). The set of graph edges colored in red forms a perfect matching

Perfect Matching

In 1965, Edmonds [Edmonds, Jack. *Paths, trees, and flowers*. Can. J. Math., 1965] invented the *Blossom algorithm* that solves the problem of optimum perfect matching in polynomial time. A straightforward implementation of Edmonds' algorithm requires $\mathcal{O}(\#V^2\#E)$ operations

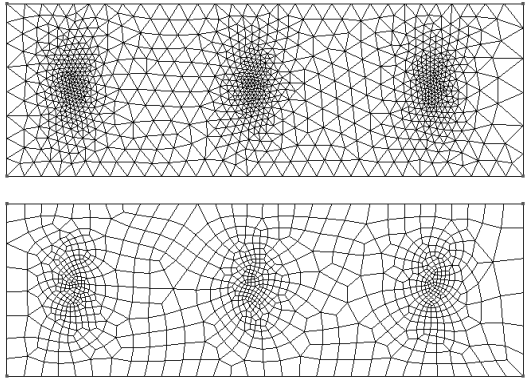
Since then, the worst-case complexity of the Blossom algorithm has been steadily improving. The current best known result is

$$\mathcal{O}(\#V(\#E + \log \#V))$$

Gmsh uses the Blossom IV code of Cook and Rohe[1], which has been considered for several years as the fastest available

[1] Computer code available at http://www2.isye.gatech.edu/~wcook/blossom4/

Perfect Matching



Try it in Gmsh (see e.g. tutorial 11)...

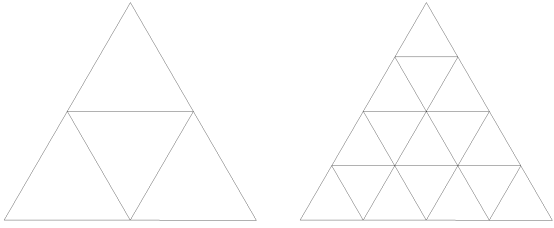
Existence of Perfect Matchings

$$n_t = 2(n_v - 1) - n_h.$$

An even number of triangles requires an even number of points on the boundary

Even if n_t is even, *there is in general no guarantee that even one single perfect matching exists in a given graph*

Tutte's theorem : A graph $G = (V, E)$ has no perfect matching if and only if there is a set $S \subseteq V$ whose removal results in more odd-sized components than the cardinality n_S of S, i.e., the number of elements in S [Pemmaraju S. and Skiena S, *Computational Discrete Mathematics'*, 2003]



Planar Graphs

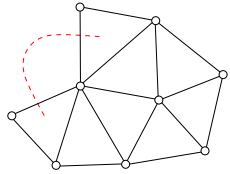
2D meshes are planar graphs. Gmsh only generates meshes in the parameter plane

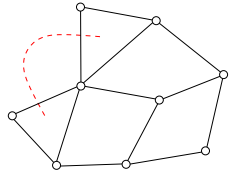
There exists an efficient algorithm (i.e., in polynomial time) that counts perfect matchings in a planar graph

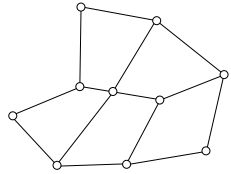
Cubic graphs, also called trivalent graphs, are graphs for which every node has exactly 3 adjacent nodes. Every cubic graph has at least one perfect matching (Oum S., *Perfect Matchings in Claw-free Cubic Graphs*). It can be proven that the number of perfect matchings in a cubic graph grows exponentially with $\#V$

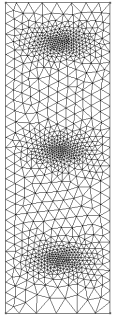
On closed surfaces, every triangular mesh has a perfect matching!

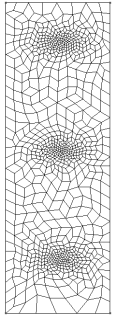
Extra Edges

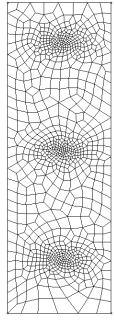


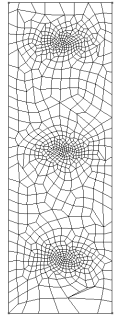


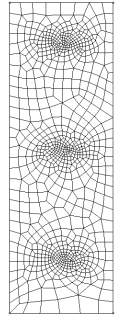






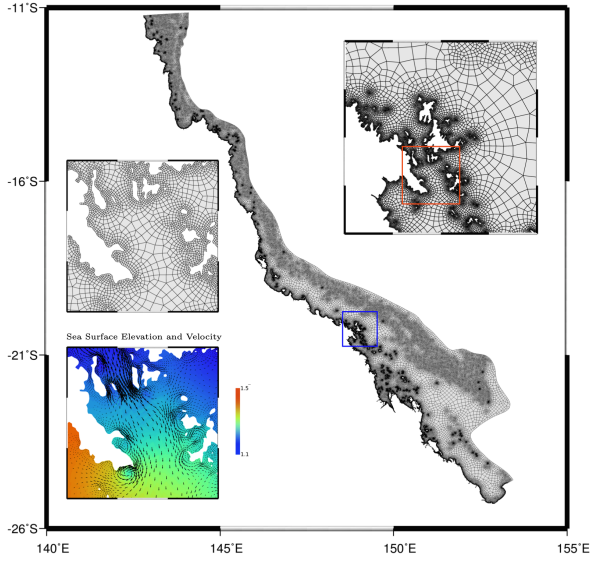






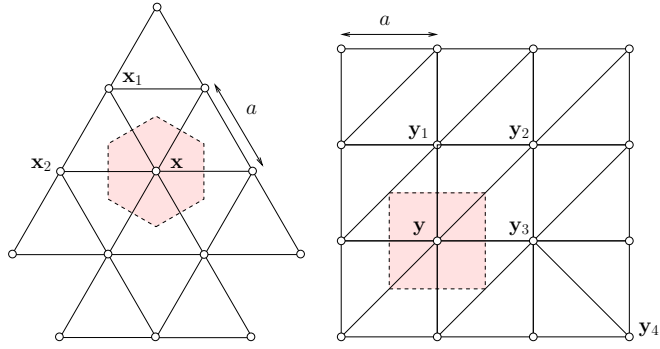
Initial triangulation	Raw Blossom application	Vertex smoothing	Topological optimization	Final mesh

Great Barrier Reef



Gmsh's second attempt: delquad

Remacle, J. F., Henrotte, F., Carrier-Baudouin, T., Béchet, E., Marchandise, E., Geuzaine, C., & Mouton, T. (2013). *A frontal Delaunay quad mesh generator using the* L^∞ *norm*. International Journal for Numerical Methods in Engineering, 94(5), 494-512



Left : The Voronoi cell of each vertex $\mathbf{x}$ is an hexagon of area $a^2\sqrt{3}/2$

Filling R^2 with equilateral triangles requires thus $2/\sqrt{3}$ times more vertices (i.e. about 15% more) than filling the same space with right triangles

Gmsh's second attempt: delquad

Gmsh's surface mesher is a delaunay-frontal algorithm. Largely inspired by [S. Rebay *Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm.* Journal of computational physics, 106(1), 125-138, 1993]

Combine the robustness of Bowyer-Watson and triangle quality control of frontal algorithms

Extension to surface meshing and the devil is in the details. One of Gmsh's oldest algorithms

An example speaks louder than a long speech

Gmsh's second attempt: delquad

Gmsh's frontal Delaunay algorithm tries its best to make equilateral triangles

A front edge e separates triangles that are "done" and other ones that are "not done"

A new point is added on the orthogonal bissector of e to eventually create an equilateral triangle

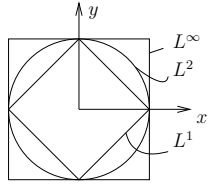
It is possible to very slightly modify the frontal algorithm to create right triangles

Triangulation in the L^∞-norm

The L^∞-norm distance

$$\|\mathbf{x}_2 - \mathbf{x}_1\|_\infty = \lim_{p\to\infty} \|\mathbf{x}_2 - \mathbf{x}_1\|_p = \max\left(|x_2 - x_1|, |y_2 - y_1|\right)$$

Unit circles

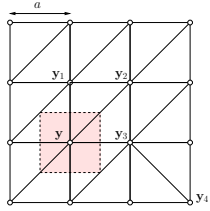


The 2-norm is the only norm that is rotationally invariant

We thus use a cross field to define a local frame at point $\mathbf{x}$

Triangulation in the L^∞-norm

In the L^∞ norm, the following mesh is made of equilateral triangles only.

