INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2009; **0**:1–24 *Prepared using* nmeauth.cls *[Version: 2002/09/18 v2.02]*

This is a preprint of an article accepted for publication in the International Journal for Numerical Methods in Engineering, Copyright ©2009 John Wiley & Sons, Ltd.

Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities

Christophe Geuzaine[1], Jean-François Remacle[2*]

[1] *Université de Liège, Department of Electrical Engineering And Computer Science, Montefiore Institute, Liège, Belgium.*

[2] *Université catholique de Louvain, Institute for Mechanical, Materials and Civil Engineering, Louvain-la-Neuve, Belgium*

SUMMARY

Gmsh is an open-source three-dimensional finite element grid generator with a build-in CAD engine and post-processor. Its design goal is to provide a fast, light and user-friendly meshing tool with parametric input and advanced visualization capabilities. This paper presents the overall philosophy, the main design choices and some of the original algorithms implemented in Gmsh. Copyright © 2009 John Wiley & Sons, Ltd.

KEY WORDS: Computer Aided Design, Mesh generation, Post-Processing, Finite Element Method, Open Source Software

1. Introduction

When we started the Gmsh project in the summer of 1996, our goal was to develop a *fast*, *light* and *user-friendly* interactive software tool to easily create geometries and meshes that could be used in our three-dimensional finite element solvers [8], and then visualize and export the computational results with maximum flexibility. At the time, no open-source software combining a CAD engine, a mesh generator and a post-processor was available: the existing integrated tools were expensive commercial packages [41], and the freeware or shareware tools were limited to either CAD [29], two-dimensional mesh generation [44], three-dimensional mesh generation [53, 21, 30], or post-processing [31]. The need for a free integrated solution was conspicuous, and several projects similar in spirit to Gmsh were also born around the same time—some of them still actively developed today [39, 6]. Gmsh however was unique in its design: it consisted of a very small kernel with four modules (geometry, mesh, solver

*Correspondence to: Prof. J.-F. Remacle, Université catholique de Louvain, Bâtiment Vinci, Place du Levant 1, B-1348 Louvain-la-Neuve, Belgium. Email: jean-francois.remacle@uclouvain.be

Copyright © 2009 John Wiley & Sons, Ltd.
Received
Revised

and post-processing), not tied to any particular computational solver, and designed from the start to be driven both using a user-friendly graphical interface (GUI) and its own scripting language.

The first public release of Gmsh occurred in 1998 after two years of somewhat haphazard on-and-off development—coding taking place at night and during week-ends and holidays since Gmsh was merely a hobby and not part of our official work duties. This version was Unix-only, distributed over the internet in binary form, with the graphics layer based on OpenGL [42] and the user interface written in Motif [20]. After several updates and a short-lived Windows-only fork in 2000, the whole user interface was rewritten using FLTK [48] in early 2001, and the code, still in binary-only form, was released for Windows and a variety of Unix operating systems. In 2003 the full source code was released under the GNU General Public License [17], and it was modified to provide native support for all major operating systems: Windows, MacOS and all Unix/X11 variants. In the summer of 2006 Gmsh underwent a major rewrite, which led to the release of version 2 of the software in February 2007. About 50% of the code in version 2 is new: an abstract geometrical and post-processing layer has been introduced, the mesh data structures and algorithms have been rewritten from scratch, and the graphics layer has also been completely overhauled.

Today Gmsh enjoys a thriving community of several hundred users and developers worldwide. It is driven by the need of researchers and engineers in academia and industry alike for a small, open-source pre- and post-processing solution for grid-based numerical methods. The aim of this paper is not to be a user's guide or a reference manual—see [16] instead. Rather, it is to present the philosophy and the original features of Gmsh which make it stand out from its free and commercial alternatives.

The paper is structured as follows. In Section 2 we outline the overall philosophy and design goals of Gmsh, as well as the main technical choices that were made in order to achieve these goals. Sections 3, 4, 5 and 6 then respectively describe the geometry, mesh, solver and post-processing modules. The paper is concluded in Section 7 with perspectives for future developments.

2. The Design of Gmsh

Gmsh is built around four modules: geometry, mesh, solver and post-processing. Each module can be controlled either interactively using the GUI or using the scripting language.

The design of all four modules relies on a simple philosophy—be *fast*, *light* and *user-friendly*.

Fast: on a standard personal computer at any given point in time Gmsh should launch instantaneously, be able to generate a "larger than average" mesh (compared to the standards of the finite element community; say, one million tetrahedra in 2008) in less than a minute, and be able to visualize such a mesh together with associated post-processing datasets at interactive speeds.

Light: the memory footprint of the application should be minimal and the source code should be small enough so that a single developer can understand it. Installing or running the software should not depend on any non-widely available third-party software package.

User-friendly: the graphical user interface should be designed in such a way that a new user

Copyright © 2009 John Wiley & Sons, Ltd.

can create simple meshes in a matter of minutes. In addition, the code should be robust, portable, scriptable, extensible and thoroughly documented—all features contributing to a user-friendly experience.

In the following sections we describe the technical choices that we made to achieve these sometimes conflicting design objectives. Although major parts of the code have been rewritten over the years, the overall initial architecture and design from 1996 have always stayed the same.

2.1. Fast and Light

In order to be fast and light in the sense just described above, Gmsh is entirely written in standard C++ [50]—both the kernel and the user interface.

The kernel uses BLAS [7] and LAPACK for most of the basic linear algebra, and both the kernel and the interface make extensive use of the Standard Template Library classes for all data containers (e.g. vectors, lists, maps and trees). To keep them easy to understand the algorithms have not been overly optimized for speed or memory usage, yet Gmsh currently generates about a million tetrahedra per minute and per 150 Mb of RAM on a standard personal computer, which makes it powerful enough for many academic and engineering applications.

The graphical interface is built using FLTK [48] and OpenGL [42]. Using FLTK instead of a larger or more complex widget toolkit, like for example Java, TCL/TK, GTK or QT, allows to link Gmsh *statically* with the toolkit. This tremendously reduces the launch time, memory footprint and installation complexity (installing Gmsh requires copying a *single* executable file), as well as the build time—a statically linked, ready to use executable is produced in a few minutes on a standard personal computer. Analogously, directly using OpenGL instead of a more complex graphics library like Inventor [49] or VTK [40] makes Gmsh lightweight, without sacrificing rendering performance (Gmsh makes extensive use of OpenGL vertex arrays).

The design of the solver and scripting interfaces follows the same strategy: the solver interface is written using standard Unix and TCP/IP sockets instead of, e.g., Corba [54], and the scripting interface is built using Lex/Flex and Yacc/Bison [24] instead of using an external scripting language like, e.g., Python.

2.2. User-friendly

Although Gmsh can be built as a library (which can then be linked with other software tools)[†], it is usually distributed as a stand-alone software, ready to be used by end users. This stand-alone version can be run either interactively using the graphical user interface or in a non-interactive mode—either from the command line or via the scripting language.

Achieving "user-friendliness" has been an important driving factor behind key technical

[†]Gmsh provides an API for accessing geometry, mesh and post-processing data and algorithms. The geometrical and mesh API consists of a `GModel` class that relies on abstract base classes such as the `GEdge` and `GFace` classes described in section 3.2, each deriving from a base geometrical entity class `GEntity`. Each `GEntity` possesses its mesh, described by its vertices (`MVertex`) and elements (`MElement`). The post-processing module is based on list of `PViews` (see section 6), which are built upon abstract `PViewData` containers, that own the data and can point to `GModels` for their underlying mesh.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

choices. We detail some of these choices hereafter, focusing on the list of desirable features given at the beginning of the section.

2.2.1. Robustness and Portability To achieve robustness, i.e., working for the largest possible range of input data and being as tolerant as possible to erroneous user input, we use robust geometrical predicates [43] in critical portions of the algorithms, and strive to provide useful error messages when an unmanageable exception is triggered.

In order to easily produce a native version of the code on all major operating systems, Gmsh is written entirely in standard C++, and uses portable toolkits for its GUI (FLTK) and graphics rendering (OpenGL). Open-sourcing Gmsh under the GNU General Public License [17] also helped portability, as the code was made part of several official Linux distributions (most notably Debian [5]), and thus benefited from their extensive automated testing infrastructure. Either with the graphical user interface or in batch mode, the same version of Gmsh now runs on most computers, from laptops to workstations and large HPC clusters.

2.2.2. Scriptability Gmsh is scriptable so that all input data can be parametrized, and so that Gmsh can be easily inserted as a component inside a larger computational chain. As mentioned above, scripting is implemented in Gmsh using Lex and Yacc. The tight integration with the resulting language means that full access to internal capabilities is provided, including bidirectional access to more than 500 internal options fine-tuning the behaviour of the four modules. The scripting language allows for example to fully parametrize all geometrical entities, to interface external solvers without modifying the source code, or to automate all post-processing operations, e.g., to create complex animations or perform off-screen rendering [32].

