Gmsh and GetDP in Academia and Industry

C. Geuzaine
University of Liège, Belgium

OctConf 2015 - Darmstadt, September 23 2015
Some Background

• I am a professor of Electrical Engineering and Computer Science at the University of Liège in Belgium, where I lead the ACE research group

• Our research interests: modeling, analysis, algorithm development, and simulation for problems arising in various areas of engineering and science

• We write quite a lot of codes, mostly PDE solvers in C++/Python

• Two codes released under GNU GPL:
  • Gmsh mesh generator: http://gmsh.info
  • GetDP finite element solver: http://getdp.info

• These are long term efforts (both started in 1997)
Some Background

Today, Gmsh and GetDP represent

- half a million lines of (mostly C++) code
- still only 3 core devs; but about 100 with repo write access
- about 1000 people on mailing lists
- about 5000 binary downloads per week (80% Windows)
- about 400 (google scholar) citations per year
Some Background

Today, Gmsh and GetDP represent

- half a million lines of (mostly C++) code
- still only 3 core devs; but about 100 with repo write access
- about 1000 people on mailing lists
- about 5000 binary downloads per week (80% Windows)
- about 400 (google scholar) citations per year

Let’s have a look!
Quick overview of Gmsh

- Gmsh is based around four modules: Geometry, Mesh, Solver and Post-processing; 3 levels of use:
  - Developer: through the (undocumented...) C++ or Python API
  - Advanced user: through the dedicated “.geo” language
  - Novice user: through the GUI (which translates most actions into “.geo” file commands)
- Main characteristic: all algorithms are written in terms of abstract CAD entities, using a “Boundary REPresentation” approach
Figure 5. Multiscale Laplace partitioning method of an aorta ($G = 0$, $N_B = 1$, $⌘ = 1_7$). In this example there are $n = 6$ different levels on which harmonic maps are computed. The red line shows the partition line that recursively splits the mesh into two area balanced mesh partitions (see the resulting mesh partition in Fig. 6).
Any 3-D model can be defined using its Boundary Representation (BRep): a volume 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 $G^0_i$ that are topological entities of dimension 0,

2. Model Edges $G^1_i$ that are topological entities of dimension 1,

3. Model Faces $G^2_i$ that are topological entities of dimension 2,

4. Model Regions $G^3_i$ that are topological entities of dimension 3.
Quick overview of Gmsh

Model entities are topological entities, i.e., they only deal with adjacencies in the model, and we use a bi-directional data structure for representing the graph of adjacencies.

Schematically, we have

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

Any model entity is able to build its list of adjacencies using local operations.
Quick overview of Gmsh

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 $p \in R^d \mapsto x \in R^3$:

1. The geometry of a model vertex $G^0_i$ is simply its 3-D location $x_i = (x_i, y_i, z_i)$.

2. The geometry of a model edge $G^1_i$ is its underlying curve $C_i$ with its parametrization $p(t) \in C_i$, $t \in [t_1, t_2]$.

3. The geometry of a model face $G^2_i$ is its underlying surface $S_i$ with its parametrization $p(u, v) \in S_i$.

4. The geometry associated to a model region is $R^3$. 
Quick overview of Gmsh

Point $p$ located on the curve $C$ that is itself embedded in surface $S$
Quick overview of Gmsh

CAD kernel idiosyncrasies: *seam* edges and *degenerated* edges
Quick overview of Gmsh

CAD kernel idiosyncrasies: *seam* edges and *degenerated* edges
Quick overview of Gmsh

- For the geometry:

  - GModel
  - GVertex
  - GEdge
  - GFace
  - GRegion

Concrete implementation for each CAD kernel (e.g. gmshFace, OCCFace, parasolidFace, fourierFace, levelsetFace, discreteFace).

Direct access via CAD kernel APIs: never translate/convert formats!
Quick overview of Gmsh

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
  //..
};
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;
    //...
};
Quick overview of Gmsh

- For the mesh:
  
  MElem
  MVertex

Each GEntity stores its “internal” vertices. Parallel I/O through GModel.