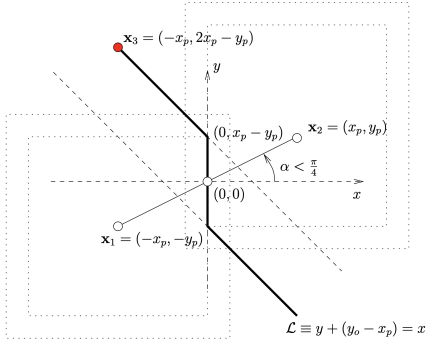


It is possible to use the same frontal-delaunay algorithm by computing orthogonal bisectors in the L^∞-norm

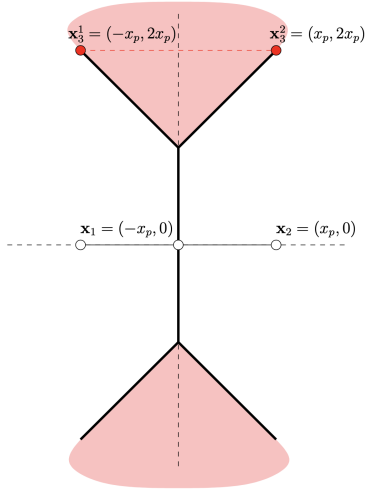
Bisectors in the L^∞-norm

The perpendicular bisector, or bisector of the segment delimited by the points $\mathbf{x}_1 = (-x_p, -y_p)$ and $\mathbf{x}_2 = (x_p, y_p)$ is by definition the set of points $\mathbf{x} = (x, y)$ equidistant to $\mathbf{x}_1$ and $\mathbf{x}_2$

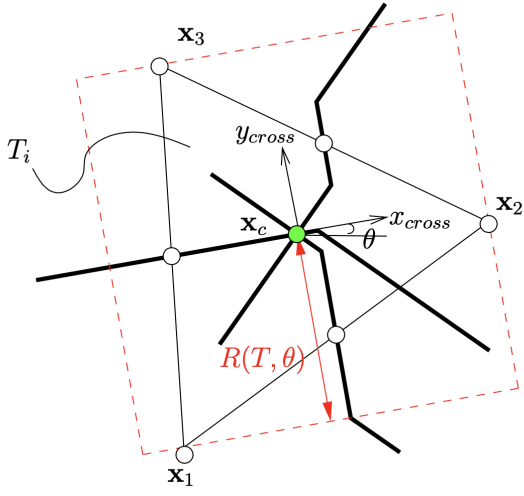
It is the union of the intersections of circles centered at $\mathbf{x}_1$ and $\mathbf{x}_2$ and having the same radius



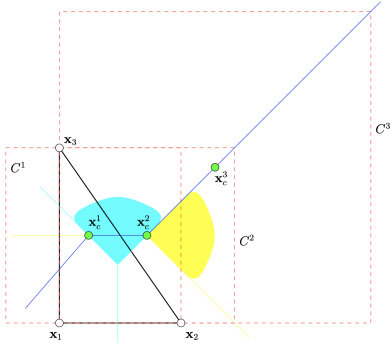
Bisectors in the L^∞-norm



Circumcenter in the L^∞-norm



Circumcenters in the L^∞-norm



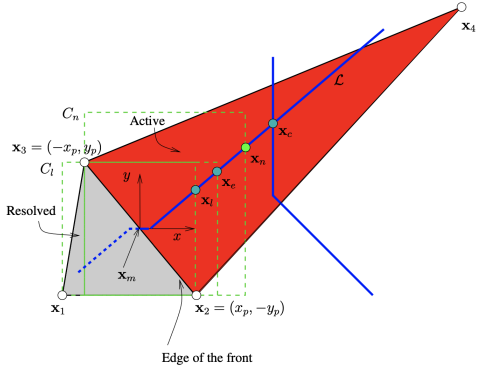
A right triangle. Perpendicular bisectors of the three segments are coloured in yellow (edge $\mathbf{x}_1\mathbf{x}_3$), blue (edge $\mathbf{x}_2\mathbf{x}_3$) and cyan (edge $\mathbf{x}_1\mathbf{x}_2$)

Points $\mathbf{x}_c^1$, $\mathbf{x}_c^2$ and $\mathbf{x}_c^3$ are three circumvents that correspond to the three circumsquares C^1, C^2 and C^3

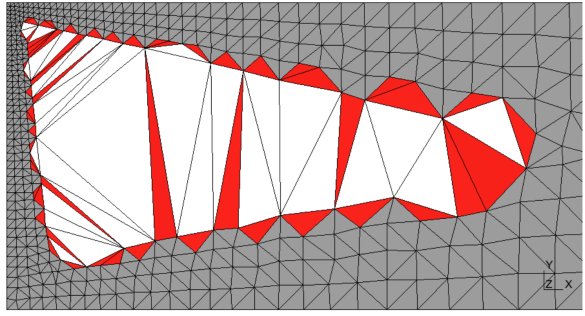
Circumcenter and circumsquare are unique when the points are in general position

Delquad

- The new point should not be placed beyond the center $\mathbf{x}_c$ of the circumsquare of the active triangle (red triangle), as this would create a triangle with a small edge $\mathbf{x}_n\mathbf{x}_4$
- The new point should not be placed below the intersection $\mathbf{x}_l$ of the bisector $\mathcal{L}$ and the circumsquare C_l of the resolved triangle $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$. Inserting a point inside C_l would make the resolved triangle invalid by means of the Delaunay criterion
- If $\delta'(\mathbf{x}_m) = \|\mathbf{x}_3 - \mathbf{x}_2\|_\infty$, then the optimal point is $\mathbf{x}_n = \mathbf{x}_e$. It corresponds to the largest triangle $T_i(\mathbf{x}_e, \mathbf{x}_2, \mathbf{x}_3)$ that verifies $R_\infty(T_i, \theta) = \delta'(\mathbf{x}_m)$



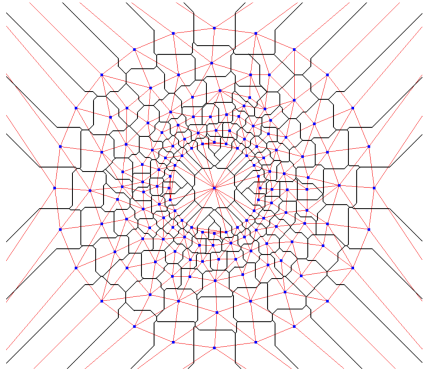
Delquad



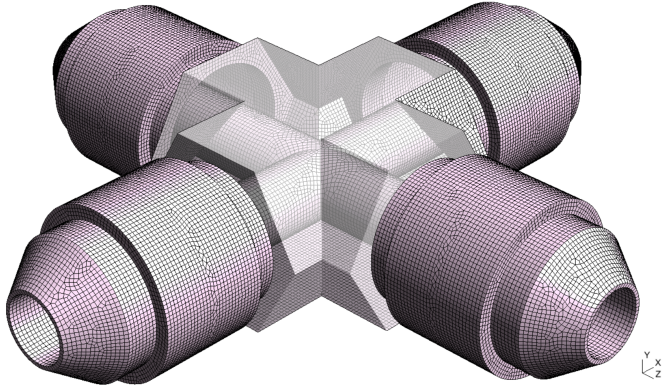
Delquad

We use standard Bowyer-Watson to connect the points i.e. we do Delaunay in the 2-norm

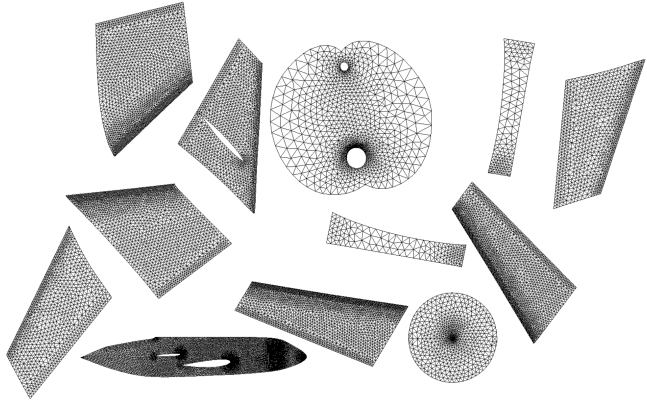
Yet, it has been observed experimentally that, in the case of finite element meshes with decent point distribution properties, the Delaunay kernel in the standard L^2-norm and the Delaunay kernel in the L^∞-norm give similar triangulations



