FreeFEM3D Documentation.

Stéphane Del Pino, Olivier Pironneau
http://www.freefem.org

Version 1.0pre10 (May 11, 2011)

Contents

Acknowledgements

We want here to address special thanks to Robert Li and Alban Pages for there interest, remarks and encouragements.

Chapter 1

Generalities

FreeFEM3D (aka ff3d) stands for "*FREE Finite Element Method in 3 Dimensions*".

This software will assist you in solving problems which are modeled by partial differential equations. As the name indicates it is free, (subject to the GPL guidelines), it uses FEM (the finite element method) and it is for three dimensional problems.

Some mathematical knowledge is needed to use FreeFEM3D, since you are required to input the partial differential equations which describe your problem.

Based on our experience with FreeFEM 3.4, FreeFEM+, FreeFEM++ we believe that the best way to describe a problem is through a language adapted to partial differential equations (PDE). Thus for each problem one needs to write a program and submit it to FreeFEM3D which will compile it and run it, and/or report bugs. Therefore it is impossible to use FreeFEM3D without reading some part of this manual or going through some of the examples.

There are 3 steps to solve a PDE

1. Input the geometry and the coefficients
2. build and solve the linear or non-linear discrete systems
3. display graphically the output.

The input of the geometry for a tri-dimensional problem is a formidable task; the entire CAD industry is busy with it. Realizing this, FreeFEM3D relies on another program to define the geometry: POV-Ray.

POV-Ray is an image synthesis software which is also free and also runs on a number of operating systems. You will need to learn to use POV-Ray to use FreeFEM3D.

Finally 3D graphics to display the solution of PDEs is also a formidable problem and so FreeFEM3D produces output files which can be visualized with Medit (a simple package written by Pascal Frey), ParaView (from KitWare), MayaVi or OpenDx (data explorer), IBM's display software. While Medit or ParaView run in MS-Windows, MacOS and Unix (so long as OpenGL is installed), OpenDx is really a unix package and so to run it in windows one needs to install an X11 package (such as Xfree86) and cygwin.

1.1 Configurations

This software requires at least 64 Megabytes of RAM and runs on the following machines:

- Macintosh with MacOS X 10.1 or later;

- PC compatible with MS-Windows (any version using `cygwin`);
- and GNU/Linux (and others Unix systems)

Note that *compilation* of `FreeFEM3D` may require nearly 500 Megabytes of memory when using the full optimization mode (see below). The standard optimized mode (`-O2`) needs nearly 150 Mb.

1.2 Contacts

Their are several ways for contacting `FreeFEM3D`'s team. The first is to join directly one of its member, but it should be preferred to use the appropriated mailing lists and report bugs using the BTS — all of those features being hosted by the Savannah project.

1.2.1 `FreeFEM3D`'s team

`FreeFEM3D`'s team is composed of the following members:

- Project Director:
  Olivier Pironneau `mailto:Olivier.Pironneau@math.jussieu.fr`,
- Main author:
  Stéphane Del Pino `mailto:Stephane.DelPino@math.jussieu.fr`,
- Mesh improvements:
  Cécile Dobrzynski `mailto:dobrzynski@ann.jussieu.fr`,
- Contributor:
  Pascal Havé `mailto:Pascal.Have@math.jussieu.fr`,
- Contributor and Debian Packager:
  Christophe Prud'homme `mailto:prudhomm@debian.org`.

1.2.2 Project pages

Being a member of the `FreeFEM` softwares family, `FreeFEM3D` was developed at the *Laboratoire Jacques-Louis Lions* of the University of Paris VI. `FreeFEM3D` is hosted at the `FreeFEM` head quarter: `http://www.freefem.org`. The link that can be used to access directly to the project page is `http://www.freefem.org/ff3d`.

Since `FreeFEM3D` is a *free* software, it is developed transparently: every one can access its source code using `cvs`. We have chosen to use Savannah as a project development tool. Savannah provides to hosted free softwares:

- code source archiving using `cvs`,
- mailing lists management,
- Bug Tracking System (BTS),
- and a lot more...

`FreeFEM3D`'s Savannah page is `http://savannah.nongnu.org/projects/ff3d`, any relevant information to use Savannah and `FreeFEM3D` conjointly will be found there.

Mailing-lists

Four mailing-lists are hosted by Savannah:

- `ff3d-users`: for `FreeFEM3D`'s usage related questions or suggestions;
- `ff3d-dev`: for developers discussions. BTS messages are copied there;
- `ff3d-cvs`: a read-only list that logs `cvs` messages to inform developers of what changes in the sources, and
- `ff3d-announce`: a read-only low traffic list, used to announce `FreeFEM3D`'s related events...

It is recommended to subscribe at least to `ff3d-users` and `ff3d-announce`.

The Bug Tracking System

The BTS is the best place to report bugs or wishes. Using it, developers will keep a trace and poster will be automatically informed of any change related to his request, or see what priority has been assigned to it...

1.3 Requirements and Installation

1.3.1 Requirements

If you want to use a pre-compiled version of `FreeFEM3D`, you will only have to install, the pre-processing and and post-processing tools:

- the `POV-Ray` package (`http://www.povray.org`) which will help you in preparing the geometry.
- A visualization software like `medit`. It is free to download and use. Written by P. Frey and can be download from:

  `http://www.ann.jussieu.fr/ frey/logiciels/medit.html`.
- One can also use `ParaView` which require more investment and can be freely download from

  `http://www.paraview.org`.
- Another interesting tool based on VTK is `MayaVi`

  `http://mayavi.sourceforge.net`.
- `OpenDx`, another free visualization software, very powerful and open source but somewhat complex; it uses the same pipeline approach as AVS (check full information at `http://www.opendx.org`).

1.3.2 Getting pre-complied binaries

Binary files for common architectures and systems can be downloaded at

http://www.freefem.org/ff3d/binaries.html.

If you are the happy owner of a Debian GNU/Linux[1] system, you can install it by the simple command:

```
apt-get install freefem3d
```

For more details with the Windows OS see Appendix 6.

1.3.3 Getting the sources (optional)

Since FreeFEM3D is developed under the terms of the General Public Licence, it is possible to download, modify and even redistribute its sources[2].

There are two ways of downloading the sources.

Archive files

The first way consists in getting an archive file from the web site at

http://www.freefem.org/ff3d/sources/.

After you unpacked the downloaded archive, a new directory called FreeFEM3D, containing the sources will be created.

Using the cvs repository

This second way is probably the best if you want to recompile your own version. Cvs allows you to keep your source code version up to date, it means that after any bug-fix version you man just download automatically the modifications. It also provides the possibility to retrieve old versions of the code just specifying a date. Moreover it is possible to download the development version of the code.

In what follows, we will only describe the Unix-like procedure, users of WinCVS should adapt easily it.

Before giving minimalist hints that will help you to download the cvs source tree, we have to point out on some special Savannah's configuration. To improve security, Savannah's hackers have chosen to allow only SSHv2 access to their servers. You have to ensure that your CVS_RSH variable is set to ssh. With a Bourn-like shell (sh, ksh, bash, zsh...) use the instruction:

```
export CVS_RSH=ssh
```

If you run a C-like shell:

```
setenv CVS_RSH ssh
```

[1] At the time of writing these lines the freefem3d package is only available in the Sarge release.

[2] Note that **binary-only** distributions are forbidden!

Since you will need this to be set *before* executing *each* `cvs` command, it is recommended that you set it once for all in the appropriate shell start-up file. Reading `cvs` documentation may be a useful help: check its web site at http://www.cvshome.org to get more information.

Let us now recall the basic `cvs` commands that will allow you to maintain your own `cvs`-tree synchronized.

Checking-out the code is required only once. You will have to do it to get your first copy of the sources. Executing the instruction line:

```
cvs -d :ext:anoncvs@savannah.gnu.org:/cvsroot/ff3d co ff3d
```

will create a `ff3d` directory containing the current development version of the package.

Keeping `FreeFEM3D` up to date will only require the command:

```
cvs -z3 update
```

See http://savannah.nongnu.org/cvs/?group=ff3d for more details.

1.3.4 Building the code

`FreeFEM3D` needs several tools to be built. Some of them are optional, others are just essential. Note that the code compilation requires lots of memory when building an optimized version.

All of the following softwares are common Unix packages. They should also be found on MacOS X and are provided by `cygwin` when building for MS-Windows.

Optional packages

So far three packages are optional: a POSIX thread library, `Qt` and `VTK`.

POSIX thread library Use of `pthread` allows a better usage of SMP[3] machines, and allows to run simultaneous tasks on multi-task systems. So far, only a few procedures really take advantage of multi-thread programming. Priority is given to linear algebra which is the bottle neck in this kind of number crunching application.

Qt This package is a *C++* class library optimized for graphical user interface development. It is developped by `Trolltech`. This is used to built the **experimental** GUI of ff3d. To get more information about it, check http://www.trolltech.com.

VTK `VTK` stands for "the Visualization Toolkit", it is an opensource *C++* library developed by `KitWare` that provides high level facilities to perform scientific graphics in 2D or 3D. Its presence in `FreeFEM3D` is still **experimental** and **undocumented**. To get this library, connect to http://www.vtk.org.

Required packages

Only one package is really required to compile `FreeFEM3D`:

[3] Symmetric Multi-Processor

Bison This is the GNU implementation of `yacc` (which stands for *yet another compiler compiler*). This software is a great help for the construction of languages parsers. Since `FreeFEM3D` uses two languages (one for the problem description, the other for the geometry) it uses such a tool to generate their compilers. Being a part of the GNU project, `bison` can be downloaded from http://www.gnu.org.

Automake/Autoconf These packages are used to generate the `Makefiles`. They are in charge of the package configuration and build dependencies. Each of the packages of this family have to be installed and particularly `libtool`.

C++ An ANSI *C++* compiler is also required. `FreeFEM3D` use some bleeding edge *C++* constructions that need good compiler. Recent GNU GCC compilers offer very good ANSI *C++* implementation. `FreeFEM3D` has been developed using those tools. We recommend the use of the most recent `g++` version to build the binary. *Any feedback concerning the use of any other compiler is appreciated.*

Make A version of `make` is obviously needed to build `FreeFEM3D`. The GNU version, sometimes called `gmake` is recommended as well.

Building instructions

`FreeFEM3D` uses a `configure` script to detect your configuration and generate `Makefiles`. First download an archive, then this script should be found in the `ff3d` directory. If it is not there or if you used `cvs` to get the code, you have to generate it. This is very simple. Enter the `ff3d` directory — all the following commands will be performed from this particular place. Now, type the command

```
autoreconf -i
```

This can produce warnings saying that some files are replaced, this is not an error. You can now call the `configure` script.

Configuring Being built with `automake` and `autoconf`, the `configure` script uses all standard options. To get the complete list of them, type

```
./configure --help
```

at the shell prompt. We will now discuss the `FreeFEM3D` special options. Note that most of "`--enable-`" option have an opposite "`--disable-`" option.

`--enable-real_t` is used to change the type of variable to store reals. Default value is `double`, this can be changed to `float` using `--enable-real_t=float`. Others types should be added in the future.

`--enable-exec` permits the use of the `exec` instruction in `FreeFEM3D` files. It is enabled by default. Since it allows execution of external programs within `FreeFEM3D`, it is a potentially dangerous for security. This is why it can be deactivated.

`--enable-debug` is used to build a debugging version. An optimized (`-O2`) version is generated if this option is omitted.

`--enable-optimize` generates an *even more* optimized version (using none standard g++ options). This option conflicts with the `--enable-debug` option.

`--enable-gui` allows the generation of the code with `GUI` support. This option is automatically enabled if `VTK` and `Qt` are detected — but can still be deactivated using the `--disable-gui` option

`--enable-pthread` compiles `FreeFEM3D` using POSIX Thread this allows to proceed some tasks in parallel. It is enabled by default if the pthread library is found.

The execution of the `configure` script, creates the `Makefiles`. Typing

```
make
```

will generate the executable called `ff3d` or `ff3d.exe`, for the MS-Windows version, after a few[4] compilation time.

1.4 Legal conditions

1.4.1 Warning

`FreeFEM3D` is a scientific product to help you solve Partial Differential Equations in 3 dimensions; it assumes a basic knowledge and understanding of the Finite Element Method and of the Operating System used. It is also necessary to read carefully this documentation to understand the possibilities and limitations of this product. The authors are **not responsible** for any errors or damages due to wrong results.

