Freefem++: Manual

version 1.25 (Under construction)

http://www.ann.jussieu.fr/~hecht/freefem++.htm

http://www.freefem.org (out of order)

F. Hecht[1], O. Pironneau [2],
Université Pierre et Marie Curie,
Laboratoire Jacques-Louis Lions,
175 rue du Chevaleret ,
PARIS XIII,

K. Ohtsuka [3],
Hiroshima Kokusai Gakuin University,
Hiroshima, Japan

August 17, 2002

[1]mailto:hecht@ann.jussieu.fr
[2]mailto:pironneau@ann.jussieu.fr,
[3]mailto:ohtsuka@barnard.cs.hkg.ac.jp

Contents

Chapter 1

Getting Started

1.1 Introduction

A partial differential equation (PDE) is a relation between a function of several variables and its (partial) derivatives. Many problems in physics, engineering, mathematics and even banking are modeled by one or several partial differential equations.

`Freefem` is a software to solve numerically these equations. As its name tells, it is public domain software based on the Finite Element Method (FEM). FEM is the method to find the numerical approximation of PDE based on *variational method*. The variational method has been developed from the *least-energy principle*, so FEM is natural method from scientific view.

Many phenomena involve several different fields. Fluid-structure interactions, Lorenz forces in liquid aluminium and ocean-atmosphere problems are three such systems. These require approximations of different degrees possibly on different meshes. Some algorithms such as Schwarz' domain decomposition method require also the interpolation of data on different meshes within one program. Thus `freefem++` can handle these difficulties: *arbitrary finite element spaces on arbitrary unstructured and adaptated meshes*

Warning

`freefem++` is a scientific product to help you solve Partial Differential Equations in 2 dimensions; it assumes a basic knowledge and understanding of the FEM (see Section 1.4) and of the Operating System used. It is also necessary to read carefully this documentation to understand the possibilities and limitations of this product. We believe that the result by `freefem++` will be available to write a paper. However, the authors are not responsible for any errors or damage due to wrong results.

License

1.1.1 Overview

In scientific problems, there are many steps from the real world to obtaining numerical results and its visualization such as, mathematical modeling, theoretical results on mathematical modeling, theory on numerical calculation, programming, computer graphics for visualizations, so on. Each field is too far from others. Even in programming, we need too many skills, structured programming, knowledge on C, object oriented (OO) programming, C++, Java, OO software designing, etc. We are designing computer programming language which solve PDEs by *mathematical expression* with FEM. This is the *fundamental design philosophy* in **freefem project**. Such software is of great utility for the education and the research on applied mathematics, physics, engineering and even banking. Because theoretical results will draw easily to numerical results by `freefem++`.

The first attempt to realize our philosophy is `FreeFEM` (refer to [7, Chapter I] for detail). It was released in 1995 by O. Pironneau, D. Bernardi, F. Hecht together with C. Prud'homme and P. Parole as a follow-up of `MacFem, PCfem`, written in Pascal. The first version written in C was `freefem 3.4`; it had mesh adaptation but on a *single mesh*. It is based on a finite element solver, *mesh adaptation* was introduced later in the software after completion by M. Castro's thesis.

But mesh adaptation became an important feature of FreeFEM. Furthermore, originally written in C, we made so many changes in C++ that the software became difficult to manage.

In `FreeFem+`, the main members are O. Pironneau, F. Hecht, and K. Ohtsuka. The general philosophy of `FreeFEM` is kept but there are a number of new features which required some modification to the syntax powered over `FreeFEM` 3.4 as follows.

- It can handle multiple meshes within one program.
- It has an extremely powerful interpolator from one mesh to another, then functions defined on one mesh can be used on any other mesh (see Section 2.4.1). Interpolation from one unstructured mesh to another is not easy because it has to be fast and non-diffusive; one has to find for each point the triangle which contains it and that is one of the basic problems of computational geometry (see Preparata & Shamos[8] for example). To do it in the minimum number of operations is the challenge. We use an implementation which is $O(n \log n)$, based on a quadtree.
- It has a very robust and versatile mesh adaptation module (see Section 2.3.3).
- It can handle systems of PDE's both in strong and weak forms.

`freefem++` is a proper implementation of the philosophy underlying in FreeFEM project and is an entirely new program written in C++ and based on `bison` for an easy modification of the freefem language.

The language syntax of `freefem++` is redesigned and makes use of STL [14] templates and `bison` for its implementation. The outcome of this redesign is:

- It becomes a versatile software in which any new finite element can be included in a few hours; but a recompiling is then necessary.

Authors will add new libraries of finite elements available in `freefem++`, therefore `freefem++` will depend on the version number. We hope users will add new libraries into `freefem++`.

- It supports mathematical data types (complex numbers, complex valued functions, vector and matrix etc.) and C++ like programming is available. However users need not deep knowledge of C++ programming, if one want purely solve PDEs. It's enough to know the fundamental knowledge on mathematics, FEM and programming technique that are written in this manual.
- *Except strong forms*, everything by `freefem+` are also done in `freefem++` in this version, but we must modify `freefem+` program partly.

We take a Poisson equation with Dirichlet boundary condition as a brief introduction for explanation of `freefem++` programming: Let us consider a domain $\Omega \subset \mathbb{R}^2$. For a given function f, find the function u such that

$$-\Delta u = f \quad \text{in } \Omega, \qquad \Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}, \tag{1.1}$$

$$u = 0 \quad \text{on } \partial\Omega. \tag{1.2}$$

Poisson equation appears in many scientific problems, for example, Stationary flow of an incompressible viscous fluid in a cylinder duct, Newtonian potential in electromagnetism, Electrostatics, Elastic membrane, etc. In the case $f = 0$, (1.1) is called Laplace equation.

Converting the *strong form* (1.1) to the *weak form*, we use the *divergence theorem*, that is, for a smooth vector-valued function $\vec{w} = (w_1, w_2)$,

$$\int_\Omega \mathrm{div}\vec{w} = \int_{\partial\Omega} \vec{w} \cdot \vec{n} \tag{1.3}$$

where $\mathrm{div}\vec{w} = \partial w_1/\partial x + \partial w_2/\partial y$ and $\vec{n} = (n_1, n_2)$ the outward unit normal of $\partial\Omega$.

Multiplying both sides of (1.1) by v satisfying $v = 0$ on $\partial\Omega$, we have

$$\int_\Omega -\Delta u\, v = \int_\Omega f\, v$$

Here, v is called *test function*. By the divergence theorem and using the normal derivative defined by

$$\partial u/\partial n = n_1(\partial u/\partial x) + n_2(\partial u/\partial y)$$

we have the following (notice that $\Delta u = \mathrm{div}\nabla u$),

$$\int_\Omega \nabla u \cdot \nabla v - \int_{\partial\Omega} (\partial u/\partial n)\, v = \int_\Omega f\, v$$

Since $v = 0$ on $\partial\Omega$, the problem (1.1) with (1.2) is equivalent to find u satisfying the identity

$$\int_\Omega \left\{ \frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} - fv \right\} = 0. \tag{1.4}$$

for all *test functions* v ($v = 0$ on $\partial\Omega$), which is called the *weak form* of (1.1)-(1.2).

FEM is devided into three main steps:

Triangulation (Mesh): Ω is approximated by the family $\mathcal{T}_h$ of triangles T_i, $i = 1, 2, \cdots, nt$ (nt = number of triangles),

$$\Omega \approx \Omega_h = \cup_{i=1}^{nt} T_i$$

Here h means the size of the triangulation and Ω_h is as in Fig. 1.1 when $\Omega = \{(x, y) : x^2 + y^2 < 1\}$. In `freefem++` we call $\mathcal{T}_h$ by *mesh*.

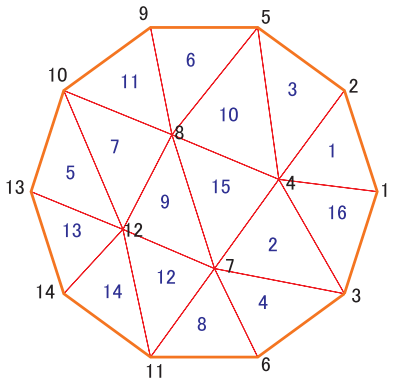


Figure 1.1: A triangulation of the unit disk

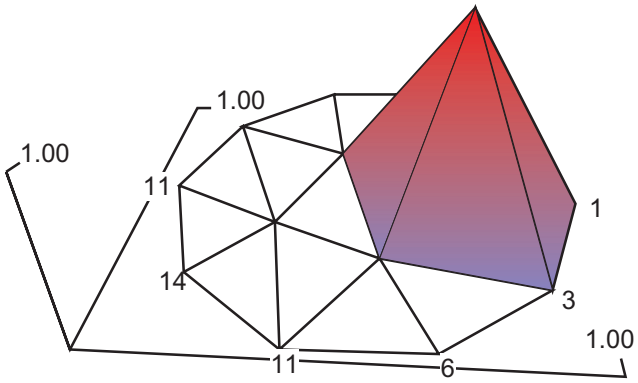


Figure 1.2: φ_4 when $l = 1$

Finite Element Space (FE-space): In solving the problem by FEM, a solution u and test functions v are approximated as follows

$$v(x, y) \approx v_1\varphi_1(x, y) + v_2\varphi_2(x, y) + \cdots + v_m\varphi_m(x, y) \tag{1.5}$$

using constants v_i and functions φ_i, $i = 1, \cdots, m$. The linear space

$$V_h = \{v : \ v \text{ satisfying (1.5) for arbitrary numbers } v_1, \cdots, v_m\} \tag{1.6}$$

is called *finite element space* (FE-space).

Linear System: Using $\mathcal{T}_h$, V_h, a finite element approximation u_h

$$u_h(x, y) = u_1\varphi_1(x, y) + u_2\varphi_2(x, y) + \cdots + u_m\varphi_m(x, y) \tag{1.7}$$

of u is given by the numerical solution of the linear system

$$AU_h^T = F_h^T \qquad U_h = (u_1, u_2, \cdots, u_m) \tag{1.8}$$

By `freefem++` we can easily construct a triangulation (mesh) $\mathcal{T}_h$, FE-space V_h and the stiffness matrix A in (1.8). We show them in the following simple example solving (1.1)-(1.2) with the given function $f = xy$, inside the unit circle.

Example 1

```
border C(t=0,2*pi){x=cos(t); y=sin(t);}
mesh Th = buildmesh (C(50));
```

```
fespace Vh(Th,P2);
func f= x*y;
Vh u,v;
problem Poisson(u,v,solver=LU) =
    int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
    - int2d(Th)( f*v )
    + on(C,u=0)  ;

real cpu=clock();
Poisson; //  SOLVE THE PDE
plot(u);
cout << " CPU = " << clock()-cpu << endl;
```

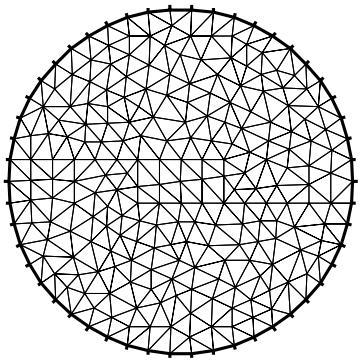


Figure 1.3: The mesh `Th`.

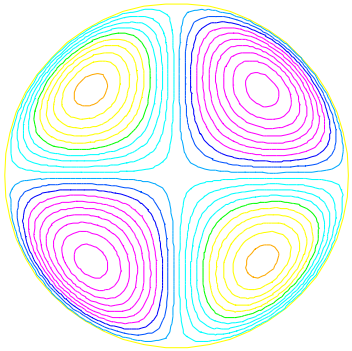


Figure 1.4: Contour lines of `u`.

Basically `freefem++` is a *compiler*, the language is typed, polymorphic and reentrant. Every variable must be typed, declared in a statement; each statement separated from the next by a semicolon '`;`'. *Comments* in `freefem++` start from "//" to the end of a line and is ignored by `freefem++`.

The standard process in `freefem++` is the following:

Step1(Define Boundary) Define a part of the boundary with keyword **`border`** by 1-parameter functions. The boundary is drawn by one curve or by multiple curves. In Example 1, the boundary is defined by analytic description (by opposition to CSG) such as

```
border C(t=0,2*pi){x=cos(t); y=sin(t); }
```

and identified by the name "`C`". If the boundary $\partial\Omega$ has the form $\partial\Omega = \cup_{j=1}^{J}\Gamma_j$ and Γ_j and Γ_{j+1} meet only one point, then we write them

$$\texttt{border}\quad \Gamma_j\texttt{(t=}a_j,b_j\texttt{)\{ x=}\phi_{j,x}\texttt{(t); y=}\phi_{j,y}\texttt{(t); \}}$$

Here $(\phi_{j,x}(a_j), \phi_{j,y}(a_j))$ and $(\phi_{j,x}(b_j), \phi_{j,y}(b_j))$ are endpoints of Γ_j.

Step2(Mesh Generation) See Section 2.3.

By writing,

```
mesh Th = buildmesh (C(50));
```

we have the mesh "Th"(see Fig. 1.3) automatically, which is generated based on the Delaunay-Voronoi algorithm with inner points generated with a density proportional to the density of points on the boundary, here we take 50 points (hence refinement is obtained by augmenting the number of boundary points). In general case $\partial\Omega = \cup_{j=1}^{J}\Gamma_j$, we write

```
mesh  Mesh_Name = buildmesh(Γ1(m1) + ··· + ΓJ(mj));
```

where m_j are numbers of marked points on Γ_j, $j = 1, \cdots, J$. As Mesh_Name, authors often use "Th" which means the symbol $\mathcal{T}_h$ widely used in the books on FEM.

Step3(FE-space and FE-function) See Section 2.4.

As in Example 1, we can define the finite element space (FE-space) in freefem++ by

```
fespace Vh(Th,P2);
```

with the mesh name Th which has 50 points on the unit circle C.

Here Vh expresses the mathematical expression V_h, but you can use an arbitrary name of finite element space. The keyword P2 determine the finite element basis functions, and in this version, we have discontinuous piecewise constant FE (finite element) P0, continuous piecewise linear FE P1, FE P2 of degree 2, Raviart-Thomas FE RT0, piecewise linear FE P1nc continuous at the middle of edge only. In what follows, we call by *FE-function* the function in FE-space.

Step4(Mathematical Formula) See Section 2.1.9.

New keyword func allows users to determine a function without arguments, if independent variables are x and y. Here, the given function f in Example 1 is defined by

```
func f=x*y;
```

Notice that the function determined by func is a mathematical expression, and is different from FE-function. The keyword func is used also in producing modules.

Step5(Weak Formulation) See Section 2.4.2.

For the time being, freefem++ *support only weak forms*. Before the formulation, we must declare the unknown function u and the symbol v used as a test function as follows,

```
Vh u,v;
```

The following give the weak form (1.4) with the name 'Poisson':

```
problem Poisson(u,v,solver=LU) =
    int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
    - int2d(Th)(f*v)
    + on(C,u=0)  ;
```

- The integral in the left-hand side of (1.4)

$$\int_\Omega \{(\partial u/\partial x)(\partial v/\partial x) + (\partial u/\partial y)(\partial v/\partial y) - fv\}$$

  is written as

```
int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
- int2d(Th)(f*v)
```

- By setting '**solver**=LU' in

  ```
  problem Poisson(u,v,solver=LU)
  ```

  we can select the solver of the linear system, here a Gauss LU factorization (see [7, Section 1.4]) is selected. In the case when

  ```
  problem Poisson(u,v)
  ```

  the default solver is LU. For the present, we can use `CG` (Conjugate Gradient (see [7, Chapter VI,5.6])), `Crout` (Crout's LU factorization), `Cholesky` (see [7, Chapter VI,2.3]), `GMRES` (see [7, Chapter VI,2]).

Step6(Solve and Visualization) See Section 2.4.2 and Section 2.4.7.
The real solving and its visualization are done by

```
Poisson; //  SOLVE THE PDE
plot(u);
```

Online graphics can be saved as Postscript file and has zooming and other feature (see Section 2.4.7). We can see the result by **plot**(u) in Fig. 1.4. More visualizations (containing 3d) in `freefem++` are the planning and preparation stage. There is a prototype of more visualization containing 3d-view and animation for `FreeFem+`, which is used for writing this book.

Step7(Reuse of Results) See Examples in index.
In `freefem++` we can reuse the obtained results easily, because the obtained results are expressed as a symbol in FE-space. For example, the obtained result is reused as a given function, function inside a cost functional in optimal control, evolution problems by FEM in space and finite differences in time, so on (see the examples for the detail).

Other(Output of Memory) In `freefem++`, the output/input are similar to C++. For example, here, the measurement of CPU time is done by

```
real cpu=clock();
...
cout << " CPU = " << clock()-cpu << endl;
```

Under the system (MS-Windows XP, Pentium(R) 4 CPU 1.70GHz and 512MB), this shows '`CPU = 0.235` (second)' that is the time used only for solving and visualization (compile 0.078s, execution 0.297s). If we comment out online graphic as follows

```
real cpu=clock();
Poisson; //  SOLVE THE PDE
// plot(u);
cout << " CPU = " << clock()-cpu << endl;
```

then it becomes '`CPU = 0.078`'. In most machine, the time will be 1 second or less.

The *stiffness matrix* A in (1.8) is obtained in `freefem++` by

```
varf a(u,v) = int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v))
              + on(C,u=0) ;
matrix A=a(Vh,Vh);
```

The keyword **varf** defines the *variational formula* "a", and "a(Vh,Vh)" gives the matrix (A_{ij}), $A_{ij} = \int_\Omega \nabla\phi_j \nabla\phi_i$ with the condition $u = 0$ by penalty method. The *load vector* $\vec{F}_h$ in (1.8) is also given in `freefem++` by

```
Vh v, f; //  given function must be FE-function to define below
varf b([v],[f]) = int2d(Th)(v*f); //  =∫_Ω fv
f = x*y;  //  define the given function
matrix B=b(Vh,Vh);
F[]=B*f[];     //  load vector
```

The **Step6** now become the following

```
u[]=A^-1*F[];  //  solve AU⃗h = F⃗.
plot(u);
```

These absolutely new feature in `freefem++` make the algorithms in **Step5** and **Step6** clearer.

Example 2

```
border C(t=0,2*pi) { x = cos(t); y = sin(t); }
mesh Th = buildmesh(C(50));
fespace Vh(Th,P2);
Vh u,v,f,F;
varf a(u,v) = int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v))
              + on(C,u=0) ;
varf b([v],[f]) = int2d(Th)(v*f);

f = x*y;
matrix A=a(Vh,Vh);  //  stiffness matrix
matrix B=b(Vh,Vh);
F[]=B*f[];     //  load vector
u[]=A^-1*F[];  //  solve AU⃗h = F⃗, see (1.8)
plot(u);
```

Perhaps, users can get the numerical results of Poisson equation defined on various domains (see Section 2.3.5) and for various given functions only by reading Overview.

Non-homogeneous boundary condition

Consider the problem

$$-\Delta u = f \ \text{ in } \Omega, \qquad u = g \ (g \neq 0) \ \text{ on } \partial\Omega \tag{1.9}$$

The `freefem++` program of (1.9) is following:

Example 3

```
mesh Th = square(10,10);
fespace Vh(Th,P2);
func f= x*y;
func g= sin(5*pi*(x-0.5)*(y-0.5));
Vh u,v;
solve Poisson(u,v) =
    int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
    - int2d(Th)(f*v)
    + on(1,2,3,4,u=g)  ;
plot(u);
```

Here the command **square** (see Section 2.3.1) give the mesh of square $]0,1[^2$ as shown in 1.5 with the label.

The following give the example of Non-homogeneous Neumann condition:

Example 4

$$-\Delta u = 0 \quad on\ \Omega, \tag{1.10}$$

$$\partial u/\partial n = 10(1-x^2)\ on\ \mathtt{T}, \partial u/\partial n = -10(1-x^2)\ on\ \mathtt{B}. \tag{1.11}$$

Here the domain Ω is given in Fig. 2.24. If the parameter `d` *become small, the domain Ω enclosed by these lines approximate the domain with cut. The singularity appear at the corner points* `C1` $\cap$ `C2` *and* `C2` $\cap$ `C3`*. The* `freefem++` *programming of this problem is*

```
fespace Vh(Th,P2);
Vh u,v;
problem Laplace(u,v)  =
        int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
        +int1d(Th,T)(10*(1-x^2)*v)+int1d(Th,B)(-10*(1-x^2)*v)
        +on(R,u=0);

Laplace;
plot(Th,u,ps="Neaumann.eps"); //  Fig. 1.6
```

The statement `int1d(Th,T)(10*(1-x^2)*v)` *express the integral*

$$\int_{T\cap\partial\Omega} 10(1-x^2)v$$

over `T` *oriented as $\partial\Omega$ with the triangulation* `Th`*.*

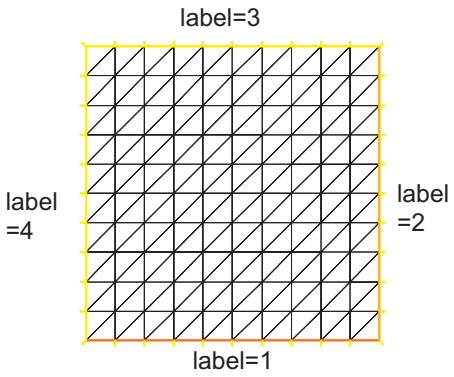


Figure 1.5: The mesh by `@square`

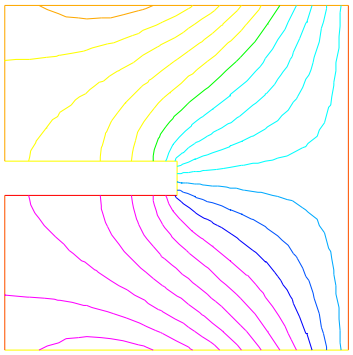


Figure 1.6: Solution of Non-Homogeneous Neumann condition

More Problems

Using `freefem++`, we can solve *evolution equations in time* such as the heat equation $\partial_t u - \Delta u$, in which we need the knowledge on finite difference in time (see Section 1.4.5) and use `for-loop`. The fluid mechanics is big field in scientific problems. `freefem++` implements the Characteristic-Galerkin method for convection operators named by `convect` (see Section 1.4.6), which is available for Convection-Diffusion, Navier-Stokes, etc. `freefem++` also solve numerically the partial differential system such as elasticity.