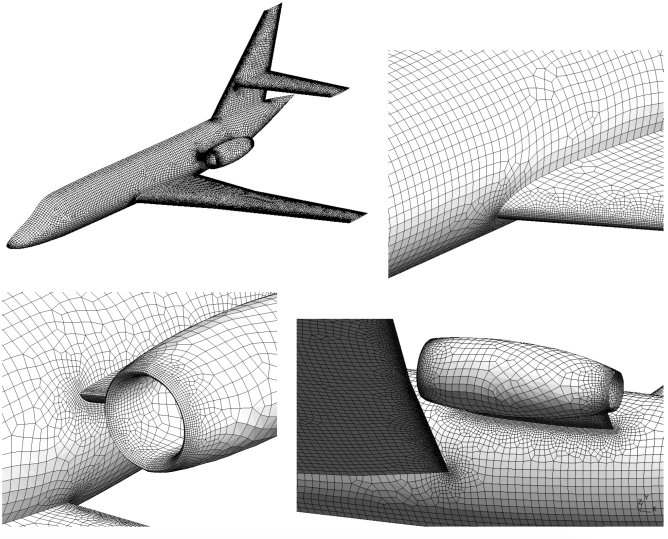
Delquad



Delquad

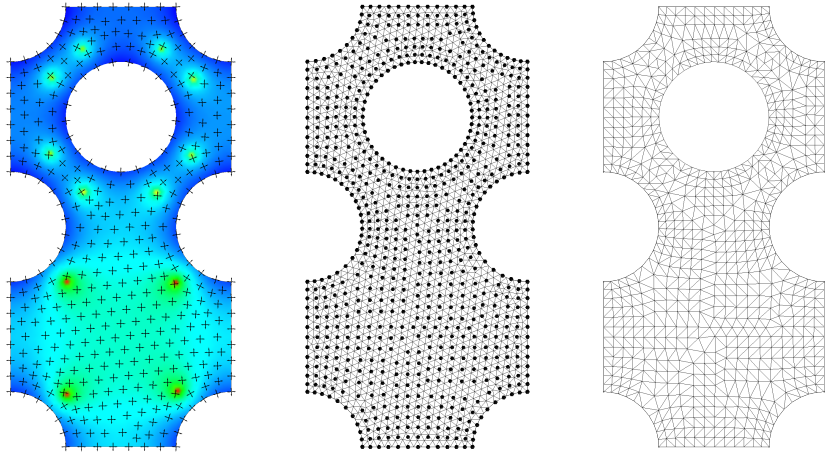


Delquad

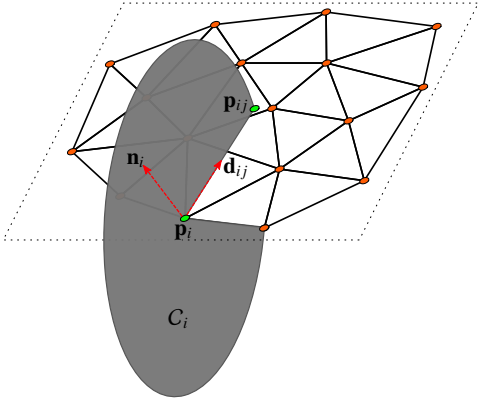


Gmsh's third (& final) attempt: pack

Baudouin, T. C., Remacle, J. F., Marchandise, E., Henrotte, F., & Geuzaine, C. (2014). *A frontal approach to hex-dominant mesh generation.* Advanced Modeling and Simulation in Engineering Sciences, 1, 1-30



Gmsh's third (& final) attempt: pack

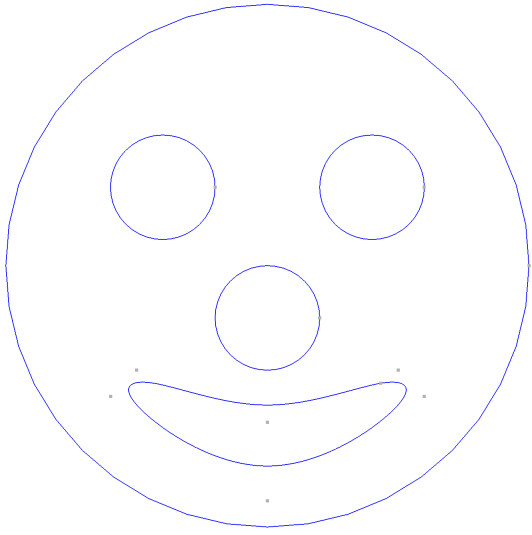


Algorithm 1 Frontal point insertion algorithm.

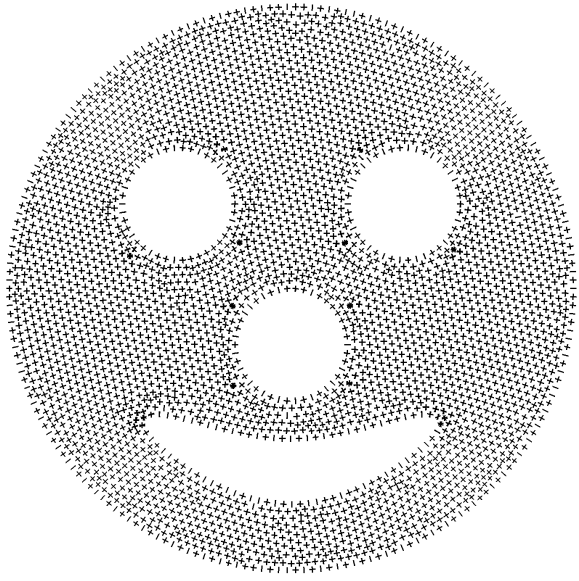
Input: Initial triangulation $\mathcal{T}_0$
cross field $\mathbf{f}$
mesh size field function $h(\mathbf{x})$
Output: Array of points P

Place boundary points in a queue
while queue is not empty **do**
  pop the first point $\mathbf{p}_i$ out of the top of the queue
  interpolate $\mathbf{f}$ and h at this point
  for $2N_d$ directions **do**
    Compute point $\mathbf{p}_{ij}$ by intersecting $\mathcal{T}_0$ with a circle
    Find set of neighboring points P_f
    for $\mathbf{p}_f \in P_f$ **do**
      if $\|\mathbf{p}_{ij} - \mathbf{p}_f\| > \alpha h(\mathbf{p}_{ij})$ **then**
        add $\mathbf{p}_{ij}$ in P
        push $\mathbf{p}_{ij}$ in the back of the queue
      else
        delete $\mathbf{p}_{ij}$
      end if
    end for
  end for
end while