Minimal storage:
- 44 bytes per vertex, 28 bytes per tetrahedron (12 Mtets/Gb)
- Enriched for specific algorithms
- MEdge and MFace created on demand

MElem provides access to mapping, Jacobian and integration
Quick overview of Gmsh

Recent features:

• Reparameterization of surfaces (“STL remeshing”)

• Coarse grained (distributed, via MPI) and fine-grained (shared memory, via OpenMP) parallel 3D Delaunay meshing algorithm

• Automatic quad and hex-dominant meshing

• Anisotropic meshes and boundary layers

• Homology and cohomology solver
Quick overview of GetDP

- GetDP language (".pro" files) for the natural expression of finite element problems (explicit function spaces and weak forms, ...)
- Solving $\nabla \cdot (a \nabla u) = f$ on domain $\Omega$ translates into:

```
Formulation{
    { Name F; Type FemEquation;
      Quantity {
          { Name u; Type Local; NameOfSpace H1_0; }
      }
      Equation {
        Galerkin { [ a[] * Dof{d u}, {d u} ]; In Omega; ... }
        Galerkin { [ f[], {u} ]; In Omega; ... }
      }
    }
}
```

i.e. a quite direct transcription of the weak form of the problem:

Find $u \in H^1_0(\Omega)$ such that $\int_{\Omega} a \nabla u \cdot \nabla u' \, d\Omega + \int_{\Omega} f u' \, d\Omega = 0$, $\forall u' \in H^1_0(\Omega)$
Quick overview of GetDP

- No distinction between 1D, 2D or 3D; static, transient, time-(multi-)harmonic, eigenproblems
- Easy coupling of fields and formulations (physics), staggered or monolithic, e.g. for explicit Jacobian matrices/sensitivity analysis of strongly coupled nonlinear problems
- Natural handling of non-local (global, integral) quantities, e.g. for circuit coupling
- Linear algebra through PETSc/SLEPc and/or Sparksit/Arpack
Quick overview of GetDP

• Recent developments:
  • Use of Gmsh library for IO, post-processing, mesh-to-mesh interpolation
  • Large scale calculations through domain decomposition methods (> 1 billion DoFs on 10,000 CPUs for time-harmonic wave scattering)
  • High-order eigenvalue problems
  • Built-in Octave and Python interpreters
Gmsh and GetDP in academia and in industry

Actual use is difficult to assess, but today we estimate that

- Gmsh is probably the most popular open source mesh generator; it is used in hundreds of universities, research centers and commercial companies around the world

- GetDP is used intensively in a few dozens universities and companies

Several commercial tools use or integrate (with dual licensing) the codes, e.g. http://www.nxmagnetics.de
Gmsh and GetDP in academia and in industry

Actual use is difficult to assess, but today we estimate that

- Gmsh is probably the most popular open source mesh generator; it is used in hundreds of universities, research centers and commercial companies around the world

- GetDP is used intensively in a few dozens universities and companies

Several commercial tools use or integrate (with dual licensing) the codes, e.g. http://www.nxmagnetics.de

Where do we go from here? The ONELAB project: http://onelab.info
Context of the ONELAB project

**Economic**
- Growing importance of numerical simulation in education and industry
- Prohibitive cost of commercial packages for a significant subset of potential users (SMEs, education, occasional use)

**Scientific**
- High quality of free/open-source software developed in universities and research centers
- Sometimes ahead of commercial equivalents

**Practical**
- No user-friendly interface and/or poor documentation for most open source Finite Element Analysis (FEA) codes
General goal of the ONELAB project

Develop a platform for integrating free Finite Element Analysis (FEA) software:

• allowing the integration (by co-simulation) of any open-source code, whatever their characteristics

• with an intuitive GUI allowing newbie users to get started and guided into the world of FE modeling — but with the possibility to construct sophisticated, upgradable, multi-code, multi-platform scripts for the specialized user