1.2 Installing FreeFem++ on Your Computer

You visit the site
http://www-rocq.inria.fr/Frederic.Hecht/freefem++.htm
You find the following in the page

- All the sources and makefile freefem++.tgz (tar+gzip)
- MacOS Powerpc (carbon) feefem++.sit (stuffit)
- Window 95,98,NT feefem++.zip

1.2.1 Unix and Linux

You select "freefem++.tgz" and download into the directory where you want install. By typing

```
%tar zxvf freefem++.tgz
```

source codes and examples are unarchived from "freefem++.tgz" as shown here:

```
FreeFem++v1.25/
FreeFem++v1.25/BUGS
FreeFem++v1.25/COPYRIGHT
FreeFem++v1.25/DOC/
.......................
FreeFem++v1.25/examples++/
.......................
FreeFem++v1.25/examples++-bug/
.......................
FreeFem++v1.25/examples++-tutorial/
.......................
FreeFem++v1.25/src/
.......................
FreeFem++v1.25/TODO
tar: Child returned ...
```

As above, this make the directory "freefem++v*x.xx*" (*x.xx* is the version number) and its subdirectories

freefem++: After compiling by "Makefile", you can get two programs under unix. FreeFem++ execute with visualization using X11graphic, and FreeFem++-nw execute without graphic X11 (no graphics window) but all the postscript plot are created.

/DOC: the document of `freefem++` .

/examples++ & -tutorial: the examples in the document and additional. The sources of `freefem++` program are files with extensions ".edp" (**e**quations aux **d**erivees **p**artielles in french). You can test all the exemples under unix by typing

```
cd examples++; ../FreeFem++  all.edp
cd examples++-tutorial; .././FreeFem++  all.edp
```

/examples++-bug: We found bugs in these examples, however we eliminated there bugs.

/examples++-mpi: Please visit the following site if you want know what is MPI.

`http://` *`www-unix.mcs.anl.gov/mpi/`*

Examples are tests of parallisation of `freefem++` under MPI.

src: Here there are source codes of `freefem++` , which are tested under Linux with the compilers g++ version 2.95.2, 2.95.3, 3.01. Edit the Makefile, and use the gnu make (gmake or make) and/or change the compiler, change the location of X11R6 include and library. The program compile with gcc version 2.95.2, 2.95.3 and 3.02 under Linux.

1.2.2 MS-Windows

If you only want to get the compiled program, then you select "freefem++.zip". After unarchive, this make subdirectories ($x.xx$ is version number),

```
freefem++
+--FreeFem++vx.xx_Win
   +--examples++
   +--examples++-bug
   +examples++-tutorial
```

The executable program "FreeFem++.exe" is in `FreeFem++v`$x.xx$`_Win` with the manual "manual.pdf".

If you want to compile `freefem++` , then you need also "freefem++.tgz".

For compile on MS-Windows, we need the Metrowerks Codewarrior compiler pro 7 (see http://www.metrowerks.com) we give the project for MS-Windows:

ffpc.mpc.zip Window95 or higher version of the Project.

We remark that the project can be old and name of file can be change. For example, under the version of codewarroir 7.

Click "FreeFem++.exe", then the dialog box will appear to select files (xxxxx.edp). You select the subdirectory "examples++" or "examples++-tutorial" and enter the one of them. You can find "all.edp", then you select and click it.

We can also use `freefem++` in an *MS-DOS command window (command prompt)* by typing

```
cd examples++
..\FreeFem++  all.edp
cd ..\examples++-tutorial;
..\FreeFem++  all.edp
```

On MS-Windows, if you install "Cygwin", then you can get an environment of programs similar to *Unix*.

1.2.3 MacOs

If you want the compiled program, You select "feefem++.sit (stuffit)" and download.

If you want compile it, then you need the metrowerks Codewarrior compiler pro 7. We given different project of different target.

FreeFem++.mpc.sit Carbon version for G3 and G4 Mac under MacOS 8.6 or newer

FreeFem++-nw.mpc.sit Command line MacOSX version without graphic but with postscript files.

1.3 Modeling–Edit–Run–Visualize–Revise

`freefem++` provide many examples and its documentation, so you can easily calculate mathematical models by FEM (finite element method) and study them. Explanations for these examples are given in this book. If you are a beginner of FEM, you start from Quick Tour of `freefem++` . The numerical simulation of scientific problem will be done as follows.

Modeling: Make a mathematical model describing scientific problems. Mathematical modeling is a deep and fruitful one, with many important implications for scientific problems (refer to Chapter 3).

Programming: Translate the mathematical model to `freefem++` source code, which is easy because `freefem++` includes many clever techniques in FEM with mathematical writing.

Run: Next step is to run it to see if it works. If we provisionally give the name of the source code to "something.edp", we can execute it by the typing

```
% freefem++ something.edp
```

An important part in programming is to keep aware of collections of programs that are available, and this manual contains many examples you can use freely. So we hope you to run these examples and their representing mathematical models, which are contained in the package in `freefem++` .

Visualization: The numerical calculation by FEM make huge data, so the easy way to check the obtained result is their visualization. `freefem++` can display the mesh and the contour lines of obtained functions. If you want to use these visualization after execution, you add the filename of PostScript to the commands "plot" (see Section 2.4.7).

Debugging: If the boolean value of "wait" is true (default is 'false'), then `freefem++` will stop at the information in visual form. Write the following, execute it and make a change the line "`wait=true`" to "`wait=false`".

```
wait = true;  //  set "true" if you want see each plotting
mesh Th = square(10,10,[-1+2*x,-1+2*y]); //  ]-1,1[^2
plot(Th);  //  plot the mesh
fespace Vh(Th,P2);
Vh f = sin(pi*x)*cos(pi*y);
plot(f);  //  plot the function f
Vh g = sin(pi*x + cos(pi*y));
plot(g);  //  plot the function g
```

If there is a fatal error in your source code, `freefem++` will end and cause an error message to appear. In MS-Windows, `freefem++` will open the message file by notepad. For example, if you forget parenthesis as in

```
mesh Th = square(10,10;
plot(Th);
```

then you will get the following message from `freefem++`,

```
mesh Th = square(10,10;
 Error line number 0, in file xxxxxx.edp, before  token ;
parse error
Compile error : parse error
        line number :0, ;
 at exec line  0
error Compile error : parse error
        line number :0, ;
```

If you use the same symbol twice as in

```
real aaa =1;
real aaa;
```

then you will get the message

```
real aaa =1;
    1 : real aaa; The identifier aaa exist
```

Notice that the line number start from 0. If you find that the program isn't doing what you want it to do, then you check the line number and try to figure out what's wrong. We give two techniques; One is *trace* by **plot** for *meshes* and (FE-)functions with **wait=true**, and by **cout** for scalar, vectors and matrices (see Section 2.1.2). Another is to *comment out* by "//". If you find a doubtful line in your source code, you comment out as follows,

```
real aaa =1;
//  real aaa;
```

1.4 A Quick Tour

Here we explain FEM in some detail. We use some mathematical tools from functional analysis. However, readers need not understand deeply.

1.4.1 Mathematical Preliminaries

In this book, we use the set $\mathbb{R}$ of *real numbers*. Computer user only think that $\mathbb{R}$ expresses the set of all considerable floating-points without limitation by machines. For example, a

current record of computation on π is $3 * 2^{36}$ decimal digits $3.141592653589\cdots$, however nobody think π equals to the calculated number. So, for strictly speaking of π, we need the knowledge on $\mathbb{R}$. Let ρ_{2002} be the record of computation on π in year 2002, and the record ρ_{2003} in year 2003. To show $\rho_y \to \pi$ as the year y tending to ∞, we need the measurement of proximity, which is called the *norm* $\|\cdot\|$. The norm is the distance from the zero point, that is, $\|a\|$ is the absolute value $|a|$ of $a \in \mathbb{R}$. In 2-dimensional space $\mathbb{R}^2$, $\|a\| = \sqrt{a_1^2 + a_2^2}$ for $a = (a_1, a_2) \in \mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. In general, $\|a\| = \sqrt{a_1^2 + a_2^2 + \cdots a_n^2}$ in n-dimensional space $\mathbb{R}^n = \mathbb{R}^{n-1} \times \mathbb{R}$. The norm is defined by the following properties: (1) $\|a\| > 0$ for $a \neq O$, $\|O\| = 0$, where $O = (0, \cdots, 0)$; (2) $\|\lambda a\| = |\lambda| \cdot \|a\|$, where λ is an arbitrary real or complex number; (3) $\|a + b\| \leq \|a\| + \|b\|$ (triangle inequality). In finite-dimensional space $\mathbb{R}^n$, there are following norms;

$$(1)\ \|a\|_{\ell^2} = \sqrt{a_1^2 + a_2^2 + \cdots a_n^2}; \qquad (2)\ \|a\|_{\ell^\infty} = \max_{i=1,\cdots,n} |a_i|; \qquad (3)\ \|a\|_{\ell^1} = \sum_{i=1}^{n} |a_i|;$$

and more. A mathematical proof shows that arbitrary norms satisfying (1)–(3) are equivalent.

Now we introduce a norm to a linear space of functions (infinite dimensional space). We can check $\mathcal{H}$ is a linear space or not by the following properties: (1) $\lambda f + \mu g \in \mathcal{H}$, for all real or complex numbers λ, μ and f, $g \in \mathcal{H}$; (2) There is a zero element O for which $0 \cdot f = O$ with the zero number 0. If we can define the norm $\|\cdot\|_{\mathcal{H}}$ in $\mathcal{H}$, then the distance $d(f, g)$ between f and g in $\mathcal{H}$ is given by $d(f, g) = \|f - g\|_{\mathcal{H}}$. For example, the exponential function e^x is approximated by the polynomial

$$f_{1000}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^{1000}}{1000!}$$

by Maclaurin's theorem. Now we introduce L^2-norm on the open interval $\Omega =]-1, 1[$, that is,

$$\|g\|_{L^2(\Omega)} = \sqrt{\int_{-1}^{1} |g(x)|^2\, dx}$$

Since $e^x - f_{1000}(x) = e^{\theta x} x^{1001}/1001!$ with $0 < \theta < 1$, we have

$$\|e^x - f_{1000}\|_{L^2(\Omega)} \leq \frac{e}{1001!} \left\{ \int_{-1}^{1} x^{2002} dx \right\}^{1/2} \leq \frac{e}{1001!} \frac{1}{\sqrt{1000}}$$

The step to construct the space of functions are following:

Step1: Introduce the norm $\|\cdot\|_{\mathcal{H}}$ and a linear space $\mathcal{H}$ of functions.

Step2: Consider a sequence f_n, $n = 1, 2, \cdots$ in $\mathcal{H}$ satisfy that $\|f_n - f_m\|_{\mathcal{H}} \to 0$ as $m, n \to \infty$, called *Cauchy's sequence*. Mathematics ensure the existence of function f_∞ such that $\|f_n - f_\infty\|_{\mathcal{H}} \to 0$ as $n \to \infty$. Now consier the linear space $\tilde{\mathcal{H}}$ adding the limit f_∞ of all Cauchy's sequnce in $\mathcal{H}$.

The linear space $\tilde{\mathcal{H}}$ is called the *completion* of $\mathcal{H}$. For example, let us consider a sequence of floating point numbers a_1, a_2, $\cdots$ satisfying $\|a_n - a_m\| \to 0$ as $m, n \to \infty$. Then there is a limit a_∞. Now we get $\mathbb{R}$ by applying Step2 for a set of floating points.

For a function f defined on $\Omega \subset \mathbb{R}^2$, we define L^2-norm by

$$\|f\|_{0,\Omega} = \left\{ \int_\Omega |f(x,y)|^2 \right\}^{1/2} \tag{1.12}$$

Here we omitted the volume element $dxdy$, for simple notation. Let us say $L^2(\Omega)$ the completion of functions defined on Ω by $\|\cdot\|_{0,\Omega}$.

For a function f defined on $\Omega \subset \mathbb{R}^2$, we define H^1-norm by

$$\|f\|_{1,\Omega} = \left\{ \|f\|^2_{0,\Omega} + \int_\Omega \left(\left| \frac{\partial f(x,y)}{\partial x} \right|^2 + \left| \frac{\partial f(x,y)}{\partial y} \right|^2 \right) \right\}^{1/2} \tag{1.13}$$

We denote by $H^1(\Omega)$ the completion of functions defined on Ω by $\|\cdot\|_{1,\Omega}$. Also we can define $H^2(\Omega)$ by the completion using the norm

$$\|f\|_{2,\Omega} = \left\{ \|f\|^2_{1,\Omega} + \int_\Omega \left(\left| \frac{\partial^2 f}{\partial x^2} \right|^2 + \left| \frac{\partial f^2}{\partial y^2} \right|^2 + + \left| \frac{\partial f^2}{\partial x \partial y} \right|^2 \right) \right\}^{1/2}$$

In `freefem++`, we can calculate numerically norms $L^2(\Omega)$, $H^1(\Omega)$ easily.

Example 5 *The function $f = \sin(\pi x)\cos(\pi y)$ is approximated by $g = (\pi x - (\pi x)^3/6)(1 - (\pi x)^2/2)$, where Maclaurin's expansion is used. Let us consider a sequence of domains $\Omega_r = \{(x,y) : \ 0 < x < r, 0 < y < r\}$, $r = 0.1, 0.2, \cdots, 1$. By the property of Maclaurin's expansion, $\|f - g\|_{\Omega_r}/\|f\|_{\Omega_r} \le \|f - g\|_{\Omega_{r'}}/\|f\|_{\Omega_{r'}}$ if $r < r'$. We make `freefem++` program to check them:*

```
verbosity=0;
for (real r=1.; r>=0.1; r-=0.1) {
  mesh Th=square(40,40,[r*x,r*y]);
  fespace Vh(Th,P2);
  Vh f=sin(pi*x)*cos(pi*y);
  Vh g=(pi*x-pi^3*x^3/6)*(1-pi^2*y^2/2);
  real L2 = sqrt( int2d(Th)(f^2) );
  real L2g = sqrt( int2d(Th)((f-g)^2) )/L2;
  cout <<"r="<<r<<", L2-norm of f = "<<L2<< ",Err =
"<<L2g<<endl;
  real H1 = sqrt( L2^2 + int2d(Th)(dx(f)^2+dy(f)^2) );
  real H1g = sqrt( L2g^2
              + int2d(Th)((dx(f)-dx(g))^2+(dy(f)-dy(g))^2) )/L2;
  cout << "H1-norm of f = "<<H1<< ", Err = "<<H1g<<endl;
}
```

Here's the output of this example.

```
r=1, L2-norm of f = 0.5, Err = 2.11331
H1-norm of f = 2.27702, Err = 16.5187
r=0.9, L2-norm of f = 0.447562, Err = 1.09646
H1-norm of f = 2.05929, Err = 9.35197
 (an omission of a middle part)
r=0.3, L2-norm of f = 0.129507, Err = 0.0141417
H1-norm of f = 0.757607, Err = 0.235007
r=0.2, L2-norm of f = 0.0653616, Err = 0.00267188
H1-norm of f = 0.561006, Err = 0.0723682
r=0.1, L2-norm of f = 0.0176678, Err = 0.000162122
H1-norm of f = 0.304707, Err = 0.0117296
times: compile 0.031s, execution 3.25s
```

1.4.2 Mathematical Expression and its `freefem++` Expression

There is `freefem3d` in freefem project solving 3-dimensional problem, but it's in developing. For the time being, `freefem++` can solve only 2-dimensional scientific problems. `freefem++` has the computer programming language with *mathematical exapression* and *structures*. We believe that *mathematics is the best language to express scientific problems in this time*, and the time has come when we should *use mathematics with computers*. Many mathematical expressions are independent of Dimensions as follows:

Let Ω be a domain in $\mathbb{R}^n$, $n = 1, 2, 3, \cdots$. The *Poisson's euqation* is described for a given function f and an unknown function u,

$$-\Delta u = f \quad \text{in } \Omega \tag{1.14}$$

where

$$\Delta = \frac{\partial^2}{\partial x_1^2} + \cdots + \frac{\partial^2}{\partial x_n^2}$$

is called *the Laplacian* or *the Laplace operator* of dimension n, x_i, $i = 1, \cdots, n$ are the components of Cartesian coordinate system of n-dimensional space $\mathbb{R}^n$ and $\partial/\partial x_i$, $i = 1, \cdots, n$ are the partial derivative with respect to x_i.

In scientific problems, it is interesting in the cases $n = 2, 3$. So we use the default coordinate symbols `x,y,z` instead of x_1, x_2, x_3. For a domain $\Omega \subset \mathbb{R}^2$, let $\mathcal{T}_h(\Omega)$ be a triangulation of Ω. The triangulation $\mathcal{T}_h(\Omega)$ is created automatically using the keyword **`mesh`** and **`buildmesh`** (see Section 2.3). Let u be a function in a finite element space (FE-space) $V_h(\mathcal{T}_h(\Omega), P_m)$, and be called FE-function. The FE-space $V_h(\mathcal{T}_h(\Omega), P_m)$ is created by **`fespace`** (see Section 2.4). Here m in the second parameter P_m describes the order of FE-functions. For $u \in V_h(\mathcal{T}_h(\Omega), P_m)$, the partial derivatives $\partial u/\partial x_i$, $i = 1, 2, 3$ are denoted by `dx(u), dy(u), dz(u)` respectively in `freefem++`. The symbol `z` is only reserved into the future. Please, notice that sometimes we use symbols x_i, $i = 1, 2$ in explanations of mathematical models and `x, y` in `freefem++` programming. The integral $\int_\Omega u$ of u over $\mathcal{T}_h(\Omega)$ is calculated numerically in `freefem++` by `int2d(`$\mathcal{T}_h$`)(`u`)` (see Section 2.4.5). If $\partial\Omega = \cup_{j=1}^{J}\Gamma_j$, then we also calculate the integral $\int_{\Gamma_j} u$ over the line Γ_j numerically by `int1d(`Γ_j`)(`u`)`.

It is better that readers are familiar with *fundamental theory on FEM*, however readers need not much skill of programming. `freefem++` help readers to acquire practical use of FEM for scientific problems and high level knowledge on FEM.

1.4.3 Fundamental theory on FEM

In Overview (see Section 1.1.1), the brief explanation of FEM is given. We discuss Poisson equation (1.1) all in detail using `freefem++`.

By the use of the space

$$H_0^1(\Omega) = \{v \in H^1(\Omega) : \ v = 0 \text{ on } \partial\Omega\}$$

the problem (1.1) become:

problem **Poisson**: For a given $f \in L^2(\Omega)$, find $u \in H_0^1(\Omega)$ such that

$$\begin{aligned} &a(u,v) + \ell(v) = 0 \quad \text{with } u = 0 \text{ on } \partial\Omega, \\ &a(u,v) = \int_\Omega \left\{ \frac{\partial u}{\partial x_1}\frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2}\frac{\partial v}{\partial x_2} \right\}, \qquad \ell(v) = \int_\Omega -f\, v \end{aligned} \tag{1.15}$$

for all test function $v \in H_0^1(\Omega)$. Here $L^2(\Omega)$ is the set of functions f defined on Ω such that

$$\|f\|_{0,\Omega} = \sqrt{\int_\Omega |f(x,y)|^2} < \infty$$

and $H^1(\Omega)$ is characterized by

$$\|f\|_{1,\Omega} = \sqrt{\|f\|_{0,\Omega}^2 + \int_\Omega \left(\left| \frac{\partial f(x,y)}{\partial x} \right|^2 + \left| \frac{\partial f(x,y)}{\partial y} \right|^2 \right)} < \infty$$

(see Section 1.4.1 for more detail).

Discrete the problem

Consider the unit disk D with the boundary C,

$$C = \{(x,y);\ x = \cos t,\ y = \sin t,\ 0 \le t \le 2\pi\}. \tag{1.16}$$

`freefem++` create the mesh of Ω by

```
border C(t=0,2*pi) { x = cos(t); y = sin(t); }
mesh Th = buildmesh(C(10));
```

`freefem++` creates the mesh of the unit disk with 10 points on the boundary C, and generate the mesh automatically based on the Delaunay-Voronoi algorithm with inner points generated with a density proportional to the density of points on the boundary. The created mesh is named by "`Th`"($=\mathcal{T}_h$) (see Fig. 1.1.

For an explanation, we use the continuous piecewise linear `P1` instead of `P2` in Example 1, which is made by

```
fespace Vh(Th,P1);
```

The function v in "Vh" is expressed

$$v(x) = v_1\varphi_1(x) + \cdots + v_{n_v}\varphi_{n_v}(x)$$

using the *hat functions* φ_j, $j = 1, \cdots, n_v$ (see Fig.1.2), where n_v denotes the number of vertexes and $v_1, \cdots, v_{n_v}$ are real numbers. Here the j-th hat function φ_j associated with j-th node is defined in the following way:

1. φ_j is continuous function on "Th".
2. φ_j is linear on each triangle K_l, $l = 1, \cdots, n_t$ of "Th".
3. $\varphi_j\left(q^k\right) = \delta_{jk}$ where q^k denotes the k-th vertex, for all $k = 1, \cdots, n_v$

In this example, $n_v = 14$ and $n_t = 16$, and so

$$\mathsf{Vh} = \{c_1\varphi_1(x) + \cdots + c_{14}\varphi_{14}(x) : \ c_1, \cdots, c_{14} \in \mathbb{R}\}. \tag{1.17}$$

The index h in Th ($=\mathcal{T}_h$) and Vh ($=V_h$) stands for the size of mesh, that is,

$$h = \max_{i=1,\cdots,n_t} \mathrm{diam}(K_i), \quad \mathrm{diam}K_i = \text{the length of longest side of } K_i.$$

Here $\Omega_h = \cup_{j=1,\cdots,n_t} K_j$, $K_j \in T_h$. Denote by I_Ω the set of indices of the internal nodes of the mesh, and the number of I_Ω by N_h. The calculation $a(u_i\varphi_i, \varphi_j)$ and $\ell(\varphi_j)$ for $i, j \in I_\Omega$ leads

$$\begin{aligned} a(u_i\varphi_i, \varphi_j) &= A_{ij}u_j,\ A_{ij} = \int_{\Omega_h} \nabla\varphi_j \cdot \nabla\varphi_i, \\ \ell(\varphi_j) &= -\int_\Omega f\varphi_j. \end{aligned} \tag{1.18}$$

For the sake of taking $u_i = 0$, $q^i \in \partial\Omega$, we put for a very large number E

$$A_{ii} = E \text{ for } i \notin I_\Omega,\ A_{ij} = \int_{\Omega_h} \nabla\varphi_j \cdot \nabla\varphi_i \text{ for } i \neq j \text{ and } 1 \leq j \leq n_v. \tag{1.19}$$

Let us set

$$\vec{F} = (F_1, \cdots, F_{n_v}),\ F_i = \int_{\Omega_h} f\varphi_i \text{ for } i = 1, \cdots, n_v. \tag{1.20}$$

We assume that there is a solution $\vec{U_h} = (u_1, \cdots, u_{n_v})$ of (1.8), that is,

$$A\vec{U_h}^T = \vec{F}^T, \qquad A = (A_{ij}) \text{ given in (1.18) and (1.19)},$$

where $\vec{U_h}^T$, $\vec{F}^T$ are the transpose of $\vec{U_h}$, $\vec{F}$. The matrix A given in (1.18) and (1.19) is called *stiffness matrix* and $\vec{F}$ the *load vector*, from early applications of FEM in structural mechanics. Then $\tilde{u}_h = \sum_{i\in I_\Omega} u_i\varphi_i$ and $\tilde{v} = \sum_{i\in I_\Omega} c_i\varphi_i$, $\forall c_i \in \mathbb{R}$ satisfty the identity

$$a(\tilde{u}_h, \tilde{v}) = \int_{\Omega_h} f\tilde{v}$$

and

$$Eu_i + \sum_{i\neq j,\ 1\leq j\leq n_v} u_j \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j = 0 \text{ for } i \notin I_\Omega.$$

Since E is the very large number, we then have

$$u_i = -\frac{1}{E} \sum_{i\neq j,\ 1\leq j\leq n_v} u_j \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i = O(1/E) \approx 0 \qquad \text{for } i \notin I_\Omega. \tag{1.21}$$

In freefem++ , the statement

```
fespace Vh(Th,P1);
Vh u,v,f;
problem Poisson(u,v) =
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
    - int2d(Th)(f*v)
    + on(C)(u=0);
```

create the stiffness matrix A and the load vector $\vec{F}$.