2.2.3. Extensibility We tried to ease the modification and addition of features by users and developers. Such extensibility takes different forms for each of the four modules:

Geometry: the abstract, object-oriented geometry layer permits to write all the algorithms independently of the underlying CAD representation. At the source code level Gmsh is thus easily extensible by adding support for additional CAD engines. Currently two engines are interfaced: the native Gmsh CAD engine and OpenCascade [38]. Adding support for other engines like, e.g., Parasolid [47], can be done simply by deriving four abstract classes—see Section 3. At the scripting level users can then transparently mix and match geometrical parts represented internally by different CAD engines. For example, it is possible to extend an OpenCascade model of an airplane with a terrain model defined in the scripting language.

Mesh: using the abstract geometrical interface it is also possible to interface additional meshing kernels. Currently, in addition to its own meshing algorithms (see Section 4), Gmsh is interfaced with Netgen [39] and Tetgen [46].

Solver: a socket-based communication interface allows to interface Gmsh with various solvers without changing the source code; tailored graphical user interfaces can also easily be added when more fine-grained interactions are needed.

Post-processing: the post-processor can be extended with user-defined operations through dynamically loadable plug-ins. These plug-ins act on post-processing datasets (called

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

views) in one of two ways: either destructively changing the contents of a view, or creating one or more views based on the current view.

All source code-level extensions can be enabled or disabled at compile time thanks to an autoconf-based build mechanism [52], which selects which parts of the code to include/exclude.

2.2.4. Documentation and Open File Formats Documentation is provided both in the source code and in the form of a reference manual, including several hands-on tutorial examples and a comprehensive web site with several mailing lists and a wiki.

Another important feature contributing to user-friendliness is the availability of standard input and output file formats. Gmsh uses open or *de facto* standard formats whenever possible, from standard bitmap graphics formats (JPEG, GIF, PNG) and vector formats [15] (SVG, PostScript, PDF) to mesh formats (Ideas UNV, Nastran BDF). Combined with the open-source release of the code this greatly facilitates the integration of Gmsh with other computational tools.

3. CAD Interface

Gmsh never had the ambition of becoming a solid modeling platform that competes with the few well-established, state of the art CAD engines [47, 51]. The native Gmsh CAD engine thus has only a limited set of features, well suited for dealing with simple academic geometries. Yet, over the years, Gmsh has been used by an increasing number of people in industry, and a strong demand emerged from this user community for Gmsh to be able mesh industrial CAD models.

One option for addressing this demand is to use exchange files, such as IGES (Initial Graphics Exchange Specification), VRML (Virtual Reality Markup Language) or STEP (STandard for the Exchange of Product model data). However, the use of such exchange file formats has always been a cause of trouble in engineering design offices, mainly because internal data structures of CAD systems are usually much richer than those in the exchange formats. The necessary simplifications of geometries as well as the importance of modeler tolerances that are not taken into account in exchange files lead to the time-consuming need to "fix" most of these exchange files before any meshing can be performed.

In Gmsh, we thus chose to deal with CAD engines differently, by providing *native* access to the underlying CAD models—without translation files (a similar approach is used in CAPRI [18]). For that, the geometry module is based on a set of abstract data structures that enables us to represent the topology of *any* solid model. As mentioned in Section 2.2.3, Gmsh can then be extended to use new CAD engines simply by deriving these abstract data structures for each new engine.

3.1. Topological Entities

Any 3-D model can be defined using its Boundary Representation (BRep): a volume (called *region*) is bounded by a set of surfaces, and a surface is bounded by a series of curves; a curve is bounded by two end points. Therefore, four kinds of *model entities* are defined:

1. Model Vertices G_i^0 that are topological entities of dimension 0,

Copyright © 2009 John Wiley & Sons, Ltd.

2. Model Edges G_i^1 that are topological entities of dimension 1,
3. Model Faces G_i^2 that are topological entities of dimension 2,
4. Model Regions G_i^3 that are topological entities of dimension 3.

Model entities are topological entities, i.e., they only deal with adjacencies in the model, and we use a bi-directional data structure [1] for representing the graph of adjacencies. In this representation, a model entity G_i^d of dimension d holds one lists of upward adjacencies $G_j^{d+1}(G_i^d)$, i.e., all its adjacent entities of dimension $d+1$, and one list of downward adjacencies of dimension $d-1$, $G_j^{d-1}(G_i^d)$. Schematically, we have

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

This representation is said to be complete because any model entity is able to build its list of adjacencies of any dimension using local operations, i.e., without having to do a complete traversal of the adjacency graph of the model.

3.2. Geometrical Description

Each model entity G_i^d has a shape, a geometry. More precisely, it is a manifold of dimension d that is embedded in 3-D space. (Note that the overall geometric model may itself be non-manifold: Gmsh supports non-manifold features such as embedded curves and surfaces and connected volumes. Some non-manifold examples will be shown in the mesh generation Section 4.)

The geometry of a model entity depends on the solid modeler for its underlying representation. Solid modelers usually provide a parametrization of the shapes, i.e., a mapping $\vec{p} \in R^d \mapsto \vec{x} \in R^3$:

1. The geometry of a model vertex G_i^0 is simply its 3-D location $\vec{x}_i = (x_i, y_i, z_i)$.
2. The geometry of a model edge G_i^1 is its underlying curve $\mathcal{C}_i$ with its parametrization $\vec{p}(t) \in \mathcal{C}_i$, $t \in [t_1, t_2]$.
3. The geometry of a model face G_i^2 is its underlying surface $\mathcal{S}_i$ with its parametrization $\vec{p}(u,v) \in \mathcal{S}_i$. Note that, for any curve $\mathcal{C}_j$ that is on a surface $\mathcal{S}_i$, mesh generation procedures require the ability to reparametrize any point $\vec{p}(t) \in \mathcal{C}_j$ on the surface $\mathcal{S}_i$, i.e., to compute the mapping $u = u(t)$ and $v = v(t)$. Gmsh either uses a brute force algorithm to compute the direct mapping $x = x(t)$, $y = y(t)$ and $z = z(t)$ and its inverse $u = u(x,y,z)$ and $v = v(x,y,z)$ (see Figure 1), or, when the underlying CAD system provides it, the direct reparametrization of a point on a model face (i.e., a function that directly computes $u = u(t)$ and $v = v(t)$).
4. The geometry associated to a model region is R^3.

Solid modelers usually provide an API for the creation, manipulation, interrogation and storage of 3-D models. To perform mesh generation only a small subset of this API has to be interfaced—only some of the interrogation functions are necessary. In order to get the full functionality of Gmsh, only five CAD-system dependent interrogation functions have to be implemented for the model edge (see Figure 2). For example, it is mandatory to be able to evaluate the mapping $\vec{p}(t) \in \mathcal{C}$ on the curve as well as the tangent vector $\vec{t}(t) = \partial_t \vec{p}(t)$. For the model face, only four functions have to be overloaded in order to enable 2-D mesh generation (see Figure 3). Note that the default 2-D algorithm does not require the computation

Copyright © 2009 John Wiley & Sons, Ltd.

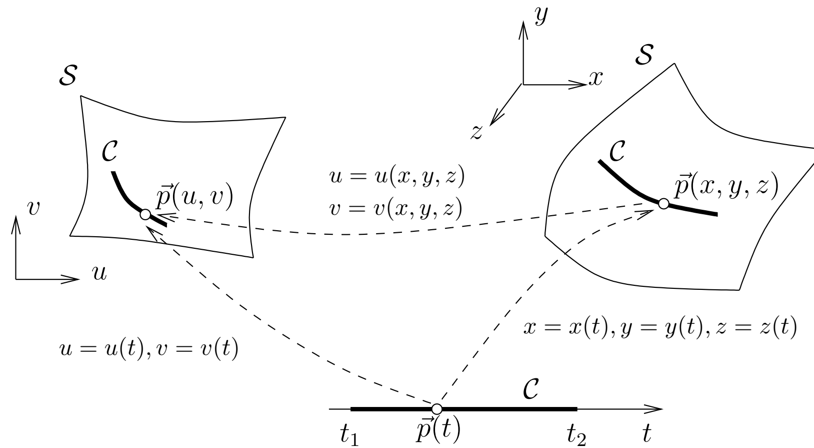


Figure 1. Point $\vec{p}$ located on the curve $\mathcal{C}$ that is itself embedded in surface $\mathcal{S}$.

```
class GEdge : public GEntity{
  // bi-directional data structure
  GVertex *v1, *v2;
  std::list<GFace*> faces;
public:
  // pure virtual functions that have to be overloaded for every
  // solid modeler
  virtual std::pair<double> parRange() = 0;
  virtual Point3 point(double t) = 0;
  virtual Vector3 firstDer(double t) = 0;
  virtual Point2 reparam(GFace *f, double t, int dir) = 0;
  virtual bool isSeam(GFace *f) = 0;
  // other functions of the class are non pure virtual
  // ...
};
```

Figure 2. A part of the model edge class description. `GEdge::parRange` returns the range for the parameter in the curve. `GEdge::point` returns the 3-D point $\vec{p}(t)$ that is located on the curve $\mathcal{C}$ for a given parameter t. `GEdge::firstDer` evaluates the tangent vector $\partial_t \vec{p}(t)$ for a given parameter t. `GEdge::reparam` computes the local parameters of the point $\vec{p}(t)$ on a model face `f` that has $\mathcal{C}$ in its closure, `GEdge::isSeam` tells if the curve is or is not a seam of the face `f`. Generally, seam edges are used to maintain consistency of data structure for periodic surfaces.

of derivatives of the surface parametrization, so that the function `GFace::firstDer` is not strictly required (a description of this 2-D algorithm is presented in Section 4.3). The other 2-D algorithms available in Gmsh, as well as most of the available 2-D meshers (e.g. `bamg` [19] or `blsurf` [23]), make use of derivatives of the parametrization.