• and with the possibility to construct both education- and business-specific tools
The solution should overcome two difficulties associated with free FEA software:

(1) The heterogeneity of the tools

(2) The missing “expert layer”, top-down validation and documentation found in commercial offerings
State of the art

- Many closed, commercial tools (COMSOL, Ansys Workbench, ...)
- More open tools, e.g. GiD (http://gid.cimne.upc.es), but not free
- Closest free software: SALOME (http://salome-platform.org), but very large project, not well suited for building “fast and light” domain-specific applications
- Other open source projects: “multi-physic” codes (Elmer, etc.) still mainly focused on a single domain (CFD, solids, E-M); the implementation of new physics leads to bare-bones features, far from the refinement of specialized codes; no easy-to-use interface and no driving of other codes
ONELAB guiding principles

• Don’t reimplement, interface the existing!

• Make it as small, lightweight and as easy to maintain as possible (no solver-dependent code in the interface)

• Make it easy to provide templates, with interactive parameter modification

• ONELAB role = data centralization, (optional) modification and redispaching
ONELAB guiding principles

• Don’t reimplement, interface the existing!
• Make it as small, lightweight and as easy to maintain as possible (no solver-dependent code in the interface)
• Make it easy to provide templates, with interactive parameter modification
• ONELAB role = data centralization, (optional) modification and redispaching

Issues of completeness and consistency of the parameter set are completely dealt with on the solver side
ONELAB features

(1) Heterogeneity of the tools

(2) Missing “expert” layer, top-down validation and documentation
ONELAB features

(1) Abstract interface to FEA codes
ONELAB features

(1) **Abstract interface** to FEA codes

- CAD & meshing; physical properties, constraints & code drivers; post-processing
ONELAB features

(1) **Abstract interface** to FEA codes

- CAD & meshing; physical properties, constraints & code drivers; post-processing
- Implemented in Gmsh:
ONELAB features

(1) **Abstract interface** to FEA codes

- CAD & meshing; physical properties, constraints & code drivers; post-processing
- Implemented in Gmsh:
  - Parameter exchange library
ONELAB features

(1) **Abstract interface** to FEA codes

- CAD & meshing; physical properties, constraints & code drivers; post-processing
- Implemented in Gmsh:
  - Parameter exchange library
  - Native C++ and Python clients; Parser for non-native clients
ONELAB features

(1) **Abstract interface** to FEA codes
   - CAD & meshing; physical properties, constraints & code drivers; post-processing
   - Implemented in Gmsh:
     - Parameter exchange library
     - Native C++ and Python clients; Parser for non-native clients

(2) **Development and documentation of templates** ("meta-models")
ONELAB features

(1) **Abstract interface** to FEA codes
   - CAD & meshing; physical properties, constraints & code drivers; post-processing
   - Implemented in Gmsh:
     - Parameter exchange library
     - Native C++ and Python clients; Parser for non-native clients

(2) Development and documentation of templates ("meta-models")
   - Model: blackbox, parameterizable via abstract interface
ONELAB features

(1) **Abstract interface** to FEA codes
   - CAD & meshing; physical properties, constraints & code drivers; post-processing
   - Implemented in Gmsh:
     - Parameter exchange library
     - Native C++ and Python clients; Parser for non-native clients

(2) **Development and documentation of templates** (“meta-models”)
   - Model: blackbox, parameterizable via abstract interface
   - Meta-model: set of models + selection logic
ONELAB implementation

Client-server:

- Clients: CAD kernels, meshers, solvers, post-processors
- Server: Gmsh (currently) + database

Abstract interface:

- The server **has no a priori knowledge of the clients** (no meta-language or exchange file format)
- The server **does not write input files for (native) clients**: the client communicates with the server to define what information should be exchanged
ONELAB implementation

Abstract interface to physical properties, constraints & code drivers:

- Library for parameter exchange:
  - Reference server in C++ for portability, e.g. on iOS/Android (onelab::server)
  - Clients in C++ (onelab::client) or Python
  - Exchange parameters (onelab::parameter) through TCP/IP or Unix sockets, or in-memory
ONELAB implementation

Abstract interface to physical properties, constraints & code drivers:

- Library for parameter exchange:
  - Reference server in C++ for portability, e.g. on iOS/Android (onelab::server)
  - Clients in C++ (onelab::client) or Python
  - Exchange parameters (onelab::parameter) through TCP/IP or Unix sockets, or in-memory

- “Native” clients use C++ or Python directly
- “Non-native” clients use Python, by instrumenting their input files
  - Currently: Elmer, OpenFOAM, Code_Aster, Abaqus, Gnuplot
ONELAB implementation

Native client: overloading of existing functions (GetDP)

...  
DefineConstant[ Numstep = \{ 50, Name "Elmer/Number of time steps"\} ];
DefineConstant[ TimeStep = \{ 0.1, Name "Elmer/Time step"\} ];
...

Non-native clients: instrumentation the input files of the client (Elmer)

...
OL.line NumStep.number(50, Elmer/, Number of time steps);
OL.line TimeStep.number(0.1, Elmer/, Time step);
Simulation
  Simulation Type = Transient
  Timestep sizes = OL.get(TimeStep)
  Timestep Intervals = OL.get(NumStep)
...

Preprocessing: conversion into a valid input file for the client (Elmer)

...
Simulation
  Simulation Type = Transient
  Timestep sizes = 0.1
  Timestep Intervals = 50
...
ONELAB implementation

onelab::parameter

- name as ‘/’-separated path
- dynamic dependency list of clients and status change
- decorations (help, bounds, choices, ...)
- serialization and deserialization
ONELAB implementation

onelab::parameter

- name as ‘/’-separated path
- dynamic dependency list of clients and status change
- decorations (help, bounds, choices, ...)
- serialization and deserialization

Example for native Gmsh & GetDP clients (in .geo or .pro files):

    DefineConstant[ N = {32, Name "Number of slices"} ];

Example for Python client:

    c = onelab.client()
    N = c.defineNumber('Number of slices', value=32)
ONELAB implementation

onelab::parameter

- name as ‘/’-separated path
- dynamic dependency list of clients and status change
- decorations (help, bounds, choices, ...)
- serialization and deserialization

Example for native Gmsh & GetDP clients (in .geo or .pro files):

DefineConstant[ N = {32, Name “Number of slices”} ];

Example for Python client:

```python
c = onelab.client()
N = c.defineNumber('Number of slices', value=32)
```

Let’s have another look!
Conclusion

• Growing use of Gmsh and GetDP in academia and industry

• “Vulgarization” requires quite a bit of work, hence the ONELAB project:
  • A simple (trivial?) way to interface FEA solvers
  • Interactive, based on Gmsh, and free
  • And now available on iOS and Android

• Give it a try:

  http://onelab.info

• Wishlist: we want an Octave server (and client)!
PS: Doing open source is rewarding!

Comment about Gmsh on http://www.fltk.org (sic):

>From Anonymous, 20:33 May 18, 2004 (score=1)
Je suis outre du programme pour des intellectuels vous devrez avoir plus d’imagination vous faite onte au genie informatique
PS: Doing open source is rewarding!

Comment about Gmsh on [http://www.fltk.org](http://www.fltk.org) (sic):

>From Anonymous, 20:33 May 18, 2004 (score=1)
Je suis outre du programme pour des intellectuels
vous devrez avoir plus d’imagination vous faite
onte au genie informatique

Translation (including misspellings!) for the non-french speaking:

>From Anonymous, 20:33 May 18, 2004 (score=1)
I am ashamed of the program for intelectuals you
should have more imagination you are the schame of
computer science

😊
Thanks for your attention!

cgeuzaine@ulg.ac.be