1.4.2 GNU Genaral Public Licence

It is in its name: `FreeFEM3D` is a *free software*. It is distributed under the GNU GPL guidelines as said here:

> *This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.*
>
> *This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.*
>
> *You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.*

See the `COPYING` file in ff3d's root directory, the exact terms of this licence can also be consult online at GNU project official web site: http://www.gnu.org/copyleft/gpl.html.

[4] or a bit more ;-)

Chapter 2

An overview of FreeFEM3D

In this chapter we will focus on a fairly complex example so that the user will learn the existence of the main possibilities of FreeFEM3D. In the next chapters we will use simpler examples while focusing on different features. Finally in the last chapter more complex examples will be proposed.

2.1 The problem

Consider the following coupled problem: find (u, v) such that

$$\left|\begin{array}{l} -\Delta u + v = f \text{ in } \mathcal{O}, \\ u = x(x-1) + y(y-1) + z(z-1) \text{ on } \partial\mathcal{O}, \\ \\ v - \Delta v + u = g \text{ in } \mathcal{O}, \\ v = \sin(\pi x)\sin(\pi y)\sin(\pi z) \text{ on } \partial\mathcal{O}. \end{array}\right. \tag{2.1}$$

where $f = -6 + \sin(\pi x)\sin(\pi y)\sin(\pi z)$,
and $g = (3\pi^2 + 1)(\sin(\pi x)\sin(\pi y)\sin(\pi z)) + x(x-1) + y(y-1) + z(z-1)$.

which has an analytical solution:

$$(u, v) = (x(x-1) + y(y-1) + z(z-1),\ \ \sin(\pi x)\sin(\pi y)\sin(\pi z))$$

whatever the domain $\mathcal{O}$ and its boundary $\partial\mathcal{O}$ are.

2.2 The program using PDEs

To solve (2.1) in the domain $\mathcal{O}$ that is the sphere centered in 0 and whose radius is 0.8 we can write the following program (example1.ff).

example1.ff

```
1 vector n = (20, 20, 20);
2 vertex a = (-1,-1,-1);
3 vertex b = ( 1, 1, 1);
4 double pi = 4*atan(1);
5
6 scene S = pov("example1.pov"); // loading POV-Ray geometry file
```

```
7
8  mesh M = structured(n,a,b);
9
10 domain O = domain(S,inside(<1,0,0>));
11
12 function uexact = (x*(x-1) + y*(y-1) + z*(z-1));
13 function vexact = sin(pi*x)*sin(pi*y)*sin(pi*z);
14
15 solve(u,v) in O by M method(type=penalty)
16 {
17    pde(u)
18      - dx(dx(u)) - dy(dy(u)) - dz(dz(u)) + v
19        = -6 + vexact;
20      u = uexact on <1,0,0>;
21
22    pde(v)
23      v - div(grad(v)) + u
24         = (3*pi^2 + 1)*vexact + uexact;
25      v = vexact on <1,0,0>;
26 }
27
28 function KiO = one(<1,0,0>);
29
30 double I=int[M](KiO*(uexact-u)^2);
31 double J=int[M](KiO*uexact^2);
32 cout << sqrt(I/J) << "\n";
33 I = int[M](KiO*(vexact-v)^2);
34 J = int[M](KiO*vexact^2);
35 cout << sqrt(I/J) << "\n";
36
37 mesh T = tetrahedrize(O,M);
38
39 save(medit, "u", u, T);
40 save(medit, "u", T);
41 save(medit, "v", v, T);
42 save(medit, "v", T);
43
44 save(raw, "u.dat", u, M);
45 save(raw, "v.dat", v, M);
```

The geometry is given in a POV-Ray file (example1.pov)

example1.pov

```
1 sphere {
2   <0,0,0>, 0.8
3   pigment { color rgb <1,0,0> }
4 }
```

2.3 Describing the program step by step

From this example we see that there are different *types* such as

```
vector, vertex, scene, mesh, domain, double, function
```

and that each instruction is ended using a *semicolon*. The syntax is borrowed from the *C/C++* language, whenever possible.

Let us go through the example:

example1.ff

```
1 vector n = (20, 20, 20);
2 vertex a = (-1,-1,-1);
3 vertex b = ( 1, 1, 1);
4 double pi = 4*atan(1);
```

Those four lines define `n`, `a` and `b` as $\mathbb{R}^3$ elements and the *double precision number* `pi` as π. These will be used later to define a box from the two points a and b meshed by an (n_x, n_y, n_z) uniform Cartesian grid.

example1.ff

```
5
6 scene S = pov("example1.pov"); // loading POV-Ray geometry file
```

The *scene* `S` is defined using the `POV-Ray` description contained in the file `example1.pov`. Since no path was given here, the file must be in the current directory, the same that contains `example1.ff`. Note also that *C++*-like commentaries can be used.

example1.ff

```
7
8 mesh M = structured(n,a,b);
```

Here we construct the background cartesian mesh `M` of the fictitious embedded domain using $20 \times 20 \times 20$ vertices in each direction (this is given by the value of `n`). Vectors `a` and `b` are two vertices of the diagonal defining a box. The frame (x, y, z) is *direct* so if (Ox) is horizontal from left to right and (Oy) is vertical bottom to top on the screen then (Oz) points towards you. This is the *right hand rule*.

example1.ff

```
9
10 domain O = domain(S,inside(<1,0,0>));
```

The computational *domain* `O` is declared. Its definition uses the object of label `<1,0,0>` of the *scene* `S`. Here the computational domain is the interior of the object.

example1.ff

```
12 function uexact = (x*(x-1) + y*(y-1) + z*(z-1));
13 function vexact = sin(pi*x)*sin(pi*y)*sin(pi*z);
```

`uexact` and `vexact` are two analytical functions defined as:

$$\begin{aligned} u_{\text{exact}} &= x(x-1) + y(y-1) + z(z-1) \\ v_{\text{exact}} &= \sin(\pi x)\sin(\pi y)\sin(\pi z) \end{aligned}$$

One can see that `FreeFEM3D` supports the algebra of functions. `x`, `y` and `z` have to be seen as functions by abuse of notation.

example1.ff
```
15 solve(u,v) in O by M method(type=penalty)
16 {
17    pde(u)
18      - dx(dx(u)) - dy(dy(u)) - dz(dz(u)) + v
19        = -6 + vexact;
20      u = uexact on <1,0,0>;
21
22    pde(v)
23      v - div(grad(v)) + u
24         = (3*pi^2 + 1)*vexact + uexact;
25      v = vexact on <1,0,0>;
26 }
```

This instruction bloc defines *both* the PDE problem and how to *solve* it! Let us now focus to the details.

example1.ff
```
15 solve(u,v) in O by M method(type=penalty)
16 {
```

means that we are going to *solve* a *coupled* PDE problem whose *unknowns* are the *functions* `u` and `v`, defined on `M`, and that to solve the problem we will use the *mesh* `M`.

`method(type=penlaty)` is optional, it is here to say to the solver to use *penalty* method for boundary conditions. Many options can be passed to the solver.

The next block defines the two coupled PDEs. The first

example1.ff
```
17    pde(u)
18      - dx(dx(u)) - dy(dy(u)) - dz(dz(u)) + v
19        = -6 + vexact;
```

defines the PDE on `u`:

$$-\partial_x\partial_x u - \partial_y\partial_y u - \partial_z\partial_z u + v = -6 + v_{\text{exact}} \text{ in } \mathcal{O}.$$

Using this formulation we illustrate the fact that *general second order* operator can be approximated. We will see that non constant coefficient can be used. The computational domain is provided on line 17.

Then,

example1.ff
```
20      u = uexact on <1,0,0>;
21
```

defines the *Dirichlet* boundary conditions associated to the unknown u : $u = u_{\text{exact}}$ on $\partial\mathcal{O}$, where $\partial\mathcal{O}$ is the boundary of objects which *rgb color* is `<1,0,0>` (red).

Now, the second equation is given, using a more compact form

example1.ff
```
22    pde(v)
23      v - div(grad(v)) + u
24         = (3*pi^2 + 1)*vexact + uexact;
25      v = vexact on <1,0,0>;
```

Which is

$$\left|\begin{array}{l} v - \nabla\cdot\nabla v + u = (3\pi^2+1)v_{\text{exact}} + u_{\text{exact}} \text{ in } \mathcal{O}. \\ \text{with } v = v_{\text{exact}} \text{ on } \partial\mathcal{O}. \end{array}\right.$$

Remark 1 *One should note that boundary condition on each variable are given while describing the associated PDE. The boundary condition process has to be understood precisely (see section 4.5.3). Moreover, in the case of* PDE systems *users may find* variational formulae *better suited and less prone to errors.*

example1.ff

```
27
28 function KiO = one(<1,0,0>);
```

defines the *function* `KiO` as the *indicator* function of *objects* whose color is `<1,0,0>` in the `POV-Ray` file:

$$\texttt{KiO}(x,y,z) = \left|\begin{array}{l} 1 \text{ if } (x,y,z) \text{ is inside } \textbf{at least one} \text{ object of color } \texttt{<1,0,0>}. \\ 0 \text{ elsewhere.} \end{array}\right.$$

Remark that `function` allows the declaration of *analytical* functions. *There is no approximation at this point!* `KiO` is the *exact* function.

Finally to check the results against the analytical solution we print the relative L^2 errors[1]

$$\left|\begin{array}{l} (\int_M \mathbf{1}_{\mathcal{O}}(u-u_{\text{exact}})^2 / \int_M \mathbf{1}_{\mathcal{O}} v_{\text{exact}}^2)^{\frac{1}{2}} \text{ and,} \\ (\int_M \mathbf{1}_{\mathcal{O}}(v-v_{\text{exact}})^2 / \int_M \mathbf{1}_{\mathcal{O}} v_{\text{exact}}^2)^{\frac{1}{2}} \end{array}\right.$$

computed the following way:

example1.ff

```
30 double I=int[M](KiO*(uexact-u)^2);
31 double J=int[M](KiO*uexact^2);
32 cout << sqrt(I/J) << "\n";
33 I = int[M](KiO*(vexact-v)^2);
34 J = int[M](KiO*vexact^2);
35 cout << sqrt(I/J) << "\n";
```

The last lines are the saving instructions of the solution for the post-processing step. Visualization is a complex task in 3D, but the use of embedded fictitious domain method makes it much more difficult. Here is an example of strategy that can be used

example1.ff

```
37 mesh T = tetrahedrize(0,M);
38
```

A visualization mesh `T` is built. This mesh is defined as the "intersection" of the domain $\mathcal{O}$ and a tetrahedrization of the mesh `M`[2]. The mesh `T` only contains tetrahedra that do intersect the domain $\mathcal{O}$. Doing so, there is no parasite values. Note also that the mesh `T`

- does not fit exactly the geometry of $\mathcal{O}$, and

[1] $\int_{\mathcal{O}}$ will be implemented later
[2] each hexahedron is divided into five tetrahedra.

- does not contain boundary informations.

It cannot be used for the computation. Such approach may be possible in the future.

example1.ff

```
39 save(medit, "u", u, T);
40 save(medit, "u", T);
41 save(medit, "v", v, T);
42 save(medit, "v", T);
```

are the saving instructions in the `medit` format for the post-processing. For each variable, the values and the mesh have to be saved separatly.

example1.ff

```
44 save(raw, "u.dat", u, M);
45 save(raw, "v.dat", v, M);
```

These final lines give an other example of saving the data. This time the `raw` format is used: the only stored information is the values of the functions. The saving mesh is `M`. These files can be post-processed using `OpenDx`, for instance, using the cartesian grid structure of `M`.

2.4 Running the program

When configured properly running this example will be

```
ff3d.exe example1.ff
```

(under MS-Windows), and

```
./ff3d example1.ff
```

under any Unix (if `ff3d` and `example1.ff` are in the same directory). Since `FreeFEM3D` is quite verbose, the output is not reproduced here.

The PDE system is discretized by the embedded fictitious domain method (EFDM) combined with the Q^1-Finite Element Method on the uniform grid of the fictious domain. Unless otherwise specified the linear system is solved by a Conjugate Gradient Method not preconditioned.