Remark 1 *It is impossible to write as*

```
int2d(Th){dx(u)*dx(v)+dy(u)*dy(v)-f*v)
+on(C)(u=0);
```

Because

```
int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
+on(C){u=0}
```

create the stiffness matrix *and*

```
- int2d(Th)(f*v)+on(C){u=0}
```

create the load vector $\vec{F}$.

Remark 2 *We feel that the technique used in (1.19) is poor in mathematical sense. However, the numerical calculation contain errors and this technique has the merit of applicability for widely boudary conditions, as follows:*

If the boundary condition is $u = g$ on $\partial\Omega$ for a given function g, then we put

$$F_i = Eg(q^i) \text{ for } i \notin I_\Omega. \tag{1.22}$$

The solution $\vec{U}_h = (u_i)$ of (1.8) satisfies

$$Eu_i + \sum_{i\neq j,\, 1\leq j\leq n_v} u_j \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j = g(q^i) \text{ for } i \notin I_\Omega. \tag{1.23}$$

This means $u_i \approx g(q^i)$, $q^i \in \partial\Omega$, which is done by

```
problem Poisson(u,v) =
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
    + int2d(Th)(-f*v)
    + on(C)(u=g);
```

Now consider the mixed boundary value problem, $u = g$ on Γ_D and $\partial u/\partial n = 0$ on Γ_N, where $\partial\Omega = \overline{\Gamma_D} \cup \overline{\Gamma_N}$ and $\Gamma_D \cap \Gamma_N = \emptyset$. In this case, we put

$$F_i = Eg(q^i) \text{ for } q^i \in \overline{\Gamma_D}, \text{otherwise } F_i = \int_\Omega f\varphi_i.$$

The default numerical solver of the linear system (1.8) is LU-factorization (see [7, Chapter VI,1.4]) in freefem++ . If you want change the solver of (1.8), you write in *problem* as follows,

```
problem Poisson(u,v,solver=Cholesky) =
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
    + int2d(Th)(-f*v)
    + on(C)(u=g);
```

Here Choleski factorization is used (see [7, Chapter VI,1]).

Example 6 The following `freefem++` program slove **Poisson** (1.1) with $f = -1$:

```
//  solve Poisson equation with Dirichlet boundary condition.
//  first, define the boundary as follows.
border C(t=0,2*pi) { x = cos(t); y = sin(t); } //   see (1.16)
mesh Th = buildmesh(C(10));                    //  see Fig. 1.1
fespace Vh(Th,P1);                             //  see (1.17)
Vh u,v;          //  unknown function u, test function v
func f = -1;    //  given function f = -1.
problem Poisson(u,v) =  //  weak form of (1.1), see (1.15)
   int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v))   //  bilinear a(u,v)
    + int2d(Th)( -f*v )                    //  linear form ℓ(v)
    + on(C,u=0) ;                          //  u = 0 on ∂Ω

 Poisson;        //  solve (1.8) by LU factorization
 plot(u);       //  see Fig. 1.7
```

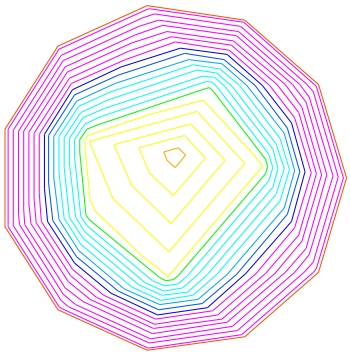


Figure 1.7: The level line of `u`

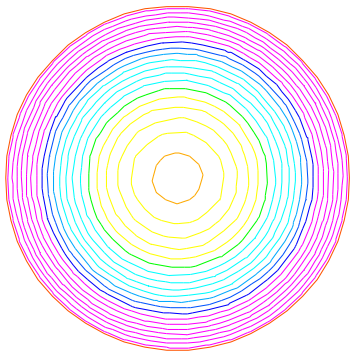


Figure 1.8: `u` by `buildmesh(C(50))`

Example 6 shows the entire program to solve **Poisson** (1.1) with $f = -1$.

Error measured by L^2-norm and H^1-norm

This problem has the analytical solution $u_e = (x^2 + y^2 - 1)/4$. You can see the difference between the exact solution and calculated solution "u" on the screen by adding the following in the last.

```
plot(u);
femp1 err =u-(x^2+y^2-1)/4;
plot(err,value=true,wait=true);
```

It is very difficult to make visual identification of the difference "err". The measurement of the difference is done by L^2-norm

$$\|\mathtt{u} - u\|_{0,D} = \left\{ \int_D \left| \mathtt{u}(x,y) - (x^2 + y^2 - 1)/4 \right|^2 dxdy \right\}^{1/2} \tag{1.24}$$

which is calculated numerically by freefem++ as follows,

```
sqrt( int2d(Th)((u-(x^2+y^2-1)/4)^2) )
```

Here $\sqrt{a}$ is calculated numerically by sqrt(a) and $\int_D f$ is done by int2d(Th)(f). The measurement of higher order is done by H^1-norm

$$\|\mathtt{u} - u_e\|_{1,D} = \left\{ \|\mathtt{u} - u_e\|_{0,D}^2 + \int_\Omega \left| \frac{\partial}{\partial x}(\mathtt{u} - u_e) \right|^2 + \int_\Omega \left| \frac{\partial}{\partial y}(\mathtt{u} - u_e) \right|^2 \right\}^{1/2}$$

which is calculated numerically as follows in freefem++

```
( int2d(Th)((u-(x^2+y^2-1)/4)^2)
       + int2d(Th)((dx(u)-x/2)^2) + int2d(D)((dy(u)-y/2)^2) )^(1/2)
```

Here dx(u) and dy(u) construct P_1-approximation of $\partial u/\partial x$ and $\partial u/\partial y$, respectively.

freefem++ statements just above are only calculations. We must display the calculated value on the (text) screen.

Output

The cout statement is the most common way to output data in freefem++ program. The statement

```
cout << ''hello freefem++'' << endl;
```

display the string "hello freefem++" on the screen. The cout statement takes a list of output separated by operator<<. The statement cout<< endl is to insert a newline character in the output sequence controlled by cout. The numerical calculation of L^2-norm is displayed by

```
cout << "Error L2-norm = " <<
         ( int2d(Th)((u-(x^2+y^2-1)/4)^2) )^0.5 << endl;
```

and H^1-norm by

```
cout << "Error H1-norm = " <<
     ( int2d(Th)((u-(x^2+y^2-1)/4)^2)
       + int2d(Th)((dx(u)-x/2)^2) + int2d(Th)((dy(u)-y/2)^2)
)^0.5
      << endl;
```

The obtained result in the case of buildmesh(C(10)) is following,

```
Error L2-norm = 0.0497833
Error H1-norm = 0.191856
```

and in the case of buildmesh(C(50))

```
Error L2-norm = 0.00205646
Error H1-norm = 0.0373978
```

1.4.4 More about stiffness matrix

`freefem++` has the command to get the stiffness matrix. Here, we show how to get the stiffness matrix using `freefem++` The first command `verf` is to define the *variational formula*.

Example 7 The example 6 (mesh size is changed) is also rewritten as follows:

```
border C(t=0,2*pi) { x = cos(t); y = sin(t); } //   see (1.16)
mesh Th = buildmesh(C(7));                     //
fespace Vh(Th,P1);                             //  see (1.17)
Vh u,v,f,F;
varf a(u,v) = int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v))
              + on(C,u=0) ;
varf b([v],[f]) = int2d(Th)(v*f);

f = x*y;
matrix A=a(Vh,Vh);  //  stiffness matrix, see (1.18), (1.19)
matrix B=b(Vh,Vh);
F[]=B*f[];     //  load vector, see (1.20)
cout << "F=" << F[] << endl;
cout <<"A="<< A << endl;
u[]=A^-1*F[];  //  solve AUh = Fh, see (1.8)
plot(u);
```

In this example, the number of vertices is 8 and the obtained stiffness matrix `A` is

	1	2	3	4	5	6	7	8
1	10^{30}	−0.31	0	0	−0.46	−0.51	0	0
2	−0.31	10^{30}	−0.23	0	0	−0.71	0	0
3	0	−0.23	10^{30}	−0.31	0	−0.71	0	0
4	0	0	−0.31	10^{30}	0	−0.51	−0.46	0
5	−0.46	0	0	0	10^{30}	−0.35	0	−0.54
6	−0.51	−0.71	−0.71	−0.51	−0.35	3.47	−0.35	−0.30
7	0	0	0	−0.46		−0.35	10^{30}	−0.54
8	0	0	0	0	−0.54	−0.30	−0.54	10^{30}

Inside of `freefem++` , the numbering of vertices start from 0 and end at 7, however we changed here the numbering and discard all digits to the right of the third decimal point for fitting this text. You can see the *penalization* (see (1.19)) with $E = 10^{30}$ and $A_{ii} = E$ for $i \notin I_\Omega$ ($6 \in I_\Omega$).

Mathematical results indicate that the Poisson equation with Neaumann boundary condition has not unique solution, whose weak form is same to (1.14) except the boundary condition:

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v dx$$

without the pernarization by $u = 0$, which is created by

```
varf a(u,v) = int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v))
matrix A=a(Vh,Vh); //  stiffness matrix
```

and the obtained stiffness matrix is the following

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \left[\begin{array}{cccccccc} 1.29 & -0.31 & 0 & 0 & -0.46 & -0.51 & 0 & 0 \\ -0.31 & 1.26 & -0.23 & 0 & 0 & -0.71 & 0 & 0 \\ 0 & -0.23 & 1.26 & -0.31 & 0 & -0.71 & 0 & 0 \\ 0 & 0 & -0.31 & 1.29 & 0 & -0.51 & -0.46 & 0 \\ -0.46 & 0 & 0 & 0 & 1.35 & -0.35 & 0 & -0.54 \\ -0.51 & -0.71 & -0.71 & -0.51 & -0.35 & 3.47 & -0.35 & -0.30 \\ 0 & 0 & 0 & -0.46 & 0 & -0.35 & 1.35 & -0.54 \\ 0 & 0 & 0 & 0 & -0.54 & -0.30 & -0.54 & 1.38 \end{array}\right] \end{array}$$

The determinant of this matrix is $-1.7082274230870981 \times 10^{-9} \approx 0$ (The matrix here differ from original one by omitting from third decimal decimal point). Perhaps, `freefem++` cannot solve the Poisson equation with Neumann boundary condition directly.

1.4.5 Evolution problems

`freefem++` also solve evolution problems such as the heat problem

$$\begin{aligned} &\frac{\partial u}{\partial t} - \mu \Delta u = f \quad \text{in } \Omega \times]0,T[, \\ &u(\vec{x},0) = u_0(\vec{x}) \quad \text{in } \Omega; \qquad (\partial u/\partial n)(\vec{x},t) = 0 \quad \text{on } \partial\Omega \times]0,T[. \end{aligned} \tag{1.25}$$

with a positive viscosity coefficient μ and homogeneous Neumann boundary conditions. We solve (1.25) by FEM in space and finite differences in time. We use the definition of the partial derivative of the solution in the time derivative,

$$\frac{\partial u}{\partial t}(x,y,t) = \lim_{\delta t \to 0} \frac{u(x,y,t) - u(x,y,t-\delta t)}{\delta t}$$

which indicate that $u^m(x,y) = u(x,y,m\delta t)$ imply

$$\frac{\partial u}{\partial t}(x,y,m\delta t) \simeq \frac{u^m(x,y) - u^{m-1}(x,y)}{\delta t}$$

The time discrezation of heat equation (1.26) is as follows:

$$\begin{aligned} &\frac{u^{m+1} - u^m}{\delta t} - \mu \Delta u^{m+1} = f^{m+1} \quad \text{in } \Omega \\ &u^0(\vec{x}) = u_0(\vec{x}) \quad \text{in } \Omega; \qquad \partial u^{m+1}/\partial n(\vec{x}) = 0 \quad \text{on } \partial\Omega, \quad \text{for all } m = 0, \cdots, [T/\delta t], \end{aligned} \tag{1.26}$$

which is so-called *backward Euler method* for (1.26). Multiplying the test function v both sides of the formula just above, we have

$$\int_\Omega \{u^{m+1}v - \delta t \Delta u^{m+1} v\} = \int_\Omega \{u^m + \delta t f^{m+1}\} v\,.$$

By the divergence theorem, we have

$$\int_\Omega \{u^{m+1}v + \delta t \nabla u^{m+1} \cdot \nabla v\} - \int_{\partial\Omega} \delta t \left(\partial u^{m+1}/\partial n\right) v = \int_\Omega \{u^m v + \delta t f^{m+1} v\}.$$

By the boundary condition $\partial u^{m+1}/\partial n = 0$, it follows that

$$\int_\Omega \{u^{m+1}v + \delta t \nabla u^{m+1} \cdot \nabla v\} - \int_\Omega \{u^m v + \delta t f^{m+1} v\} = 0. \tag{1.27}$$

Using the identity just above, we can calculate the finite element approximation u_h^m of u^m in a step-by-step manner with respect to t.

Example 8 *We now solve the following example with the exact solution* $u(x, y, t) = tx^4$.

$$\frac{\partial u}{\partial t} - \mu\Delta u = x^4 - \mu 12tx^2 \ \ in\ \Omega\times]0,3[,\ \Omega =]0,1[^2$$
$$u(x, y, 0) = 0 \quad on\ \Omega, \qquad u|_{\partial\Omega} = t * x^4$$

```
mesh Th=square(16,16);
fespace Vh(Th,P1);

Vh u,v,uu,f,g;
real dt = 0.1, mu = 0.01;
problem dHeat(u,v) =
    int2d(Th)( u*v + dt*mu*(dx(u)*dx(v) + dy(u)*dy(v)))
    + int2d(Th) (- uu*v - dt*f*v )
    + on(1,2,3,4,u=g);

real t = 0; //  start from t=0
uu = 0;     //  u(x,y,0)=0
for (int m=0;m<=3/dt;m++)
{
   t=t+dt;
   f = x^4-mu*t*12*x^2;
   g = t*x^4;
   dHeat;
   plot(u,wait=true);
   uu = u;
   cout <<"t="<<t<<"L^2-Error="<<sqrt( int2d(Th)((u-t*x^4)^2) ) << endl;
}
```

In the last statement, the L^2*-error* $\left(\int_\Omega |u - tx^4|^2\right)^{1/2}$ *is calculated at* $t = m\delta t, \delta t = 0.1$. *At* $t = 0.1$, *the error is 0.000213269. The errors increase with* m *and 0.00628589 at* $t = 3$.

The iteration of the backward Euler (1.27) is made by **for loop** *(see Section 2.2).*

Remark 3 *The stiffness matrix in loop is used over and over again.* `freefem++` *support reuses of stiffness matrix.*

1.4.6 Convection

The hyperbolic equation

$$\partial_t u - \vec{\alpha} \cdot \nabla u = f; \quad \text{for a vector-valued function } \vec{\alpha},\ \partial_t = \frac{\partial}{\partial t}, \tag{1.28}$$

appear frequently in scientific problems, for example, Navier-Stokes equation, Convection-Diffusion equation, etc.

In the case of 1-dimensional space, we can easily find the general solution $(x,t) \mapsto u(x,t) = u^0(x - \alpha t)$ of the following equation, if α is constant,

$$\partial_t u + \alpha \partial_x u = 0, \qquad u(x,0) = u^0(x), \tag{1.29}$$

because $\partial_t u + \alpha \partial_x u = -\alpha \dot{u}^0 + a\dot{u}^0 = 0$, where $\dot{u}^0 = du^0(x)/dx$. Even if α is not constant construction, the principle is similar. One begins the ordinary differentielle equation (with convention which α is prolonged by zero apart from $(0,L) \times (0,T)$):

$$\dot{X}(\tau) = -\alpha(X(\tau), \tau), \quad \tau \in (0,t) \quad X(t) = x$$

In this equation τ is the variable and x, t is parameters, and we denote the solution by $X_{x,t}(\tau)$. Then it is noticed that $(x,t) \to v(X(\tau),\tau)$ in $\tau = t$ satisfy the equation

$$\partial_t v + \alpha \partial_x v = \partial_t X \dot{v} + a \partial_x X \dot{v} = 0$$

and by the definition $\partial_t X = \dot{X} = -\alpha$ and $\partial_x X = \partial_x x$ in $\tau = t$, because if $\tau = t$ we have $X(\tau) = x$. The general solution of (1.29) is thus the value of the boundary condition in $X_{x,t}(0)$, it is has to say $u(x,t) = u^0(X_{x,t}(0))$ if $X_{x,t}(0)$ is on the x axis, $u(x,t) = u^0(X_{x,t}(0))$ if $X_{x,t}(0)$ is on the axis of t.

In higher dimension $\Omega \subset R^d,\ d = 2, 3$, the equation of the convection is written

$$\partial_t u + \vec{\alpha} \cdot \nabla u = 0 \text{ in } \Omega \times (0,T)$$

where $\vec{a}(x,t) \in R^d$. `freefem++` implements the Characteristic-Galerkin method for convection operators. Recall that the equation (1.28) can be discretized as

$$\frac{Du}{Dt} = f \ \text{ i.e. } \frac{du}{dt}(X(t),t) = f(X(t),t) \ \text{ where } \frac{dX}{dt}(t) = \vec{a}(X(t),t)$$

and where D is the total derivative operator. So a good scheme is one step of backward convection by the method of Characteristics-Galerkin

$$\frac{1}{\delta t}\left(u^{m+1}(x) - u^m(X^m(x))\right) = f^m(x) \tag{1.30}$$

where $X^m(x)$ is an approximation of the solution at $t = m\delta t$ of the ordinary differential equation

$$\frac{d\vec{X}}{dt}(t) = \vec{\alpha}^m(\vec{X}(t)),\ \vec{X}((m+1)\delta t) = x.$$

where $\vec{\alpha}^m(x) = (\alpha_1(x, m\delta t), \alpha_2(x, m\delta t))$. Because, by Taylor's expansion, we have

$$\begin{aligned} u^m(\vec{X}(m\delta t)) &= u^m(\vec{X}((m+1)\delta t)) - \delta t \sum_{i=1}^{d} \frac{\partial u^m}{\partial x_i}(\vec{X}((m+1)\delta t)) \frac{\partial X_i}{\partial t}((m+1)\delta t) + o(\delta t) \\ &= u^m(x) - \delta t \vec{\alpha}^m(x) \cdot \nabla u^m(x) + o(\delta t) \end{aligned} \tag{1.31}$$

where $X_i(t)$ are the i-th component of $\vec{X}(t)$, $u^m(x) = u(x, m\delta t)$ and we used the chain rule and $x = \vec{X}((m+1)\delta t)$. From (1.31), it follows that

$$u^m(X^m(x)) = u^m(x) - \delta t \vec{\alpha}^m(x) \cdot \nabla u^m(x) + o(\delta t). \tag{1.32}$$

Also we apply Taylor's expansion for $t \mapsto u^m(x - \vec{\alpha}^m(x)t)$, $0 \le t \le \delta t$, then

$$u^m(x - \vec{\alpha}\delta t) = u^m(x) - \delta t \vec{\alpha}^m(x) \cdot \nabla u^m(x) + o(\delta t).$$

Putting

$$\textbf{convect}\,(\vec{\alpha}, \delta t, u^m) = u^m\,(x - \vec{\alpha}^m \delta t)\,,$$

we can get the approximation

$$u^m\,(X^m(x)) \approx \textbf{convect}\,([a_1^m, a_2^m], \delta t, u^m) \quad \text{by } x = X((m+1)\delta t).$$

Example 9 A classical convection problem is that of the "rotating bell" (quoted from [7][p.16]). Let Ω be the unit disk centered at 0, with its center rotating with speed $\alpha_1 = y$, $\alpha_2 = -x$ We consider the problem (1.28) with $f = 0$ and the initial condition $u(x, 0) = u^0(x)$, that is, from (1.30)

$$u^{m+1}(x) = u^m(X^m(x)) \approx \texttt{convect}(\vec{\alpha}, \delta t, u^m).$$

The exact solution is $u(x, t) = u(\vec{X}(t))$ where $\vec{X}$ equals x rotated around the origin by an angle $\theta = -t$ (rotate in clockwise). So, if u^0 in a 3D perspective looks like a bell, then u will have exactly the same shape, but rotated by the same amount. The program consists in solving the equation until $T = 2\pi$, that is for a full revolution and to compare the final solution with the initial one; they should be equal.

```
border C(t=0, 2*pi) { x=cos(t);  y=sin(t); }; //  the unit cir-
cle
mesh Th = buildmesh(C(70));   //  triangulates the disk
fespace Vh(Th,P1);
Vh u0 = exp(-10*((x-0.3)^2 +(y-0.3)^2));    //  give u^0

real dt = 0.17,t=0;       //  time step
Vh a1 = -y, a2 = x;                  //  rotation velocity
Vh u; //  u^{m+1}
for (int m=0; m<2*pi/dt ; m++) {
    t += dt;
    u=convect([a1,a2],dt,u0);  //  u^{m+1} = u^m(X^m(x))
    u0=u;                      //  m++
    plot(u,cmm="convection: t="+t + ", min=" + u[].min + ",
max=" +  u[].max,wait=0);
};
```

Remark 4 *The scheme* `convect` *is unconditionally stable, then the bell become lower and lower (the maximum of* u^{37} *is* 0.406 *as shown in Fig. 1.10).*

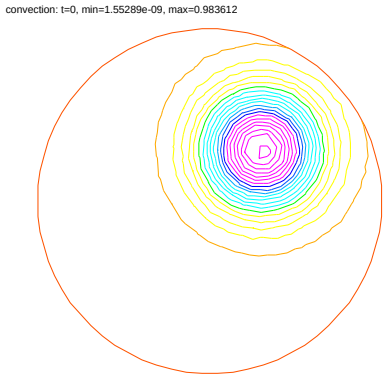


Figure 1.9: $u^0 = e^{-10((x-0.3)^2+(y-0.3)^2)}$

Figure 1.10: The bell at $t = 6.29$

1.4.7 Adaptive Mesh

In FEM, we can change the size of mesh freely in keeping the ratio $\rho_K/h_K > C$ with a constant $C > 0$ independent of meshes. Here h_K is the length of the longest side mesh size and ρ_K the diameter of the circle inscribed for $K \in T_h$.