Copyright © 2009 John Wiley & Sons, Ltd.

```
class GFace : public GEntity{
  // bi-directional data structure
  GRegion *r1, *r2;
  std::list<GEdge*> edges;
public:
  // pure virtual functions that have to be overloaded for every
  // solid modeler
  virtual std::pair<double> parRange(int dir) const = 0;
  virtual Point3 point(double u, double v) const = 0;
  virtual std::pair<Vector3> firstDer(double u, double v) const = 0;
  // other functions of the class are non pure virtual
  virtual double curvature(double u, double v) const;
  // ...
};
```

Figure 3. A part of the model face class description. `GFace::parRange` returns the range for the parameter in the surface in direction `dir`. `GFace::point` returns the 3-D point $\vec{p}(u, v)$ that is located on the surface $\mathcal{S}$ for the given parameter couple (u, v). `GFace::firstDer` evaluates two tangent vectors $\partial_u \vec{p}(u, v)$ and $\partial_v \vec{p}(u, v)$. The `GFace::curvature` function computes the divergence of the unit normal vector at (u, v). This last function is used for the definition of mesh size fields. It is not a pure virtual function: a default implementation is available using finite differences.

3.3. Solid Model

A Gmsh model is simply built as a list of model entities, each one of which possibly of different underlying geometries. In fact, as mentioned before, several CAD models can co-exist in the same Gmsh model. Mixing parts coming from different CAD engines can be very interesting in practice: for example, complex, non parametrizable parts designed with one CAD modeler (say, a full airplane model) can be extended with a parametrizable part written using the scripting language of the Gmsh native modeler (say, a radar antenna whose design one wishes to optimize). The overall model can then be discretized and optimized without any CAD translation.

4. Mesh Generation in Gmsh

For the description of the mesh generation process, let us consider the CAD model of a propeller presented in Figure 4. The model has been created with the OpenCascade solid modeler and has been loaded in Gmsh in its native format (brep). The model contains 101 model vertices, 170 model edges, 76 model faces and one model region.

4.1. Mesh Size Field and Quality Measures

Let us define the mesh size field $\delta(x, y, z)$ as a function that defines, at every point of the domain, a target size for the elements at that point. The present ways of defining such a mesh size field in Gmsh are:

Copyright © 2009 John Wiley & Sons, Ltd.

Figure 4. CAD model of a propeller (left) and its volume mesh (right)

1. mesh sizes prescribed at model vertices and interpolated linearly on model edges;
2. prescribed mesh gradings on model edges (geometrical progressions, ...);
3. mesh sizes defined on another mesh (a background mesh) of the domain;
4. mesh sizes that adapt to the principal curvature of model entities.

These size fields can then be acted on by functionals that may depend, for example, on the distance to model entities or on user-prescribed analytical functions; and when several size fields are provided, Gmsh uses the minimum of all fields. Thanks to that mechanism, Gmsh allows for a mesh size field defined on a given model entity to extend in higher dimensional entities. For example, using a distance function, a refinement based on the curvature of a model edge can extend on any surface adjacent to it.

Let us now consider an edge e of the mesh. We define the adimensional length of the edge with respect to the size field δ as

$$l_e = \int_e \frac{1}{\delta(x,y,z)} dl. \quad (1)$$

The aim of the mesh generation process is twofold:

1. Generate a mesh for which each mesh edge e is of size close to $l_e = 1$,
2. Generate a mesh for which each element K is *well shaped*.

In other words, the aim of the mesh generation procedure is to be able to build a good quality mesh that complies with the mesh size field.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

To quickly evaluate the adequation between the mesh and the prescribed mesh size field, we defined an efficiency index τ [13] as

$$\tau = \exp\left(\frac{1}{ne}\sum_{e=1}^{ne}\tau_e\right) \tag{2}$$

with $\tau_e = l_e - 1$ if $l_e < 1$ and $\tau_e = \dfrac{1}{l_e} - 1$ if $l_e \geq 1$. The efficiency index ranges in $\tau \in [0,1]$ and should be as close as possible to $\tau = 1$.

For measuring the quality of elements, various element shape measures are available in the literature [33, 27]. Here, we choose a measure based on the element radii ratio, i.e. the ratio between the inscribed and the circumcircles.

If K is a triangle, we have the following formula

$$\gamma_K = 4\frac{\sin\hat{a}\sin\hat{b}\sin\hat{c}}{\sin\hat{a}+\sin\hat{b}+\sin\hat{c}},$$

$\hat{a}$, $\hat{b}$ and $\hat{c}$ being the three inner angles of the triangle. With this definition, the equilateral triangle has a $\gamma_K = 1$ and degenerated (zero surface) triangles have a $\gamma_K = 0$.

For a tetrahedron, we have the following formula:

$$\gamma_K = \frac{6\sqrt{6}\,V_k}{\left(\sum_{i=1}^{4} a(f_i)\right)\max_{i=1,\ldots,6} l(e_i)},$$

with V_K the volume of K, $a(f_i)$ the area of the ith face of K and $l(e_i)$ the dimensional length of the ith edge of K. This quality measurement lies in the interval $[0,1]$, an element with $\gamma_K = 0$ being a sliver (zero volume).

4.2. 1-D Mesh Generation

Let us consider a point $\vec{p}(t)$ on a curve $\mathcal{C}$, $t \in [t_1, t_2]$. The number of subdivisions N of the curve is its adimensional length:

$$\int_{t_1}^{t_2} \frac{1}{\delta(x,y,z)}\|\partial_t\vec{p}(t)\|dt = N. \tag{3}$$

The $N+1$ mesh points on the curve are located at coordinates $\{T_0,\ldots,T_N\}$, where T_i is computed with the following rule:

$$\int_{t_1}^{T_i} \frac{1}{\delta(x,y,z)}\|\partial_t\vec{p}(t)\|dt = i. \tag{4}$$

With this choice, each subdivision of the curve is exactly of adimensional size 1, and the 1-D mesh exactly satisfies the size field δ. In Gmsh, (4) is evaluated with a recursive numerical integration rule.

Copyright © 2009 John Wiley & Sons, Ltd.

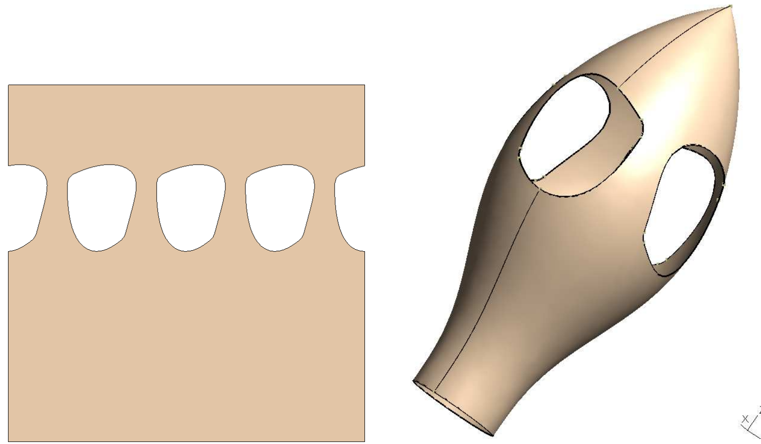


Figure 5. Geometry of a model face in parametric space (left) and in real space (right). Two seam edges are present in the face. The top model edge is degenerated in one point.

4.3. 2-D Mesh Generation

Curved surface shapes designed by CAD systems are usually defined by parametric surfaces, for example, NURBS [34]. Let us consider a model face G_i^2 with its underlying geometry, in this case a surface $\mathcal{S} \in R^3$ with its parametrization $\vec{p}(u, v) \in \mathcal{S}$, where the domain of definition of the parameters (u, v) is defined by a series of boundary curves. An example of such a surface is given in Figure 5, which shows one of the 76 model faces of the propeller in the parametric space (left) and in real space (right). Three features of surface $\mathcal{S}$, common in CAD descriptions, make its meshing non-trivial:

1. $\mathcal{S}$ is periodic. The topology of the model face is modified in order to define its closure properly. A seam is present two times in the closure of the model face. These two occurrences are separated by one period in the parametric space.
2. $\mathcal{S}$ is trimmed: it contains four holes and one of them is crossed by the seam.
3. One of the model edges of $\mathcal{S}$ is degenerated. This is done for accounting of a singular point in the parametrization of the surface. This kind of degeneracy is present in many shapes: spheres, cones and other surfaces of revolution.