2.5 Post-processing

This is the weakest point of `FreeFEM3D`!
Up to now, there is no graphical user interface in `FreeFEM3D`. Thus one needs to learn to use a visualization package also. In this part, we give a few hints for a typical visualization of the results. The visualization softwares we typically use are `Medit` and `OpenDx`. We recommend to the user to read the manuals of these packages. Furthermore to obtained good looking picture one may have to add instructions specifically for graphics such as surface meshes.

2.5.1 Using Medit

Medit is a free of charge software. Binaries of this package can be downloaded at

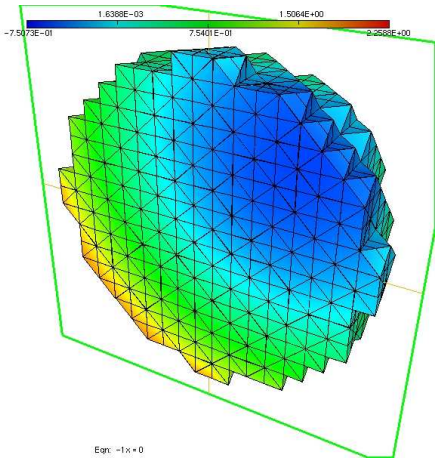
http://www-rocq1.inria.fr/gamma/medit/medit.html.

The online documentation is in *french*, but the software is very simple to use. Running FreeFEM3D on example1.ff produces 4 files for visualization, 2 meshes files (u.mesh and v.mesh) and 2 data files (u.bb and v.bb). To post-process u, use the command

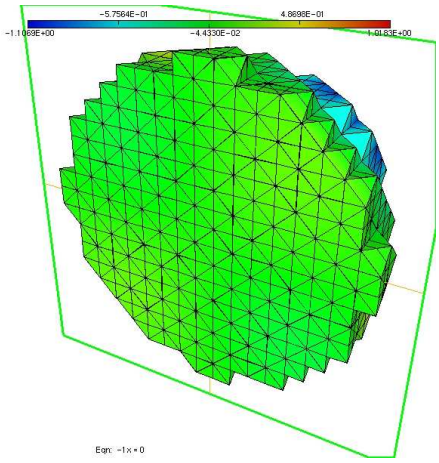
```
medit u
```

in the directory that contains the u.* files. The figures 2.1(a) and 2.1(b) are obtained the following way: once Medit is launched

- press F1 to cut the mesh,
- press m to color the mesh with the data.



(a) u on a cutting plane



(b) v on a cutting plane

Figure 2.1: Medit post-processing using the visualization mesh T

One can access the online help of Medit pressing the h key in the graphic window. As it was previously mentioned the visualization mesh T does not fit the exact geometry. It contains all the tetrahedra that intersect the computational domain.

2.5.2 Using ParaView

ParaView is an open-source software developed by Kitware that relies on VTK. Binaries and sources of this package can be downloaded at

http://www.paraview.org.

The use of this software is quite intuitive for people used to *dataflow* visualization packages. Information concerning the documentation of this software can be found on ParaView's web page.

2.5.3 Using OpenDx

Being a professional tool, OpenDx is very powerful but its use complex, reading the manual is necessary to get advantage of it and to produce good graphics. Lots of material about OpenDx can be found at http://www.opendx.org.

We will just give here the basic tips to start with OpenDx. First, remember that no specific format is implemented by now. We just use the raw output, so, only computations on cartesian structured grids can be post-process. The process is the following:

- launch OpenDx,
- click on Import Data...,
- choose Grid or Scattered file, and then
- on Describe Data...

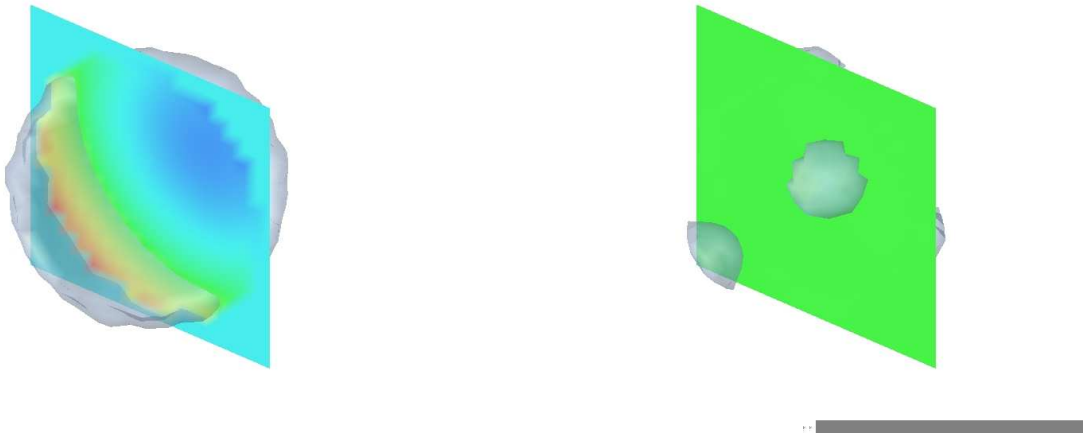
Another window appears. You will define here the geometry of the computation. Set

- the data file (for instance u.dat or v.dat) and
- the Grid size (just fill the 3 first fields).

You can optionally set the cell sizes (Grid positions). When it is done you need to save this (File and then Save as...). You can now close this window.

In the previous window the Visualize Data... button should be accessible. Clicking on it will launch a default visualization program.

For u and v, following the previous commands leads to the figures 2.2(a) and 2.2(b). The difficulties of the post-processing of embedded fictitious domain simulations is enlighten by this example. One sees on figure 2.2 that the it is not easy to identify the computed values from the parasite values (outside the computational domain).



(a) u on a cutting plane, and the iso-surface 0.75

(b) v on a cutting plane, and the iso-surface 0.75

Figure 2.2: `OpenDx` post-processing using the background mesh `M`

Chapter 3

FEM and EFDM

Unless you are familiar with the Finite Element Method (FEM) implemented in an Embedded Fictitious domain (EFDM) it is wise to read the present chapter to understand some of the peculiarities of `FreeFEM3D`.

Let us look at a simple problem:

$$\left| \begin{array}{l} \nabla \cdot \nu \nabla u = f \text{ in } \Omega \\ u|_\Gamma = u_\Gamma, \end{array} \right. \tag{3.1}$$

where $\Gamma = \partial\Omega$ is the boundary of Ω and ν is positive everywhere.

The implementations below are made with Ω a sphere centered at the origin of radius 0.5 embedded (for numerical purpose only) in the cube $(-1,1)^3$, $f = 1$ and $u_\Gamma = 0$.

3.1 Finite Element Method

The Finite Element Method (FEM) is a Galerkin method applied to the variational formulation of the problem. In the case of (3.1) the variational formulation consists in finding $u \in H^1(\Omega)$ satisfying the boundary condition ($u_{|_\Gamma} = u_\Gamma$) and

$$\int_\Omega (\nu \nabla u \cdot \nabla w - f w) = 0 \quad \forall w \in H_0^1(\Omega), \tag{3.2}$$

where H_0^1 is the subspace of H^1 of functions which vanishes on the boundaries and H^1 is the space of square integrable functions with square integrable derivatives.

The Galerkin method consists in approximating the problem by replacing $H^1(\Omega)$ by a finite subspace and a special case is the FEM of order 1 on a tetrahedral mesh of Ω:

$$H_h = \{w_h \in C^0(\Omega) \text{ and piecewise linear on each tetrahedron of the mesh of } \Omega\}. \tag{3.3}$$

Alternatively continuous piecewise linear functions in each variable x_i can be used in conjunction with a covering of Ω by box like quadrangular elements.

Then, the discretized problem can be: find $u_h \in H_h$ such that $u_h = u_\Gamma$ at the vertices of Γ and such that:

$$\int_\Omega (\nabla u_h \cdot \nabla w_h - f w_h) = 0 \quad \forall w_h \in H_h \cap H_0^1(\Omega). \tag{3.4}$$

Dirichlet boundary conditions can be implemented by *elimination* or by *penalty*; in this later case, the discrete problem is: find $u_h \in H_h$ such that

$$\int_\Omega (\nabla u_h \cdot \nabla w_h - f w_h) + \frac{1}{\epsilon} \sum_{\{i:q^i \text{ vertex of } \Gamma\}} (u_h(q^i) - u_\Gamma(q^i)) w_h(q^i) = 0 \quad \forall w_h \in H_h, \tag{3.5}$$

where ϵ is a small parameter and Γ denotes the boundary of Ω.

3.2 Embedding of a Fictitious Domain Method

3.2.1 Dirichlet Conditions

So let C such that $\Omega \subset C$ and $\partial\Omega \cap \partial C = \emptyset$. Let $\tilde{u} \in H_0^1(C)$ such that $\tilde{u} = u$ in $H^1(\Omega)$, and let $\tilde{\nu} \in L^\infty(C)$ such that $\tilde{\nu} = \nu$ in $L^\infty(\Omega)$, and $\tilde{\nu}$ is strictly positive in $C \setminus \Omega$.

To simplify notations let us now call $\tilde{u}$: u and $\tilde{\nu}$: ν.

The Embedding of a Fictitious Domain Method (EFDM) tries to avoid the difficulty of dividing Ω into non-overlapping tetraedra and so extends all functions in a simpler domain C containing Ω.

Let us denote by

$$V_h = \{w_h \text{ continuous, piecewise linear on the tetrahedral mesh of } C\} \tag{3.6}$$

and solve for $u_h, v_h \in V_h$:

$$\int_C \mathbf{1}_\Omega (\nu \nabla u_h \cdot \nabla w_h - f w_h) + \frac{1}{\epsilon} \int_\Gamma (u_h - u_{\Gamma h}) w_h = 0 \quad \forall w_h \in V_h. \tag{3.7}$$

This is essentially the same as (3.5) except that the triangulation is not conform with Γ and so the sum on vertices has been replaced by an integral. But one should note that (3.7) does *not* have a *unique* solution in V_h because the bilinear form is not strongly elliptic on C. Hence one cannot use *direct method* to solve the associated *linear system* but since we use a *Conjugate Gradient*-like method the solution of the discrete problem will converge to the solution of the continuous problem **in Ω**.

The program given in 2.2 uses this approach.

Boundary Penalty

One notes that penalty amounts to trade the Dirichlet boundary condition for the Fourier condition:

$$\frac{u - u_\Gamma}{\epsilon} + \left[\nu \frac{\partial u}{\partial n}\right] = 0 \tag{3.8}$$

where $[\cdot]$ stands for the jump across Γ.

Indeed a Robin jump condition such as in

$$\left| \begin{array}{l} \nabla \cdot (\nu \nabla u) = f \text{ in } C \setminus \Gamma \\ \beta u + \left[\nu \dfrac{\partial u}{\partial n}\right] = g \text{ on } \Gamma, \end{array} \right. \tag{3.9}$$

has the variational formulation: find $u \in H_0^1(C)$

$$\int_C (\nu \nabla u \cdot \nabla \hat{u} - f\hat{u}) + \int_\Gamma (\beta u - g)\hat{u} = 0 \quad \forall \hat{u} \in H_0^1(C). \tag{3.10}$$

Domain Penalty

A much simpler method is to extend u_Γ into $\tilde{u}_\Gamma$ in $C - \Omega$ and to solve

$$\int_C (\nu \nabla u \cdot \nabla \hat{u} - f\hat{u}) + \frac{1}{\epsilon} \int_{C-\Omega} (u - \tilde{u}_\Gamma)\hat{u} = 0 \quad \forall \hat{u} \in H_0^1(C). \tag{3.11}$$

However in theory the precision is less than with the first method.

3.2.2 Implementation by hand

In FreeFEM3D you may either do the EFDM by yourself or let it be done automatically by specifying the domain (see section **??**). There are others approaches to EFDM [**?**][**?**][**?**].

Should you decide to do it yourself then to solve (3.7) you would use a program like

penal1.ff

```
solve(u) in O by M
{
 test(w)
   int(grad(u)*grad(w))+int[<1,0,0>](10000*u*w)=int(w);
   u=0 on M;
}
save(medit,"penal",M);
save(medit,"penal",u,M);
save(raw,"u.dat",u,M);
```

It uses a weak formulation of the PDE [1].