Example 10 *The level lines of the function* $f = e^{-100((x-0.5)^2+(y-0.5)^2)}$ *on the mesh* `Th = square(5,5)` *is poor. The command*

```
mesh ATh = adaptmesh(Th,f);
```

create the new mesh `ATh` *by the Hessian* $D^2 f(x) = (\partial_{x_i}\partial_{x_j} f(x))$. *The level line of* f *on* `ATh` *is better than that on* `Th` *(see Fig. 1.11).*

```
mesh Th=square(5,5);
fespace Vh(Th,P1);
func f = exp(-100*((x-0.5)^2+(y-0.5)^2));
Vh u = f;
plot(u); //  left-hand side in Fig. 1.11

mesh ATh=adaptmesh(Th,f); //  make new mesh by Hessian of f
plot(ATh); //  middle in Fig. 1.11
fespace AVh(ATh,P1);
AVh uu = f;
plot(uu);  //  right-hand side in Fig. 1.11
```

1.4.8 Movemesh

Let Ω be a bounded domain in $\mathbb{R}^2$ and Φ the 1-1 mapping from Ω onto $\Phi(\Omega)$. If Ω is triangulated already – dubbed $T_h(\Omega)$, then we want $\Phi(\Omega)$ is also triangulated automatically. This want is satisfied by

```
mesh  Th=movemesh(Th,[Φ1,Φ2]);
```

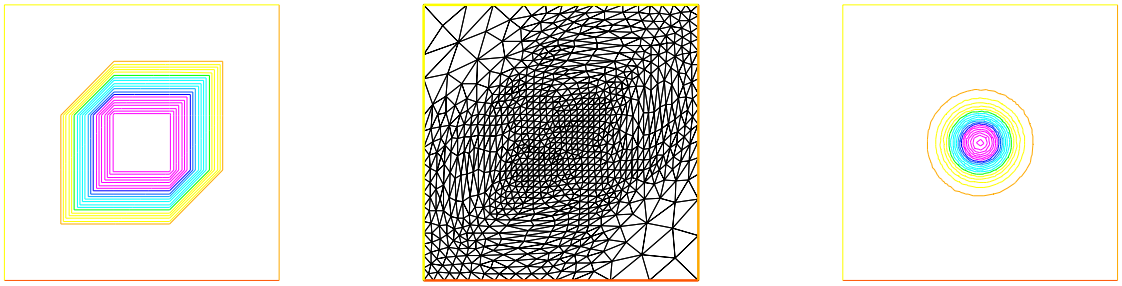


Figure 1.11: From the left-hand side, `plot(Th)`, `ATh`, `plot(ATh,f)`

where $\Phi = (\Phi_1, \Phi_2)$. This feature is used to see the deformation of plate in 2d elasticity.

Example 11

```
verbosity=4;
border a(t=0,1){x=t;y=0;label=1;};
border b(t=0,0.5){x=1;y=t;label=1;};
border c(t=0,0.5){x=1-t;y=0.5;label=1;};
border d(t=0.5,1){x=0.5;y=t;label=1;};
border e(t=0.5,1){x=1-t;y=1;label=1;};
border f(t=0,1){x=0;y=1-t;label=1;};
func uu= sin(y*pi)/10;
func vv= cos(x*pi)/10;

mesh Th = buildmesh ( a(6) + b(4) + c(4) +d(4) + e(4) + f(6));
plot(Th,wait=1,fill=1,ps="Lshape.eps");//  figure 1.12
Th=movemesh(Th,[x+uu,y+vv]);
plot(Th,wait=1,fill=1,ps="movemesh.eps");//  figure 1.13
```

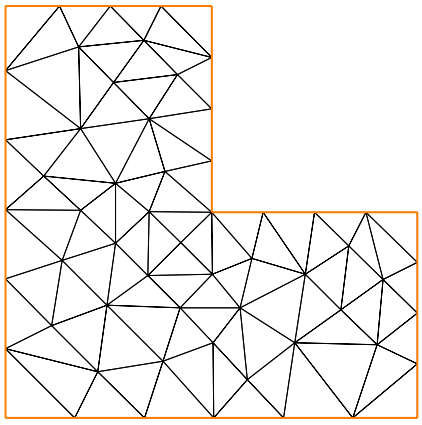


Figure 1.12: L-shape

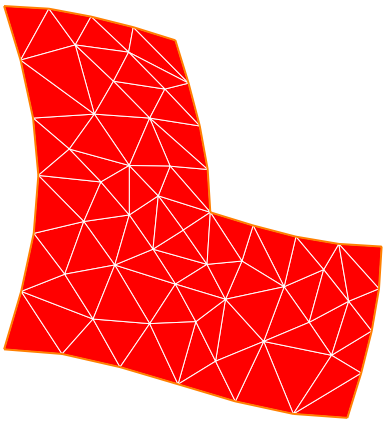


Figure 1.13: moved L-shape

Remark 5 *Consider a function u defined on a mesh Th. A statement like* `Th=movemesh(Th...` *does not change u and so the old mesh still exists. It will be destroyed when no function use*

it. Therefore a statement like $u = u$ will redefine u on the new Th and therefore destroy the old Th if u was the only function using it.

1.4.9 Finite Element Interpolator

We have theoretical error estimate [1] between finte element approximations u_h and the exact solution u of (1.1):

$$\|u_h - u\|_{0,\Omega} = \left\{ \int_\Omega |u_h - u|^2 \right\}^{1/2} \leq Ch^2 \tag{1.33}$$

with a positive constant C independent of mesh sizes h. It is very difficult to obtain the constant C in (1.33) theoretically. The constant C depend on Ω, u and f etc. For a domain Ω, the triangulations $\mathcal{T}_h(\Omega)$ and $\mathcal{T}_{h'}(\Omega)$ differ, if $h \neq h'$. This means FE-function u_h on $\mathcal{T}_h(\Omega)$ and $u_{h'}$ on $\mathcal{T}_{h'}(\Omega)$ also differ. `freefem++` has powerful interpolator (Section 2.4.1) from one mesh to another. Then we can calculate the approximation of $\|u_h - u_{h'}\|_{0,\Omega}$ numerically in `freefem++` as follows,

$$\texttt{sqrt(int2d(}\mathcal{T}_h\texttt{)(} |u_h - u_{h'}|^2 \texttt{)} \text{ if } h' < h$$

Example 12 *For the unit square domain, we can easily obtain C by the following* `freefem++` *program*

```
mesh Th0=square(200,200);
fespace Vh0(Th0,P1);
Vh0 u0, v0, uu;
solve Poisson0(u0, v0) =  //  instead of the exact sol.
        int2d(Th0)(dx(u0)*dx(v0)+dy(u0)*dy(v0))
        + int2d(Th0)(-1*v0)
        + on(1,2,3,4, u0=0);
for (int i=5; i<=100; i+=5) {
   mesh Th = square(i,i);  //  change mesh size
   fespace Vh(Th,P1);
   Vh u,v;
   solve Poisson(u, v) =
        int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
        + int2d(Th)(-1*v)
        + on(1,2,3,4, u=0);
   plot(u,ps="chkSol"+i+".eps");
   uu = u0 - u;
   cout <<"i="<<i<<", err = "<< sqrt( int2d(Th0)(uu^2) ) << endl;
   plot(uu,ps="chkErr"+i+".eps");
}
```

Here we used the keyword solve

```
solve Poisson0(u0, v0) =  ...
```

[1] This estimation is false for some non-convex domain Ω.

which define the weak form of Poisson's equation with Dirichlet and solve *it. We used the obtained result* `u0` *instead of the exact solution* u *. In the loop, we repeated to make the mesh* `Th` *with the mesh size* $h = 1/i$ *and to solve finite element solution* u_h=`u` *at* $i = 5, 10, \cdots, 100$. *The meshes* T_h *are different for each* h, *evenif* $\Omega_h =]0,1[^2$, *Then we* interpolated u_h *to FE function in FE space* `Vh0(`$T_{0.005}, P_1$`)` *by*

```
Vh0 uu = u0 - u;
```

Next we calculated the numerically $\|u_h - u\|_{0,\Omega}$ *by*

```
sqrt( int2d(Th0)(uu^2) );
```

and put the result on the screen with

```
cout <<"i="<<i<<", err = "<< sqrt( int2d(Th0)(uu^2) ) << endl;
```

The obtained results are in Table 1.1.

h	0.2000	0.1000	0.0667	0.0500	0.0400	$\cdots$	0.01
error	0.004754	0.001245	0.000558	0.000313	0.000200	$\cdots$	0.00023

Table 1.1: Mesh size h and error=$\|u_h - u\|_{0,\Omega}$

By Table 1.1, we obtain $C \approx 3.143$, *the maximum of* $\|$`uu`$\|_{0,\Omega}/h^2$ *is* 3.53 *and the minimum is* 2.91 *for* $h = 0.1, \cdots, 0.01$.

For an arbitrary mesh $\mathcal{T}_h$, we can know the size h of $\mathcal{T}_h$ using `hTriangle`.

1.4.10 Discontinuous FEM

Sometimes scientific problems contain the discontinuity of derivative. The typical example is transmission problem such as:

Let Ω_0 and Ω are bounded domains in $\mathbb{R}^2$, such that, $\Omega_0 \subset \Omega$. Let χ be the characteristic function of Ω_0, that is,

$$\chi(x) = 1 \text{ for } x \in \Omega_0; \qquad \chi(x) = 0 \text{ for } x \in \mathbb{R}^2 \setminus \Omega_0.$$

Let a_1 and a_2 be different positive constants.
For a given function f defined on Ω, find u such that,

$$\begin{aligned} &-a_1\Delta u = f \text{ in } \Omega_0, \quad -a_2\Delta u = f \text{ in } \Omega \setminus \overline{\Omega_0}, \\ &u = 0 \text{ on } \partial\Omega, \\ &a_1\partial u^-/\partial n|_{\partial\Omega_0} = a_2\partial u^+/\partial n|_{\partial\Omega_0}, \qquad u^-|_{\partial\Omega_0} = u^+|_{\partial\Omega_0} \end{aligned} \tag{1.34}$$

where $\partial/\partial n$ stands for the normal derivative on Ω_0, n the outward unit normal of $\partial\Omega_0$, and $u^\pm$ the limit of u from outside of Ω_0 and inside of Ω_0, respectively.

The weak form of (1.34) is simple:

For a given f, find u ($u = 0$ on $\partial\Omega$) such that:

$$\int_\Omega a\nabla u \cdot \nabla v = \int_\Omega fv \quad \text{for all } v,\ v = 0 \text{ on } \partial\Omega \tag{1.35}$$

where $a = a_1\chi(x) + a_2(1 - \chi(x))$. In numerical calculation, we have difficulty because P_1 FE functions are continuous. One method is to approximate χ by a P_0 FE function.

Example 13 *Consider* $\Omega =]-2,2[^2$ *and the ellipse*

$$\Omega_0 = \{(x,y) \in \mathbb{R}^2 : \ (x/0.5)^2 + (y/1.5)^2 < 1\}.$$

The creations of Ω *and* Ω_0 *are done by*

```
border B(t=0,4){x=-2+t; y=-2;}
border R(t=0,4){x=2; y=-2+t;}
border T(t=0,4){x=2-t; y=2;}
border L(t=0,4){x=-2; y=2-t;}
border C(t=0,2*pi) {x=0.5*cos(t); y=1.5*sin(t); }
int n=20;
mesh Th = buildmesh(B(n)+R(n)+T(n)+L(n)+C(40));
```

Here notice the direction by `+C(40)`*. The function* χ *is defined by* `P0` *and using* `region` *as follows:*

```
fespace Ph(Th,P0); //  constant discontinuous functions/element
fespace Vh(Th,P1); //  P1 continuous functions/element

Ph pl=region; //  define P0 function associated to region number
int inside=pl(0.0,0.0); //  get the region number of Ω0
Ph chi = (region==inside); //  define χ.
Ph a = chi + 3*(1-chi);  //  a = χ + 3(1 − χ)
```

For a given function $f = 1$*, we make the weak form and solve it.*

```
Vh u,v;
solve Transmission(u,v) = //  define and solve
        int2d(Th)(a*( dx(u)*dx(v)+dy(u)*dy(v) ))
        + int2d(Th)(-1*v)
        + on(B,R,T,L, u=0);
plot(u);
```

Chapter 2

Manual

2.1 Syntax

2.1.1 Data Types

Basically `freefem++` is a compiler, the language is typed, polymorphic and reentrant. Every variable must be typed, declared in a statement; each statement separated from the next by a semicolon '`;`'. The language allows the manipulation of basic types integers (`int`), reals (`real`), strings (`string`), arrays (example: `real[int]`), bidimensional (2D) finite element meshes (`mesh`), 2D finite element spaces (`fespace`) , definition of functions (`func`), arrays of finite element functions (`func`[*basic type*]), linear and bilinear operators, sparse matrices, vectors , etc. For instance

```
int i,n=20;                 //    i,n ∈ ℤ
real pi=4*atan(1);          //   pi ∈ ℝ
real[int] xx(n),yy(n);      //   two array of size n
for (i=0;i<=20;i++)         //  which can be used in statements such as
 { xx[i]= cos(i*pi/10); yy[i]= sin(i*pi/10); }
```

where `int`, `real`, , `complex` correspond to the C-types `long`, `double`, `complex<double>`.

The scope of a variable is the current block $\{\ldots\}$ like in `C++` , except the `fespace` variable, and the in variables local to a block are destroyed at the end of the block as follows.

Example 14

```
real r= 0.01;
mesh Th=square(10,10); //  unit square mesh
fespace Vh(Th,P1);     //  P1 lagrange finite  element space
Vh u = x+ exp(y);
func f = z * x + r * log(y);
plot(u,wait=ture);
{  //  new block
  real r= 2; //  not the same r
  fespace Vh(Th,P1);//   error because Vh is a global name
}  //  end of block
//   here r==0.01
```

The type declarations are compulsory in freefem++ because it is easy to make bugs in a language with many types. The variable name is just an alphanumeric string, the underscore character "_" is not allowed, because it will be used as an operator in the future.

2.1.2 List of major types

bool A conditional evaluates to a true or false. If the conditional is true, the statement following are executed; if the conditional is false, they are skipped. If you insert

```
plot(Th, wait=true);
```

between (2) and (3) in Example 6, the visualization of "Th" wait until your action.

int Declare an integer, C++ long integer.

string Strings are written as text enclosed within double quates, such as:

```
"This is a string in double quotes."
```

real A floating-point number such as 12.345, C++ double. The next line stores the number 1234.56789 in a variable, and print it out.

```
real aa = 1234.56789;
cout << aa << endl;
```

Here's the output of the freefem++ statements just above.

```
----CheckPtr:---init execution ---
1234.57
----CheckPtr:---end execution ----
```

complex Complex numbers, such as $1+2i$, $i = \sqrt{-1}$, corresponding to C++ complex<double>.

```
complex a =  1i, b = 2 + 3i;
cout << "a + b = " << a + b << endl;
cout << "a - b = " << a + b << endl;
cout << "a * b = " << a * b << endl;
cout << "a / b = " << a / b << endl;
```

Here's the output;

```
----------CheckPtr:-----init execution ------
a + b = (2,4)
a - b = (2,4)
a * b = (-3,2)
a / b = (0.230769,0.153846)
----------CheckPtr:-----end execution -- ----
```

ofstream ofstream to output to a file

ifstream ifstream to input from a file

real[int] Declare a variable that store multiple real numbers with integer index.

```
real[int] a(5);
a[0] = 1; a[1] = 2; a[2] = 3.3333333; a[3] = 4; a[4] = 5;
cout << "a = " << a  << endl;
```

This produces the output;

```
----------CheckPtr:-----init execution ---
a = 5   :
  1       2     3.33333   4       5
----------CheckPtr:-----end execution -- -
```

real[string] Declare a variable that store multiple real numbers with string index.

func Define a function without argument, if independent variables are `x, y`, for example

```
func f=cos(x)+sin(y) ;
```

Remark the function's type is given by the expression's type. The power of functions are given in `freefem++` such as `x^1`, `y^0.23`.

mesh This create the triangulation, see Section 2.3.

fespace This define a new type of finite element space, see Section 2.4.

P0 constant discontinuous finite element

P1 linear piecewise continuous finite element

P2 P_2 piecewise continuous finite element, where P_2 is the set of polynom of $\mathbb{R}^2$ de degrees two,

RT0 the Raviat-Thomas finite element,

P1nc nonconforming P_1 finite element,

problem to define a pde problem without solving it.

solve to define a pde problem and solve it.

varf to define a full variational form.

matrix to define a sparce matrix.

2.1.3 Globals

The names `x,y,z,label,region,P,N,nu␣triangle` are used to link the language to the finite element tools:

x the x coordinate of current point (real value)

y the y coordinate of current point (real value)

z the z coordinate of current point (real value) , (reserved for future use).

label The boundary is constucted two curves `C` and `Hole`, and the domain Ω enclosed by `C` and `Hole` is triangulated by

```
border C(t=0,2*pi) {x=cos(t); y=sin(t); }
real cx=0.2, cy=0.2, rr=0.1;//  center and radius of hole
border Hole(t=0,2*pi) {x=cx+rr*cos(t); y=cy+rr*sin(t); }
mesh Th = buildmesh(C(50)+Hole(-30));
```

In this case, the curve `C` has the label number 1 and `Hole` has 2. We can change the label number as follows,

```
border C(t=0,2*pi) {x=cos(t); y=sin(t); label=2; }
real cx=0.2, cy=0.2, rr=0.1;//  center and radius of hole
border Hole(t=0,2*pi) {x=cx+rr*cos(t); y=cy+rr*sin(t); label=1;
}
```

`label` show the label number of boundary if the current point is on a boundary otherwise 0 (int value).

`region` This function returns the region number of the current point (x,y) (int value). The following example display the region number of the domain *without hole*, here we notice that `Hole(30)` create the interior boundary and divide the unit disk to two regions.

```
border C(t=0,2*pi) { x=cos(t); y=sin(t); }
real cx=0.2, cy=0.2, rr=0.1;//  center and radius of hole
border Hole(t=0,2*pi) {x=cx+rr*cos(t); y=cy+rr*sin(t); }
mesh D = buildmesh(C(50)+Hole(30));  //  differ from Hole(-30)
fespace Ph(D,P0);  //  use P0-element
Ph regn = region;  //  define "region" as P0-function
cout <<"region no. at ("<<cx<<","<<cy<<") is "<< regn(cx,cy) <<
endl;
cout <<"region no. at ("<<cx+1.01*rr<<","<<cy+1.01*rr<<") is "
      << regn(cx+1.01*rr,cy+1.01*rr) << endl;
cout << "region no. at ("<<2<<","<<2<<") is "<< regn(2.0,2.0) <<
endl;
```

Here's the output;

```
----------CheckPtr:-----init execution ---
region no. at (0.301,0.301) is 1
region no. at (2,2) is 0
----------CheckPtr:-----end execution --
```

`P` The current point (R^2 value.) We can get the x, y components of `P` by `P.x`, `P.y`. Also `P.z` is reserved.

`N` The outward unit normal vector at the current point is on the curve define by `border` ($\mathbb{R}^3$ value). `N.x` and `N.y` are x and y components of the normal vector.

Example 15 *For a domain Ω, by the divergence theorem, we have the identity*

$$0 = \int_\Omega div\vec{1}\, dx = \int_{\partial\Omega} n_x + n_y$$

where $\vec{1} = (1,1)$ and n_x, n_y are x, y components of the outward unit vector on $\partial\Omega$. We check this by ***freefem++***

```
border C(t=-pi,pi) {x=cos(t); y=sin(t); };
mesh Th=buildmesh(C(50));
cout << int1d(Th,C)(N.x + N.y) << endl;
```

The calculated value is $1.63758e-15 \sim 0$.

`N.z` is reserved. .

cout The standard output (default output is in console). On MS-Windows, the standard output is only to console, in this time. `ostream`

cin The default is input from console (`istream`). On MS-Windows, this don't work.

true true `bool`value

false false `bool`value

pi `real` approximation of π

Here is how to show all the types, and all the operator and functions.

```
dumptable(cout);
```

To execute a system command in the string (not implemented on MacOs)

```
exec("shell command");
```

On MS-Windows, we need the full path. For example, if there is the command "ls.exe" in the subdirectory "`c:\cygwin\bin\`", then we must write

```
exec("c:\\cygwin\\bin\\ls.exe");
```

2.1.4 Array

There are 2 kinds of arrays: arrays with integer indices and arrays with string indices.

In the first case, the size of this array must be know at the execution time, and the implementation is done with the `KN<>` class so all the vector operator of `KN<>` are implemented. It is also possible to make an array of FE-function, with the same syntax.

Example 16 *In the following example, Poisson's equation is solved under 3 different given functions* $f = 1, \sin(\pi x)\cos(\pi y), |x-1||y-1|$*, whose solutions are stored in an array of FE-function.*

```
mesh Th=square(20,20,[2*x,2*y]);
fespace Vh(Th,P1);
Vh u, v, f;
problem Poisson(u,v) =
    int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v))
     + int2d(Th)( -f*v ) + on(1,2,3,4,u=0) ;
Vh[int] uu(3); //  an array of FE function
f=1;   //  problem1
Poisson; uu[0] = u;
f=sin(pi*x)*cos(pi*y);  //  problem2
Poisson; uu[1] = u;
f=abs(x-1)*abs(y-1);    //  problem3
Poisson; uu[2] = u;
for (int i=0; i<3; i++)  //  plots all solutions
  plot(uu[i], wait=true);
```

For the second case, it is just a map of the STL[1][12] so no vector operation except the selection of an item is allowed.