Techniques for generating finite element meshes on curved surfaces are of two kind:

1. techniques for which the surface mesh is generated directly in the real 3-D space;
2. techniques for which the surface mesh is generated in the parametric space.

Most algorithms working directly in the real 3-D space work by modifying an existing geometrical mesh [11]. For example, such algorithms have been used for building meshes from STL (stereolithography) data files [2] or from medical imaging [57]. The principal advantage of these algorithms is that no interface to the solid modeler is required. The main drawback of such algorithms is their relative lack of robustness, as checking the validity of a mesh

Copyright © 2009 John Wiley & Sons, Ltd.

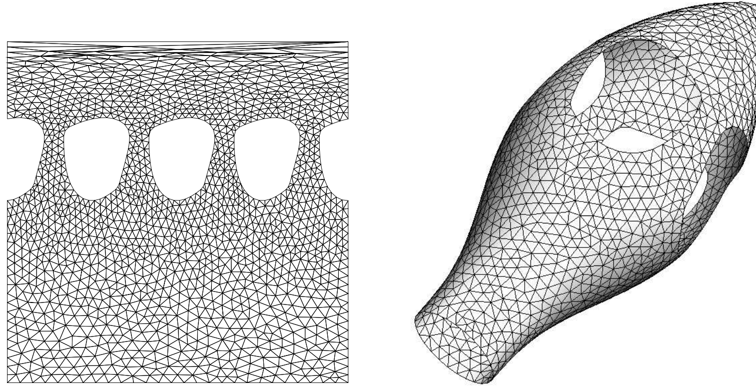


Figure 6. Mesh of a model face drawn in the parametric space (left) and in the real space (right).

modification in real space requires heuristics, as for example a maximum tolerance on angles between normals.

The second alternative can be applied only if a parametrization of the surfaces is available. If it is the case, doing the mesh in the parametric space is usually advantageous because all the meshing procedures can be applied in the parametric plane. This allows mesh operators to be highly robust: building a valid triangulation in the plane is always possible, and one can guarantee that elements do not overlap.

Yet, all surfaces do not always have a parametrization that conserves angles and lengths. Consequently, only 2-D algorithms that allow to build anisotropic meshes in the plane can be considered as good candidates for doing surface meshing. Figure 6 presents the surface mesh of the model face of Figure 5, both in the parametric space and in the real space. The mesh in the real space is isotropic and uniform while the one in the parametric space is highly anisotropic and non uniform. To solve this problem, George and Borouchaki [12] have proposed the use of a metric derived from the first fundamental form of the surface. The metric field is a second order tensor field that has the form, at any point of the parametric space, of a 2×2 matrix. The metric is used to define angles and distances in parametric space. With their Delaunay approach, the "empty circle" property, effectively becomes an "empty ellipse" property. An equivalent "metric-based" advancing front surface mesh generation algorithms is presented by Cuilliere in [4]. A more exotic metric-based approach based on packing ellipses has been devised by Yamada et al. [45] and has been used more recently by Lo and Wang in [28].

In addition to a Delaunay implementation similar to [12] and a frontal-Delaunay meshing technique inspired by [35], Gmsh provides an original surface meshing strategy based on the concept of local mesh modifications [25, 26, 36]. The main advantage of the new approach, compared to the other ones based on the Delaunay criterion, is that it does not require the computation of derivatives of the parametrization. For that reason, the new approach remains robust even when the parametrization is singular.

The algorithm works as follows. First, an initial mesh containing all the mesh points of the curves bounding the face is built in the parametric space using a divide and conquer strategy

Copyright © 2009 John Wiley & Sons, Ltd.

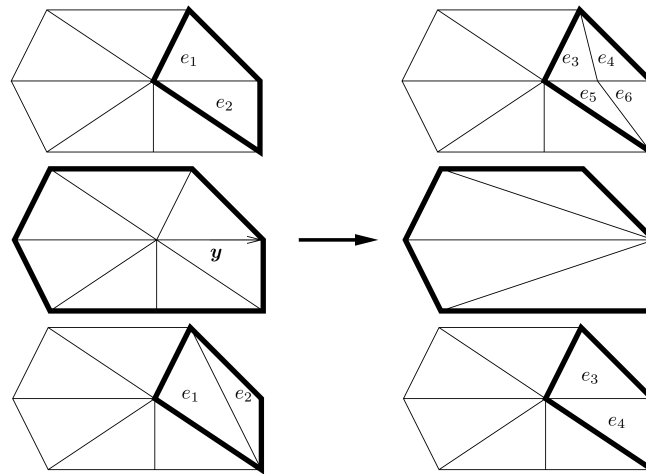


Figure 7. Illustration of local mesh modifications.

[9]. Then, all the edges of the 1-D discretization are recovered using swaps [56]. Finally, local mesh modifications are applied:

1. Each edge that is too long is split;
2. Each edge that is too short is removed using an edge collapse operator;
3. Edges for which a better configuration is obtained by swapping are swapped;
4. Vertices are re-located optimally.

More precisely, here is how these four local mesh modifications procedures are applied in Gmsh:

Edge Splitting: An edge is considered too long when its adimensional length is greater than $l_e > 1.4$. When split, the two new edges will have a minimal size of 0.7. In order to converge to a stable configuration, an edge of size $l_e = 0.7$ should not be considered as a short edge.

Edge Collapsing: An edge is considered to be short when its adimensional length is smaller than $l_e < 0.7$. An edge cannot be collapsed if one of the remaining triangles after the collapse is inverted in the parametric space.

Edge Swapping: An edge is swapped if $\min(\gamma_{e_1}, \gamma_{e_2}) < \min(\gamma_{e_3}, \gamma_{e_4})$ (see Figure 7), unless

1. it is classified on a model edge;
2. the two adjacent triangles e_1 and e_2 form a concave quadrilateral in the parametric space;
3. the angle between the triangles normals is greater than a threshold, typically 30 degrees.

Copyright © 2009 John Wiley & Sons, Ltd.

Vertex Re-positioning: Each vertex is moved optimally inside the cavity made of all its surrounding triangles. The optimal position is chosen in order to maximize the worst element quality [10].

For each of these local mesh modification procedures, the *opportunity* of doing a mesh modification is evaluated in the real space, i.e., in (x, y, z), while the *validity* of a mesh modification is evaluated in the parametric space (u, v). Therefore, Gmsh mesh generators always retain both real and parametric coordinates of any mesh vertex. To ensure robustness, all the elementary geometrical predicates make use of robust algorithmics [43].

In practice, this algorithm converges in about 6-8 iterations and produces anisotropic meshes in the parametric space without computing derivatives of the mapping.

Let us illustrate the algorithm on an example. We have meshed the model face of Figure 6 using an analytical size field

$$\delta(x, y, z) = \delta_0[1 + \cos(\pi(x + y - z)/L)] + \epsilon$$

where L is a characteristic size of the domain and $\epsilon \ll \delta_0 < L$. Figure 8 shows the mesh in the parametric space at different stages of the algorithm. Note that the derivatives of the parametrization of the underlying surface are not defined at the singular point so that any algorithm that requires to compute such derivatives would be in trouble in this case.

4.4. 3-D Mesh Generation

Once a surface triangulation is available, an automatic mesh generation procedure does not usually require an interface to a CAD system. Indeed, Gmsh interfaces several open source 3-D tetrahedral mesh generation kernels [39, 46] in addition to its own Delaunay refinement algorithm. These algorithms are standard [14, 39] and will not be explained here. We focus on two other issues instead:

1. the way Gmsh interfaces multiple mesh generation algorithms;
2. the way Gmsh optimizes the quality of 3-D meshes.

(A third issue concerns the way Gmsh handles mixed structured/unstructured grids—this is addressed in Section 4.6.)

4.4.1. Mesh Algorithm Interfaces Gmsh is able to deal with most of the standard finite element shapes: lines, triangles, quadrangles, tetrahedra, hexahedra, prisms and pyramids. The internal mesh data structures are designed to minimize the memory footprint without compromising flexibility: in addition to a integer tag and a partition/visualisation index, any element only holds its ordered list of vertices. With that simple design, Gmsh can load about 12 million tetrahedra per Gigabyte of memory (28 bytes per tetrahedron, 44 bytes per mesh vertex), including graphics representation, i.e., OpenGL vertex arrays [42].

When "in house" meshing routines are used, Gmsh derives (in the object oriented sense) enriched data structures specifically tailored for each meshing algorithm. Those derived structures contain just the right extra information necessary: the parametric coordinates of a vertex for parametric 2-D meshing algorithms, the neighbours of a tetrahedron for the 3-D Delaunay algorithm, etc. With this approach the footprint of a tetrahedron is for example extended to 84 bytes, and the 3-D Delaunay algorithm implemented in Gmsh, using a classical

Copyright © 2009 John Wiley & Sons, Ltd.