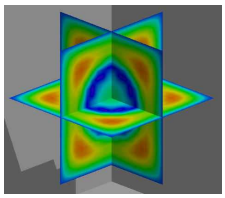
To solve (3.11) one would write

penal2.ff

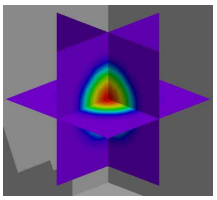
```
vector n = (20,20,20);
vector a = (-1,-1,-1);
vector b = (1,1,1);

scene S = pov("sphere.pov");
mesh M = structured(n,a,b);
domain O = domain(S,inside(<0,1,0>)&& inside(<1,0,0>));

function Id = 100*(1-one(<1,0,0>));

solve(u) in O by M{
 test(w)
```

[1] By taking ν small but positive in $C - \Omega$, the jump of $\nu \frac{\partial u}{\partial n}$ is approximately the part inside Ω so on could have implemented the same using the PDE in strong form with the approximate condition $\frac{u}{\epsilon} + \nu \frac{\partial u}{\partial n} = 0$

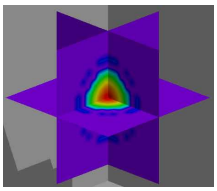
```
13      int(grad(u)*grad(w)+Id*u*w)=int(w);
14  }
15  save(medit,"penal",M);
16  save(medit,"penal",u,M);
17  save(raw,"u.dat",u,M);
```



(a) u on a 3 cutting plane computed by penal1.ff



(b) u on a 3 cutting plane computed by penal2.ff



(c) u on a 3 cutting plane computed by penal3.ff

Figure 3.1: Solution of (3.1) with Dirichlet conditions by boundary penalty (left) and domain penalty (center) and `FreeFEM3D`built-in penalty (right). The color scales are different for each picture.

Remark 2 *The file* ***`penal.pov`*** *contains*

```
sphere{
    <0,0,0>,0.5
    pigment {color rgb <1,0,0>}
} sphere{
    <0,0,0>,0.9
    pigment {color rgb <0,1,0>}
}
```

The second sphere is here only to comply with ***`FreeFEM3D`****syntax for fictitious domains.*

What `FreeFEM3D`does

Notice that the implementations above require the weak forms of the PDEs and a good knowledge of the fictitious domain methods. `FreeFEM3D`simplifies the formulation by using strong forms and it implements the first method (boundary penalty) when the following is invoked to solve (3.1):

penal3.ff

```
vector n = (20,20,20);
vector a = (-1,-1,-1);
vector b = (1,1,1);

scene S = pov("sphere.pov");
mesh M  = structured(n,a,b);
domain O = domain(S,inside(<1,0,0>));

function nu=1;

solve(u) in O by M
  method(type=penalty)
{
  pde(u)
   -div(nu*grad(u))=1;
   u = 0 on <1,0,0>;
}
save(medit,"penal",M);
save(medit,"penal",u,M);
save(raw,"u.dat",u,M);
```

3.3 Neumann Boundary Conditions

For Neumann problem the fictitious domain method is much more natural. Consider

$$\left| \begin{array}{l} u - \nabla \cdot \nu \nabla u = f \text{ in } \Omega \\ \nu \frac{\partial u}{\partial n}|_\Gamma = g. \end{array} \right. \tag{3.12}$$

The variational formulation is: find $u \in H^1(\Omega)$ such that

$$\int_\Omega (uw + \nu \nabla u \cdot \nabla w) = \int_\Omega fw + \int_\Gamma gw \quad \forall w \in H^1(\Omega). \tag{3.13}$$

Any $w \in H^1(\Omega)$ can be extended into $\tilde{w} \in H^1(C)$, so (3.13) can be written: find $\tilde{u} \in H^1(C)$

$$\int_C (I_\Omega(\tilde{u}w + \nu \nabla \tilde{u} \cdot \nabla w - fw) = \int_\Gamma gw \quad \forall w \in H^1(C). \tag{3.14}$$

The numerical approximation is by the finite element method on a triangulation of C with, say the space V_h of continuous piecewise linear functions: find $u_h \in V_h$ solution of

$$\int_C (\alpha u_h w_h + \mu \nabla u_h \cdot \nabla w_h - f w_h) = \int_\Gamma g w_h \quad \forall w_h \in V_h. \tag{3.15}$$

where $\alpha = 1, \mu = \nu$ in Ω and very small outside and f is extended by zero outside Ω.

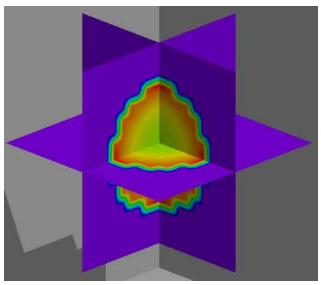
It is implemented automatically in `FreeFEM3D`by writing (see `penal4.ff`)

```
dnu(u) = g on(<1,0,0>)
```

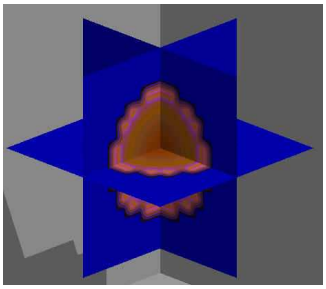
but is can also be done manually as in

penal5.ff

```
vector n = (20,20,20);
vector a = (-1,-1,-1);
vector b = (1,1,1);

scene S = pov("sphere.pov");
mesh M = structured(n,a,b);
domain O = domain(S,inside(<0,1,0>)&& inside(<1,0,0>));

function Id = one(<1,0,0>);
function nu=Id+0.01;

solve(u) in O by M
{
 test(w)
    int(Id*u*w+nu*grad(u)*grad(w))=int[<1,0,0>](w);
}
save(medit,"penal",M);
save(medit,"penal",u,M);
```



(a) u on a 3 cutting plane computed by penal1.ff



(b) u on a 3 cutting plane computed by penal2.ff

Figure 3.2: Solution of (**??**) with Neumann boundary conditions by the fictitious domain method (left) automatically implemented by `FreeFEM3D`and by the same but implemented by hand (right). The color scales are different in each picture.

Chapter 4

Solving problems with FreeFEM3D

4.1 Geometry definition

A PDE problem is defined by a set of functional equalities betwen functions and their derivatives in a computational domain and its boundary. As described in section 3, FEM uses a tetrahedrisation of the domain. To cover the domain by tetrahedra (or boxe like quadrangular elements) is a complex task that we avoid by using the *Embedded Fictitious Domain Method* (EFDM).

In some PDE problems the domain is infinite, so embedding such domain is not the same as enclosing a compact set into a bigger cube. Thus for a given set D one has to distinguish between *external* problems where the PDEs are solved in $R^3 - D$ and and *internal* problems which are set in D itself. In both cases D is plunged inside a cube (or a box) C but in the first case the points of interests are those of $C - D$ which in the second case it is those in D. Two keywords will deal with this fundamental difference:

`domain O = outside(D)` or `= inside(D)`

EFDM will require to define functions which take different values inside and outside objects and also to compute boundary integrals on the objects.

In FreeFEM3D, the geometry of the domain is given using *Virtual Reality* (VR) data. It means that the domain is defined as *set operations*[1] on simple *primitive* shapes[2]. For various reasons[3], the language chosen to describe VR is POV-Ray's.

Image synthesis softwares such as POV-Ray define *scenes* as a collection of simple objects with set operations on them. But they also worry about realistic rendering and so a number of features are irrelevant for us, such as the camera, the type of light, the textures[4]. What we need is simply to define a scene as a collection of objects, know what are the set operations that have been applied to them and name each object. Therefore, for a scene made of a sphere and a brick, the following is sufficient:

```
sphere {
   <0,0,0>, 1.5
   pigment { color rgb <0,1,0> }
}
```

[1] set operations are union, extrusion and intersection.

[2] box, sphere, cylinder, cone,...

[3] Mostly because it is a freeware

[4] One should note that those keywords are or will be **ignored** by FreeFEM3D compiler, so you don't have to modify the scene you rendered with POV-Ray. However during the development phase we suggest that POV lines not used by FreeFEM3D be started by a /**/ which will tell FreeFEM3D to ignore the whole line

```
box {
   <0,0,0>,
   <2,2,2>
   pigment { color rgb <0,0,1> }
}
```

In addition it may be helpful to visualize the fictitious computational domain. So the scene may also contain (but only for the visualization) the computational box, for example

```
box {
   <-2,-2,-2>,
   < 3, 3, 3>
   pigment { color rgbf <1,1,1,0.5> }
}
```

The color which is assigned to each object will be used by ***FreeFEM3D*** *to identify the object and its boundary condition*, so it is important to distinguish colors if boundary conditions are different.

Colors like `rgbf <a,b,c,d>` define objects which will not be identified by `FreeFEM3D`, and so are used only for graphics.

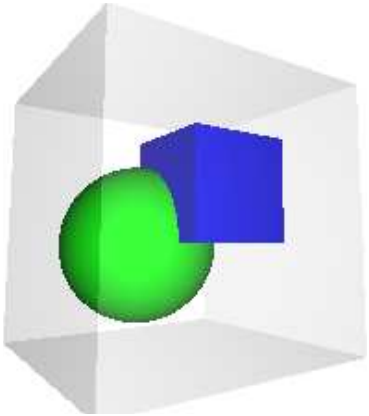


Figure 4.1: A green sphere and a blue cube. The transparent box is for graphic use only (to see the computational domain used) but `FreeFEM3D` will check that all objects are contained in it.

The user is sent to the `POV-Ray` language manual for a complete reference, but here are some *conventions* used in `FreeFEM3D` and some basis to the `POV-Ray` language.

4.1.1 `POV-Ray` Language Conventions for `FreeFEM3D`

References: For *PDE-solvers* it is common practice to define physical constants and boundary conditions using *references* (indices that characterize *degrees of freedom*).

Since the geometry description is not contained in the mesh, but comes from a `POV-Ray` file, `FreeFEM3D` uses *objects colors* as references. So the reference is not given by an *integer* but by an $\mathbb{R}^3$ vector such as:

```
<a,b,c>
```

where `a`, `b` and `c` are three *double*. The colors in `POV-Ray` being given by values of red, green and blue. So to visualize the scene before computation one should choose pleasing values `a`, `b` and `c` in $[0, 1]$ but as far as `FreeFEM3D` goes these values are unimportant since they are used only in comparison tests.

4.1.2 `POV-Ray` language basics

The following subset of the `POV-Ray`language is imlemented:

sphere:

```
sphere {
   <0,0,0>, 1
   pigment { color rgb <1,0,0> }
}
```

defines a *unit* sphere centered at $(0, 0, 0)$. The pigment is defined so that the sphere will be referenced as `<1,0,0>`.

box:

```
box {
   <0,0,0>, <1,1,1>
   translate <1,-1,1>
   pigment { color rgb <1,0,1> }
}
```

describes a box built on vertices $(0, 0, 0)$ and $(1, 1, 1)$ and *then* translate by a vector $(1, -1, 1)$. It is referenced has `<1,0,1>`.

cylinder:

```
cylinder {
   <0,0,0>, <1,1,1>, 0.2
   pigment { color rgb <1,0,1> }
}
```

is a cylinder built on the axis defined by $(0, 0, 0)$ and $(1, 1, 1)$ whose radius is 0.2. Its reference is `<1,0,1>`.

4.2 Boolean operations

The main interest of CSG is the ability of combining all primitives and built objects. `POV-Ray`'s way of doing it is given bellow.

object:

```
object {
   box {
     <0,0,0>, <1,1,1>
   }
   pigment { color rgb <0,0,1> }
}
```

Defines a box and *then* using the `object` statement the reference $(0, 0, 1)$ is assigned to it.

union:

```
union {
   box {
     <0,0,0>, <1,1,1>
   }
   box {
     <0,0,0>, <1,1,1>
     translate <1,-1,1>
   }
   pigment { color rgb <1,0,1> }
}
```

defines the union of two boxes. The second is translated by a vector $(1, -1, 1)$. The obtained object has the reference $(1, 0, 1)$. The union can be operated on n objects.