The transpose operator is ' like MathLab or SciLab, so the way to compute the dot product of two array a,b is **real** ab= a'*b.

Example 17

```
int i;
real [int] tab(10), tab1(10); //  2 array of 10 real
    real [int] tab2;    //   bug array with no size
tab = 1;                //   set all the array to 1
tab[1]=2;
cout << tab[1] << " " << tab[9] << " size of tab = "
     << tab.n << " " << tab.min << " " << tab.max << " " <<  endl;
tab1=tab;
tab=tab+tab1;
tab=2*tab+tab1*5;
tab1=2*tab-tab1*5;
tab+=tab;
cout << " dot product " << tab'*tab << endl; //   ^t tab tab
cout << tab << endl;
cout << tab[1] << " " << tab[9] <<  endl;
real[string] map;        //   a dynamique array
for (i=0;i<10;i=i+1)
  {
    tab[i] = i*i;
    cout << i << " " << tab[i] << "\n";
  };

map["1"]=2.0;
map[2]=3.0;             //   2 is automatically cast to the string "2"

cout << " map[\"1\"] = " << map["1"] << "; "<< endl;
```

[1] Standard template Library, now part of standard C++

```
cout << " map[2] = " << map[2] << "; "<< endl;
```

2.1.5 Calculations with the integers $\mathbb{Z}$

In integers, $+$, $-$, $*$ express the usual arithmetic summation (plus), subtraction (minus) and multiplication (times), respectively. The operators / and % yield the quotient and the remainder from the division of the first expression by the second. If the second number of / or % is zero the behavior is undefined. The *maximum* or *minimum* of two integers a, b are obtained by `max(`a`,`b`)` of `min(`a`,`b`)`. The power a^b of two integers a, b is calculated by writing `a^b`.

Example 18 *Calculations with the integers* $\mathbb{Z}$

```
int a = 12, b = 5;
cout <<"plus, minus of "<<a<<" and "<<b<<" are "<<a+b<<", "<<a-b<<endl;
cout <<"multiplication, quotient of them are "<<a*b<<", "<<a/b<<endl;
cout <<"remainder from division of "<<a<<" by "<<b<<" is "<<a%b<<endl;
cout <<"the minus of "<<a<<" is "<< -a << endl;
cout <<a<<" plus -"<<b<<" need bracket:"<<a<<"+(-"<<b<<")="<<a+(-b)<<endl;
cout <<"max and min of "<<a<<" and "<<b<<" is "<<max(a,b)<<","<<min(a,b)<<
endl;
cout <<b<<"th power of "<<a<<" is "<<a^b<< endl;
b=0;
cout <<a<<"/0"<<" is "<< a/b << endl;
cout <<a<<"%0"<<" is "<< a%b << endl;
```

produce the following result:

```
plus, minus of 12 and 5 are 17, 7
multiplication, quotient of them are 60, 2
remainder from division of 12 by 5 is 2
the minus of 12 is -12
12 plus -5 need bracket :12+(-5)=7
max and min of 12 and 5 is 12,5
5th power of 12 is 248832
12/0 : long long long
Fatal error : ExecError  Div by 0 at exec line  9
Exec error : exit
```

2.1.6 Calculations with the real $\mathbb{R}$

By the relation $\mathbb{Z} \subset \mathbb{R}$, the operators "$+$, $-$, $*$, /, %" and "**max**, **min**, ^" are also applicable to $\mathbb{R}$. However, % calculates the remainder of the integral parts of two real numbers.

The following example similar to 18

```
real a=sqrt(2.), b = pi;
cout <<"plus, minus of "<<a<<" and "<<pi<<" are "<< a+b <<", "<<
a-b << endl;
cout <<"multiplication, quotient of them are "<<a*b<<", "<<a/b<<
endl;
cout <<"remainder from division of "<<a<<" by "<<b<<" is "<< a%b
<< endl;
cout <<"the minus of "<<a<<" is "<< -a << endl;
cout <<a<<" plus -"<<b<<" need bracket :"<<a<<"+(-"<<b<<")="<<a
+ (-b) << endl;
```

gives the following output:

```
plus, minus of 1.41421 and 3.14159 are 4.55581, -1.72738
multiplication, quotient of them are 4.44288, 0.450158
remainder from division of 1.41421 by 3.14159 is 1
the minus of 1.41421 is -1.41421
1.41421 plus -3.14159 need bracket :1.41421+(-3.14159)=-1.72738
```

The *powers of real numbers* is given such as; $a^{n/m} = \sqrt[m]{a^n}$ for two integers m, n, for example $2^{1/2} = \sqrt{2}$, $5^{-1} = 1/5$, $7^{-2/3} = 1/\sqrt[3]{7^2}$ etc. For an arbitray real number α, $a^\alpha = \lim_{j\to\infty} a^{q_j}$ using a sequence $\{q_j\}_{j=1,2,\cdots}$ of rational numbers converging to α as $j \to \infty$, which is calculated by `a^`α in `freefem++` .

2.1.7 Calculations with the complex $\mathbb{C}$

By the relation $\mathbb{Z} \subset \mathbb{R} \subset \mathbb{C}$, the operators "$+$, $-$, $*$, $/$" and "`^`". However, "%, max, min" fall into disuse. Complex number such as `5+9i`, i= $\sqrt{-1}$, can be a little tricky. For real variables `a=2.45, b=5.33`, we must write the complex numbers `a+b*i` and `a+sqrt(2.0)*i` as

```
complex z1 = a+b*1i, z2=a+sqrt(2.0)*1i;
```

The imarginary and real parts of complex number `z` is obtained by **`imag`** and **`real`**. The conjugate of $a+bi$ (a, b are real) is defined by $a-bi$, which is denoted by **`conj`**`(a+b*1i)` in `freefem++` .

Graphical representation of complex numbers

The complex number $z = a + ib$ is considered as the pair (a, b) of real numbers a, b. Now draw the point (a, b) in the plane one draw a Cartesian rectangular system of axes and marks on the x-axis the real numbers in the usual way, on the y-axis the imaginary numbers with i as unit. By changing Cartesian coordinate (a, b) to the polar coordinate (r, ϕ), the complex number z has another expression $z = r(\cos\phi + i\sin\phi)$, $r = \sqrt{a^2+b^2}$ and $\phi = \tan^{-1}(b/a)$; r is called the *absolute value* and ϕ the *argument* of z. In the following example, we shall show them using `freefem++` programming, and *de Moivre's formula* $z^n = r^n(\cos n\phi + i\sin n\phi)$.

Example 19

```
real a=2.45, b=5.33;
complex  z1=a+b*1i, z2 = a+sqrt(2.)*1i;
func string pc(complex z) //  printout complex to (real)+i(imaginary)
{
   string r = "("+real(z);
```

```
   if (imag(z)>=0) r = r+"+";
   return r+imag(z)+"i)";
}
//  printout complex to |z|*(cos(arg(z))+i*sin(arg(z)))
func string toPolar(complex z)
{
   return abs(z)+"*(cos("+arg(z)+")+i*sin("+arg(z)+"))";
}
cout <<"Standard output of the complex "<<pc(z1)<<" is the pair "
     <<z1<<endl;
cout <<"Plus, minus of "<<pc(z1)<<" and "<<pc(z2)<<" are "<< pc(z1+z2)
     <<", "<< pc(z1-z2) << endl;
cout <<"Multiplication, quotient of them are "<<pc(z1*z2)<<", "
     <<pc(z1/z2)<< endl;
cout <<"Real/imaginary part of "<<pc(z1)<<" is "<<real(z1)<<", "
     <<imag(z1)<<endl;
cout <<"Absolute of "<<pc(z1)<<" is "<<abs(z1)<<endl;
cout <<pc(z2)<<" = "<<toPolar(z2)<<endl;
cout <<"  and polar("<<abs(z2)<<","<<arg(z2)<<") = "
     << pc(polar(abs(z2),arg(z2)))<<endl;
cout <<"de Moivre's formula: "<<pc(z2)<<"^3 = "<<toPolar(z2^3)<<endl;
cout <<"conjugate of "<<pc(z2)<<" is "<<pc(conj(z2))<<endl;
cout <<pc(z1)<<"^"<<pc(z2)<<" is "<< pc(z1^z2) << endl;
```

Here's the output from Example 19

```
Standard output of the complex (2.45+5.33i) is the pair
(2.45,5.33)
Plus, minus of (2.45+5.33i) and (2.45+1.41421i) are
(4.9+6.74421i), (0+3.91579i)
Multiplication, quotient of them are (-1.53526+16.5233i),
(1.692+1.19883i)
Real/imaginary part of (2.45+5.33i) is 2.45, 5.33
Absolute of (2.45+5.33i) is 5.86612
(2.45+1.41421i) = 2.82887*(cos(0.523509)+i*sin(0.523509))
  and polar(2.82887,0.523509) = (2.45+1.41421i)
de Moivre's formula: (2.45+1.41421i)^3
                          = 22.638*(cos(1.57053)+i*sin(1.57053))
conjugate of (2.45+1.41421i) is (2.45-1.41421i)
(2.45+5.33i)^(2.45+1.41421i) is (8.37072-12.7078i)
```

2.1.8 Logic (Boolean Algebra) and if-else

The statement "$3 < x$" is *true* if x is 4, and is *false* if x is 2. In `freefem++` , the statements are splited into two disjoint classes, which are denoted by the symbol 1 (true) or 0 (false) and are called *boolean values*. The operators `| & !` give the three boolean operators 'or' , 'and' , 'not'.

Example 20 *The following program showes the determination of these operators:*

```
bool p=(3<4), q=(2<3);
func string L(bool a, bool b)
{
   return "("+int(a)+","+int(b)+")";
}
cout <<"(p,q)="<<L(p,q)<<"|"<<L(!p,q)<<"|"<<L(p,!q)<<"|"<<L(!p,!q)<<endl;
cout <<"p and q ="<<p&q<<" |  "<<(!p)&q<<"  |  "<<p&(!q)
     <<"  |  "<<(!p)&(!q)<<endl;
cout <<"p or q = "<<p|q<<" |  "<<(!p)|q<<"  |  "<<p|(!q)
     <<"  |  "<<(!p)|(!q)<<endl;
```

Here's the output by Example 20.

```
(p,q)=(1,1)|(0,1)|(1,0)|(0,0)
p and q =1 |  0  |  0  |  0
p or q = 1 |  1  |  1  |  0
```

The **if** *conditional statement* with **true** executes a block of statements, for example,

```
int x = 4;
if ( 3<x ) { cout <<"(3<"<<x<<") is true."<< endl; }
```

In this example, the statement "$3 < x\,(x = 4)$" executes the block `{ ... }`; the message

```
(3<4) is true.
```

is printed out. However the statement "$3 < x\,(x = 2)$" do *nothing*. If you want a message in the case "$3 < x\,(x = 2)$", you must add **else** as follows:

```
int x = 2;
if ( 3<x ) { cout <<"(3<"<<x<<") is true."<< endl; }
else { cout <<"(3<"<<x<<") is false."<< endl; }
```

produces the following message

```
(3<2) is false.
```

2.1.9 Elementary functions

Power A function $f(x) = x^\alpha$, in which α is a constant, is called a *power function* such as $f(x) = x^3$, $f(x) = x^{1/2} = \sqrt{x}$, $f(x) = x^{-2/3} = 1/\sqrt[3]{x^2}$ so on, which is written by "`x^`α" in `freefem++` . The function $f^{-1}(x) = x^{-\alpha}$ is the inverse of $f(x) = x^\alpha$.

Exponent For the transcendental number $e = \lim_{n\to\infty}(1 + 1/n)^n = 2.718281...$, the function $f(x) = e^x$ is called the *exponent function* written by "**exp**`(x)`" in `freefem++` and $\frac{d}{dx}e^x = e^x$.

Logarithmic This function $f(x) = \ln x$ is the inverse of the exponent function $y = e^x$. Because the range of the exponential function $y = e^x$ is $0 < y < +\infty$, the *logarithmic function* $f(x) = \ln x$ can be defined only for *positive* value x, i.e., the domain of definition is $0 < x < +\infty$ and $\lim_{x\to 0} \ln x = -\infty$, $\lim_{x\to\infty} \ln x = +\infty$. By the relation $a^x = e^{x \ln a}$ for a positive constant $a > 0$, the logarithmic function $f(x) = \log_a x = \ln x / \ln a$. In `freefem++` , **log**`(x)` or **log10**`(x)` returns the value $\ln x$ or $\log_{10} x$ respectively.

We make a little test to see them:

```
cout <<"exp(1)="<<exp(1.)<<",exp(10^2)="<<exp(10.^2)<<endl;
cout <<"exp(10^3)="<<exp(10.^3)<<",exp(-10^2)="<<exp(-
10.^2)<<endl;
cout <<"log(1)="<<log(1.)<<",log(10)="<<log(10.)<<endl;
```

produce the output of the test

```
exp(1)=2.71828,exp(10^2)=2.68812e+43
exp(10^3)=inf,exp(-10^2)=3.72008e-44
log(1)=0,log(10)=2.30259
```

Here we notice that `exp(1)` will make error, because the argument of `exp` must be *real* (see the test above).

Trigonometric The trigonometric functions `sin(x)`, `cos(x)`, `tan(x)` depend on angles measured by *radian*. By the definition $\tan x = \sin x/\cos x$, $\tan x$ has discontinuity at $x = \pm\pi/2, 3\pi/2, \cdots$ and $\lim_{x\to\pi/2-0}\tan(x) = \infty$, $\lim_{x\to\pi/2+0}\tan(x) = -\infty$.
Now we give small test:

```
cout <<"cos(0)="<<cos(0.)<<",cos(pi/2)="<<cos(pi/2) << endl;
cout <<"tan(0)="<<tan(0.)<<",tan(pi/2)="<<tan(pi/2)<<",tan(-
pi/2)="<<tan(-pi/2)<<endl;
cout <<"tan(pi/2+0.1^15)="<<tan(pi/2+.1^15)<<",tan(-pi/2-
0.1^15)="<<tan(-pi/2-.1^15) << endl;
```

and the result:

```
cos(0)=1,cos(pi/2)=6.12303e-17
tan(0)=0,tan(pi/2)=1.63318e+16,tan(-pi/2)=-1.63318e+16
tan(pi/2+0.1^15)=-9.53295e+14,tan(-pi/2-0.1^15)=9.53295e+14
```

In `freefem++` V.1.23, we approximate π (=`pi`) by the constant,

$$3.14159265358979323846264338328$$

Circular The inverse of $\sin x$, $\cos x$, $\tan x$ are written by $\arcsin x$, $\arccos x$, $\arctan x$ and called *circular function* or *inverse trigonometric function*. Since $|\sin x| \le 1$ and $|\cos x| \le 1$, the domain of definition of $\arcsin x$ and $\arccos x$ is the interval $[-1, 1]$. However, we must take the principal values of the inverse trigonometric functions. We now check them using `freefem++` , (`asin(x)`(=$\arcsin x$), `acos(x)`(=$\arccos x$), `atan(x)`(=$\arctan x$):

```
cout <<"arcsin(-1)="<<asin(-1.)<<",arcsin(1)="<<asin(1.) <<
endl;
cout <<"arccos(-1)="<<acos(-1.)<<",arccos(1)="<<acos(1.) <<
endl;
cout <<"arctan(-10^10)="<<atan(-10.^10)
     <<",arctan(10^10)="<<atan(10^10) << endl;
```

produce the following:

```
arcsin(-1)=-1.5708,arcsin(1)=1.5708
arccos(-1)=3.14159,arccos(1)=-6.12574e-17
arctan(-10^10)=-1.5708,arctan(10^10)=1.5708
```

The principal values of `asin(x)`, `acos(x)`, `atan(x)` are

$$-\pi/2 \le \texttt{asin(x)} \le \pi/2, \quad 0 \le \texttt{acos(x)} \le \pi, \quad -\pi/2 \le \texttt{atan(x)} \le \pi/2.$$

Here the domain of definition of **atan**(x) is the interval $(-\infty < \mathtt{x} < \infty)$.

Hyperbolic By Euler's formulas

$$e^{i\phi} = \cos\phi + i\sin\phi, \qquad e^{-i\phi} = \cos\phi - i\sin\phi$$

we obtain the following

$$\sin\phi = \frac{e^{i\phi} - e^{-i\phi}}{2i}, \qquad \cos\phi = \frac{e^{i\phi} + e^{-i\phi}}{2}.$$

The correspondences give the following functions called *hyperbolic function*,

$$\sinh x = \left(e^{x} - e^{-x}\right)/2, \qquad \cosh x = \left(e^{x} + e^{-x}\right)/2.$$

and $\tanh x = \sinh x/\cosh x$. From the definition, we have the properties:

$$\cosh^2 x - \cosh^2 x = 1.$$

In freefem++ , hyperbolic functions are written by **sinh**(x) and **cosh**(x), whose inverse functions are written by **asinh**(x) and **acosh**(x).

$$\sinh^{-1} x = \ln\left[x + \sqrt{x^2+1}\right], \qquad \cosh^{-1} x = \ln\left[x + \sqrt{x^2-1}\right].$$

Elementary functions We call by *elementary functions* the class of functions consisting of the functions in this section (polynomials, exponential, logarithmic, trigonometric, circular) and the functions obtained from those listed by the four arithmetic operations

$$f(x) + g(x),\ f(x) - g(x),\ f(x)g(x),\ f(x)/g(x)$$

and by superposition $f(g(x))$, in which four arithmetic operarions and superpositions are permitted finitely many times. In freefem++ , we can create all elementary functions. The derivative of an elementary function is also elementary. However, the indefinite integral of an elementary function cannot always be expressed in terms of elementary functions.

Example 21 *The following make the mesh of the domain enclosed by* Cardioid

```
real b = 1.;
real a = b;
func real phix(real t)
{
   return (a+b)*cos(t)-b*cos(t*(a+b)/b);
}
func real phiy(real t)
{
   return (a+b)*sin(t)-b*sin(t*(a+b)/b);
}
border C(t=0,2*pi) { x=phix(t); y=phiy(t); }
mesh Th = buildmesh(C(50));
```

If you need a non-elementary function (for example, Special functions), then we have two method. One is to make it as the numerical solution of the differential equation. Another is to implement it into the source code of `freefem++` (see "AFunction.cpp").

Remark 6 *Consider hhe boundary of domains described by parametric curves*

$$\Gamma = \{(x, y) : \ x = \phi_x(t),\ y = \phi_y(t),\ a \le t \le b\}. \tag{2.1}$$

In `freefem++` *, we can use elementary functions as* ϕ_x, ϕ_y.

Remark 7 *The elementary functions* e^z, $\sin z$, $\cos z$ *are defined on the whole complex plane* $|z| < +\infty$. *Because the infinite series*

$$\begin{aligned} e^z &= 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots + \frac{z^n}{n!} + \cdots \\ \sin z &= z - \frac{z^3}{3!} + \frac{z^5}{5!} + \cdots + (-1)^k \frac{z^{2k+1}}{(2k+1)!} + \cdots \\ \cos z &= 1 - \frac{z^2}{2!} + \frac{z^4}{4!} + \cdots + (-1)^k \frac{z^{2k}}{(2k)!} + \cdots \end{aligned}$$

are converges on $|z| < +\infty$. *In* `freefem++` *, we can use complex value as the argument. Taking the principal value, we can define* $\log z$ *for* $z \neq 0$ *by*

$$\ln z = \ln |z| + \arg z.$$

Using `freefem++` *, we calculated* `exp(1+4i)`, `sin(pi+1i)`, `cos(pi/2-1i)` *and* `log(1+2i)`, *we then have*

$$\begin{aligned} -1.77679 - 2.0572i, \quad & 1.8896710^{-16} - 1.1752i, \\ 9.4483310^{-17} + 1.1752i, \quad & 0.804719 + 1.10715i. \end{aligned}$$

2.1.10 Real functions with two independent variables

The general form of real functions with two independent variables x, y is usually written as $z = f(x, y)$. In `freefem++` , `x` and `y` are reserved word in Section 2.1.3. When two independent variables are `x` and `y`, we can define a function without argument, for example

```
func f=cos(x)+sin(y) ;
```

Remark the function's type is given by the expression's type. The power of functions are given in `freefem++` such as `x^1`, `y^0.23`. In `func`, we can write an elementary function as follows

```
func f = sin(x)*cos(y);
func g = (x^2+3*y^2)*exp(1-x^2-y^2);
func h = max(-0.5,0.1*log(f^2+g^2));
```

Three dimensional view of these functions (see Fig. 2.1) are created by the visualizatio tool for `freefem+`.

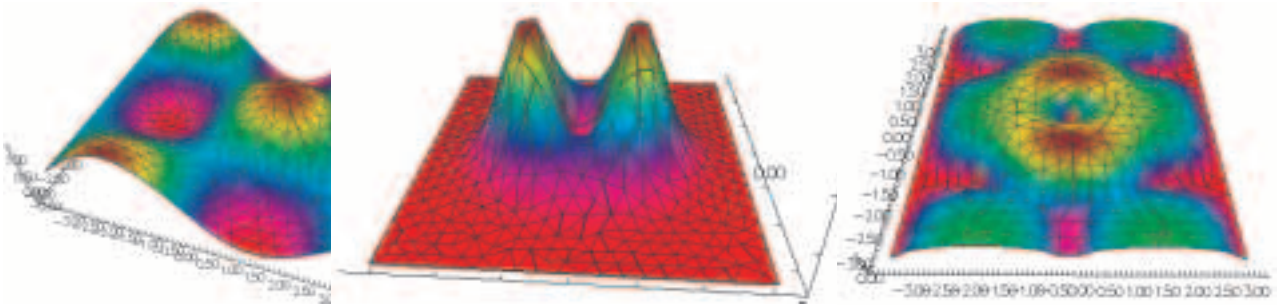


Figure 2.1: From left-hand side, `f`,`g` and `h`

Complex valued function create functions with 2 variables `x, y` as follows,

```
mesh Th=square(20,20,[-pi+2*pi*x,-pi+2*pi*y]); //  ]−π,π[²
fespace Vh(Th,P2);
func z=x+y*1i;  //  z = x + iy
func f=imag(sqrt(z));  //  f = ℑ√z
func g=abs( sin(z/10)*exp(z^2/10) ); //  g = |sin z/10 exp z²/10|
Vh fh = f; plot(fh);  //  contour lines of f
Vh gh = g; plot(gh);  //  contour lines of g
```

Remark 8 *The command* `plot` *is valid only for FE-functions.*

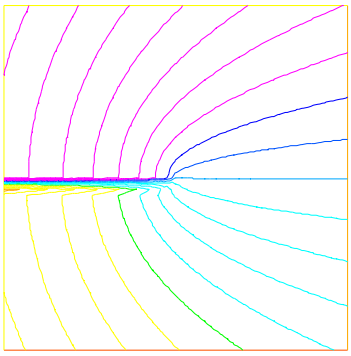


Figure 2.2: $\Im\sqrt{z}$ has branch

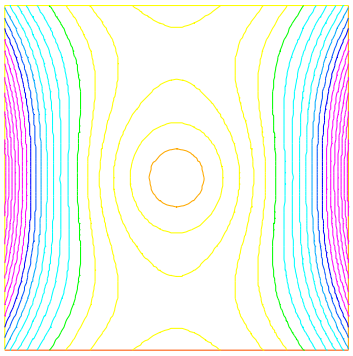


Figure 2.3: $|\sin(z/10)\exp(z^2/10)|$

2.1.11 FE-function and formula

In `freefem++`, there are arithmetic built-in functions $x\hat{}\alpha$, e^x, $\ln x$, $\sin x$, $\cos x$, $\tan x$, $\arcsin x$, $\arccos x$,$\arctan x$,$\sinh x$,$\cosh x$, $\sinh^{-1} x$, $\cosh^{-1} x$ and it is able to construct a new function by the four arithmetic operations and superposition of them (see *elementary functions*), which are called *formulas* to distinguish from FE-functions. We can add *new formulas* easily, if we want. Here, FE-function is an element of finite element space (see Section 2.4). Or to put it another way: *formulas* are the mathematical expressions combining its numerical analogs, but it is independent of meshes (triangulations).