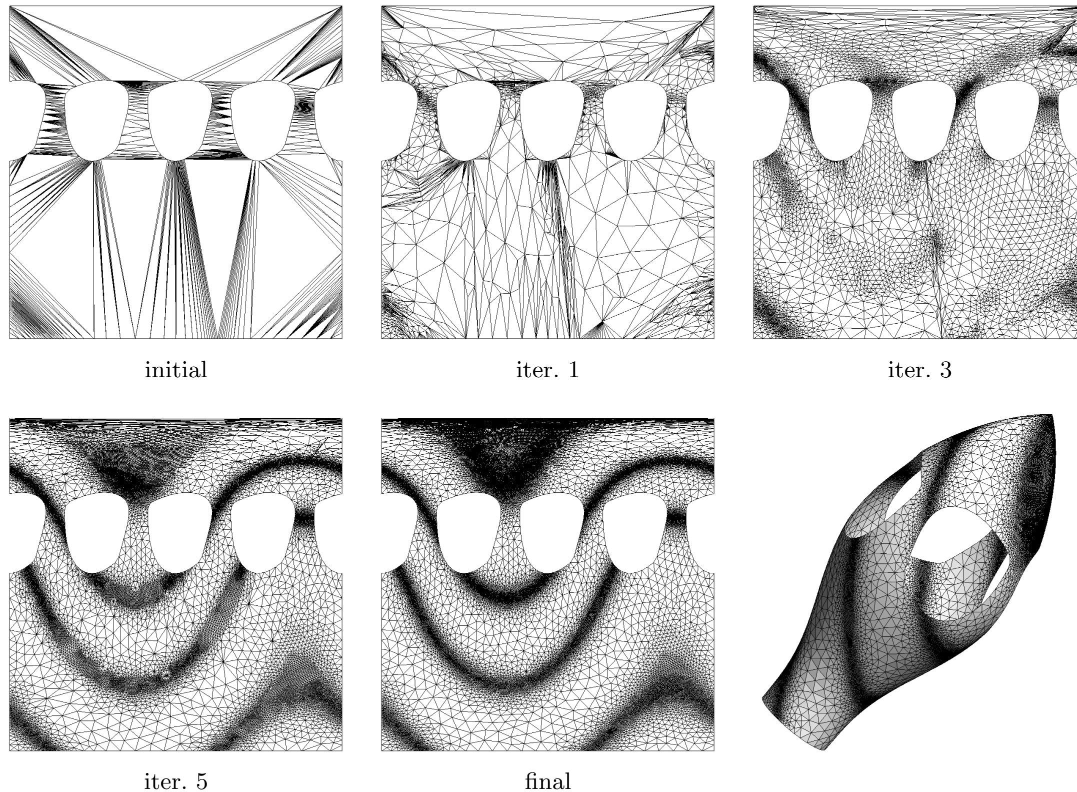


Figure 8. Illustration of the surface meshing algorithm based on local mesh modifications. The images correspond to the initial mesh containing boundary vertices, the mesh after 1, 3, 5 and 8 iterations. At iteration 8, the algorithm has converged. The size field efficiency $\tau = 0.89$ can be considered as excellent: 90 % of the elements have a radii ratio γ_K greater that 0.9.

Bowyer-Watson algorithm [55], is able to build about 7 million tetrahedron per Gigabyte of memory (including overhead like the data structures of the CAD engine).

When a third party mesh generator is invoked, Gmsh needs of course to allocate the appropriate structures required by that specific software. But thankfully, while transferring the data from the third party algorithm into Gmsh, only the minimal internal data structures need to be allocated (i.e., 28 byte per tetrahedron in the 3-D case mentioned above). This greatly reduces the overhead incurred by interfacing external algorithms.

4.4.2. Tetrahedral Mesh Improvement Tetrahedral mesh improvement is usually required to produce 3-D meshes suitable for grid-based numerical methods [10]. Unfortunately, mesh optimization procedures have a lot to do with “black magic”: even if the ingredients required to construct a mesh optimization procedure are well known (essentially swapping and smoothing), there is no known “best recipe”, i.e., no known optimal way of combining those smoothing and swapping operators.

Copyright © 2009 John Wiley & Sons, Ltd.

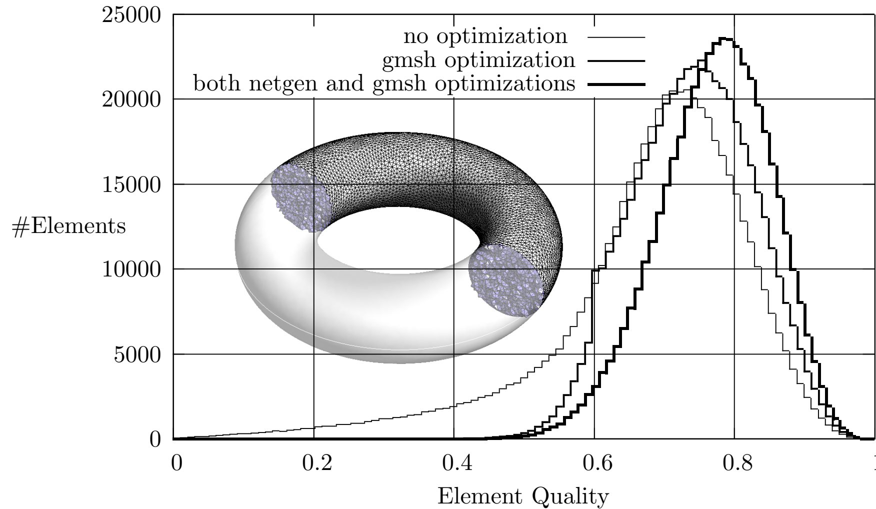


Figure 9. Distribution of γ_K in a mesh of about $600,000$ tetrahedra.

Gmsh implements its own mesh optimization procedure to enhance tetrahedral mesh quality by means of edge- and face-swappings and vertex relocations, and also interfaces third party mesh optimizers—in particular the open-source optimizer from Netgen [39]. Interestingly, applying optimization routines one after the other enables to produce better meshes than applying mesh optimizers separately. Figure 9 shows the distribution of elemental qualities on the mesh of a toroidal domain containing about 600,000 tetrahedra (the mesh was generated in about 30 seconds with the "in house" 3-D Delaunay algorithm). The unoptimized mesh contains quite a few ill shaped elements: more than 5000 elements have an aspect ratio γ_K below 0.2 and the worst shaped element has an aspect ratio of 10^{-3}. After one pass of the Gmsh mesh optimizer, which takes about 12 seconds, all slivers have disappeared and the worst element has an aspect ratio of 0.32. The distribution of element quality is enhanced, with a clear right shift of the distribution. Applying the Netgen optimizer after the Gmsh optimizer, additional improvement can be observed: the worst elemental quality is now 0.41 and another shift to the right has occurred. However, the application of the Netgen optimizer also dramatically reduced the number of elements in the mesh, and this second optimization pass took more than 200 seconds—about 15 times more than for the Gmsh optimizer. Transferring the mesh in Netgen format also doubled the memory usage.

4.5. Examples

One of the objectives of this section is to demonstrate that Gmsh is able build meshes that can be used by the finite element community. The various examples shown below can all be downloaded from the Gmsh web site. They come from different sources: native Gmsh CAD models, CAD models found on the web, or CAD models that were proposed by industrial and academic partners. The formats considered are IGES, STEP, BREP and Gmsh. Various size

Copyright © 2009 John Wiley & Sons, Ltd.

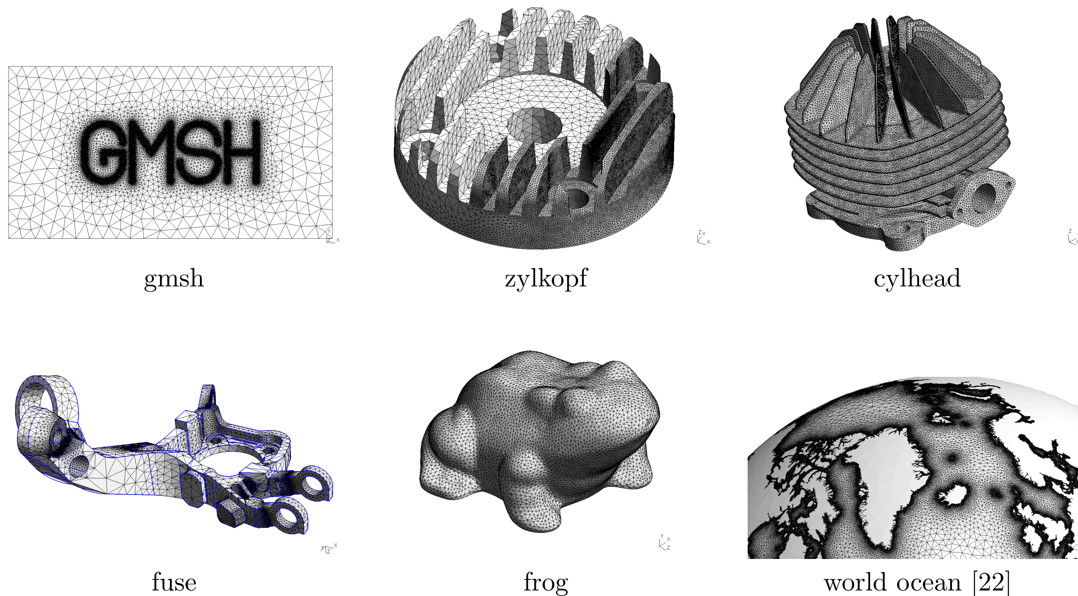


Figure 10. Some images of surface meshes.

fields have been used, uniform or not. Both 2-D and 3-D statistics are provided.