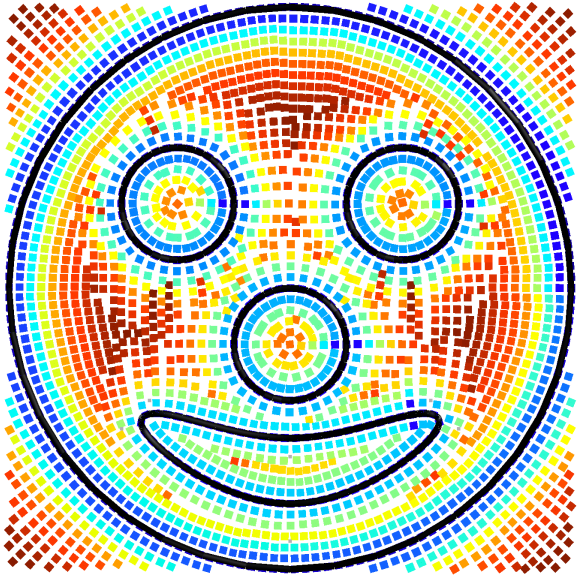
Gmsh's third (& final) attempt: pack



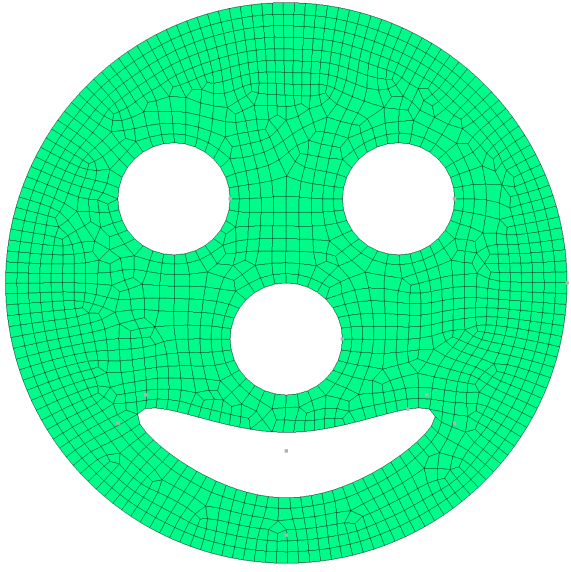
Gmsh's third (& final) attempt: pack



Gmsh's third (& final) attempt: pack

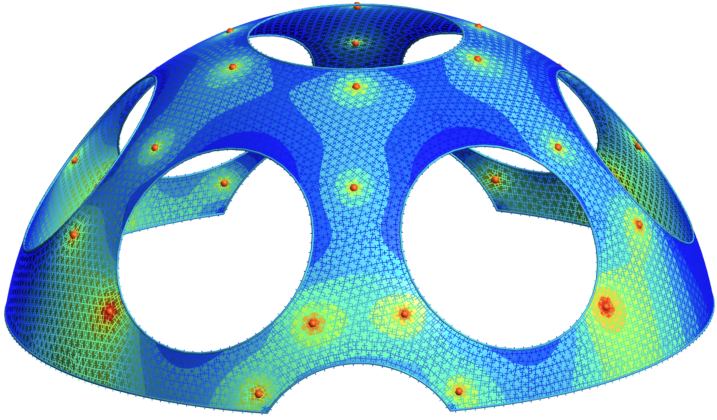


Gmsh's third (& final) attempt: pack



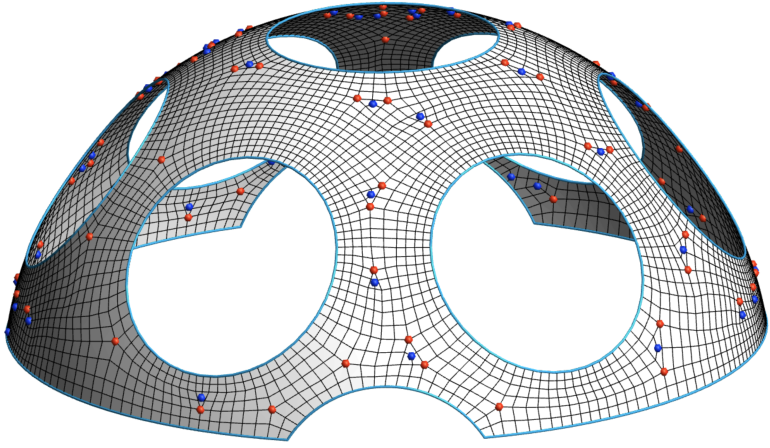
Improving pack: quadqs

[M. Reberol et al. 2021]



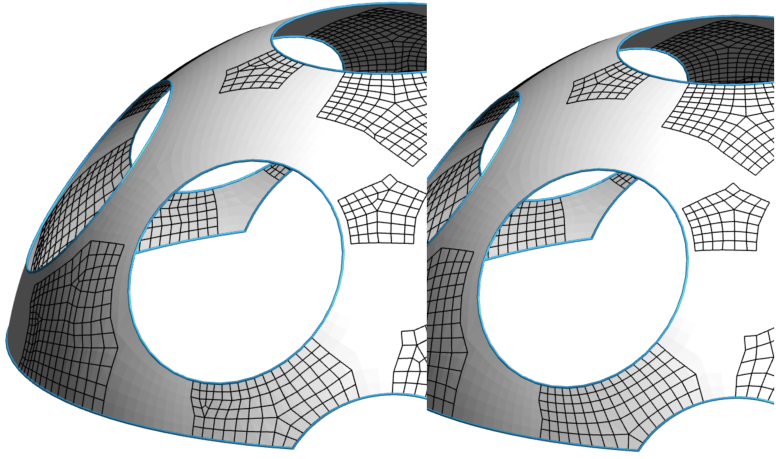
Compute a (scaled) cross-field with multilevel diffusion

Improving pack: quadqs



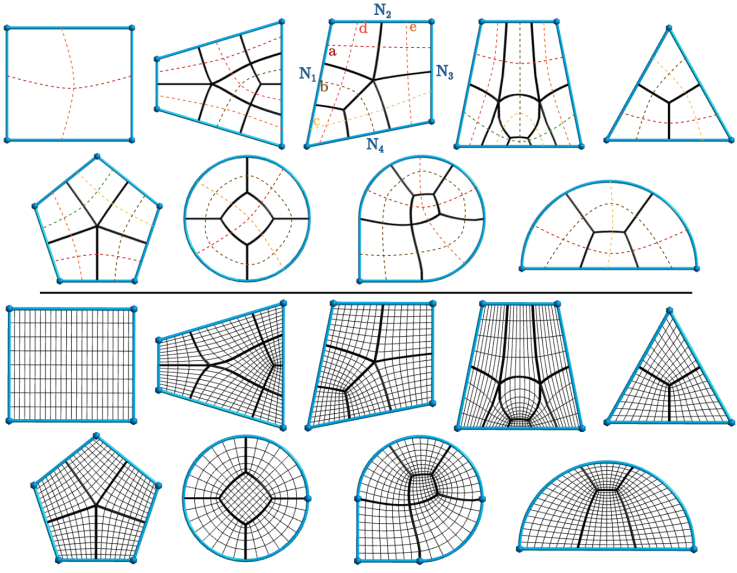
Build a unstructured quadrilateral mesh with a frontal approach guided by the scaled cross field

Improving pack: quadqs



Pattern-based quadrilateral meshing and cavity remeshing to eliminate unnecessary irregular vertices while preserving the cross field singularities

Improving pack: quadqs



Improving pack: quadqs

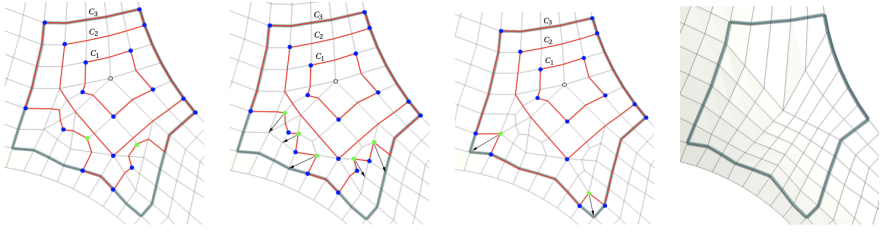
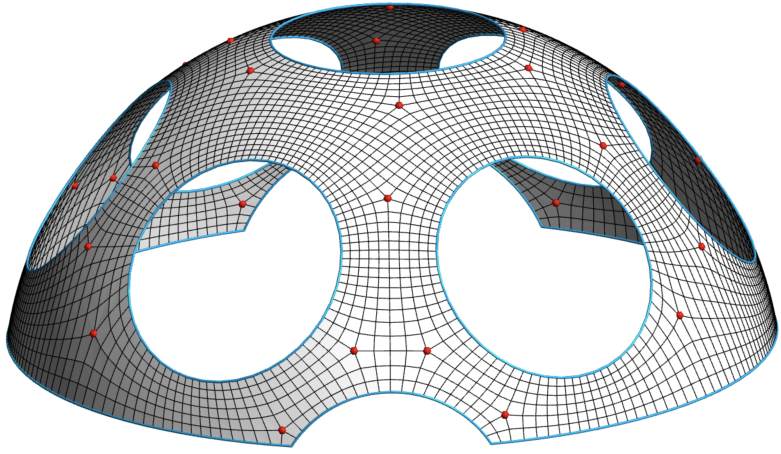


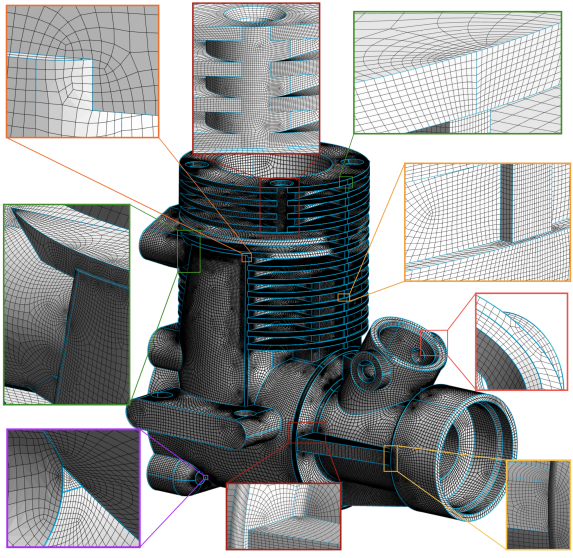
Fig. 6: Growing a cavity around one vertex of index -1 (in pink). Convex corners are in blue and concave corners are in green. The remeshed cavity (d) has one irregular vertex instead of eleven.