Remark 3 *One can substitute the keyword* `merge` *to* `union`*. There is strictly* no difference *in FreeFEM3D as opposed to POV-Ray.*

intersection:

```
intersection {
   cylinder {
     <0,0,0>, <1,1,1>, 1
   }
   box {
     <0,0,0>, <1,1,1>
   }
   pigment { color rgb <1,0,1> }
}
```

builds the intersection of a cylinder and a box. The obtain object has the reference $(1, 0, 1)$. The intersection can be operated on n objects.

difference:

```
difference {
   sphere {
     <0,0,0>, 1
   }
   box {
     <0,0,0>, <1,1,1>
   }
   pigment { color rgb <1,0,1> }
}
```

removes a box from a sphere. The obtain object has the reference $(1, 0, 1)$. The `difference` can be operated on n objects, then, the $n - 1$ objects are extruded from the first one.

4.3 Language Basics

`FreeFEM3D` derives its syntax from *C++* in a reasonable way! So, people familiar to *C* or *C++* should learn the language easily (the biggest difficulty being that one does not know which subset of *C++*is implemented). At the same time, someone unfamiliar with those languages should not be afraid since it is a *hi-level*[5] language.

Knowing that we will explain step-by-step this language, let us first recall some important rules:

- Every variable must be declared before being used.
- One cannot declare a variable twice.
- Each variable is global, so when declared, it *lives* until the end of the program execution, even if declared *in a block*. This behaviour should change in the future.

4.3.1 Simple types

In the following we describe the two basic types of variables that can be declared in `FreeFEM3D`: `double` and `vector/vertex`($\in \mathbb{R}^3$). The language supports other types such as *boolean* or *string* but they are only used internally (up to now).

Variable declaration follows the general syntax

```
   <typeid> <variableid>;
```

which *declares* a *non initialized variable*, or

```
   <typeid> <variableid> = <typevalue>;
```

which *constructs* a variable initialized to the given value.

[5] *high-level* in the sense that it manipulates "real-life" objects.

double: doubles are used to represent $\mathbb{R}$ elements. To declare a double, use the following syntax:

```
double a = 3.14159;
```

The *algebra* on $\mathbb{R}$ has been implemented, and classical function are built in the language, so one can write

```
1 double pi = 4*atan(1); double b; b = sqrt(1+pi^2*(2+sqrt(2)));
```

In this case, b will contain, as expected, the value $1 + \pi^2(2 + \sqrt{2})$ at line 3, before (line 2) its content is not determined.

FreeFEM3D syntax	mathematical meaning
abs(a)	$\|a\|$
exp(a)	e^a
log(a)	$\log(a)$
sin(a)	$\sin(a)$
cos(a)	$\cos(a)$
tan(a)	$\tan(a)$
asin(a)	$\arcsin(a)$
acos(a)	$\arccos(a)$
atan(a)	$\arctan(a)$
a^b	a^b

Table 4.1: Built-in functions on doubles. a and b are double.

The table 4.1 shows FreeFEM3D functions that can be used on double.

vector/vertex: To declare a vector one can use one of the two following syntax:

```
vector v1 = (1,0,0); vertex a = (0,1,0);
```

By now, there is no difference between vertex and vector but the vertex object could be enriched for future use.

Also, in a sense of POV-Ray compatibility, one can use indifferently the notations (a,b,c) or <a,b,c>. We suggest the use of <a,b,c> only when the vector refers to the color of a POV-Ray object.

Since $\mathbb{R}^3$ algebra is implemented, one can write the following expression:

```
vector w = 2*(1,1,1)+(-2*2.5,1,2);
```

then w will contain $(-3, 3, 4)$.

4.3.2 Complex types.

By complex types we do not mean imaginary types nor "composed" types such as structures or classes but types whose behavior needs some enlightenment.

function: Scalar functions algebra is implemented in `FreeFEM3D`. It means that the user can manipulate scalar functions in an analytic way, *i.e.* manipulating the functions and not their approximations.

There are 3 special functions/keywords: `x`, `y`, `z` which are $\mathbb{R}^3 \to \mathbb{R}$ functions which refer to the coordinate system:

$$(x,y,z) \longmapsto x, \quad (x,y,z) \longmapsto y, \quad (x,y,z) \longmapsto z. \tag{4.1}$$

One can now easily define polynomial functions:

```
function p = 2*x + y + x*z + z^3;
```

with the obvious meaning $p(x,y,z) = 2x + y + xy + z^3$.

The built-in `functions` in `FreeFEM3D` are listed in table 4.2. For example, one could redefine the $(x,y,z) \longmapsto tan(x)$ function using the name tag `tan_x` by:

```
function tan_x = sin(x)/cos(x);
```

It is possible define functions with *constant* values by assigning a `double` to the function, such as in

```
function cos1 = cos(1);
```

However computing speed could be slower compared to a

```
double cos1 = cos(1);
```

`FreeFEM3D` syntax	mathematical meaning
`abs(f)`	$\lvert f \rvert$
`exp(f)`	e^f
`log(f)`	$\log(f)$
`sin(f)`	$\sin(f)$
`cos(f)`	$\cos(f)$
`tan(f)`	$\tan(f)$
`asin(f)`	$\arcsin(f)$
`acos(f)`	$\arccos(f)$
`atan(f)`	$\arctan(f)$
`f^g`	f^g

Table 4.2: Built-in functions on `functions`. f and g are `functions`.

femfunction: This is a special type related to finite element functions. Up to now, only Q^1 functions are allowed: piecewise *tri-linear*[6] and continuous functions defined on an hexahedral mesh. Since finite element functions spaces needs a *mesh* to be defined, one has to write

[6] tri-affine for purists!

```
femfunction f(M) = sin(x);
```

where M is a previously defined mesh (see 4.3). After the instruction, f contains a finite element approximation of $(x_1, x_2, x_3) \longmapsto \sin(x_1)$. f is in fact defined by the interpolation at mesh vertices: $f(X) = \sin(X_1)$ if X is a vertex of M.

Even though a femfunction is a special kind of function, it can be used in the function algebra:

```
femfunction f(M) = exp(x+y*z); function g = x - f;
```

Again, this is a *definition*; g is not evaluated a this point. The evaluation will be performed when needed as in

```
double t = g(1,1,1);
```

which not only defines t but also triggers an evaluation of g at (1,1,1), and therefore the computation of the linear interpolation to calculate $f(1, 1, 1)$.

4.4 Instructions for the Geometry

4.4.1 scene:

VR data comes from a POV-Ray file. The geometry informations contained in this file are stored in a scene variable.

The syntax is very simple:

```
scene S = pov("scene.pov");
```

Then S contains the POV-Ray *scene* described in the file scene.pov.

One can define many *scenes* in one FreeFEM3D file. In that latter case, to avoid ambiguities, the last *used* scene is the current one. Lets look at the following example:

```
1 scene S1 = pov("scene1.pov"); scene S2 = pov("scene2.pov"); function
2 f = one(<1,0,0>); using S1; f = one(<1,0,0>);
```

After line 2, two scenes S1 and S2 are defined and *current* scene *is* S2. So at line 3, the function f, is the *indicator* function of the object <1,0,0> *of scene S2*.

Line 4, the *current* scene is set to S1! Line 5 makes the function f be the *indicator* function of the object <1,0,0> *of scene S1*.

4.4.2 domain:

How to define a computational domain?

The use of the domain keyword is always associated to Fictitious Domain-like resolutions!

> **Defining the *computational domain* is one of the most critical part of a FreeFEM3D simulation. A clever definition of the domain may simplify the syntax and make FreeFEM3D run faster.**
> **This part is very important and needs a very good understanding in order to create your own simulations.**

Several `domains` can be defined in a single `FreeFEM3D` file. `Domains` can be used to different purpose:

- they simplify the writing of the PDE. Only one domain Ω can be associated to a `solve` statement. For instance solving

$$-\Delta u = f \text{ in } \Omega \text{ with } u = 0 \text{ on } \partial\Omega$$

with a background mesh M will involve

```
solve(u) in Omega by M
```

Alternatively the problem could also be set as

$$\int_M \mathbf{1}_\Omega \nabla u \cdot \nabla v + \frac{1}{\epsilon}\int_M (1-\mathbf{1}_\Omega)uv = \int_M \mathbf{1}_\Omega f v \qquad \forall v \in H^1(M).$$

- The domain is also necessary to define `POV-Ray` boundary conditions: the only `POV-Ray` borders that can be used are those explicitly used in the keyword `domain`.
- Finally a domain can be used to define characteristic functions

$$\mathbf{1}_\Omega(x) = \left| \begin{array}{l} 1 \text{ if } x \in \Omega, \\ 0 \text{ in the other case,} \end{array} \right.$$

using the simple syntax `one(Omega)`. This can be useful for instance to define different materials using a `POV-Ray` geometry description.

Here comes now the probably most important rule to learn: use the background mesh **as much as possible** to describe your geometry.

1. This means for instance that to solve external problems you do not need to specify an outer box in the `POV-Ray` file.
2. This has also the advantage of improving the approximation at those borders since the standard finite element method is used there.

The computational *domain* is defined using a scene,

```
domain <domain> [ = domain(<scene> , <booleanexp>) ];
```

where the `booleanexp` is defined by the following:

```
<booleanexp>: {not <inoutexp> | <inoutexp>}
```

and finally

```
<inoutexp>: { ( <inoutexp> )
            | <inoutexp> and <inoutexp>
            | <inoutexp> or <inoutexp>
            | outside(<ref>)
            | inside(<ref>)}
```

Remark 4 *The keywords* `not`, `and` *and* `or` *can respectively be replaced by* `!`, `&&` *and* `||`, *as it is the case for every boolean operation.*

4.4.3 Meshes

The *embedding Fictitious Domain* is a box defined by two points (i.e. vectors) which are at both end of the main diagonal of the box, and a vector with integer coordinates to tell how many intervals there will be on each edge. For example

```
vector n = (20,30,40);
vector a = (0,0,0);
vector b = (1,2,3);
mesh M = structured(n,a,b);
```

puts 20 intervals on the x-edge of the box $(0,1) \times (0,2) \times (0,3)$ and 30 intervals on the y-edge and 40 on the z-edge.

There are two kinds of meshes in `FreeFEM3D`: volume and surface. The volume meshes are only *cartesian structured* meshes, *i.e.* a squared box regularly meshed by squared hexahedra. Surface meshes are composed of triangles or quadrangles living in $\mathbb{R}^3$, they are only used to compute integrals in the fictitious domain method.

To construct a *surface* mesh, one needs a *structured* mesh and can write

```
   mesh m = surface(<1,0,0>, S, M);
```

where `<1,0,0>` is the *reference* of a `POV-Ray` object, given by the scene `S` and `M` a *structured* mesh; all of them having previously been declared.

Remark 5 *The surface mesh is built using a* marching cubes*-like method, this is why a structured mesh is needed. In future, reading meshes in files should be allowed, also.*

4.5 Problem definition

4.5.1 The Solver Bloc

solve: The *`solve` block* is of course special to `FreeFEM3D` and should be studied with attention:

```
solve (<unknown list>) in <domain> by <mesh> [<solver options>] {
   { pde (<unknown1>)
       <pde>
       <boundary condition list>
     [ pde(< unknown2>)
       <pde>
       <boundary condition list>
       [...]
     ]
   | test(<test function list>)
     <variational formula>
     <dirichlet boundary condition list> }
}
```

Remark 6 *Note that the* `solve` bloc *can be used with two different kinds of body. The first one is* systems of PDEs*, the second is* variational formulae*.*

Lets focus on the common part and then on the `<unknown list>`, it is a set of `unknown` of the form:

```
u,v,w
```

Those `unknowns` can already have been defined as `functions`! If it is the case, this `function` will be the first guess for iterative methods, otherwise it will be 0. An instruction like

```
in <domain> by <mesh>
```

gives informations to the solver has to know which `<domain>` and which `<mesh>` use.

Remark 7 *If the* ***unknown*** *list contains n elements, n* ***pde*** *statements have to be defined (one per* ***unknown****)*

Remark 8 *The* ***unknowns*** *computed by the* ***solver*** *can be used later as if they were decalred as* ***femfunctions*** *of the solving mesh (whatever they were before).*

Remark 9 *If an* ***unknown*** *was previously defined as a function its interpolate will be taken as the initial guess for iterative methods.*

Solver options The `<solver options>` is used to pass arguments to the solver such as the type of algorithm to solve the linear system (conjugate gradient, bi-conjugate gradient) and associated parameters (type of preconditioner, maximum number of iterations, value of ϵ[7]), discretization method,...