Table I presents details for some of the models (see Figure 10) used in the Gmsh 2-D test suite. Mesh size field are defined in a variety of ways: analytic, uniform, size fields driven by distance functions (attractors), boundary layers, size fields related to the curvature of surfaces, or size fields interpolated using sizes that are defined on model vertices. Table II gives statistics for the 2-D meshes generated with the surface meshing algorithm presented in Section 4.3. In the case of planar surfaces and uniform meshes this algorithm is about three times slower than the 2-D anisotropic Delaunay mesh generator implemented in Gmsh. However, when multiple size fields are involved and/or when the surfaces are very complex, this new approach becomes competitive in terms of CPU time—and is much more robust than the anisotropic Delaunay mesher. With the caveat that the performance and robustness of mesh generation algorithms are highly dependent on their implementation, we believe this shows evidence that the new algorithm proposed in Section 4.3 is a viable alternative to classical Frontal or Delaunay approaches.

Table III presents some statistics for 3-D meshes. The first example can serve as reference: it is a unit cube that is meshed uniformely with about one million tetrahedra. Some of the examples have complex mesh size fields (linkrods or frogadapt). One has small features in the geometry (block). Some have multiple volumes (media or senzor). Complex mesh size fields such as the ones of linkrods or frogadapt make the mesh generation process slower of about 20%. This overhead is essentially due to the evaluation of the mesh size field. Mesh with strong size variations or with small geometric features require more optimization. The performance figures mentioned in Section 4.4.1 hold even for models with a large number of model regions: the model called "media", created in the native Gmsh CAD format, involves over 1000 model

Copyright © 2009 John Wiley & Sons, Ltd.

	type	n_R	n_F	n_E	n_V	δ
cube	GMSH	1	6	12	8	uniform
gmsh	GMSH	0	1	35	23	attractor
frog	IGES	1	475	950	477	uniform
frogadapt	IGES	1	475	950	477	analytic
linkrods	STEP	1	37	108	74	analytic
zylkopf	STEP	1	137	404	270	at vertices
cylhead	BREP	1	1054	2485	1445	uniform
fuse	STEP	1	249	723	476	curvature
block	STEP	1	533	1586	1048	uniform
senzor	GMSH	8	90	200	146	at vertices
world ocean	GMSH	0	1	4245	145291	boundary layer
media	GMSH	1274	8398	5779	3894	uniform

Table I. Statistics on the models that are considered: n_R,n_F,n_E and n_V are respectively the number of model regions, of model faces, of model edges and of model vertices in the model. δ is the size field.

	np	ne	$\gamma_K > 0.9$	$\min_K \gamma_K$	$\text{avg}_K \gamma_K$	$l_{\sqrt{2}}$	τ	CPU
gmsh	28041	55922	83.5%	0.287	0.946	99.0%	0.891	10 s
linkrods	55959	119922	84.2%	0.385	0.946	98.9%	0.893	61 s
zylkopf	32806	65668	86.0%	0.105	0.947	98.5%	0.860	8 s
cylhead	84014	188150	77.5%	0.050	0.915	95.5%	0.892	45 s
fuse	23485	47038	76.0%	0.010	0.919	97.3%	0.886	11 s
block	19694	55530	76.0%	0.021	0.923	96.8%	0.895	20 s
senzor	19876	40002	84.6%	0.546	0.947	98.4%	0.896	11 s
ocean	1152011	2255212	89.0%	0.211	0.950	99.1%	0.901	729 s

Table II. Surface mesh generation statistics. Here, np are ne are the number of points and triangles in the surface mesh, $\gamma_K > 0.9$ states for the percentage of triangles that have a quality measure γ_K greater that 0.9, $\min_K \gamma_K$ is the worst element quality in the surface mesh and $\text{avg}_K\gamma_K$ is the average elemental quality. The quantity $l_{\sqrt{2}}$ states for the percentage of edges that have an adimensional length in the range $1/\sqrt{2} < l_e < \sqrt{2}$. The factor τ is the efficiency index defined in Equation (2). The last column gives the CPU time (in seconds) for performing the surface mesh generation.

regions and was meshed using the 3-D Delaunay algorithm in less than one minute. All timings were measured on a standard MacBook Pro with a CPU clocked at 2.0 GHz.

4.6. Mixed Meshes

In addition to unstructured meshes, Gmsh enables the generation of simple *structured* meshes in 1-D, 2-D and 3-D, and allows to couple these with unstructured meshes. Structured technologies include transfinite and elliptic meshes as well as a variety of sweeping techniques.

For example, to build a boundary layer mesh from a set of source surfaces (see Figure 12), Gmsh

1. creates the topology of the boundary layer by sweeping zero-height volumes from all the source surfaces. (In addition to the boundary layer volumes, this creates a set of new

Copyright © 2009 John Wiley & Sons, Ltd.