Also, in `freefem++`, we can give an arbitrary symbol to FE-function combining numerical calculation by FEM.

Projection of Mathematical function to FE-function

The projection of a formula f to FE-space is done as in

```
func f=x^2*(1+y)^3+y^2;
mesh Th = square(20,20,[-2+2*x,-2+2*y]); //  square ]−2,2[²
fespace Vh(Th,P1);
Vh fh=f;  //  fh is the projection of f to Vh
```

The construction of `fh` ($=f_h$) is explained in Section 1.4.3. We need such projection when we want get the visualization of a formula f, the partial differential of f, definite integral of f.

`freefem++` is mathematical software for FEM, not formula manipulation like Mathematica, Mapple, muPAD, Maxima, RISA/ASIR, etc. The main difference is the following.

Factorization: No.

Differentiation: Not yet, but the numerical partial differential is supported after the projection.

Indefinite integral: No.

Definite integral: After interporation, we can get it numerically.

The main purpose of `freefem++` is to solve the boundary value problems of partial differential equations. In general, it is impossible to give the solution by formulas, evenif we add spetical functions. So `freefem++` give the solutions numerically keeping the mathematical structure. In `freefem++`, formulas and FE-functions coexist.

Example 22 *The following example shows the four arithmetic operations and superposition of FE-functions and formulas. The visualization of functions are valid only for FE-functions, so we give the module* `plotF` *for visualization of formulas, here. If two independent variables x and y are required for a new function $g(x,y) = |x^2 - y^2|$, we can use also the keyword* `func` *as shown here.*

```
func f=x^2*(1+y)^3+y^2;
mesh Th = square(20,20,[-2+4*x,-2+4*y]); //  square ]−2,2[²
fespace Vh(Th,P1);
fespace Vh2(Th,P2);
func int plotF(real f)  //  plot formula
{
   Vh2 fh = f; //  projection to FE-function
   plot(fh,wait=true,nbiso=40);
}
func real g(real x,real y)
{
   real r = x^2-y^2;
   if (r == 0) return 0.000000001;
   else return abs(x^2-y^2);
}
plotF(f);
Vh fh=f;  //  the projection of f
```

```
func h1 = fh+g(x,y); plotF(h1);  //  sum
func h2 = fh*g(x,y); plotF(h2);  //  product
func h3 = fh/g(x,y); plotF(h3);  //  quotient
func h4 = g(dx(fh),dy(fh));      //  superposition
//  here we cannot use dx(f), dy(f)
plotF(h4);
```

freefem++ keep mathematical structure in FEM-level, the operations become simple for FE-functions than Example 22

Example 23 *Here we use two functions, one is the function with sombrero shape*

$$f(x,y) = \sin(\pi r)/(\pi r), \quad r = \sqrt{x^2+y^2}.$$

and another is

$$g(x,y) = \frac{xy(x^2-y^2)}{x^2+y^2} \text{ if } (x,y) \neq (0,0); \qquad = 0 \text{ if } (x,y) = (0,0).$$

However, there is a danger to divide by 0, then we make adjustments to them.

```
mesh Th = square(20,20,[-2+4*x,-2+4*y]); //  square ]−2,2[²
fespace Vh(Th,P2);
wait = true;  //  wait in each plotting
func real g(real x,real y)
{
   real r;

   if ((x==0)&&(y==0)) r=0;
   else r = x*y*(x^2-y^2)/(x^2+y^2);

   if (r == 0) return 1.E-100; //  return 10^-100
   else return r;
}
func r = max(1.E-100,sqrt(x^2+y^2)); //  metric from the origin
Vh f= sin(pi*r)/(pi*r); //  shape of sombrero
plot(f);
Vh h1 = f+g(x,y); plot(h1); //  sum
Vh h2 = f*g(x,y); plot(h2); //  product
Vh h3 = f/g(x,y); plot(h3); //  quotient
Vh h4 = g(dx(f),dy(f)); plot(h4);
```

Functions providing Informations on Mesh

In calculation of error estimations, we need the informations on meshes, for example, mesh size, the number of subdomains etc. Here we explain them.

hTriangle give diamT_i (= the length of longest side) of each triangle $T_i \in \mathcal{T}_h$. Here the index h expresses the size of the mesh, which is the maximum of the each diamT_i. We use **hTriangle** as FE-function of order 0 as follows;

```
fespace Ph(Th,P0);
Ph hh = hTriangle;
```

and we get the index h of $\mathcal{T}_h =$ Th by

```
cout << "The mesh size of Th is "<<hh[].max<<endl;
```

We can get the distribution of mesh sizes for each T_i by

```
plot(hh,fill=true);
```

region give the number of subdomains of Th created by

```
mesh Th=buildmesh(....);
```

In Example 24, the unit disk Ω is devided into three subdomains $\Omega_1 = \{(x,y); x^2/0.2^2 + y^2/0.3^2 < 1\}$, $\Omega_0 = \{(x,y); (x-0.5)^2 + y^2 < 0.1^2\}$ and $\Omega_2 = \Omega \setminus \overline{\Omega_0 \cup \Omega_1}$. The munber of the subdomain is given as FE-function of order 0.

Example 24

```
{ //  example for "hTriangle"
  wait=true;
  mesh Th=square(1,2);
  fespace Ph(Th,P0);
  Ph h=hTriangle;
  savemesh(Th,"meshSqr0.msh"); // see 2.4
  plot(Th,bw=true);
  cout << h(0.1,0.75) << endl;
}
{ //  example for "region"
  border Cout(t=0,2*pi) { x=cos(t); y=sin(t); }
  border Cin1(t=0,2*pi) { x=0.2*cos(t); y=0.3*sin(t); }
  border Cin2(t=0,2*pi) { x=0.1*cos(t)+0.5; y=0.1*sin(t); }
  mesh Th=buildmesh(Cout(50)+Cin1(20)+Cin2(20));
  plot(Th,bw=true,wait=true,ps="meshNo2Holes.eps");// see 2.5
  fespace Vh(Th,P1);
  fespace Ph(Th,P0);
  Ph h=hTriangle;
  Ph R=region;
  cout <<"Regions are "<<R(-.5,0.)<<","<<R(0.,0.)<<","<<R(.5,0.) << endl;
  Ph xhi1 = (region == R(0,0));
  Ph xhi2 = (region == R(0.5,0));
  plot(xhi1,fill=true);
  plot(xhi2,fill=true);

  cout << h(0.0,0.0) << endl;
  cout <<"minimum size = "<< h[].min << ", maximum size = " << h[].max
<< endl;
}
```

Here, the output of Example 24.

```
   Nb of edges on Mortars  = 0
   Nb of edges on Boundary = 6, neb = 6
 Nb Of Nodes = 4
 Nb of DF = 4
1.11803
 Nb of common points 3
  --  mesh:  Nb of Triangles =    936, Nb of Vertices 494
 Nb of edges on Mortars  = 0
 Nb of edges on Boundary = 50, neb = 90
 Save Postscript in file 'meshNo2Hole.eps'
 Nb Of Nodes = 494
 Nb of DF = 494
 Nb Of Nodes = 936
 Nb of DF = 936
Regions are 2,1,0
0.132063
minimum size = 0.0312395, maximum size = 0.177423
```

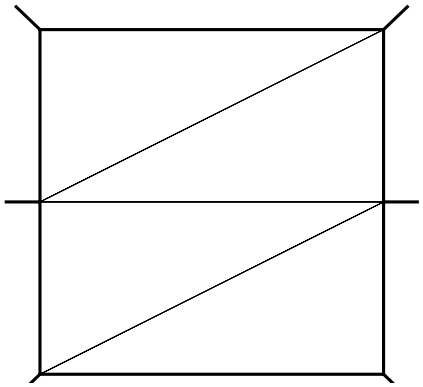


Figure 2.4: The mesh by `square(1,2)`

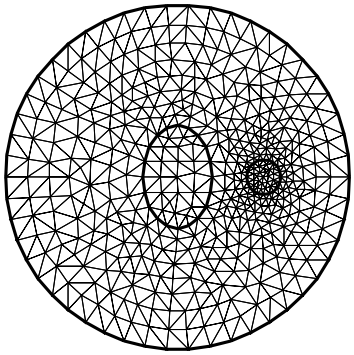


Figure 2.5: Domain divided into subdomains

2.1.12 Vectors and Matrices

There are three type of vectors in `freefem++`: First is the array of real (complex) such as "`real[int] a(5);`". Second is the coordinate of the point P such as "`R3 P;`". Third is the FE-function. Now we show them by the following example.

Example 25

```
real[int] a(5);
a=3;
cout << a << endl;
R3 P;
P.x=2.111; P.y=3.222; P.z=4.555;
cout << P << endl;
mesh Th = square(2,2);
```

				Explanation of each line
6	4	6		total of verteces, total of elements, total of edges
0	0	4		x-coordinate of 1st vertex, y-coordinate, lie on the edge 4
1	0	2		2nd vertex, lie on 4th edge
0	0.5	4		3rd vertex, lie on 4th edge
1	0.5	2		4th vertex, lie on 2nd edge
0	1	4		5th vertex, lie on 4th edge
1	1	3		6th vertex, lie on 3rd edge
1	2	4	0	1st element connected in order of 1,2,4
1	4	3	0	2nd element connected in order of 1,4,3
3	4	6	0	3rd element connected in order of 3,4,6
3	6	5	0	4th element connected in order of 3,6,5
1	2	1		line connecting 1st vertex and 2nd is assigned a number to 1
2	4	2		line connecting 2,4 is assigned a number to 2
4	6	2		line connecting 4,6 is assigned a number to 2
6	5	3		line connecting 6,5 is assigned a number to 3
3	1	4		line connecting 3,1 is assigned a number to 4
5	3	4		line connecting 5,3 is assigned a number to 4

Table 2.1: Contents of "meshSqr0.msh" and its explanation, see Fig. 2.4

```
fespace Vh(Th,P1);
Vh f=x*y;
cout << f << endl;
```

Here's the output from Example 25. The first line is the size of `a` *and the components of* `a` *in next. From 5th line to 8th line, the message by '*`meshTh = square(2,2)`*' appear.*

```
5
  3       3       3       3       3

2.111 3.222 4.555
   Nb of edges on Mortars  = 0
   Nb of edges on Boundary = 8, neb = 8
 Nb Of Nodes = 9
 Nb of DF = 9
9
  0       0       0       0     0.25
        0.5       0     0.5       1
```

We have added three operators "`' .* ./` " (like in Matlab or Scilab), where

- '`'`' is unary right *transposition* of array, matrix
- `' *` is the compound of transposition and matrix product, so it is the dot product (example `real DotProduct=a'*b`)
- '`.*`' is the term to term multiply operator.
- '`./`' is the term to term divide operator.

Example 26 *Now we show the example by the operators "' .* ./ " for the array.*

```
real[int] a=[3,7,3,5], b=[8,5,2,10], c(4);
c=3;
cout <<"a = "<< a << endl;
cout <<"c = "<< c << endl;
c=a+b;
cout <<"a+b = "<<c<<endl;
cout <<"scalar product (a,b) = "<<a'*b<<endl;
c=a.*b;
cout <<"term to term multiply of a, b = "<<c<< endl;
c=a./b;
cout <<"term to term divide of a, b = "<<c<< endl;
```

Here's the output of Example 26.

```
a = 4
  3       7       3       5
c = 4
  3       3       3       3
a+b = 4
 11      12       5      15
scalar product (a,b) = 115
term to term multiply of a, b = 4
 24      35       6      50
term to term divide of a, b = 4
0.375   1.4     1.5     0.5
```

The keyword **matrix** is used when the variational form create the matrix such as

```
Vh u,v;
varf  lap(u,v)= int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v));
matrix A = lap(Vh,Vh);
```

where Vh is FE-space created in advance. We can use the solve the linear system using '^-1' such that b = A^-1 x, but A^-1 is not matrix in this version. For the detail, see Section 2.4.6.

2.2 Programming

The syntax of freefem++ is in keeping with mathematics (theory of variational method, linear algebra, numerical analysis, etc.). The flow control in freefem++ programming is same to C++ , that is, branch (*if-then*, *if-then-else*, see Section 2.1.8), loops (*while*). Of course, there is the traditional *for* statement. And usual C++ functions are implemented: **exit, assert**

2.2.1 Loops

The `for` and `while` loops are implemented, and the semantic is the same as in `C++` with `break` and `continue` keywords.

In for-loop, there are three parameters; the INITIALIZATION of a control variable, the CONDITION to continue, the CHANGE of the control variable. While `CONDITION` is true, for-loop continue.

```
for (INITIALIZATION; CONDITION; CHANGE)
     { BLOCK of calculations }
```

The sum from 1 to 10 is calculated by (the result is in `sum`),

```
int sum=0;
for (int i=1; i<=10; i++)
   sum += i;
```

The while-loop

```
while (CONDITION) {
   BLOCK of calculations or change of control variables
}
```

is executed repeatedly until CONDITION become false. The sum from 1 to 10 is also written by **`while`** as follows,

```
int i=1, sum=0;
while (i<=10) {
  sum += i; i++;
}
```

We can exit from a loop in midstream by **`break`**. The **`continue`** statement will pass the part from *continue* to the end of the loop.

Example 27

```
for (int i=0;i<10;i=i+1)
    cout << i << "\n";
real eps=1;
while (eps>1e-5)
 { eps = eps/2;
   if( i++ <100) break;
   cout << eps << endl;}

for (int j=0; j<20; j++) {
   if (j<10) continue;
   cout << "j = " << j << endl;
}
```

2.2.2 Input/Output

The syntax of input/output statements is similar to `C++` syntax. It uses **`cout`**, **`cin`**, **`endl`**, `<<,>>`.

To write to (resp. read from) a file, declare a new variable `ofstream ofile("filename");` or `ofstream ofile("filename",append);` (resp. `ifstream ifile("filename");`

) and use `ofile` (resp. `ifile`) as `cout` (resp. `cin`). The word `append` in `ofstream ofile("filename",append);` means openning a file in append mode.

Remark 9 *The file is closed at the exit of the enclosing block,*

Example 28

```
int i;
cout << " std-out" << endl;
cout << " enter i= ? ";
cin >> i ;
{
  ofstream f("toto.txt");
  f << i << "coucou'\n";
}; //   close the file f because the variable f is delete

{
  ifstream f("toto.txt");
   f >> i;
}
{
  ofstream f("toto.txt",append);
     //  to append to the existing file "toto.txt"
  f << i << "coucou'\n";
}; //   close the file f because the variable f is delete

  cout << i << endl;
```

2.3 Mesh Generation

The following keywords are discussed in this section:

square, border, buildmesh, movemesh , adaptmesh, readmesh, trunc

,

2.3.1 Square

For easy and simple testing, we have included a macro for rectangles. All other shapes should be handled with `border+buildmesh`. The following

```
real x0=1.2,x1=1.8;
real y0=0,y1=1;
int n=5,m=20;
mesh Th=square(n,m,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
mesh th=square(4,5);
plot(Th,th,ps="twosquare.eps");
```

constructs a grid $n \times m$ in the rectangle $[1.2, 1.8] \times [0, 1]$ and a grid 4×5 in the unit square $[0, 1]^2$.

Remark 10 *The label of boundaries are* $1, 2, 3, 4$ *for bottom, right, top, left side (before the mapping* $[x0 + (x1 - x0) * x, y0 + (y1 - y0) * y]$ *(the mapping can be non linear).*

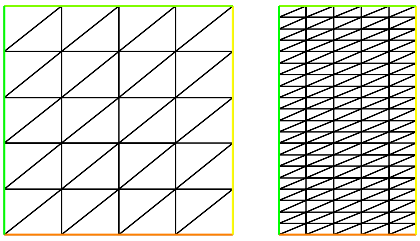


Figure 2.6: The 2 square meshes

2.3.2 Border

A domain Ω is defined by a parametrized curve

$$\textbf{border}\ \ \Gamma(t=a,b)\{x=\phi_x(t);y=\phi_y(t);\}$$

or curves Γ_j, $j=1,\cdots,J$ such as $\partial\Omega=\cup_{j=1}^{J}\Gamma_j$.

In **mesh- buildmesh**,

$$\textbf{mesh}\ \ \mathcal{T}_h=\textbf{buildmesh}(\Gamma_1(m_1)+\cdots+\Gamma_J(m_J));$$

$|m_j|$ in $\Gamma_j(m_j)\,(j=1,\cdots,J)$ expresses the number of vertices on the boundary Γ_j, and Ω is on the left (right) of Γ_j, if $m_j>0$ $(m_j<0)$ (see Fig, 2.7).

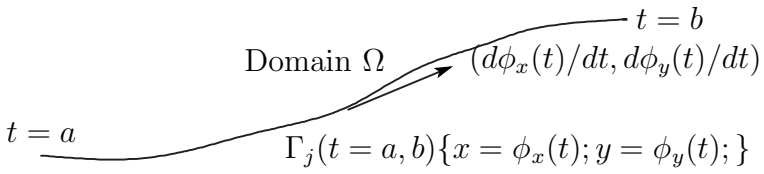


Figure 2.7: Γ_j and its positive orientation.

The vertices on Γ_j are taken at the point

$$(\phi_x(a+t_k),\phi_y(a+t_k));\ t_k=a+k(b-a)/m_j,\ k=0,\cdots,m_j$$

The order of Γ_j effects the numbering of verteces in $\mathcal{T}_h$.

Example 29 $\Omega=]0,9[^2$ *and* $\Omega=\cup_{j=1}^{4}G_j$. *The following make the triangulations of* Ω *under the the changing order of* j. *In plotting, each triangulations are arranged from left-hand side using* **movemesh**.

```
border G1(t=0,9){x=t;y=0;};
border G2(t=0,9){x=9;y=t;};
border G3(t=9,0){x=t ;y=9;};
border G4(t=9,0){x = 0; y = t;};
mesh Th = buildmesh(G1(3)+G2(3)+G3(3)+G4(3));
mesh Th01 = buildmesh(G2(3)+G1(3)+G3(3)+G4(3));
mesh Th02 = buildmesh(G1(3)+G3(3)+G4(3)+G2(3));
mesh Th03 = buildmesh(G1(3)+G4(3)+G2(3)+G3(3));
mesh Th1 = movemesh(Th01,[x+10,y]);
mesh Th2 = movemesh(Th02,[x+20,y]);
mesh Th3 = movemesh(Th03,[x+30,y]);
plot(Th,Th1,Th2,Th3,ps="border-chk.eps"); //  see Fig. 2.8
```

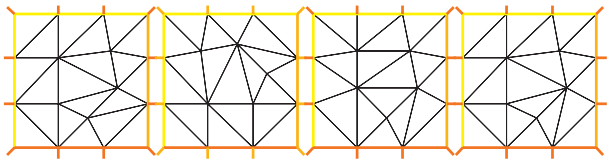


Figure 2.8: From the left-hand side, `Th, Th1, Th2, Th3`

Example 30 *For instance the unit circle with a small circular hole inside would be*

```
real pi=4*atan(1);
border a(t=0,2*pi){ x=cos(t); y=sin(t);label=1;}
border b(t=0,2*pi){ x=0.3+0.3*cos(t); y=0.3*sin(t);label=2;}
mesh Thwithouthole= buildmesh(a(50)+b(+30));
mesh Thwithhole   = buildmesh(a(50)+b(-30));
plot(Thwithouthole,wait=1,ps="Thwithouthole.eps"); // figure 2.9
plot(Thwithhole,wait=1,ps="Thwithhole.eps"); //  figure 2.10
```

Remark 11 *Notice the use of* `ps="fileName"` *to generate a postscript file identical to the plot shown on screen.*

Polygons are best described as unions of straight segments such as below for the $(0,2) \times (0,1)$ rectangle:

```
border a(t=0,2){x=t; y=0;label=1;}; //  the label can be depend of t
border b(t=0,1){x=2; y=t;label=1;};
border c(t=2,0){x=t; y=1;label=1;};
border d(t=1,0){x=0; y=t;label=1;};
border sq = [ label=1 ,[0,0],[0,2], [1,2,label=2] [1,0], [0,0,label=0]];
int n = 20;
mesh th= buildmesh(a(2*n)+b(n)+c(2*n)+d(n));
mesh Th= buildmesh(sq(8*n));
```

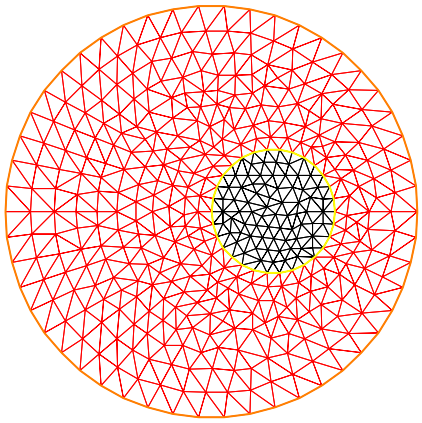


Figure 2.9: mesh without hole

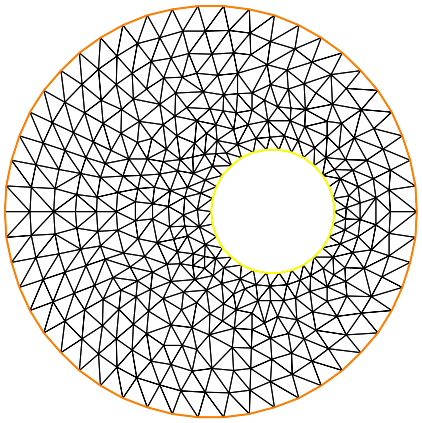


Figure 2.10: mesh with hole

Remark 12 *On the mesh* Th *the label of the polygon line with vertices* $|_0^0$, $|_2^0$, $|_2^1$ *is* 1 *, the label of the polygon line of vertices* $|_2^1$, $|_0^1$, $|_0^0$ *, to close the polygon the two labels corresponding are the same.*

Note that boundaries must cross at end points only.