Below is a list of solver options[8]

The syntax of options is

```
[ <option name> ( <suboption> = <value> [,
                  <suboption> = <value> ]...) [,
...]]
```

Here comes the complete list of options.

`method` *use to tune the discretization method*

- selectable parameters
  - `type` *selects the discretization method.* Default value is `penalty`. Available values are
    - `penalty`: *sets Dirichlet boundary conditions to be computed by penalty*
    - `eliminate`: *sets Dirichlet boundary conditions using elimination*
    - `fatBoundary`: *sets boundary conditions using FBM (***not implemented)**

`penalty` *sets penalty parameters*

[7] A condition is often used by iterative methods to stop the process. Solving linear system this condition is often of the form $|Au_n - b| < \epsilon|Au_0 - b|$.

[8] the list is extracted from the code automatically to make it up-to-date. The draw back of this is that reading it may be boring, it is to be considered as a reference.

- double parameters
  - ◇ `epsilon` ϵ*'s value. (*ϵ *coming from* $\frac{1}{\epsilon}\int_\Gamma (u-g)v$*).* Default value is 1E-3

`memory` *sets memory management options*

- selectable parameters
  - ◇ `matrix` *sets matrix type.* Default value is `sparse`. Available values are
    - ○ `sparse`: *used for sparse matrices, cost is approximatly* $27 \times n_v \times {n_u}^2$*, where* n_v *is the number of vertices and* n_u *the number of unknown (for a* Q_1 *discretization)*
    - ○ `none`: *do not store the matrix. Cost no memory, but is slower*

`krylov` *used to modify krylov solver*

- selectable parameters
  - ◇ `type` *is used to select the type of solver.* Default value is `cg`. Available values are
    - ○ `cg`: *selects the conjugate gradient*
    - ○ `bicg`: *selects the bi-conjugate gradient (for non symetric problems)*
    - ○ `bicgstab`: *selects the bi-conjugate gradient stabilized (for non symetric problems)*
    - ○ `ilufact`: *selects the iterative LU factorization*
  - ◇ `precond` *is used to select the preconditioner.* Default value is `diagonal`. Available values are
    - ○ `diagonal`: *preconditions with the diagonal of the operator*
    - ○ `ichol`: *incomplete choleski factorization*
    - ○ `multigrid`: *multigrid finite difference solver. By now, the grid must be* $(2^{n_x}+1) \times (2^{n_y}+1) \times (2^{n_z}+1)$*.*
    - ○ `none`: *no preconditioning*

`fatBoundary` *options for fat boundary method. Not implemented yet!*

`eliminate` *Options for elimination method (by now none)*

`multigrid` *to change multigrid options (****Not working anymore!****)*

- integer parameters
  - ◇ `maxiter` *maximum number of iterations.* Default value is 1
  - ◇ `level` *grid level.* Default value is 3
  - ◇ `nu1` ν_1*.* Default value is 2
  - ◇ `nu2` ν_2*.* Default value is 2
  - ◇ `mu1` μ_1*.* Default value is 1
  - ◇ `mu2` μ_2*.* Default value is 1
- double parameters
  - ◇ `epsilon` *the factor of reduction of the residu.* Default value is 1E-4
  - ◇ `omega` ω*, the relaxation parameter for Jacobi solver.* Default value is 2./3.

`cg` *specifies conjugate gradient options*

- integer parameters
  - ◇ `maxiter` *sets maximum number of iteration.* Default value is 500
- double parameters
  - ◇ `epsilon` *sets the required reduction factor of the residu.* Default value is 1E-5

`bicgstab` *to change bi-conjugate gradient stabilized options*

- integer parameters
  - ◇ `maxiter` *the maximum number of iterations.* Default value is 500
- double parameters
  - ◇ `epsilon` *the factor of reduction of the residu.* Default value is 1E-5

`bicg` *to change bi-conjugate gradient options*

- integer parameters
  - ◇ `maxiter` *the maximum number of iterations.* Default value is 500
- double parameters
  - ◇ `epsilon` *the factor of reduction of the residu.* Default value is 1E-5

`FreeFEM3D` treats the options the following way: the parser reads the option set and builds a tree associated to it. Then when `solve` starts, each *parametrizable* object reads its options when it is built. It looks first in the tree and if an option is not specified here, it uses the default value.

Knowing the rules, lets look at some examples.

```
solve(u) in Omega by M
  krylov(precond=diagonal)
{
```

Parsing this leads to the modification of a Krylov solver option: it changes the preconditioner from the default `none` to `diagonal`. This behaves as expected. Lets now look to a more confusing case:

```
solve(u) in Omega by M
  bicg(epsilon=1E-10, maxiter=1000)
{
```

Parsing will modify *bi-conjugate gradient*'s options, `epsilon` will be 10^{-10} and the *maximum number of iterations* (`maxiter`) 1000.

There is no mistake here, but this option will have *no effect* during execution! The reason is simple. The *Krylov solver*'s options are not modified and that default linear system solver is *conjugate gradient* (`cg`). So, *bi-conjugate gradient* will never start and will have no opportunity to read its options.

To have those options used, one has to specify the use of `bicg`:

```
  solve(u) in Omega by M
    bicg(epsilon=1E-10, maxiter=1000),
    krylov(type=bicg)
  {
```

To conclude with options, consider the code

```
1  solve(u) in Omega by M
2    cg(epsilon=1E-10),
3    cg(epsilon=1E-3,maxiter=15),
4    cg(maxiter=200),
5  {
```

First the `epsilon` parameter of `cg` is set to 10^{-10}, at line 2. Then it is change to 10^{-3} and `maxiter` becomes 15. Finally, at line 4, `maxiter` is modified to become 200. So *conjugate gradient* will run with $\epsilon = 10^{-3}$ and `maxiter`=200.

Between the two brackets (`{` and `}`) comes the problem description. We will first focus one *PDE system*-like descriptions and then on *Variational formula*-like descriptions.

PDE and System of PDEs. The general principle is that equations and boundary conditions are defined variable per variable, even in the case of a coupled system. Each new PDE is announced using the `pde(<unknownid>)` structure.

Remark 10 *It helps to rememeber that FreeFEM3D uses this information to construct a variational formulation* [9]*. If the informations are not given at the right place, the reconstructed variational formulation may not be the right one, and yet no error message will appear. Moreover, one may need to repeat informations when defining PDE systems, this will be clear in the second example below.*

Here is a simple example for illustration

```
solve (u) in O by M {
   pde (u)
     -div(grad(u)) = 1;
     u = 1 on M;
}
```

it stands for solving:

$$\left|\begin{array}{l} -\Delta u = 1 \text{ in } \Omega, \\ u = 1 \text{ on } \partial\Omega \cap \partial M, \\ \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega \setminus \partial M. \end{array}\right.$$

FreeFEM3D and the mathematical syntax are quite close, but that the condition $\frac{\partial u}{\partial n} = 0$ on $\partial\Omega \setminus \partial M$ is *implicit* in FreeFEM3D because the user has forgotten to specify what boundary

[9] This variational formulation is needed by the *finite element* discretization process.

condition to apply on that boundary. For more details on boundary conditions see section 4.5.2, and 4.5.3. Uniqueness of the solution assumes that the measure of $\partial\Omega \cap \partial M$ is positive.

Now look a second example which focuses on the a system description and the *underlying variational problem* as suggested in remark 10.

Lets solve the following problem

$$\left|\begin{array}{l} -\Delta u - \frac{1}{2}\Delta v = f_1 \text{ in } \Omega, \\ -\frac{1}{2}\Delta u - \Delta v = f_2 \text{ in } \Omega, \\ u = u_0 \text{ on } \Gamma_1, \\ \frac{\partial u}{\partial n} = u_1 \text{ on } \Gamma_2, \\ v = v_0 \text{ on } \Gamma_1, \\ \frac{\partial v}{\partial n} = v_1 \text{ on } \Gamma_2, \end{array}\right. \tag{4.2}$$

where Ω is the cube $]0,1[^3$ (for seek of simplicity), $\overline{\Gamma_1 \cup \Gamma_2} = \partial\Omega$, $\Gamma_1 \cap \Gamma_2 = \emptyset$, Γ_2 being the face $x = 0$, and $f_1, f_2, u_0, v_0, u_1, v_1$ given functions such that the problem (4.2) is well posed.

Here comes the associated code in `FreeFEM3D`:

example3.ff

```
23 solve(u,v) in Omega by M
24 {
25   pde(u)
26   -div(grad(u))-div(0.5*grad(v))=f1;
27   dnu(u) = u1 on M xmin;
28   dnu(v) = 0.5*v1 on M xmin;
29   u=u0 on M xmax;
30   u=u0 on M ymin;
31   u=u0 on M ymax;
32   u=u0 on M zmin;
33   u=u0 on M zmax;
34
35   pde(v)
36   -div(0.5*grad(u))-div(grad(v))=f2;
37   dnu(u) = 0.5*u1 on M xmin;
38   dnu(v) = v1 on M xmin;
39   v=v0 on M xmax;
40   v=v0 on M ymin;
41   v=v0 on M ymax;
42   v=v0 on M zmin;
43   v=v0 on M zmax;
44 }
```

The lines 28 and 37 are very important! Even if they are redundant they must be provided by the user since the Green formula is not computed by `FreeFEM3D`, but only a correspondence is made between PDE operators and variational operators.

This comes from the fact that the variational formula associated to (4.2) is

$$\int_\Omega \nabla u\cdot\nabla w_1 - \int_{\Gamma_2}\nabla u\cdot \mathrm{n}\, w_1 + \frac{1}{2}\int_\Omega \nabla v\cdot\nabla w_1 - \boxed{\frac{1}{2}\int_{\Gamma_2}\nabla v\cdot \mathrm{n}\, w_1} + \frac{1}{2}\int_\Omega \nabla u\cdot\nabla w_2 - \boxed{\frac{1}{2}\int_{\Gamma_2}\nabla u\cdot \mathrm{n}\, w_2} + \int_\Omega \nabla v\cdot\nabla w_2 - \int_{\Gamma_2}\nabla v\cdot \mathrm{n}\, w_2 = \int_\Omega f_1 w_1 + \int_\Omega f_2 w_2. \quad (4.3)$$

For given w_1 and w_2.

The boxed terms are the one which should not be forgotten! Using the information coming from (4.2) boundary conditions, one writes:

$$\left|\begin{array}{l} \frac{1}{2}\int_{\Gamma_2}\nabla u\cdot \mathrm{n}\, w_2 = \frac{1}{2}\int_{\Gamma_2} u_1 w_2, \text{ and} \\ \frac{1}{2}\int_{\Gamma_2}\nabla v\cdot \mathrm{n}\, w_1 = \frac{1}{2}\int_{\Gamma_2} v_1 w_1. \end{array}\right.$$

This leads to the final variational formula:

$$\int_\Omega \nabla u\cdot\nabla w_1 + \frac{1}{2}\int_\Omega \nabla v\cdot\nabla w_1 + \frac{1}{2}\int_\Omega \nabla u\cdot\nabla w_2 + \int_\Omega \nabla v\cdot\nabla w_2 = \int_\Omega f_1 w_1 + \int_\Omega f_2 w_2 + \int_{\Gamma_2} u_1 w_1 + \boxed{\frac{1}{2}\int_{\Gamma_2} v_1 w_1} + \boxed{\frac{1}{2}\int_{\Gamma_2} u_1 w_2} + \int_{\Gamma_2} v_1 w_2. \quad (4.4)$$

It would not be an easy task to make `FreeFEM3D` automatically compute (4.4), which means not forgetting the boxed terms. So those terms have to be provided explicitly by the user following the example. Some of the translations are given bellow.

`FreeFEM3D`	leading to	comment
`-div(grad(u))`	$\int_\Omega \nabla u\cdot\nabla w_1$	it is w_1 since we are describing the `pde(u)`
`dnu(u)=u1`	$\int_\Omega u_1 w_1$	goes to right hand side
`dnu(v)=0.5*v1`	$\frac{1}{2}\int_\Omega v_1 w_1$	*it will not be deduced from the* `pde(v)` *bloc!*

One then understands the logic behind it and can look at the table 4.3 for the complete list of domain operators interpretation.