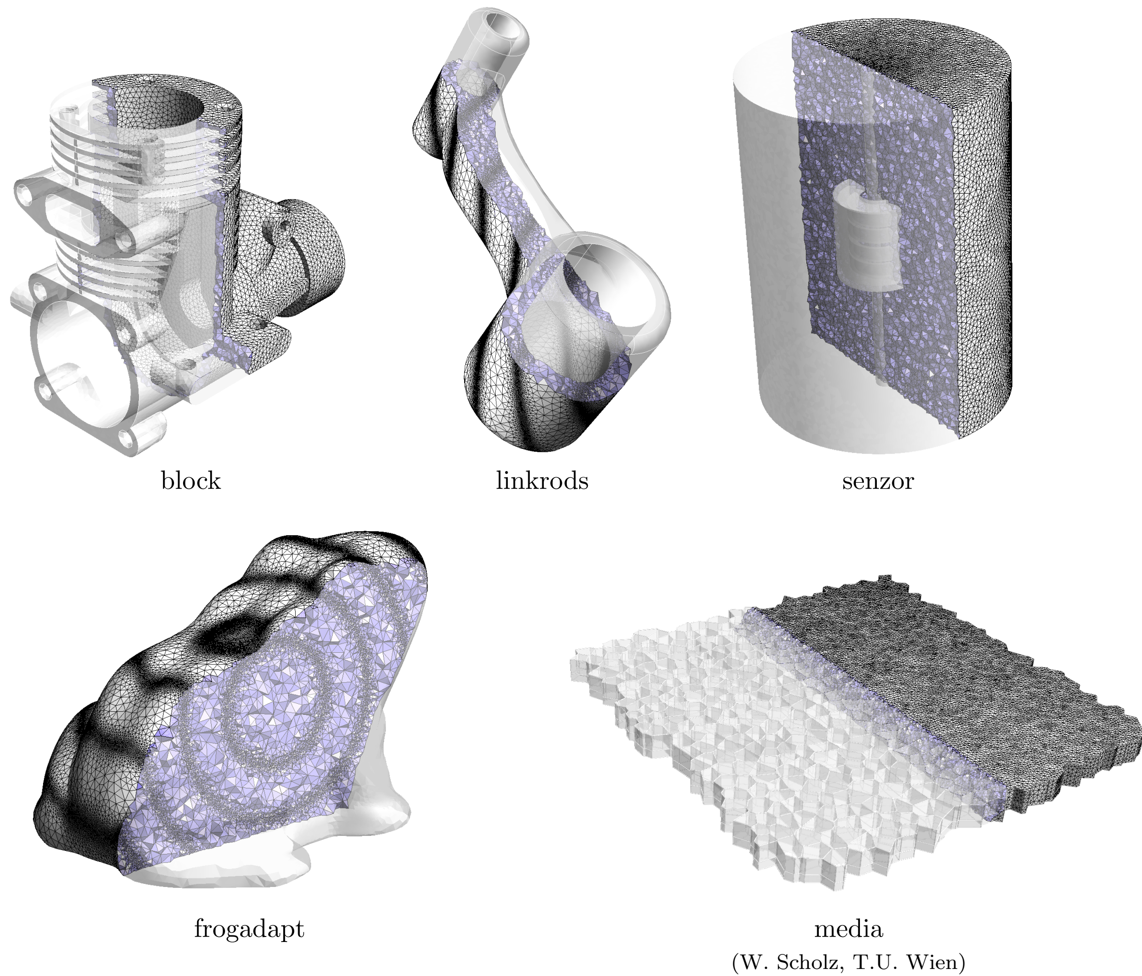


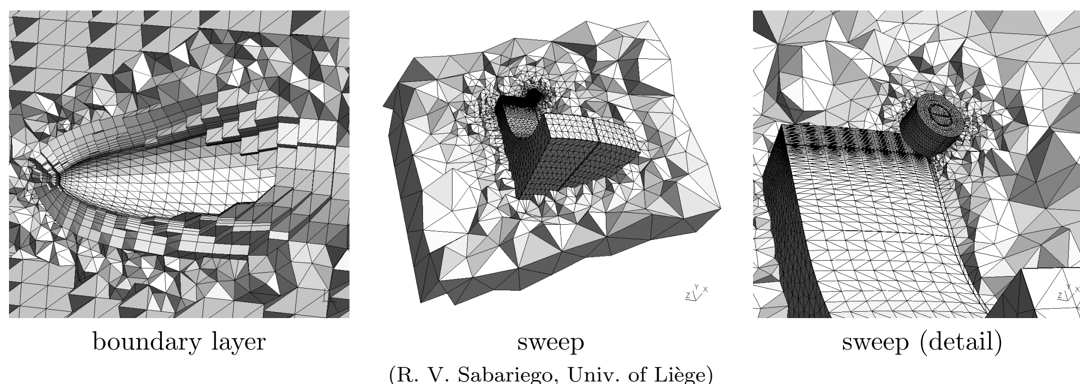
Figure 11. Some images of volume meshes.

	np	ne	$\min_K \gamma_K$	$\text{avg}_K \gamma_K$	CPU (mesh)	CPU (opti)
cube	195,671	1,098,530	0.235	0.717	49 sec.	36 s
linkrods	341,297	1,836,634	0.347	0.756	111 s	117 s
block	48,897	221,090	0.012	0.660	12 s	14 s
senzor	143,799	805,392	0.222	0.765	35 s	27 s
frogadapt	403,947	2,381,969	0.172	0.691	116 s	180 s
media	164,517	890,756	0.071	0.696	55 s	31 s

Table III. Volume mesh generation statistics. Here, np are ne are the number of points and tetrahedron in the volume mesh, $\min_K \gamma_K$ is the worst element quality in the volume mesh and $\text{avg}_K \gamma_K$ is the average elemental quality. The last two columns give mesh generation and mesh optimization timings.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls



boundary layer — sweep (R. V. Sabariego, Univ. of Liège) — sweep (detail)

Figure 12. Some images of hybrid volume meshes.

boundary layer points, curves and surfaces. These new points, curves and surfaces will only acquire a concrete representation during the meshing process.)

2. meshes the source surfaces and computes (unique) normals at the mesh vertices;
3. sweeps the boundary layer points along the normals, and remeshes all the non-boundary-layer curves connected to these points, and then meshes the boundary layer curves by sweeping along the normals;
4. meshes the non-source surfaces, then the boundary layer surfaces (again by sweeping along the normals), and finally the volumes.

Once all the structured parts are meshed, the remaining parts are meshed using the unstructured algorithms, resulting in conforming mixed meshes. In order for the final mesh to be conforming, the structured algorithm splits hexahedra, prisms or pyramids into simplices when needed.

5. Solver Interface

When Gmsh is built as a library, finite element solvers can be directly linked to it and use the Gmsh API to load geometries, generate meshes or post-process data. This approach is very powerful, but requires a significant programming investment on the part of the solver developer.

Following our "user-friendly" design goal, Gmsh provides a less pervasive, but still powerful, alternative: external solvers can be interfaced with Gmsh through Unix and TCP/IP sockets. This permits to launch external computations and to directly collect and process the results from within the post-processing module (see Section 6), with only minimal changes required on the solver side. The scripted interface allows to add support for a new solver simply by editing options in a text file, and the standard stand-alone versions of Gmsh can thus be directly used, unchanged. The default solver interfaced with Gmsh is GetDP [8] (see Figure 13), and templates for solvers written in several languages (C, C++, Perl and Python) are also available.

Copyright © 2009 John Wiley & Sons, Ltd.

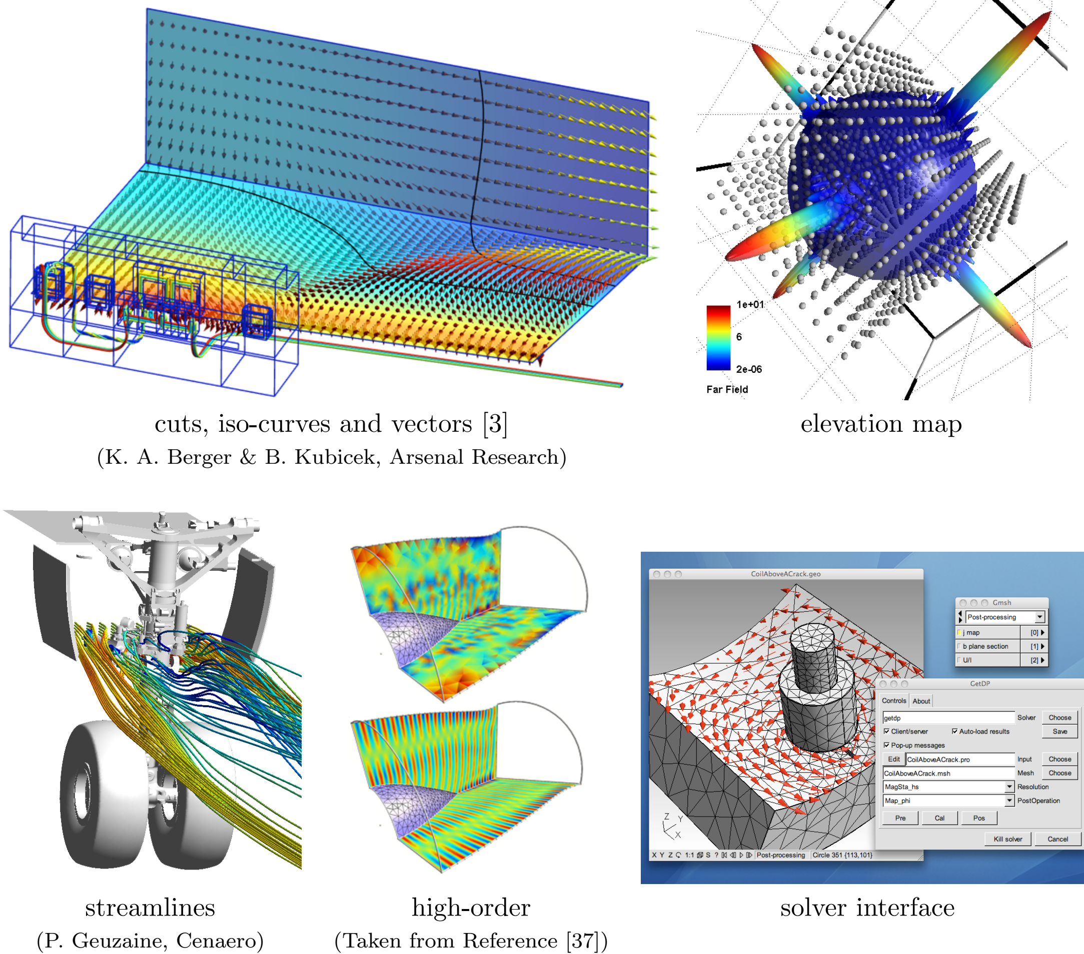


Figure 13. Some images from the solver and post-processing modules.

6. Post-Processing in Gmsh

The post-processing module can load, transform and display multiple post-processing datasets (called "views") at once, along with the geometry and the mesh. Each view can contain a mix of scalar, vector and tensor data as well as text annotations. Views can be manipulated either globally or individually (each view has its own button in the GUI and can be referred to by its index in a script), and each one possesses its own set of display options. Internally, the view is an abstract class that can access a variety of underlying representations, from the node-based data sets used in standard finite element codes to high-order, discontinuous data sets used, e.g., in discontinuous Galerkin or finite volume solvers [37].

Scalar fields are represented by iso-surfaces or color maps, while vector fields are represented

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

by three-dimensional arrows or displacement maps (see Figure 13). The graphics display code is written using OpenGL, and all representations are stored internally as vertex arrays to improve rendering performance.

Display options are non-destructive (they do not modify the dataset) and can be changed on the fly. These options include for example choosing the plot type and the number of iso-surfaces to display, modifying the type and range of the scale and the colormap, or applying complex geometrical transformations—changes of coordinates based on functional expressions, e.g. to exploit symmetry or to apply an geometrical offset based on the values in the dataset. In addition to the non-destructive display options, the post-processing module provides a plug-in architecture to enable the application of destructive modifications to views. Gmsh ships with about thirty default plug-ins, that perform operations such as computing sections, elevation maps and stream lines, extracting boundaries, components and time steps, applying differential operators, calculating eigenvalues and eigenvectors, or triangulating point datasets.