Improving pack: quadqs



The final quad mesh is very similar to the one obtained with the global parametrization approach and has the same number of irregular vertices

Improving pack: quadqs



- "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

Ongoing: Pragmatic quad mesher

- Boundary curves are discretized first, surfaces are then meshed using the 1D discretization of the curves.
- For a surface to be meshed exclusively with quadrilaterals, the total number of subdivisions assigned to the curves bounding that surface must be even.
- While enforcing an even number of segments on every individual curve is a sufficient condition, it typically induces substantial changes to the 1D mesh.
- Idea – change the 1D mesh as few as possible

Ongoing: Pragmatic quad mesher

- Each face F_j is bounded by a set of curves $J_j \subset \{1, \dots, E\}$. We call s_i the final number of subdivisions that fulfills the following parity conditions:

$$\forall j, \qquad \sum_{i \in J_j} s_i \equiv 0 \pmod 2.$$

- We thus minimize the following weighted-cost function:

$$\min_{n \in \mathbb{Z}^m_{\geq 0}} \quad \sum_{i=1}^{m} w_i \, \mathbf{1}_{\{s_i \neq S_i\}} \quad \text{subject to parity constraints.}$$

  Here, $\mathbf{1}_{\{s_i \neq S_i\}}$ is an *indicator function*:

$$\mathbf{1}_{\{s_i \neq S_i\}} = \begin{cases} 1 & \text{if } s_i \neq S_i, \\ 0 & \text{if } s_i = S_i. \end{cases}$$

- We define a weight $w_i > 0$ typically chosen so that $w_i = \frac{1}{S_i}$.

Ongoing: Pragmatic quad mesher

- Let us first define the *face–curve incidence matrix*

$$A \in \{0,1\}^{F \times E}.$$

The entry A_{ji} is defined as

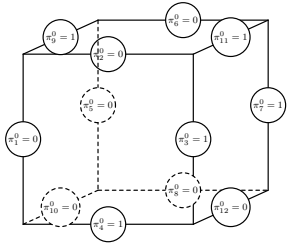
$$A_{ij} = \begin{cases} 1 & \text{if curve } E_j \text{ lies on the boundary of face } F_i, \\ 0 & \text{otherwise.} \end{cases}$$

- A sum of integers is even if and only if the number of odd terms in the sum is even, we define a set of unknowns $\pi_i = s_i \pmod 2$: $\pi_i = 1$ if s_i is odd and $\pi_i = 0$ if s_i is even.
- Parity conditions

$$A\pi \equiv 0 \pmod 2.$$

Ongoing: Pragmatic quad mesher

Initial solution:



- A cube :

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

Ongoing: Pragmatic quad mesher

- We perform Gaussian elimination on A over the finite field $\mathbb{F}_2$ to get

$$A \sim \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

- Row 6 is null – for every closed volume, one equation is redundant.
- Initial parities given by Gmsh are $\pi^0 = (0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0)$.
- Heuristic – unknowns are sorted – $S_1 > S_2 \ldots$.
- We can use the "initial" solution to fix $(\pi_6, \ldots, \pi_{12})$.

Ongoing: Pragmatic quad mesher

We start with $\pi = (\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, 0, 1, 0, 1, 0, 1, 0)$.

$$\pi_1 = \underbrace{\pi_6}_{0} + \underbrace{\pi_7}_{1} + \underbrace{\pi_8}_{0} + \underbrace{\pi_9}_{1} + \underbrace{\pi_{10}}_{0} \pmod 2 \quad \rightarrow \quad \pi_1 = 0$$

$$\pi_2 = \underbrace{\pi_6}_{0} + \underbrace{\pi_9}_{1} + \underbrace{\pi_{11}}_{1} \pmod 2 \quad \rightarrow \quad \pi_2 = 0$$

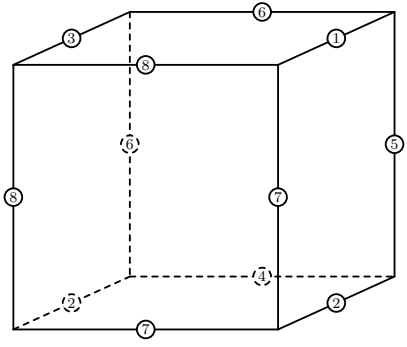
$$\pi_3 = \underbrace{\pi_7}_{1} + \underbrace{\pi_{11}}_{1} + \underbrace{\pi_{12}}_{0} \pmod 2 \quad \rightarrow \quad \pi_3 = 0$$

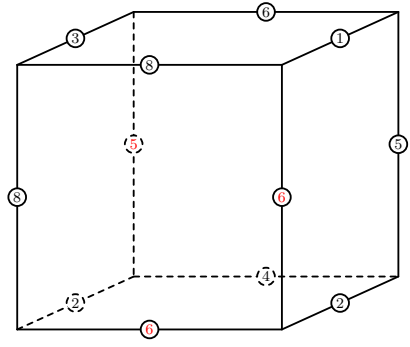
$$\dots$$

$$\pi = (0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0).$$

Ongoing: Pragmatic quad mesher

Initial Solution





Ongoing: Pragmatic quad mesher

Improved solution $\pi^{(i)} = \pi + \sum_{\ell=1}^{7} \alpha_\ell \, v^{(\ell)}$,

- π is the initial solution and the $v^{(\ell)}$ form a basis of the null space of A and $\alpha_l \in \{0, 1\}$.

$$A \sim \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

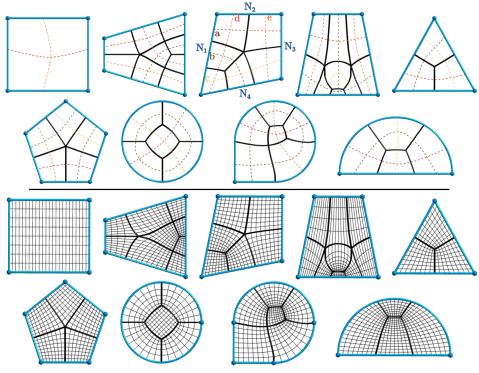
- Do not look for all $v^{(\ell)}$, choose a small subset. Here just choose $i = 6$ and start with $v^{(1)} = (v_2^{(1)}, v_2^{(1)}, v_3^{(1)}, v_4^{(1)}, v_5^{(1)}, 1, 0, 0, 0, 0, 0, 0)$. and thus

$$v^{(1)} = (1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0)$$

Ongoing: Pragmatic quad mesher

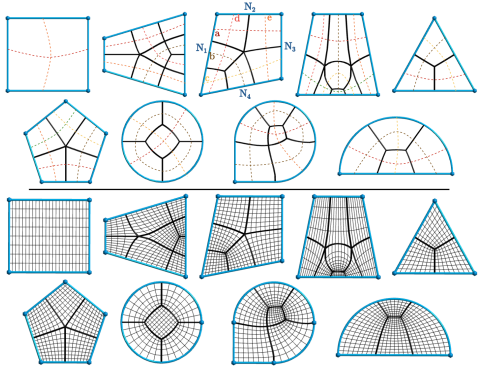
Patterns:

- CAD models are made of many patches, lots of them of *simple topology*.
- In Gmsh, we only encoded a few of those, more will come.

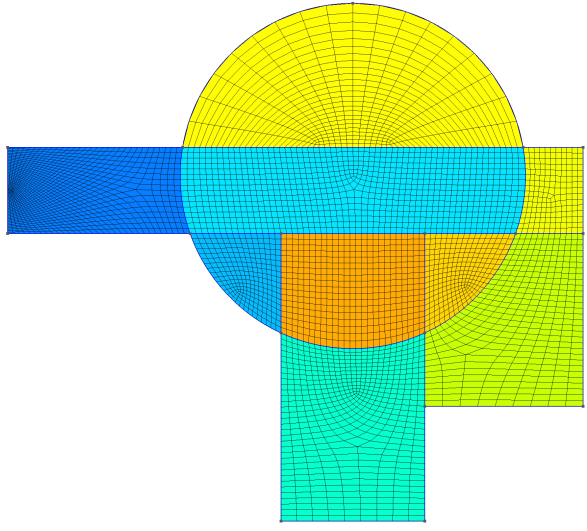


Ongoing: Pragmatic quad mesher

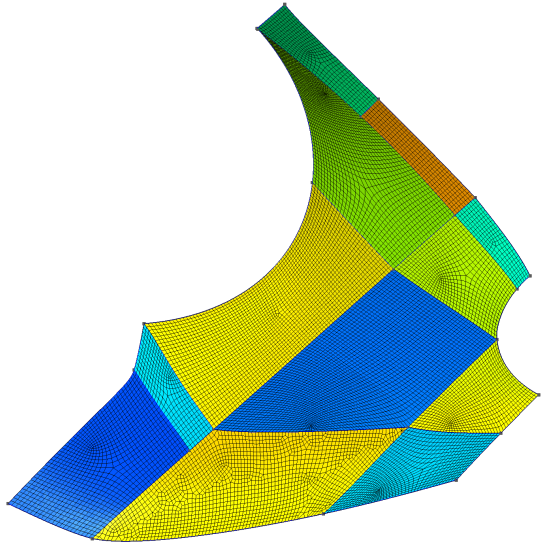
- Look for faces with $\{0, \ldots, 5\}$ convex corners, no holes, no concave corners.
- Apply pattern if all chords are positive
- Smooth using enhanced winslow (now works on surfaces, could be taylored for very anisotropic meshes)
- If mesh is bad, restore the unstructured one.