2.3.3 Adaptmesh

It is possible to adapt the mesh to a function f by redefining the distance between two points with the Hessian matrix $\nabla\nabla f$.
Let M be a positive definite $d \times d$ matrix (d=2 in 2D).
The distance associated with M is

$$\|x\|^2 = x^T M x, \quad d(x,y) = \|x - y\|$$

The Delaunay criteria will now be applied with this new distance. Notice that circles become ellipes. In the algorithms above the notion of distance appears also when the edges are divided.

We recall that the interpolation error from u_h to u is bounded by

$$\|u - u\|_{E,0} \leq c\|\nabla\nabla u\|_{E,\infty} h^2$$

where E recalls that the Sobolev norms are defined with the Euclidian distance. for L^2 and for L_∞.
Thus we see that the error is large if $\|\nabla\nabla u\|$ is large. So the idea is to keep $\|h^T \nabla\nabla u h\|$ small. Note however that there are no error estimates which take into account the anisotropy of $\nabla\nabla u$. So the proof of the efficiency of this approach is open.
Similarly if we wish to adapt the mesh to two function we can consider the matrix $(\nabla\nabla u)(\nabla\nabla v)$.

However $x^T M y$ is a proper scalar product only if M is positive definite.

Given any symmetric matrix we can compute its eigen values and eigen vectors and construct a positive definite matrix from it by

$$\tilde{M} = Q^T|\Lambda|Q, \quad |\Lambda| = \begin{pmatrix} |\lambda_1| + \epsilon & 0 \\ 0 & |\lambda_2| + \epsilon \end{pmatrix}$$

where Q is the rotation matrix which makes M diagonal and ϵ is a small number.
In practice the function u which is used to adapt the mesh is computed on a coarse mesh. So there are several interpolations to do, the moment we pass from one mesh to another. This is a difficult task.

Remark 13 *No Metric Specified*

The user could also wish to specify a function but not a metric. Then we can still use the previous approach but with the identity matrix multiplied by the user-specified function $h(x, y$. This will generate an isotropic refinment with density adjusted to h.

Mesh adaptation is a very powerful tool which should be used whenever possible for best results. `freefem++` uses a variable metric/Delaunay automatic mesh algorithm which takes for input one or more functions (see in the following example) and builds a mesh adapted to the second differentiated field of the prescribed function.

```
verbosity=2;
mesh Th=square(10,10,[10*x,5*y]);
fespace Vh(Th,P1);
Vh u,v,zero;

u=0;
u=0;
zero=0;
func f= 1;
func g= 0;
int i=0;
real error=0.1, coef= 0.1^(1./5.);

problem Problem1(u,v,solver=CG,init=i,eps=-1.0e-6) =
    int2d(Th)(  dx(u)*dx(v) + dy(u)*dy(v))
  + int2d(Th) ( v*f )
  + on(1,2,3,4,u=g)  ;

real cpu=clock();

for (i=0;i< 10;i++)
{
  real d = clock();
  Problem1;
  plot(u,zero,wait=1);
  Th=adaptmesh(Th,u,inquire=1,err=error);
  cout << " CPU = " << clock()-d << endl;
  error = error * coef;
```

```
} ;

 cout << " CPU = " << clock()-cpu << endl;
```

The method is described in detail in [4]. It has a number of default parameters which can be overwritten for better results.

hmin= Sets the value of the minimal edge size. (`val` is of type double precision and default value is related to the size of the domain to be meshed and the precision of the mesh generator).

hmax= Sets the value of the maximal edge size. (`val` is of type double precision and the default value is the diameter of the domain to be meshed)

err= Sets the level of the P^1 interpolation error (0.01 is the default's value).

errg= Sets the value of the relative error on geometry. By default this error is 0.01, and in any case this value must be greater than $1/\sqrt{2}$. Remark that mesh size created by this option can be smaller than the `-hmin` argument due to geometrical constraint.

nbvx= Sets the maximal number of vertices generated by the mesh generator (9000 is the default's value).

nbsmooth= Set the number of iterations of the smoothing procedure (5 is the default's value).

nbjacoby= Set the number of iterations in a smoothing procedure during the metric construction, 0 imply no smoothing (6 is the default's value).

ratio= Set the ratio for a prescribed smoothing on the metric. If the value is 0 or less than 1.1 no smoothing on the metric is done (1.8 is the default's value).

If `ratio` > 1.1 the speed of mesh size variation is bounded by $log($`ratio`$)$. Remark: As `ratio` is closer to `1`, the number of vertices generated increases. This may be useful to control the thickness of refined regions near shocks or boundary layers .

omega= Set the relaxation parameter of the smoothing procedure (1.0 is the default's value).

iso= Forces the metric to be isotropic or not (false is the default's value).

abserror= If false the metric is evaluated using the criterium of equi-repartion of relative error (false is the default's value). In this case the metric is defined by

$$\mathcal{M} = \left(\frac{1}{\texttt{err coef}^2} \quad \frac{|\mathcal{H}|}{max(\texttt{CutOff}, |\eta|)} \right)^p \tag{2.2}$$

otherwire, the metric is evaluated using the criterium of equidistribution of errors. In this case the metric is define by

$$\mathcal{M} = \left(\frac{1}{\texttt{err coef}^2} \quad \frac{|\mathcal{H}}{sup(\eta) - inf(\eta)} \right)^p . \tag{2.3}$$

cutoff= Sets the limit value of the relative error evaluation (1.0e-6 is the default's value).

verbosity= Sets the level of printing (verbosity , which can be chosen between 0 and ∞) verbosity, and change the value of the global variable verbosity (obsolete).

inquire= To inquire or not the mesh (false is the default's value).

splitpbedge= If is true then split in two all internal edges with two boundary vertices (true is the default's value).

maxsubdiv= Change the metric such that the maximal subdivision of a background's edge is bound by the `val` number (always limited by 10,and 10 is also the default's value).

rescaling= the function with respect to which the mesh is adapted is re-scaled to be between 0 and 1 (true is the default's value).

keepbackvertices= if true will try to keep as many vertices of the previous mesh as possible (true is the default's value).

isMetric= if it is true the metric is given by hand (false is the default's value). So 1 or 3 functions given defining directly a symmetric matrix field (if one function is given then isotropic mesh size is given directly at every point through a function) , otherwise for all given functions, its Hessian is computed to define a metric.

power= exponent power of the Hessian to compute the metric (1 is the default's value).

Example 31 (`adaptindicatoredp.edp`) *In L-shape domain Ω, consider the problem*

$$-\Delta u + \epsilon u = f \qquad in\ \Omega, \qquad \partial u/\partial n = 0,$$

where $f = x - y$ and $\epsilon = 10^{-10}$. The solution u is in $H^2(\Omega \setminus B_\delta(\gamma))$ for the open disc $B_\delta(\gamma)$ centered at $\gamma = (0.5, 0.5)$ with redius δ. However, u has the following expression in $B_\delta(\gamma)$

$$u(r,\theta) = K(\gamma) r^{2/3} \cos(2\theta/3) + u_R, \qquad u_R \in H^2(B_\delta(\gamma) \cap \Omega)$$

where $r = |x - \gamma|$ and $0 \le \theta \le 3\pi/2$ is the angle from the segment `bc` to $\overrightarrow{\gamma x}$ and $K(\gamma)$ is a constant.

```
border ba(t=0,1.0){x=t;   y=0;  label=1;};//  bottom
border bb(t=0,0.5){x=1;   y=t;  label=2;};
border bc(t=0,0.5){x=1-t; y=0.5;label=3;};
border bd(t=0.5,1){x=0.5; y=t;  label=4;};
border be(t=0.5,1){x=1-t; y=1;  label=5;};
border bf(t=0.0,1){x=0;   y=1-t;label=6;};//  left
mesh Th = buildmesh (ba(6) + bb(4) + bc(4) +bd(4) + be(4) + bf(6));
savemesh(Th,"th.msh");
fespace Vh(Th,P1);
fespace Nh(Th,P0);
Vh u,v;
real error=0.01;
```

```
func f=(x-y);
problem Probem1(u,v,solver=CG,eps=1.0e-6) =
    int2d(Th,qforder=2)( u*v*1.0e-10 + dx(u)*dx(v) + dy(u)*dy(v))
  - int2d(Th,qforder=2)( f*v);

Nh eta;
varf indicator2(uu,eta) =
 intalledges(Th)(eta*lenEdge*square(jump(N.x*dx(u)+N.y*dy(u))))
 +int2d(Th)(eta*square(f*hTriangle));
for (int i=0;i< 4;i++)
{
  Probem1;
   cout << u[].min << " " << u[].max << endl;
   plot(u,wait=1);
   cout << " indicator2 " << endl;

   eta[] = indicator2(0,Nh);
   eta=sqrt(eta);
   cout << eta[].min << " " << eta[].max << endl;
   plot(eta,fill=1,wait=1);
   Th=adaptmesh(Th,u,err=error);
   plot(Th,wait=1);
   u=u;
   eta=eta;
  error = error/2;
} ;
```

Here, the maximum and minimum of P_0 function η in the loop just above is

```
0.00487519 0.0360467
0.000789029 0.0196461
0.000624047 0.0119285
0.00024993 0.00642755
```

2.3.4 Trunc

A small operator

$$\texttt{mesh}\ \ \mathcal{T}_h' = \texttt{trunc}(\mathcal{T}_h, CONDITION, parameters);$$

create a truncated mesh $\mathcal{T}_h'$ from a mesh $\mathcal{T}_h$ with respect to "CONDITION = true".

The two named parameter

label= sets the label number of new boundary item (one by default)

split= sets the level n of triangle splitting. each triangle is splitted in $n \times n$ (one by default).

To create the mesh `Th3` where alls triangles of a mesh `Th` are splitted in 3×3 , just write:

```
mesh Th3 = trunc(Th,1,split=3);
```

Example 32 (`truncmesh.edp`) *The exemple construct all "trunc" mesh to the support of the basic function of the space* `Vh` *(cf.* `u>0`*), split all the triangles in* 5×5*, and put a label number to* 2 *on new boundary.*

```
mesh Th=square(3,3);
fespace Vh(Th,P1);
Vh u;
int i,n=u.n;
u=0;
for (i=0;i<n;i++)  //  all degre of freedom
 {
  u[][i]=1;        //   the basic function i
  plot(u,wait=1);
  mesh Sh1=trunc(Th,abs(u)>1.e-10,split=5,label=2);
  plot(Th,Sh1,wait=1,ps="trunc"+i+".eps");  //   plot the mesh  the func-
tion's support
  u[][i]=0;      //  reset
 }
```

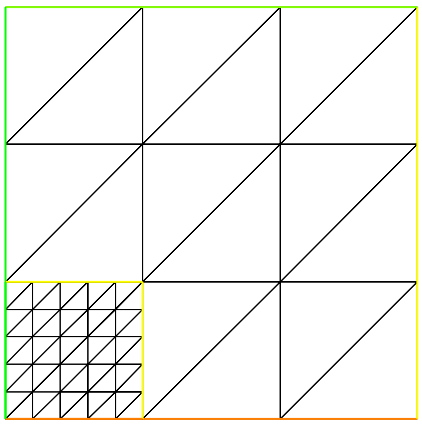


Figure 2.11: mesh of support the function P1 number 0, splitted in 5×5

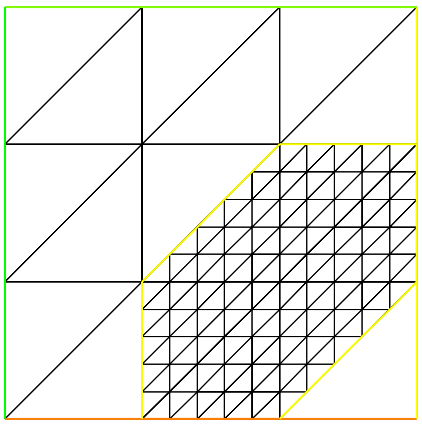


Figure 2.12: mesh of support the function P1 number 6, splitted in 5×5

2.3.5 Meshing examples

Here, we pick up characteristic meshes, which give some really useful advice on users.

Example 33 (Domain with L-shape) *This is the famous test domain in which corner singularity occur at* $bc \cap bd$*.*

```
border ba(t=0,1){x=t;y=0;label=1;};
border bb(t=0,0.5){x=1;y=t;label=1;};
border bc(t=0,0.5){x=1-t;y=0.5;label=1;};
```

```
border bd(t=0.5,1){x=0.5;y=t;label=1;};
border be(t=0.5,1){x=1-t;y=1;label=1;};
border bf(t=0,1){x=0;y=1-t;label=1;};

mesh rh = buildmesh (ba(6)+bb(4)+bc(4)+bd(4)+be(4)+bf(6));
plot(rh,ps="L-shape.eps"); //  Fig. 2.13
```

Example 34 (Two rectangles touching by a side) *A domain Ω_1 touch with another domain Ω_2, whose boundaries are given such as $\partial\Omega_1 = a \cup b \cup c \cup d$ and $\partial\Omega_1 = h \cup e \cup f \cup g$. The contact boundary is $b = h$. The domain Ω_1 is used in Deformation of Beam (see Section 3.9) and $\Omega_1 \cup \Omega_2$ is used in Fluid-structure Interaction (see Section 3.11).*

```
border a(t=2,0)  { x=0; y=t ;} //   left of beam
border b(t=0,10) { x=t; y=0 ;} //  bottom
border c(t=0,2)  { x=10; y=t ;}  //   rigth
border d(t=0,10) { x=10-t; y=2;} //   top
border h(t=0,10) {x=t ; y=0;}    //  top of water bath
border e(t=0,10) { x=t; y=-10;}  //  bottom
border f(t=0,10) { x=10; y=-10+t;} //  right
border g(t=0,10) { x=0; y=-t;}  //  left
int n=1;
mesh th = buildmesh(a(10*n)+b(10*n)+c(10*n)+d(10*n));
mesh TH = buildmesh ( h(10*n) + e(5*n) + f(10*n) + g(5*n) );
plot(th,TH,ps="TouchSide.esp"); //  Fig. 2.14
```

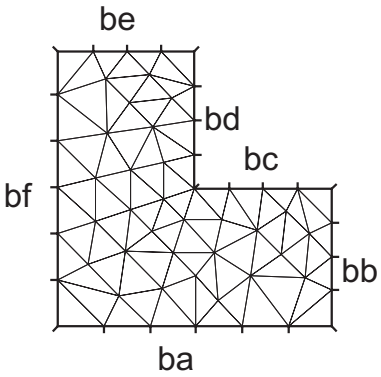


Figure 2.13: Domain with L-shape

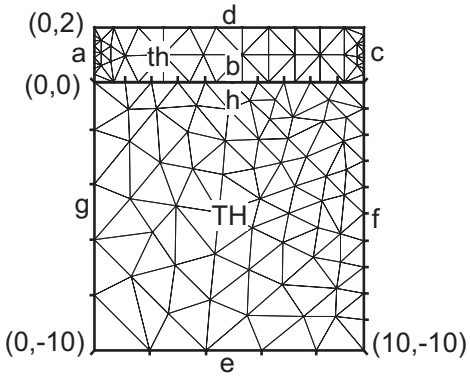


Figure 2.14: Two rectangles touching by a side

Example 35 (Meshes for domain decompositions with overlapping) *Test domain used in Schwarz algorithm for domain decomposition with overlapping (see Section 3.7.2). Domain decomposition method (DDM) is an important field in parallisation of PDE. A domain Ω is decomposed to 2 overlapping domains Ω_1 and Ω_2 such as $\Omega_1 \cap \Omega_2 \neq \emptyset$ whose meshes are* `TH` *and* `th` *respectively. Here the inside part of boundaries $\partial\Omega_1 \cap \Omega_2$ and $\Omega_1 \cap \partial\Omega_1$ are labeled with inside/outside.*

```
int inside = 2;  //   inside boundary
int outside = 1; //   outside boundary
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
border e(t=0, pi/2){ x= cos(t); y = sin(t);label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=outside;};
int n=2;
mesh th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh( e(5*n) + e1(25*n) );
plot(th,TH,ps="schwarz-th.ps",bw=1);  //  Fig. 2.15
```

Example 36 (Meshes for domain decompositions without overlapping) *Test domain used in Schwarz DDM without overlapping (see Section 3.7.3).* $\Omega_1 \cap \Omega_2 = \emptyset$ *and* $\partial\Omega_1 \cap \partial\Omega_2 = d = e$.

```
int outside = 1, inside = 2;
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
border e(t=0, 1){ x= 1-t; y = t;label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=outside;};
int n=2;
mesh th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh ( e(5*n) + e1(25*n) );
plot(th,TH,wait=1,ps="schwarz-no-th.eps",bw=1);
```

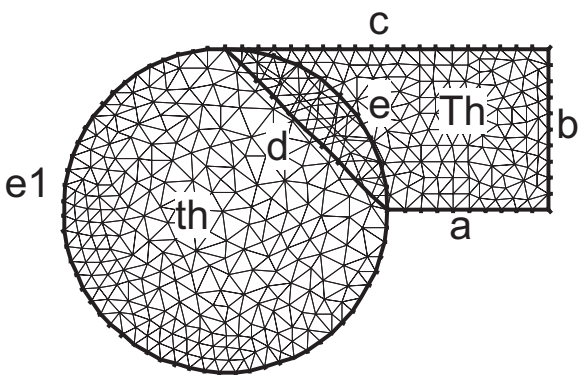


Figure 2.15: Test domain for Schwarz DDM with overlapping

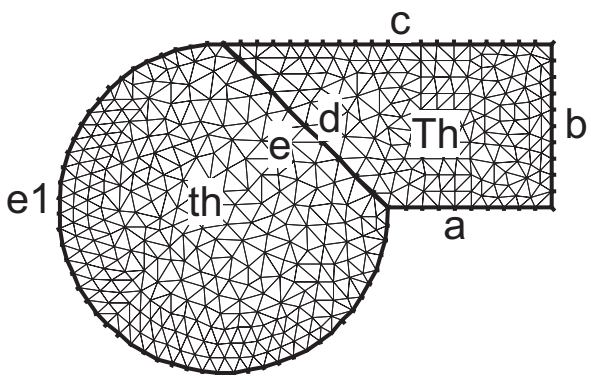


Figure 2.16: Test domain for Schwarz DDM without overlapping

Example 37 (NASA0012 Airfoil)

```
//  a NACA0012 airfoil.
border upper(t=0,1) { x = t;
     y = 0.17735*sqrt(t)-0.075597*t
- 0.212836*(t^2)+0.17363*(t^3)-0.06254*(t^4); }
border lower(t=1,0) { x = t;
     y= -(0.17735*sqrt(t)-0.075597*t
-0.212836*(t^2)+0.17363*(t^3)-0.06254*(t^4)); }
border c(t=0,2*pi) { x=0.8*cos(t)+0.5;  y=0.8*sin(t); }
mesh Th = buildmesh(c(30)+upper(35)+lower(35));
plot(Th,ps="NACA0012.eps",bw=1);  //  Fig. 2.17
```

Example 38 (Cardioid)

```
real b = 1, a = b;
border C(t=0,2*pi) { x=(a+b)*cos(t)-b*cos((a+b)*t/b);
                          y=(a+b)*sin(t)-b*sin((a+b)*t/b); }
mesh Th = buildmesh(C(50));
plot(Th,ps="Cardioid.eps",bw=1); //  Fig. 2.18
```

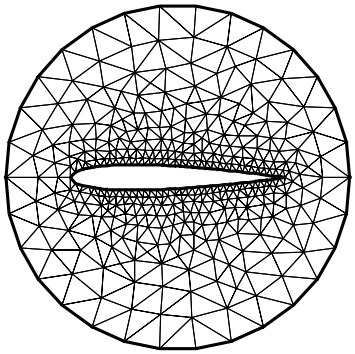


Figure 2.17: NACA0012 Airfoil

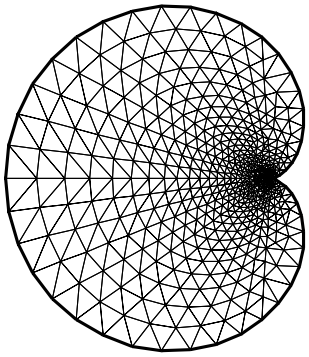


Figure 2.18: Domain with Cardioid curve boundary

Example 39 (Cycloid Curve)

```
real b = 1, a = 4*b;
border C(t=0,2*pi) { x=(a+b)*cos(t)-b*cos((a+b)*t/b);
                          y=(a+b)*sin(t)-b*sin((a+b)*t/b); }
border C0(t=0,2*pi) { x=cos(t); y=sin(t); }
mesh Th = buildmesh(C(50)); //  without auxiliary curve
mesh th = buildmesh(C(50)+C0(20));
plot(Th,ps="Cycloid1.eps",bw=1); //  Fig. 2.19
plot(th,ps="Cycloid2.eps",bw=1); //  Fig. 2.20
```