Variational problem description. Entering a problem with a variational formula is quite different from giving its PDE system. First, there is *only* one variational formula (even in the case of systems) and second, only the Dirichlet conditions are given outside that formula, since the Neuman and Robin conditions are included in the variational formula.

For example to solve the problem: find u in $H^1(\Omega)$ such that

$$\left|\begin{array}{l} -\nabla\cdot\mu\nabla u = 0 \text{ in } \Omega, \\ u + \dfrac{\partial u}{\partial n} = g \text{ on } \Gamma_1, \\ u = u_0 \text{ on } \Gamma_2 = \partial\Omega\setminus\Gamma_1. \end{array}\right.$$

one sets the variational problem

$$\int_\Omega \mu \nabla u \cdot \nabla w - \int_{\partial\Omega} \mu \nabla u \cdot n \; w = 0 \qquad \forall w.$$

Using the fact that

$$\int_{\partial\Omega} \mu \nabla u \cdot n \; w = \int_{\partial\Omega} \mu (g - \alpha u) w$$

the variational problem is written, find $u \in H^1(\Omega)$ such that $u = u_0$ on Γ_2 and

$$\int_\Omega \mu \nabla u \cdot \nabla w + \int_{\partial\Omega} \mu \; uw = \int_{\partial\Omega} \mu \; gw \qquad \forall w, \tag{4.5}$$

It is this formula (4.5) that must be entered in the FreeFEM3D code

example4.ff

```
13 solve(u) in Omega by M
14 {
15   test(w)
16   int(mu*grad(u)*grad(w)) + int[M xmin](mu*u*w) = int[M xmin](mu*g*w);
17   u=uexact on M xmax;
18   u=uexact on M ymin;
19   u=uexact on M ymax;
20   u=uexact on M zmin;
21   u=uexact on M zmax;
22 }
```

Note the presence of the `test(w)` statement. It is here to define a (list of) test functions used in the *bilinear* forms. The *test function variables* only live within the solve bloc[10].

Then comes the variational formula. As one can see, it is really close to the mathematics, but still needs some explanation:

`int(mu*grad(u)*grad(w))` defines $\int_\Omega \mu \nabla u \cdot \nabla w$. The integration domain is implicit;

`int[M xmin](mu*u*w)` corresponds to $\int_{\Gamma_1} \mu \, uw$ with Γ_1 being the face $x = 0$ of the domain; and

`int[M xmin](mu*g*w)` which is $\int_{\Gamma_1} \mu \, gw$.

The last lines define the Dirichlet boundary conditions. Note that as in section 4.5.3, only Dirichlet conditions are allowed at this level of the problem description.

This example shows what is a "FreeFEM3D variational formula". Basically, it is an equation made of linear and bilinear terms. A linear form is

$$w \longmapsto l(w)$$

where w *must* be a *test function*. A bilinear form is

$$(u, w) \longmapsto a(u, w)$$

[10] This is an exception but test function names are not really variables, but tags...

where w *must* be a *test function* too and u *must* be an *unknown*. *All others combination are forbidden!*

So the general form of a "FreeFEM3D variational formula" is

$$\sum_j \sum_i a_{ij}(u_i, w_j) = \sum_j l_j(w_j).$$

Where (a_{ij}) is a family of *bilinear forms*, (l_j) is a family of *linear forms*, (u_i) is a family of *unknowns* and (w_j) is a family of *test functions*.

Bilinear and linear forms will be described more precisely in the sections 4.5.4 and 4.5.5.

Lets now reconsider the problem (4.2) and solve it using a variational formula.

The associated variational formula is still given by (4.4). So, the FreeFEM3D code is written immediately by

example5.ff

```
23 solve(u,v) in Omega by M
24  memory(matrix=none)
25 {
26   test(w1,w2)
27   int(grad(u)*grad(w1)) + int(0.5*grad(v)*grad(w1))
28   + int(0.5*grad(u)*grad(w2)) + int(grad(v)*grad(w2))
29   = int(f1*w1) + int(f2*w2)
30     + int[M xmin](v1*w2 + u1*w1)
31     + int[M xmin](0.5*v1*w1 + 0.5*u1*w2);
32   u=u0 on M xmax;
33   u=u0 on M ymin;
34   u=u0 on M ymax;
35   u=u0 on M zmin;
36   u=u0 on M zmax;
37   v=v0 on M xmax;
38   v=v0 on M ymin;
39   v=v0 on M ymax;
40   v=v0 on M zmin;
41   v=v0 on M zmax;
42 }
```

This really looks like (4.4)!

4.5.2 PDE system syntax.

The PDE structure uses the syntax

```
[-] <pdeoperator> [ {+|-} <pdeoperator> ] ... = <function>;
```

Supported PDE operators are shown on the table 4.3.

Some examples:

```
u - div(grad(u)) = f;
```

FreeFEM3D operator	mathematical	bilinear form ($\forall v$)
`mu*u`	μu	$\int \mu uv$
`dx(u)`	$\partial_x u$	$\int \partial_x uv$
`dy(u)`	$\partial_y u$	$\int \partial_y uv$
`mu*dz(u)`	$\mu\partial_z u$	$\int \mu\partial_z uv$
`div(grad(u))`	$\nabla\cdot\nabla u = \Delta u$	$-\int \nabla u\nabla v$
`dx(dx(u))+dy(dy(u))+dz(dz(u))`	$\nabla\cdot\nabla u = \Delta u$	$-\int \partial_x u\partial_x v + \partial_y u\partial_y v + \partial_z u\partial_z v$
`dx(mu*dy(u))`	$\partial_x\mu\partial_y u$	$-\int \mu\partial_y u\partial_x v$

Table 4.3: partial differential operators. `u` is an `unknown` and `mu` a `function` representing μ. The third column shows the bilinear operator that will be used to discretize the partial differential operator, note that in the case of second order operators a Green formula is used. **This means that boundary integral terms are to be supplied by users when needed**, through boundary conditions.

stands for $u - \nabla\cdot\nabla u = f$.

```
-dx(dy(u)) - dy(dx(u)) + dz(u) = f;
```

stands for $-\partial_x\partial_y u - \partial_y\partial_x u + \partial_z u = f$.

4.5.3 Boundary Conditions

To define boundary conditions, one has to use

```
<condition> on <border>;
```

The `<condition>` is defined using the followings:

Dirichlet: $u = g$ is written `u=g`,

Neumann: $\nu\partial_n u = g$ is written `dnu(u)=g`,

Robin (Fourrier): $\alpha u + \nu\partial_n u = g$ is written `alpha*u+dnu(u)=g`.

Remark 11 *`dnu(u)` denotes the co-normal derivative, the term which arises in the variational form when applying the Green formula to the second order operator. In the case of $-\nabla\cdot\mu\nabla u$ the term is $\mu\nabla u$, coming from*

$$-\int_\Omega \nabla\cdot\mu\nabla uv = \int_\Omega \mu\nabla u\cdot\nabla v + \int_{\partial\Omega} \mu\nabla u\cdot\mathbf{n}\, v \qquad \forall v.$$

To refer to borders one uses the following syntax:

- `on <a,b,c>` when the condition is applied on the border of the object having a `POV-Ray` reference `<a,b,c>`.
- `on M` (where `M` is a *structured mesh*) when the condition is to be applied on the border of `M`.

- `on M <modifier>` where `<modifier>` is one of `xmin`, `xmax`, `ymin`, `ymax`, `zmin` or `zmax`, means that it will be applied to the corresponding face on the *structured mesh* `M`.
- `on S` (where `S` is a *surface mesh*) is used to impose a condition on an already built *surface mesh* (read in a file or previously built).

For example:

```
u = 0 on M;
```

$u = 0$ on ∂`M`.

```
dnu(u) = g on <1,0,0>;
```

imposes a Neumann condition on the border of objects `<1,0,0>`, the co-normal derivative of `u` will be equal to `g` in a weak sense. Similarly

```
  u + dnu(u) = g on S;
```

a Robin condition on the surface meshed by `S`.

4.5.4 Bilinear forms.

Bilinear forms implemented in `FreeFEM3D` are of the type

$$a(u,w) = \int_{\mathcal{O}} \mathcal{A}(u,w),$$

where $\mathcal{O}$ is an $\mathbb{R}^3$ domain or an $\mathbb{R}^3$ surface, and $\mathcal{A}$ is such that

$$\mathcal{A}(u,w) = \sum_i g^i \mathcal{D}_u^i(u)\mathcal{D}_w^i(w),$$

with $\mathcal{D}_u^i$ and $\mathcal{D}_w^i$ being two partial differential operators of order 0 or 1; and u, w two functions. To be a "`FreeFEM3D` bilinear form", it is required for u to be an *unknown* and for w to be a *test function*.

The possible choices for operators $\mathcal{D}_u$ and $\mathcal{D}_w$ are given at table 4.4.

`FreeFEM3D` operator	mathematical meaning
`v`	v (order 0 operator)
`dx(v)`	$\partial_x v$
`dy(v)`	$\partial_y v$
`dz(v)`	$\partial_z v$
`grad(v)`	∇v

Table 4.4: partial differential operators. `v` is an *unknown* or a *test function*.

Note that using variational formula, one can discretize $\int \partial_{x_i} uv$ or $\int u\partial_{x_i} v$ while only $\int \partial_{x_i} uv$ is used in PDEs (see table 4.3).

4.5.5 Linear forms.

In the same way, linear forms implemented in FreeFEM3D are defined as

$$l(w) = \int_{\mathcal{O}} \sum_i \mathcal{L}^i(w).$$

Where $\mathcal{O}$ is an $\mathbb{R}^3$ domain or an $\mathbb{R}^3$ surface. $\mathcal{L}$ can be of the following forms:

$$\left| \begin{array}{l} \mathcal{L}(w) = gw, \text{ or} \\ \mathcal{L}(w) = g\nabla f \cdot \nabla w. \end{array} \right.$$

Let us write some examples combining linear and bilinear forms.

```
int[<1,0,0>](2*u*alpha*v)+int(grad(v)*grad(u))-int(v)=0;
```

is associated to $\int_\Omega \nabla u \nabla v + \int_{\Gamma_i} 2\alpha uv = \int_\Omega v$, where Γ_i is the border referenced by <1,0,0>. By the way, solving

```
int(u*v)=int(f*v*g);
```

for all v, makes u the L^2 projection of fg on the finite element space.

4.5.6 Convection Operator

The convection operator

$$\partial_t \varphi + u_i \cdot \partial_{x_i} \varphi, \tag{4.6}$$

can be implemented by using a discrete method of characteristics[?]. To call this operator one uses

```
convect([ux,uy,uz],dt,phi);
```

phi is the transported function at the speed (ux, uy, uz) during the timestep dt.

One has to note that this function is evaluated *when needed*. This means that if one writes

```
1  function f = convect([ux,uy,uz],dt,phi);
```

f *is* "convect([ux,uy,uz],dt,phi)".
the convect operator adapts to the context:

```
1  function f = convect([ux,uy,uz],dt,phi);
2  femfunction g(M) = f;
3  solve (h) in Omega by M
4  {
5    test(v)
6    int(h*v)=int(f*v);
7  }
```

The g function will be a Q_1 function with values at vertices of the mesh M which are the same as those of f, but to compute h, the values of f will be evaluated at the quadrature vertices.

4.6 Other Instructions

4.6.1 Input and output

Built-in. A *built-in* input and output instruction set can be used to read/write files using complex formats. User cannot really change them but can provide options.

Those functions are typically variants of `save` and `read`.

The `save` instruction. The syntax of the `save` instruction is

```
save(<format>, <filename>, <function-or-field-list>, <mesh>[, <filetype>]);
```

With the following options:

- `<format>` is the data storage *format*, one can refer to the table 4.5 for the list of supported formats,

Format	Identifier
Raw	`raw`
MEdit (mesh)	`medit`
`ParaView / MayaVi`	`vtk`

Table 4.5: Supported file formats

- `<filename>` is a *string* containing the name of the file stored on the disk,
- `<function-or-field-list>` is used to refer to a *function* (usually a *function variable*) or a field of functions, or a list of both of them.
- `<mesh>` is the mesh that will be used to proceed saving (automatic interpolation is possible).
- The last parameter `<filetype>` is the second *optional* parameter, it is used to define the file type. Possible file types are given on table 4.6.1. Note that up to now, `binary` is ignored.