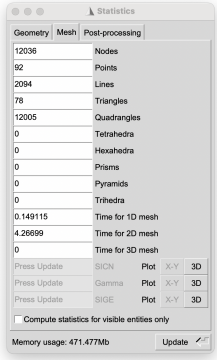
Ongoing: Pragmatic quad mesher

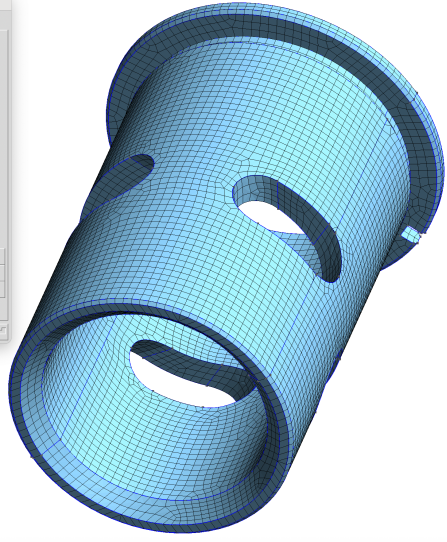


Ongoing: Pragmatic quad mesher



Ongoing: Pragmatic quad mesher





Unstructured hex meshing

Frontal approach

- Create a 3D frame/size field
- Generate points on surfaces & on volumes using the same approach
- Tetrahedralize the points (+ recover features)
- Subdivide tetrahedra into hexahedra
- Create a all-hex mesh?

```
gmsh Kolben.stp -clmin .3 -clmax .3 -hybrid -3 -nt 8
```

Subdividing a hexahedron into tetrahedra

Bounds on the number of tetrahedra:

$$n_v - n_e + n_f - n_t = 1$$

We have $n_v = 8$,

$$n_e = n_{ie} + n_{be} \quad \text{with} \quad n_{be} = 12 + 6 = 18$$

$$n_f = n_{if} + n_{bf} \quad \text{with} \quad n_{bf} = 2 = 12$$

$$4n_t = 2n_{if} + n_{bf} \quad \rightarrow \quad n_{if} = 2n_t - 6$$

All together (H. Edelsbrunner et al, *Tetrahedrizing point sets in three dimensions*, Journal of Symbolic Computation 10 (1990) 335–347)

$$8 - n_{ie} - 18 + (2n_t - 6) + 12 - n_t = 1 \quad \rightarrow \quad n_t = n_{ie} + 5$$

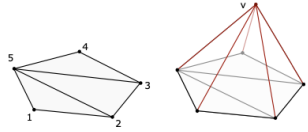
Since there are at most $n_{ie} = \binom{8}{2} - n_{be} = 10$ interior edges, we have the bounds

$$\boxed{5 \leq n_t \leq 15}$$

174 Subdivisions

Pellerin, J., Verhetsel, K., & Remacle, J. F. (2018). *There are 174 Subdivisions of the Hexahedron into Tetrahedra*. ACM Transactions on Graphics (TOG), 37(6), 1-9

A triangulation of the 2-sphere can be constructed from the triangulation of a 2-ball by building a cone



The inverse transformation, the removal of one point v of the sphere triangulation as well as all triangles incident to v, permits to obtain the triangulation of a ball

The 3-sphere is defined as the 3-dimensional boundary of a 4-dimensional ball

There are 1296 triangulations of the 3-sphere with 9 points [Altshuler et al, *The classification of simplicial 3-spheres with nine vertices into polytopes and nonpolytopes.* Discrete Mathematics 31, 2 (1980), 115–124)]

Dual complex

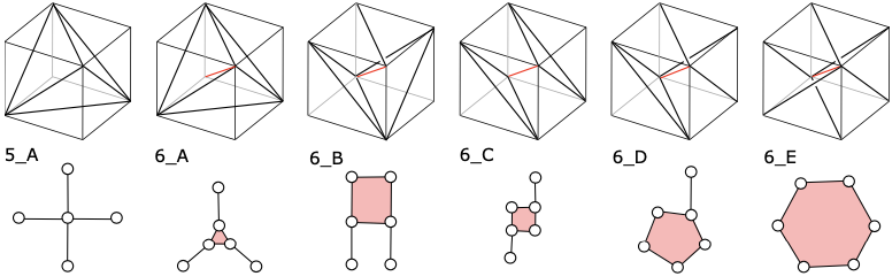
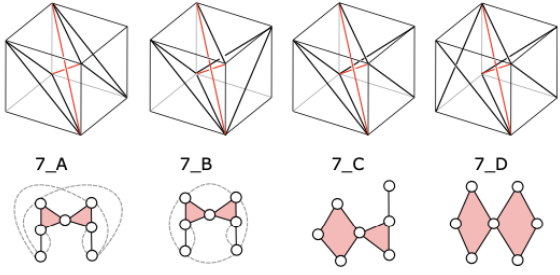


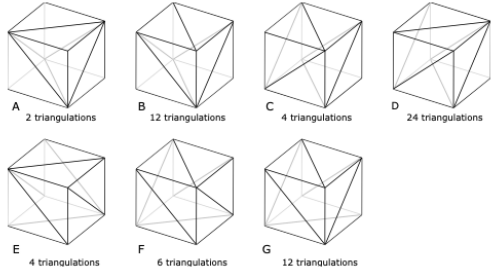
Figure 5: The six types of tetrahedrizations of the 3-cube and their dual complex representation.



174 Subdivisions

Nine triangulations of the 3-ball with eight vertices can be built from each of the 1296 triangulations byremoving one of thevertices $v_i, i = 1, \ldots, 9$ and its link, i.e. all tetrahedra incident to v_i

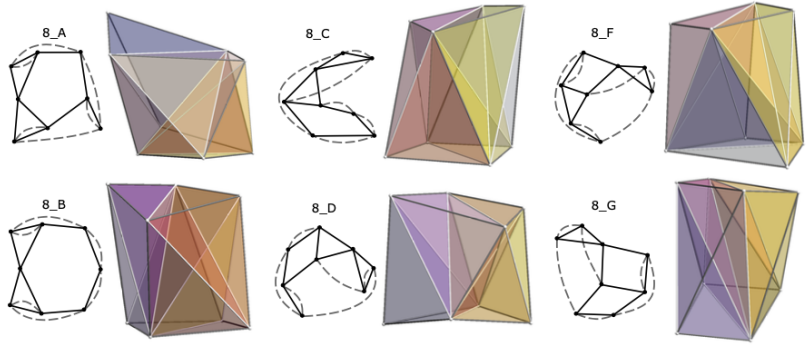
The triangulation of the boundary of a hexahedron has 8 vertices and 18 edges. Among these, 12 are fixed and there are 2 possibilities to place the remaining 6 diagonals of the quadrilateral facets. We have then $2^6 = 64$ possible triangulations. These triangulations can be classified into 7 equivalence classes, i.e. there are 7 triangulations of the hexahedron boundary up to isomorphism



174 Subdivisions

The hexahedron has 174 combinatorial triangulations up to isomorphism that do not contain any boundary tetrahedra

Among those 174 combinatorial triangulations, the 171 triangulations that admit an oriented matroid have a realization. The other ones cannot be realized



174 Subdivisions

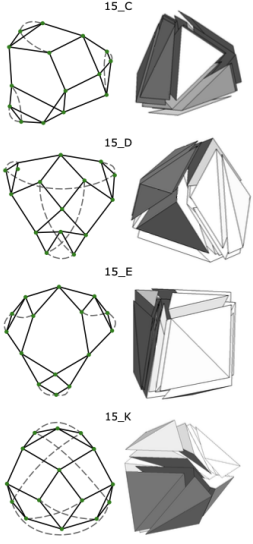


Fig. 13. The four realizable triangulations with 15 tetrahedra and with points in convex position. The hexahedra are valid, their Jacobian is strictly positive.

Table 4. Number of triangulations patterns per number of tetrahedra counted in the Delaunay triangulations of random point sets.