All the post-processing features can be accessed either interactively or through the scripting language, which permits to automate all operations, as for example to create animations. Gmsh provides a large number of raster output formats, as well as vector output for high-quality technical renderings using GL2PS [15], which is especially useful for 2-D scenes. Raster files can also be created at sizes larger than what the screen resolution allows by using offscreen rendering [32].

7. Perspectives and Future Developments

In this paper, we have presented the overall architecture of Gmsh as well as some of its specific features. Gmsh is evolving rapidly and many new developments are planned for the near future: anisotropic 3-D algorithms and more advanced boundary layer capabilities, parallel meshing, surface reparametrization and high-order meshes—to name a few.

Amongst these, the development of robust high-order mesh generation procedures is paramount. Indeed, high-order methods in engineering have become an intensive field of research and yet, very little has been done in the domain of curvilinear mesh generation. This problem is far from being trivial: snapping high-order vertices on the geometry may cause elements to become invalid and complex mesh untangling procedures have to be put in place in order to restore mesh validity.

Together with the growing open-source community of Gmsh users and developers, we believe that we are on the right track to implement these features without compromising the original design goals laid out in this article.

Acknowledgements

The authors gratefully acknowledge Électricité de France (EDF) for their financial support. We would also like to thank all persons that have contributed to the development, testing and debugging of Gmsh. Among those, we especially thank David Colignon for being such an indefatigable beta-tester.

Copyright © 2009 John Wiley & Sons, Ltd.

REFERENCES

1. M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.
2. E. Bechet, J.-C. Cuillere, and F. Trochu. Generation of a finite element mesh from stereolithography (stl) files. *Computer-Aided Design*, 34(1):1–17, 2002.
3. K. A. Berger and B. Kubicek. Magnetic field of a 30kV/400V-substation. Private communication, Arsenal Research, Austria.
4. J.-C. Cuilliere. An adaptive method for the automatic triangulation of 3d parametric surfaces. *Computer-Aided Design*, 2:139–149, 1998.
5. Debian. Debian linux. `http://www.debian.org`.
6. G. Dhondt and K. Wittig. Calculix: A free software three-dimensional structural finite element program, 1998. `http://www.calculix.de`.
7. J. Dongarra. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.
8. P. Dular and C. Geuzaine. GetDP: a general environment for the treatment of discrete problems, 1997. `http://www.geuz.org/getdp/`.
9. R. A. Dwyer. A simple divide-and-conquer algorithm for computing delaunay triangulations in o(n log log n) expected time. In *Proceedings of the second annual symposium on Computational geometry*, pages 276 – 284, 1986.
10. L. A. Freitag and C. Ollivier-Gooch. Tetrahedral mesh improvement using face swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21):3979–4002, 1998.
11. P. J. Frey. About surface remeshing. In *9th International Meshing Roundtable*, 2000.
12. P.-L. George and H. Borouchaki. *Delaunay Triangulation and Meshing: Application to Finite Elements*. Hermes, 1998.
13. P.-L. George and P. Frey. *Mesh Generation*. Hermes, 2000.
14. P.-L. George, F. Hecht, and E. Saltel. Automatic mesh generator with specified boundary. *Computer Methods in Applied Mechanics and Engineering*, 92(3):269–288, 1991.
15. C. Geuzaine. GL2PS: an OpenGL to PostScript printing library, 2000. `http://www.geuz.org/gl2ps/`.
16. C. Geuzaine and J.-F. Remacle. Gmsh: a finite element mesh generator with built-in pre- and post-processing facilities, 1996. `http://www.geuz.org/gmsh/`.
17. GNU. The GNU general public license, 1988. `http://www.gnu.org/licenses/gpl.html`.
18. R. Haimes. CAPRI: Computational analysis programming interface (a solid modeling based infra-structure for engineering analysis and design). Technical report, Massachusetts Institute of Technology, 2000.
19. F. Hecht. Bamg: Bidimensional anisotropic mesh generator, 2006. `http://www.freefem.org/ff++`.
20. D. Heller, P. M. Ferguson, and D. Brennan. *Motif Programming Manual*, volume 6A. O'Reilly, second edition, 1994.
21. B. Joe. GEOMPACK – a software package for the generation of meshes using geometric algorithms. *Adv. Eng. Software*, 13:325–331, 1991.
22. J. Lambrechts, R. Comblen, V. Legat, C. Geuzaine, and J.-F. Remacle. Multiscale mesh generation on the sphere. *Ocean Dynamics*, 58:461–473, 2008.
23. P. Laug and H. Borouchaki. Blsurf-mesh generator for composite parametric surfaces-user's manual, 1999. Technical Report, INRIA, France.
24. J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly, 1992.
25. X. Li. *Mesh Modification procedure for General 3-D Non-Manifold Domains*. PhD thesis, Renselear Polytechnic Indtitute, 2003.
26. X. Li, J.-F. Remacle, N. Chevaugeon, and M. S. Shephard. Anisotropic mesh gradation control. In *13th International Meshing Roundtable*, 2004.
27. A. Liu and B. Joe. Relationship between tetrahedron shape measures. *BIT Numerical Mathematics*, 34(2):268–287, 1994.
28. H. Lo and W. X. Wang. Generation of anisotropic mesh by ellipse packing over an unbounded domain. *Engineering with Computers*, 20(4):372–383, 2005.
29. M. Muuss. BRL-CAD, 1984. Army Research Laboratory.
30. C. F. Ollivier-Gooch. GRUMMP — generation and refinement of unstructured, mixed-element meshes in parallel, 1998. `http://tetra.mech.ubc.ca/GRUMMP/`.
31. F. Ortega. GMV: The general mesh viewer, 1996. `http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html`.
32. B. Paul. The Mesa 3D graphics library, 1995. `http://www.mesa3d.org/`.
33. P. P. Pebay and T. J. Baker. Analysis of triangle quality measures. *Mathematics of Computation*, 72(244):1817–1839, 2003.
34. L. Piegl and W. Tiller. *The Nurbs Book*. Springer, 1997.

Copyright © 2009 John Wiley & Sons, Ltd.

35. S. Rebay. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *Journal of Computational Physics*, 106:25–138, 1993.
36. J.-F. Remacle, X. Li, M. S. Shephard, and J. E. Flaherty. Anisotropic adaptive simulation of transient flows using discontinuous galerkin methods. *International Journal for Numerical Methods in Engineering*, 62(7):899–923, 2005.
37. J.-F. Remacle, E. Marchandise, N. Chevaugeon, and C. Geuzaine. Efficient visualization of high order finite elements. *International Journal for Numerical Methods in Engineering*, 69(4):750–771, 2007.
38. Open CASCADE S.A.S. Open cascade. `http://www.opencascade.org`.
39. J. Schöberl. Netgen, an advancing front 2d/3d-mesh generator based on abstract rules. *Comput. Visual. Sci.*, 1:41–52, 1997.
40. W. Schroeder, K. Martin, and B. Lorensen. *The visualization toolkit*. Prentice Hall, 1998.
41. SDRC. I-DEAS master series, 1993.
42. M. Segal and K. Akeley. The OpenGL graphics system: A specification. Technical report, Silicon Graphics Computer Systems, 1992.
43. J. R. Shewchuk. Robust Adaptive Floating-Point Geometric Predicates, May 1996.
44. J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.
45. K. Shimada, A. Yamada, and T. Itoh. Anisotropic triangular meshing of parametric surfaces via close packing of ellipsoidal bubbles. In *6th International Meshing Roundtable*, pages 375–390,, 1997.
46. H. Si. Tetgen a quality tetrahedral mesh generator and three-dimensional delaunay triangulator, 2004. `http://tetgen.berlios.de/`.
47. Siemens PLM Software. Parasolid. `http://www.parasolid.com`.
48. B. Spitzak. FLTK, the fast light tool kit, 1998. `http://www.fltk.org`.
49. Paul S. Strauss. IRIS Inventor, a 3D graphics toolkit. *ACM SIGPLAN Notices*, 28(10):192–200, 1993.
50. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.
51. Dassault Systèmes. Catia. `http://www.3ds.com`.
52. G. V. Vaughan and T. Tromey. *GNU Autoconf, Automake and Libtool*. New Riders Publishing, 2000.
53. S. Vavasis. QMG: mesh generation and related software, 1995. `http://www.cs.cornell.edu/home/vavasis/qmg-home.html`.
54. S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.
55. D. F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoï polytopes. *the Computer Journal*, 24(2):167–175, 1981.
56. N. P. Weatherill. The integrity of geometrical boundaries in the two-dimensional delaunay triangulation. *Communications in Applied Numerical Methods*, 6(2):101–109, 1990.
57. M. A. Yerry and M. S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal for Numerical Methods in Engineering*, 20(11):1965–1990, 1984.

Copyright © 2009 John Wiley & Sons, Ltd.