Format	Identifier
Unix	`unix`
MS-DOS/Windows	`dos`
Mac OS	`mac`
Binary	`binary`

Table 4.6: File types

Remark 12 (`raw` format) *The `raw` format does not store any information concerning the mesh. The generated file only contain the function values. It is mainly used to post-process data using `OpenDx` on cartesian meshes.*

Remark 13 (`vtk` format) *File extensions are generated automatically according to the type of the mesh, for instance `.vti` (cartesian mesh of hexahedra) or `.vtu` (unstructured grid).* ***The mesh is saved with the function in one file.*** *Up to now, the `vtk` format is the only one that supports lists of functions and fields. The instruction*

```
save(vtk,"foo",{[u,v,w],p,rho},M);
```

saves the field (u, v, w) *and the scalar functions* p *and* ρ *into one file:* `foo.vtu` *(if* `M` *is unstructured) or* `foo.vti` *(if* `M` *is a cartesian mesh).*

Remark 14 (`medit` format) *Again, the extension (*`.bb`*) is added automatically. However, the mesh requires to be stored in another file with the same affix. This is let to the user that use the same syntax omitting the function or field. For instance, to save the field of functions* `u`*,* `v` *and* `w` *one might write*

```
save(medit,"foo",[u,v,w],M); // save the field
save(medit,"foo",M);         // save the mesh
```

This code creates the two files `foo.bb` *and* `foo.mesh`*.*

The `read` instruction. In fact, there is two kind of read instruction:

- to read a mesh (an tetrahedral unstructured grid for instance)
- to read a function defined on a given mesh

The syntax to read a mesh is

```
read(<format>, <filename> [, <filetype>]);
```

- The `<format>` is one of
  - `gmsh` format from `http://www.geuz.org/gmsh/`,
  - `medit` INRIA mesh format,
  - `vtk` the VTK XML format.

An example of use is

```
mesh M = read(gmsh,"mesh.msh");
```

The syntax of the `read` *function* instruction is

```
read(<format>, <filename>[:<function-or-field-name>][:<number>],
     <mesh> [, <filetype>]);
```

- The `<format>` is given in table 4.5,
- `<mesh>` is the mesh where the function is defined,
- `<filename>` is the file name containing the function values,
- if the format support multiple data (VTK for instance) one must precise which field or function is to be read.

- The read instruction only returns scalar functions, so a field is to be read component by component, `<number>` is the component number.

Let `M` be an arbitrary mesh.

```
function f = read(medit,"data.bb",M);
```

reads a function from the file `data.bb`.

```
function u = read(medit,"velocity.bb":0,M);
function v = read(medit,"velocity.bb":1,M);
function w = read(medit,"velocity.bb":2,M);
```

places the three components of a field stored in the file `velocity.bb` into three functions `u`, `v` and `w`.

```
function u = read(medit,"ns.vti":"[u,v,w]":0,M);
function v = read(medit,"ns.vti":"[u,v,w]":1,M);
function w = read(medit,"ns.vti":"[u,v,w]":2,M);
function p = read(medit,"ns.vti":"p",M);
```

reads the field labeled `[u,v,w]` and the function `p` from the same file.

User "defined" This provides more basic stuffs to allow the user to read/write to console or files. The way to use them is similar to *C++* streams so one manipulates low level objects and builds his own format.

The syntax is the following

```
<ostream> [ << <expression> ] ... ;
```

By now, only *output* `streams` are implemented, *input* `streams` will be introduced in the future.

Let us now introduce the `cout` stream. It is a predefined *output* `stream` that behaves like in *C++*. For instance, the following code

```
double i = 10;
cout << "i=";
cout << i << "\n";
cout << "----\n";
```

produces the output:

```
i=10
----
```

It is also possible to manipulate output stream to file (`ofstream`) variables.

```
ofstream <ofstream> [ = ofstream(<filename>) ];
```

Thus, the code

```
ofstream fout;
double i = 0;
do {
  fout  = ofstream("file".i);
  fout << i << "\n";
  i++;
} while(i<10);
```

will create a set of 10 files named `file0`, `file1`,... to `file9` that contain their number.

Remark 15 (`ofstream` synchronization) *`FreeFEM3D`'s `ofstream` is not buffered: this means that it should be used*

4.6.2 Statements

Syntax for statements in `FreeFEM3D` follows the rules of *C* or *C++*.

4.6.3 Conditional statements.

In `FreeFEM3D`, only the `if` statement is implemented. Its syntax is

```
if (<boolexp>) { <instruction>; | <bloc> }
[ else  { <instruction>; | <bloc> } ]
```

4.6.4 Loops.

Standard `do-while`, `while` and `for` structures are implemented:

```
do { <instruction> | <bloc> } while (<boolexp>);
```

```
while (<boolexp>) { <instruction>; | <bloc> }
```

```
for (<instruction>;<boolexp>;<instruction>) { <instruction>; | <bloc> }
```

The following contains a set of examples:

statements.ff

```
1  for (double i=0; i<5; i=i+1)
2    if (i<3)
3      cout << i << " ";
4    else {
5      double j=0;
6      while (j<i) {
7        cout << j-i << " ";
8        do {
9                j=j+1;
10       } while (0>1);
11     }
12   }
13 cout << "\n";
```

It produces the following output

```
0 1 2 -3 -2 -1 -4 -3 -2 -1
```

Chapter 5

Examples

5.1 A simple example: the Poisson problem in a cube

Find u such that

$$\left| \begin{array}{l} -\Delta u = f \text{ in } \Omega, \\ u_{|\Gamma} = 0, \end{array} \right. \tag{5.1}$$

where $f \in L^2(\Omega)$, and $\Gamma = \partial\Omega$. Lets assume that $\Omega = (-1,1)^3$, and choose $f = 1$.

Here the domain is a cube so there will be no informations coming from the `POV-Ray`file. The mesh is built by

```
mesh M = structured((10,10,10),(-1,1,-1),(1,-1,1));
```

We recall that the mesh `M` is built in a box specified by its two opposite corners $(-1,1,-1)$ and $(1,-1,1)$ and $(10,10,10)$ specifies the number of discretization points on each edge.

For more clarity one may prefer the following notations:

example2.ff

```
1  vector n = (10, 10, 10);
2  vertex a = (-1, 1,-1);
3  vertex b = ( 1,-1, 1);
4  mesh M = structured(n,a,b);
```

Since the geometry is very simple here one has to create an empty *scene*. This `POV-Ray` scene will be describe by an empty file: `"empty.pov"`. The *domain* `O` will be declared by

example2.ff

```
6  scene S = pov("void.pov"); // the pov-ray file for the geometry
7  domain O = domain(S);
```

The PDE is specified by

example2.ff

```
9  solve(u) in O by M
10 {
11    pde(u)
12      - div(grad(u)) = 1;
13      u = 0  on M;
14 };
15 save(opendx,"u.dat",u,M);
```

The last line is used to save the data in the file `"u.dat"`.

Chapter 6

Appendix A

6.1 Using ff3d within Windows and Cygwin

Step 1 Download and install the unix-in-windows software `cygwin`. (Note that this step is not essential and that if you do not want to do it, you can download cygwin1.dll which is the dynamic lib for cygwin compatibility and place it next to ff3d.exe.)

Step 2 Download and install a graphic package compatible with ff3d such as `medit, openDX, T3D`.

Step 3 Download ff3d and the examples.

Step 4 Optional: download and install `POV-Ray`

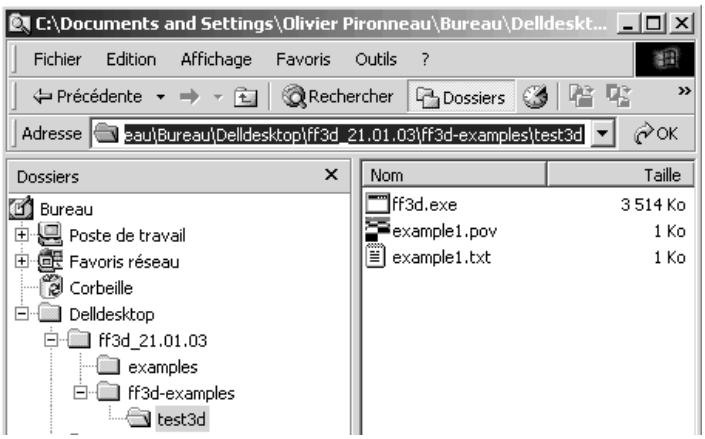


Figure 6.1: Subfolders displayed in Windows

6.1.1 Run an Example

1. Create a folder and for this illustration call it `test3d`
2. Put in folder `test3d` the files `ff3d.exe, example1.pov, example1.txt` and (optional) `medit.exe`.

3. Copy the full name (path and folder name) from the title of the folder window as shown in black on figure 1.

Figure 6.2: `FreeFEM3D`in the windows-cygwin environment

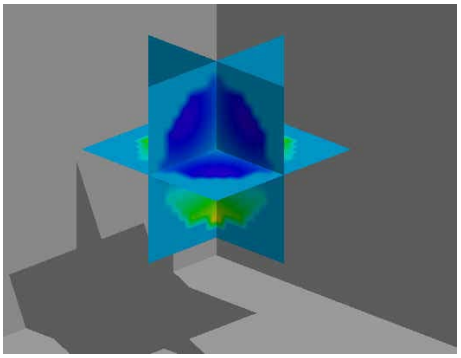
4. Start `cygwin` (or open a dos command window) and do a change directory command `cd "paste name here"`

5. type the command
   `./ff3d example1.txt`
   The following should appear

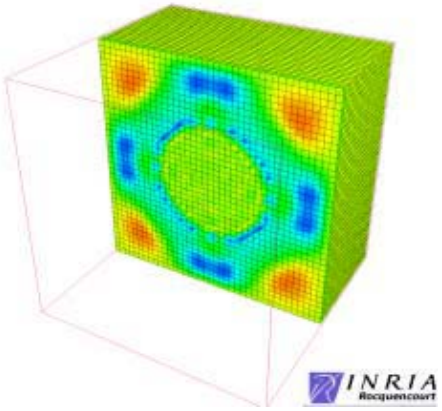
6. Do a graphic command on the new files generated by ff3d in this case `u.dat,v.dat`

6.1.2 Graphics

Creating good graphics with the results of `ff3d` is difficult; it is one of the weak point of the fictitious domain method. There is provision in `ff3d` for creating surface meshes that can be read by `medit`. For example if the following is added at the end of the file `example1.txt`:



(a) output generated by T3D



(b) Output generated by medit

Figure 6.3: Two ways to display results in Windows (the two figures do not display the same case).

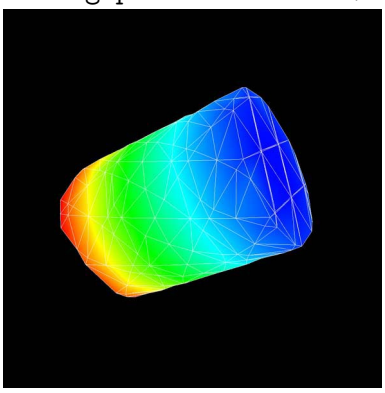
```
save(medit,"uu",gsphere,dos);   // to save the mesh
save(medit,"uu",u,gsphere,dos); // to save the values
```

where `gsphere` is defined in the `.pov` file which then becomes

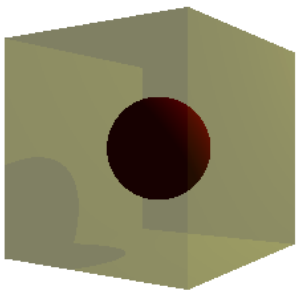
```
sphere {<0,0,0>, 0.5 pigment { color rgb <1,0,0> }}
cylinder { <0, 0,0>,<0.3, 0.3, 0.3>, 0.2 pigment{ color rgb <2,0,0> }}
```

and the file `example1.txt` contains the li

```
mesh  sphere = surface(<1,0,0>,0,M
mesh gsphere = surface(<2,0,0>,g0,
```



(a) output generated by medit



(b) Geometry displayed with POV-Ray

Figure 6.4: `POV-Ray` is also quite powerful. For instance on the right above is a visualization of the computational domain using a transparent texture for the computational box. The cylinder on which the solution is displayed could have been included as well.

Index