#vertices	5	6	7	8	9	10	11	12	13	14	15	Total
3,000	1	5	2	7	13	16	4	0	0	0	0	51
10,000	1	5	5	7	13	19	10	2	0	0	0	62
20,000	1	5	5	7	13	19	15	2	0	0	0	67
100,000	1	5	5	7	13	20	24	5	0	0	0	80
500,000	1	5	5	7	13	20	28	12	1	0	0	92
1,000,000	1	5	5	7	13	20	30	14	0	0	0	95
2,000,000	1	5	5	7	13	20	30	15	4	1	0	101
5,000,000	1	5	5	7	13	20	30	16	4	1	0	102
10,000,000	1	5	5	7	13	20	31	16	6	1	0	105

Table 3. Number of triangulations patterns per number of tetrahedra counted in the available input data of [Pellerin et al. 2018].

Model	#vert .	5	6	7	8	9	10	11	12	13	14	15	Total
Cube	127	1	5	2	1	0	0	0	0	0	0	0	9
Fusee	11,975	1	5	5	7	13	9	4	0	0	0	0	44
CShaft	23,245	1	5	5	7	13	17	6	0	0	0	0	54
Fusee_1	71,947	1	5	5	7	7	1	0	0	0	0	0	26
Caliper	130,572	1	5	5	7	7	1	0	0	0	0	0	26
CShaft2	140,985	1	5	5	7	8	0	1	0	0	0	0	27
Fusee_2	161,888	1	5	5	7	7	1	0	0	0	0	0	26
FT47_b	221,780	1	5	5	7	8	2	0	0	0	0	0	28
FT47	370,401	1	5	5	7	7	2	0	0	0	0	0	27
Fusee_3	501,021	1	5	5	7	8	4	0	0	0	0	0	30
Los1	583,561	1	5	5	7	7	2	0	0	0	0	0	27
Knuckle	3,058,481	1	5	5	7	8	2	0	0	0	0	0	28

Combining

Pellerin, J., Johnen, A., & Remacle, J. F. (2017). *Identifying combinations of tetrahedra into hexahedra: a vertex based strategy.* Procedia engineering, 203, 2-13

1. A set of mesh vertices V is initially sampled in the domain
2. A tetrahedral mesh T is built by connecting V, e.g. using a Delaunay kernel like
3. A set H of potential hexahedra that can be constructed by combining some tetrahedra of T is created
4. A maximal subset $H_c \subset H$ of compatible hexahedra is determined It has been shown that this stage can be formally written as a maximal clique problem
5. The tetrahedra, T', that are not combined into hexahedra are combined into prisms, pyramids, or remain unchanged in the final *hex-dominant mesh*

Combining

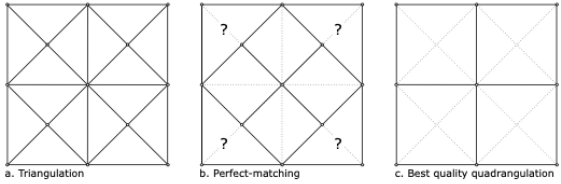


Figure 1: Combining pairs of triangles into quadrilaterals may not lead to the best quadrilateral mesh.

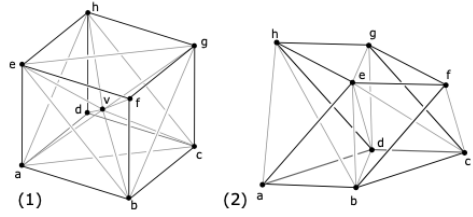


Figure 2: Two potential hexahedra that are not identified by existing combination methods. (1) A decomposition with an interior vertex v. (2) A decomposition into eight tetrahedra. This is a counter example to the claim of [7] that there is no decomposition into more than 7 interior tetrahedra.

Combining

Eight vertices of the tetrahedral mesh T define a potential hexahedron if (1) the twelve hexahedron edges are edges of T and if (2) the six quadrilateral hexahedron faces can be formed by merging two triangular facets of T

This starting point is quite general and allows to automatically detect potential hexahedra without having to define a priori decomposition patterns into tetrahedra

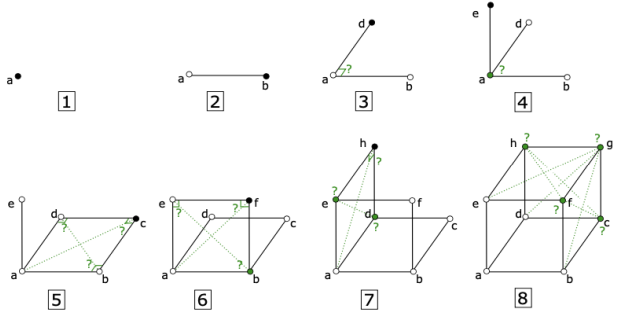


Figure 9: Vertex based search algorithm to build one hexahedron. Starting from one vertex a, the 7 other vertices are added one after another searching for vertices that are that are adjacent through edges of the tetrahedral mesh. Tests on the existence of triangulations of faces ensure the existence of the hexahedron boundary. Tests on the quality of 2D face angles and 3D hexahedron angles help the quick discard of bad hexahedra.

Maximal independent set

In graph theory, an independent set is a set of vertices in a graph, no two of which are adjacent

Create a graph – nodes at the potential hexes and an edge exist between two hexes if they share a tet

The optimization problem of finding a maximum independent set is a strongly NP-hard problem (in 2D, Blossom is polynomial!)

Greedy algorithm: choose the best hex h, remove all hexes that are connected to h i.e. that share a tet with h, choose the best remaining hex and so on

Hex-dominant

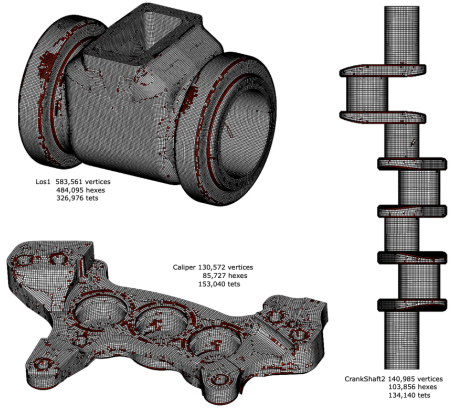
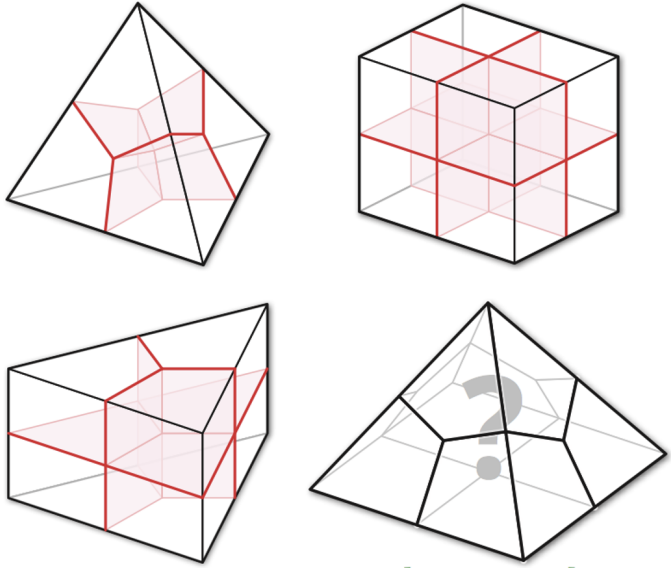


Figure 10: Hexahedral dominant meshes generated by greedy selection of the best quality hexahedra among those identified by our algorithm (white: hexahedra, red: tetrahedra). All potential hexahedra are identified in typically less than a minute, the greedy selection runs in a few seconds.

```
gmsh Los1.stp -clmin 1.5 -clmax 1.5 -hybrid -3 -nt 8
```

Full Hex?



Full Hex?

Verhetsel, K., Pellerin, J., & Remacle, J. F. (2019). *A 44-element mesh of Schneiders' pyramid Bounding the difficulty of hex-meshing problems. Computer Aided Design*

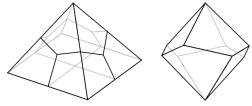


Fig. 1: Left: Schneiders'pyramid. Right: The octogonal spindle.

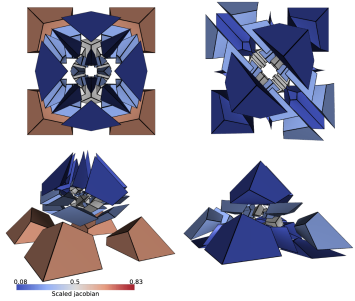


Fig. 2: Comparison of our 44-element mesh of Schneiders' pyramid (left) with the smallest known 36-element solution (right). Both admit two planar symmetries.

Full Hex?

Erickson J. (2014). *Efficiently hex-meshing things with topology.* Discrete & Computational Geometry 52, 3 (2014), 427–449

Verhetsel, K., Pellerin, J., & Remacle, J. F. (2019). *Finding hexahedrizations for small quadrangulations of the sphere.* ACM Transactions on Graphics (TOG), 38(4), 1-13

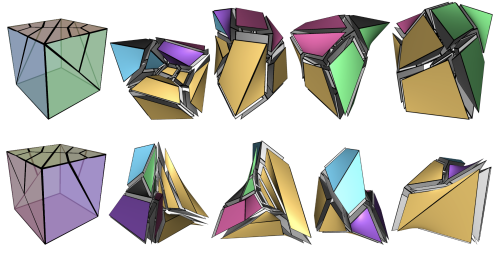


Fig. 10. Hexahedrizations of the two types of buffer cubes used to mesh arbitrary domains in the algorithm of Erickson [2014]. (top) 37 hexahedra to mesh the 20-quadrangle cell; (bottom) 40 hexahedra to mesh the 22-quadrangle cell. Colors correspond to the different sides of the original cubes (shown on the left).

Any ball-shaped domain bounded by n quadrangles can be meshed with no more than $78n$ hexahedra. This paper gives bounds that are very significantly lowers the previous upper bound of $5396n$

Gmsh-based solvers

GmshFEM and GmshDDM

C++ finite element and domain decomposition libraries based on the Gmsh API [A. Royer et al. 2022]

- Symbolic symbolic high-level description of weak formulations
- General coupled formulations in 1D, 2D, 2D-axi and 3D

GmshFEM and GmshDDM

C++ finite element and domain decomposition libraries based on the Gmsh API [A. Royer et al. 2022]

- Symbolic symbolic high-level description of weak formulations
- General coupled formulations in 1D, 2D, 2D-axi and 3D
- Arbitrarily high-order Lagrange and hierarchical basis functions
- Scalar and vector fields (L2, H1, H(curl)) on hybrid, curved meshes
- Real and complex arithmetic, single or double precision

GmshFEM and GmshDDM

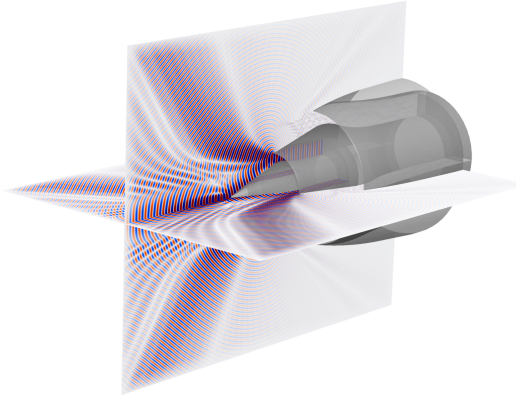
C++ finite element and domain decomposition libraries based on the Gmsh API [A. Royer et al. 2022]

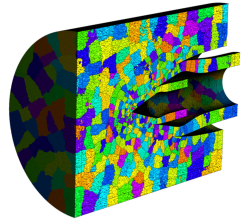
- Symbolic symbolic high-level description of weak formulations
- General coupled formulations in 1D, 2D, 2D-axi and 3D
- Arbitrarily high-order Lagrange and hierarchical basis functions
- Scalar and vector fields (L2, H1, H(curl)) on hybrid, curved meshes
- Real and complex arithmetic, single or double precision
- Parallelization and linear algebra backends:
  - GmshFEM multi-threaded using OpenMP, linear algebra using Eigen and PETSc, eigensolver using SLEPc
  - GmshDDM: distributed memory parallelization using MPI, iterative Krylov solver using PETSc (incl. HPDDM)

GmshFEM and GmshDDM

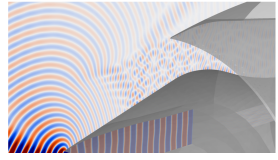
```
// Domains
Domain omega("omega"), gammaScat("scat"), gammaExt("ext");

// Finite element field
Field<Scalar, form::Form0> u("u", omega,
                             functionSpaceH1::HierarchicalH1,
                             6); // polynomial degree 6

// Dirichlet constraint
complex<double> im = complex<double>(0., 1.);
double k = 50;
Function<complex<double>, Degree::Degree0> uInc =
  exp<complex<double>>(im * k * z<complex<double>>());
u.addConstraint(gammaScat, -uInc);

// Weak formulation
Formulation<Scalar> f("helmholtz");
const string g = "Gauss12";

f.integral(      grad(dof(u)), grad(tf(u)), omega, g);
f.integral(- k * k *  dof(u) ,      tf(u) , omega, g);
f.integral(- im * k * dof(u) ,      tf(u) , gammaExt, g);
```

GmshFEM and GmshDDM





(b)



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

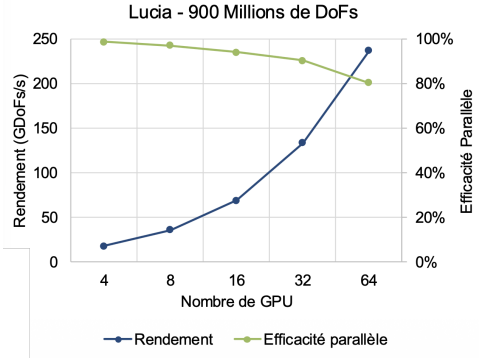
Acoustic noise from turbofan engine exhaust (4096 partitions, LUMI)

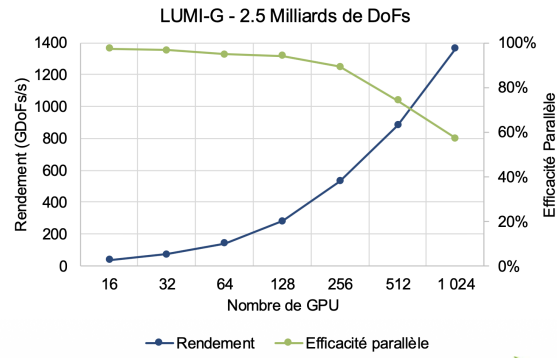
GmshDDM & GmshDG on GPU

- GmshDDM is being ported to GPU to speed up the iterative process

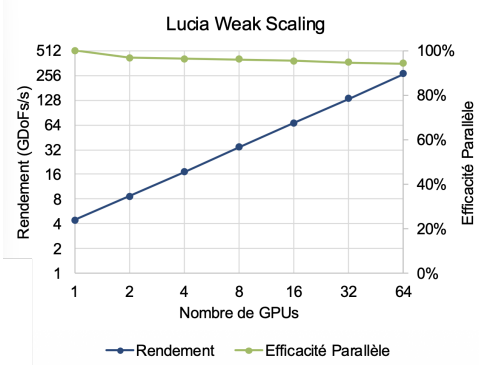
GmshDDM & GmshDG on GPU

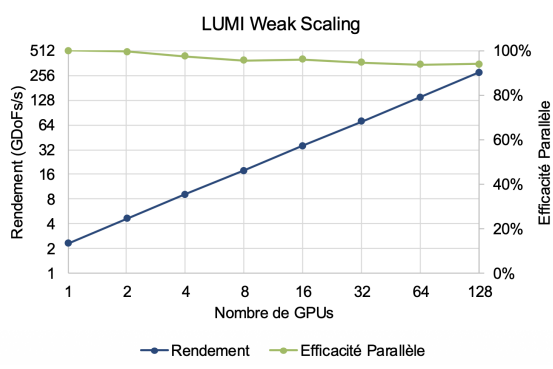
- GmshDDM is being ported to GPU to speed up the iterative process
- We have also added multi-GPU support to our time-domain Discontinuous Garlerkin code GmshDG for Maxwell:





GmshDDM & GmshDG on GPU





Conclusions and perspectives

Conclusions and perspectives

- Overview of Gmsh:
  - Unstructured triangulations
  - Surface meshing, parametrizations, high-order meshes
  - Optimal meshes
  - Unstructured quad and hex meshing
  - Gmsh-based solvers

Conclusions and perspectives

- Overview of Gmsh:
  - Unstructured triangulations
  - Surface meshing, parametrizations, high-order meshes
  - Optimal meshes
  - Unstructured quad and hex meshing
  - Gmsh-based solvers

- Exciting perspectives:
  - Improved high-order meshing
  - 3D boundary layers
  - Polygons and polyhedra?

Conclusions and perspectives

- Overview of Gmsh:
  - Unstructured triangulations
  - Surface meshing, parametrizations, high-order meshes
  - Optimal meshes
  - Unstructured quad and hex meshing
  - Gmsh-based solvers

- Exciting perspectives:
  - Improved high-order meshing
  - 3D boundary layers
  - Polygons and polyhedra?

- **First Gmsh User Meeting**: 8-9 July 2026 in Liège!