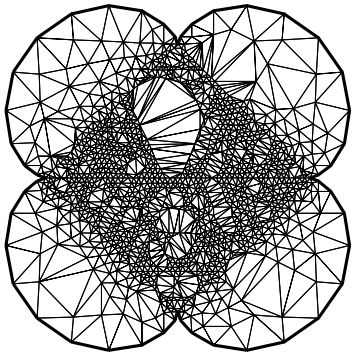


Figure 2.19: Domain with Cycloid curve boundary

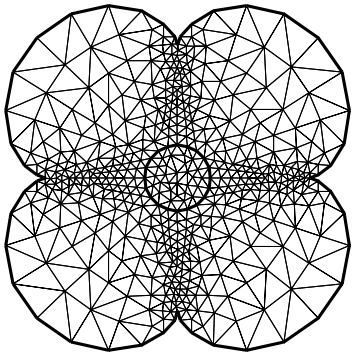


Figure 2.20: Inside, auxiliary curve `C0` is added

Example 40 (By cubic Bezier curve)

```
//  A cubic Bezier curve connecting two points with two control points
func real bzi(real p0,real p1,real q1,real q2,real t)
{
  return p0*(1-t)^3+q1*3*(1-t)^2*t+q2*3*(1-t)*t^2+p1*t^3;
}

real[int] p00=[0,1], p01=[0,-1], q00=[-2,0.1], q01=[-2,-0.5];
real[int] p11=[1,-0.9], q10=[0.1,-0.95], q11=[0.5,-1];
real[int] p21=[2,0.7], q20=[3,-0.4], q21=[4,0.5];
real[int] q30=[0.5,1.1], q31=[1.5,1.2];
border G1(t=0,1) { x=bzi(p00[0],p01[0],q00[0],q01[0],t);
                   y=bzi(p00[1],p01[1],q00[1],q01[1],t); }
border G2(t=0,1) { x=bzi(p01[0],p11[0],q10[0],q11[0],t);
                   y=bzi(p01[1],p11[1],q10[1],q11[1],t); }
border G3(t=0,1) { x=bzi(p11[0],p21[0],q20[0],q21[0],t);
                   y=bzi(p11[1],p21[1],q20[1],q21[1],t); }
border G4(t=0,1) { x=bzi(p21[0],p00[0],q30[0],q31[0],t);
                   y=bzi(p21[1],p00[1],q30[1],q31[1],t); }
int m=5;
mesh Th = buildmesh(G1(2*m)+G2(m)+G3(3*m)+G4(m));
plot(Th,ps="Bezier.eps",bw=1);  //  Fig 2.21
```

Example 41 (Section of Engine)

```
real a= 6., b= 1., c=0.5;
border L1(t=0,1) { x= -a; y= 1+b - 2*(1+b)*t; }
border L2(t=0,1) { x= -a+2*a*t; y= -1-b*(x/a)*(x/a)*(3-2*abs(x)/a );}
border L3(t=0,1) { x= a; y=-1-b + (1+ b )*t; }
border L4(t=0,1) { x= a - a*t;   y=0; }
border L5(t=0,pi) { x= -c*sin(t)/2; y=c/2-c*cos(t)/2; }
border L6(t=0,1) { x= a*t;  y=c; }
border L7(t=0,1) { x= a;  y=c + (1+ b-c )*t; }
border L8(t=0,1) { x= a-2*a*t; y= 1+b*(x/a)*(x/a)*(3-2*abs(x)/a); }
```

```
mesh Th = buildmesh(L1(8)+L2(26)+L3(8)+L4(20)+L5(8)+L6(30)+L7(8)+L8(30));
plot(Th,ps="Engine.eps",bw=1); //  Fig. 2.22
```

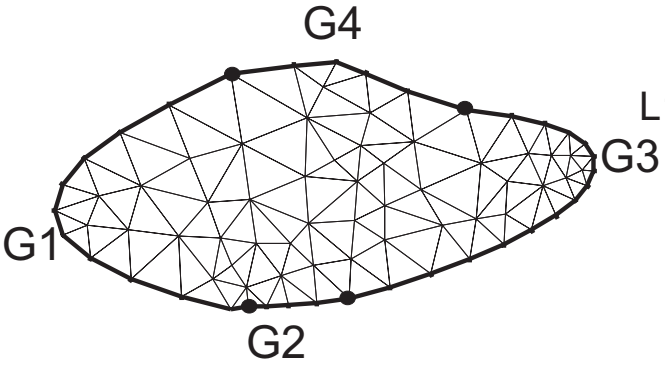


Figure 2.21: Boundary drawed by Bezier curves

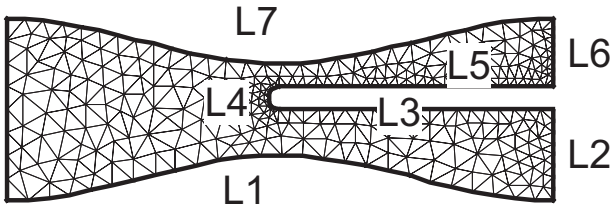


Figure 2.22: Section of Engine

Example 42 (Smiling face)

```
real d=0.1;
int m=5;
real a=1.5, b=2, c=0.7, e=0.01;
border F(t=0,2*pi) { x=a*cos(t); y=b*sin(t); }
border E1(t=0,2*pi) { x=0.2*cos(t)-0.5; y=0.2*sin(t)+0.5; }
border E2(t=0,2*pi) { x=0.2*cos(t)+0.5; y=0.2*sin(t)+0.5; }
func real st(real t) {
   return sin(pi*t)-pi/2;
}
border C1(t=-0.5,0.5) { x=(1-d)*c*cos(st(t)); y=(1-d)*c*sin(st(t)); }
border C2(t=0,1){x=((1-d)+d*t)*c*cos(st(0.5));y=((1-d)+d*t)*c*sin(st(0.5));}
border C3(t=0.5,-0.5) { x=c*cos(st(t)); y=c*sin(st(t)); }
border C4(t=0,1) { x=(1-d*t)*c*cos(st(-0.5)); y=(1-d*t)*c*sin(st(-0.5));}

border C0(t=0,2*pi) { x=0.1*cos(t); y=0.1*sin(t); }
mesh Th=buildmesh(F(10*m)+C1(2*m)+C2(3)+C3(2*m)+C4(3)
                  +C0(m)+E1(-2*m)+E2(-2*m));
plot(Th,ps="SmileFace.eps",bw=1);  //  see Fig. 2.23
}
```

Example 43 (Domain with U-shape channel)

```
real d = 0.1; //  width of U-shape
border L1(t=0,1-d) { x=-1; y=-d-t; }
border L2(t=0,1-d) { x=-1; y=1-t; }
border B(t=0,2) { x=-1+t; y=-1; }
border C1(t=0,1) { x=t-1; y=d; }
```

```
border C2(t=0,2*d) { x=0; y=d-t; }
border C3(t=0,1) { x=-t; y=-d; }
border R(t=0,2) { x=1; y=-1+t; }
border T(t=0,2) { x=1-t; y=1; }
int n = 5;
mesh Th = buildmesh (L1(n/2)+L2(n/2)+B(n)+C1(n)+C2(3)+C3(n)+R(n)+T(n));
plot(Th,ps="U-shape.eps",bw=1); //  Fig 2.24
```

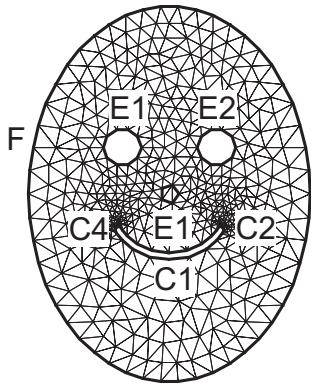


Figure 2.23: Smiling face (Mouth is changeable)

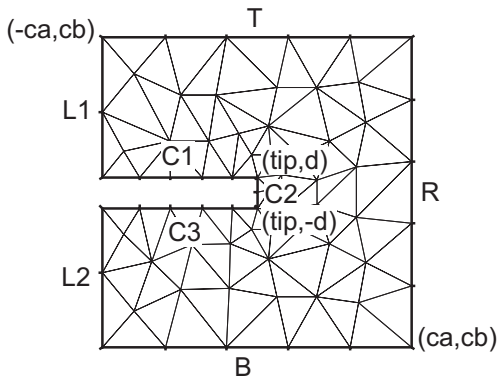


Figure 2.24: Domain with U-shape channel changed by `d`

2.4 Finite Elements

To use a finite element, one needs to define a finite element space with the keyword `fespace` (short of finite element space) like

```
fespace IDspace(IDmesh,<IDFE>)
```

or with k pair of periodic boundary condition

```
fespace IDspace(  IDmesh,<IDFE>,
                  periodic=[[la1,sa1],[lb1,sb1],...,[lak,sak],[lbk,sbk]])
```

where `IDspace` is the name of the space for example `Vh`, `IDmesh` is the name of the associated mesh and `<IDFE>` is a identifier of finite element type, where a pair of periodic boundary condition is defined by `[la`$_i$`,sa`$_i$`],[lb`$_i$`,sb`$_i$`]`. The `int` expressions `la`$_i$ and `lb`$_i$ are defined the 2 labels of the piece of the boundary to be equivalence, and the `real` expressions `sa`$_i$ and `sb`$_i$ a give two common abcissa on the two boundary curve.

As of today, the known types of finite element are

P0 piecewise constante discontinuous finite element

$$P0_h = \{v \in L^2(\Omega) : \forall K \in \mathcal{T}_h \quad v_{|K} = \alpha_K \text{constant of } \mathbb{R}\} \tag{2.4}$$

P1 piecewise linear continuous finite element

$$P1_h = \{v \in H^1(\Omega) : \ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_1\} \tag{2.5}$$

P2 piecewise P_2 continuous finite element,

$$P2_h = \{v \in H^1(\Omega) : \ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_2\} \tag{2.6}$$

where P_2 is the set of polynomials of $\mathbb{R}^2$ of degrees at most 2.

RT0 Raviart-Thomas finite element

$$RT0_h = \{\mathbf{v} \in H(div) : \ \forall K \in \mathcal{T}_h \quad \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix}\alpha_K\\ \beta_K\end{smallmatrix}\right. + \gamma_K \left|\begin{smallmatrix}x\\ y\end{smallmatrix}\right. \} \tag{2.7}$$

where $H(div)$ is the set of function of $L^2(\Omega)$ with divergence in $L^2(\Omega)$, and where $\alpha_K, \beta_K, \gamma_K$ are real numbers.

P1nc piecewise linear element continuous at the middle of edge only.

To define the finite element spaces

$$X_h = \{v \in H^1(]0,1[^2) : \ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_1\}$$

$$X_{ph} = \{v \in X_h : \ v(\left|\begin{smallmatrix}0\\ \cdot\end{smallmatrix}\right.) = v(\left|\begin{smallmatrix}1\\ \cdot\end{smallmatrix}\right.), v(\left|\begin{smallmatrix}\cdot\\ 0\end{smallmatrix}\right.) = v(\left|\begin{smallmatrix}\cdot\\ 1\end{smallmatrix}\right.)\}$$

$$M_h = \{v \in H^1(]0,1[^2) : \ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_2\}$$

$$R_h = \{\mathbf{v} \in H^1(]0,1[^2)^2 : \ \forall K \in \mathcal{T}_h \quad \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix}\alpha_K\\ \beta_K\end{smallmatrix}\right. + \gamma_K \left|\begin{smallmatrix}x\\ y\end{smallmatrix}\right. \}$$

where $\mathcal{T}$ is a mesh 10×10 of the unit square $]0,1[^2$,

the corresponding `freefem++` definitions are:

```
mesh Th=square(10,10);  //  border label: 1 down, 2 left, 3 up, 4 right
fespace Xh(Th,P1);      //   scalar FE
fespace Xph(Th,P1,periodic=[[2,y],[4,y],[1,x],[3,x]]);   //   bi-periodic
FE-space
fespace Mh(Th,P2);      //   scalar FE
fespace Rh(Th,RT0);     //   vectorial FE
```

so `Xh,Mh,Rh` are finite element spaces (called FE-spaces). Now to use functions $u_h, v_h \in X_h$ and $p_h, q_h \in M_h$ and $U_h, V_h \in \mathbb{R}_h$ one can define the FE-function like this

```
Xh uh,vh;
Xph uph,vph;
Mh ph,qh;
Rh [Uxh,Uyh],[Vxh,Vyh];
Xh[int] Uh(10); //   array of 10 function in Xh
Rh[int] [Wxh,Wyh](10); //   array of 10 functions in Rh.
```

The functions U_h, V_h have two components so we have

$$U_h = \left|\begin{smallmatrix}Uxh\\ Uyh\end{smallmatrix}\right. \quad \text{and} \quad V_h = \left|\begin{smallmatrix}Vxh\\ Vyh\end{smallmatrix}\right.$$

.

Like in the previous version, freefem+, the finite element functions (type FE functions) are both functions from $\mathbb{R}^2$ to $\mathbb{R}^N$ and arrays of real.

To interpolate a function, one writes

```
 uh = x^2 + y^2;                //   ok uh is scalar FE-function
[Uxh,Uyh] = [sin(x),cos(y)];    //  ok vectorial FE-function
Uxh = x;    //  error: impossible to set only 1 component of  a vec-
tor FE-function.
vh = Uxh;  //  ok
Th=square(5,5);
vh=vh;     //   re-interpolates vh on the new mesh square(5,5);
vh([x-1/2,y])= x^2 + y^2;  //  interpole vh = ((x − 1/2)^2 + Y^2)
```

To get the value at a point $x = 1, y = 2$ of the FE-function uh, or uh,[Uxh,Uyh],one writes

```
real value;
value = uh(2,4);       //   get value= uh(2,4)
value = Uxh(2,4);      //  get value= Uxh(2,4)
//   ------  or ------
x=1;y=2;
value = uh;       //  get value= uh(1,2)
value = Uxh;      //  get value= Uxh(1,2)
value = Uyh;      //  get value= Uyh(1,2).
```

To get the value of the array associated to the FE-function uh, one writes

```
real value = uh[][0] ; //  get the value of degree of freedom 0
real maxdf = uh[].max; //   maximal value of degree of freedom
int size = uh.n; //  the number of degree of freedom
real[int] array(uh.n)= uh[]; //   copy the array of the function uh
```

The other way to set a FE-function is to solve a 'problem' (see below).

Remark 14 *It is possible to change a mesh to do a convergence test for example, but see what happens in this trivial example. In fact a FE-function is three pointers, one pointer to the values, second a pointer to the definition of* fespace*, third a pointer to the* fespace*. This* fespace *is defined when the FE-function is set with operator* = *or when a* problem *is solved.*

Example 44

```
mesh Th=square(2,2);
fespace Xh(Th,P1);
 Xh uh,vh;
 vh= x^2+y^2;  //  vh
 Th = square(5,5); //  change the mesh
 //   Xh is unchange
 uh = x^2+y^2; //  compute on the new Xh
```

```
//  and now uh use the 5x5 mesh
//  but the fespace of vh is alway the 2x2 mesh
plot(vh,ps="onoldmesh.eps");  //  figure 2.25
vh = vh; //  do a interpolation of vh (old) of 5x5 mesh
         //  to get the new vh on 10x10 mesh.
plot(vh,ps="onnewmesh.eps"); //  figure 2.26
```

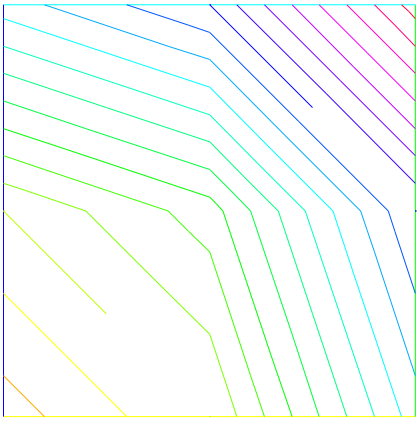


Figure 2.25: vh Iso on mesh 2×2

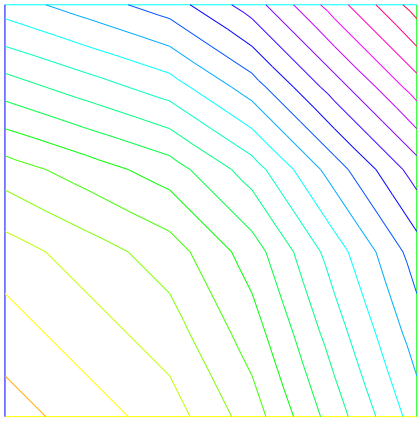


Figure 2.26: vh Iso on mesh 5×5

2.4.1 A Fast Finite Element Interpolator

In practice one may discretize the variational equations by the Finite Element method. Then there will be one mesh for Ω_1 and another one for Ω_2. The computation of integrals of products of functions defined on different meshes is difficult. Quadrature formulae and interpolations from one mesh to another at quadrature points are needed. We present below the interpolation operator which we have used and which is new, to the best of our knowledge. Let $\mathcal{T}_h^0 = \cup_k T_k^0, \mathcal{T}_h^1 = \cup_k T_k^1$ be two triangulations of a domain Ω. Let

$$V(\mathcal{T}_h^i) = \{C^0(\Omega_h^i) \ : \ f|_{T_k^i} \in P^1\}, \quad i = 0, 1$$

be the spaces of continuous piecewise affine functions on each triangulation.

Let $f \in V(\mathcal{T}_h^0)$. The problem is to find $g \in V(\mathcal{T}_h^1)$ such that

$$g(q) = f(q) \quad \forall q \text{ vertex of } \mathcal{T}_h^1$$

Although this is a seemingly simple problem, it is difficult to find an efficient algorithm in practice. We propose an algorithm which is of complexity $N^1 \log N^0$, where N^i is the number of vertices of $\mathcal{T}_h^i$, and which is very fast for most practical 2D applications.

Algorithm

The method has 5 steps. First a quadtree is built containing all the vertices of mesh $\mathcal{T}_h^0$ such that in each terminal cell there are at least one, and at most 4, vertices of $\mathcal{T}_h^0$.

For each q^1, vertex of $\mathcal{T}_h^1$ do:

Step 1 Find the terminal cell of the quadtree containing q^1.

Step 2 Find the the nearest vertex q_j^0 to q^1 in that cell.

Step 3 Choose one triangle $T_k^0 \in \mathcal{T}_h^0$ which has q_j^0 for vertex.

Step 4 Compute the barycentric coordinates $\{\lambda_j\}_{j=1,2,3}$ of q^1 in T_k^0.

- – if all barycentric coordinates are positive, go to Step 5
- – else if one barycentric coordinate λ_i is negative replace T_k^0 by the adjacent triangle opposite q_i^0 and go to Step 4.
- – else two barycentric coordinates are negative so take one of the two randomly and replace T_k^0 by the adjacent triangle as above.

Step 5 Calculate $g(q^1)$ on T_k^0 by linear interpolation of f:

$$g(q^1) = \sum_{j=1,2,3} \lambda_j f(q_j^0)$$

End

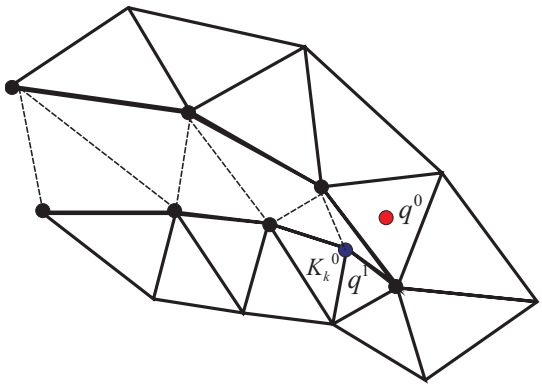


Figure 2.27: To interpolate a function at q^0 the knowledge of the triangle which contains q^0 is needed. The algorithm may start at $q^1 \in T_k^0$ and stall on the boundary (thick line) because the line q^0q^1 is not inside Ω. But if the holes are triangulated too (doted line) then the problem does not arise.

Two problems needs to solved:

- *What if q^1 is not in Ω_h^0 ?* Then Step 5 will stop with a boundary triangle. So we add a step which test the distance of q^1 with the two adjacent boundary edges and select the nearest, and so on till the distance grows.

- *What if Ω_h^0 is not convex and the marching process of Step 4 locks on a boundary?* By construction Delaunay-Voronoi mesh generators always triangulate the convex hull of the vertices of the domain. So we make sure that this information is not lost when $\mathcal{T}_h^0, \mathcal{T}_h^1$ are constructed and we keep the triangles which are outside the domain in a special list. Hence in step 5 we can use that list to step over holes if needed.

Remark Step 3 requires an array of pointers such that each vertex points to one triangle of the triangulation.

2.4.2 Problem and Solve

As shown in Section 1.1.1, the Poisson equation with Dirichelt boundary condition (see (1.1)),

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega. \tag{2.8}$$

become the variational formula: For a given f, find u satisfying $u = 0$ on $\partial\Omega$ and

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega fv$$

for all test functions v ($v = 0$ on $\partial\Omega$). The variational form is divided into the bilinear form

$$\int_\Omega \{(\partial u/\partial x)(\partial v/\partial x) + (\partial u/\partial y)(\partial v/\partial y)\}$$

the linear form $\int_\Omega fv$ and the boundary condition "$u = 0$ on $\partial\Omega$".

The step to solve (2.8) is the following:

Triangulation(Mesh): (see Section 2.3) Define the boundary of Ω by **border**. Make the triangulation $\mathcal{T}_h$ of Ω by **mesh- buildmesh**. If Ω is a square, we can easity make the mesh by **mesh- square**.

FE-space: (see Section 2.4) Define FE-space V_h by **fespace** where you want to solve it.

Unknown and test functions: Let the unknown function u and test functions v be in V_h.

Given function: Define the given function f (see Section 2.1.9).

Problem Define the variational form as the sum of *bilinear form(s), linear form(s)* and *boundary condition(s)* using **problem**.

Solve Write the name of the problem given in previous step, then the problem is solved. The solution u is given as the FE-function in V_h.

Problem+Solve We can use **solve** if we want at the same time to formulate the problem and to solve it (see following example).

Example 45

```
mesh Th=square(10,10);
fespace Vh(Th,P1);      //  P1 FE-space
Vh uh,vh;      //  unknown and test function.
```

```
    func f=1;    //   right hand side function
    func g=0;    //   boundary condition function

    solve laplace(uh,vh) =   //   definion of  the problem and solve
       int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) ) //   bilinear form
     + int2d(Th)( -f*vh )        //   linear form
     + on(1,2,3,4,uh=g) ;        //   a lock boundary condition form

    f=x+y;   //  change the given function
    laplace; //  solve again the problem with this new f
    plot(uh,ps="Laplace.eps",value=true); //  to see the result
```

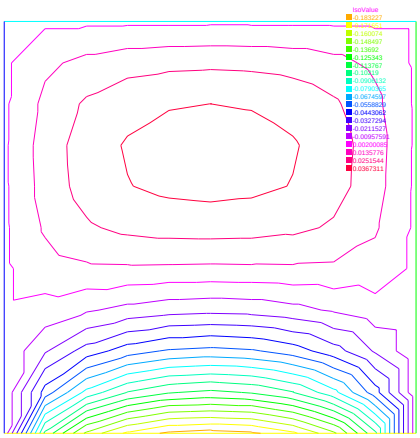


Figure 2.28: Isovalues of the solution

Example 46 *A laplacian in mixed finite formulation .*

```
   mesh Th=square(10,10);
   fespace Vh(Th,RT0);
   fespace Ph(Th,P0);

   Vh [u1,u2],[v1,v2];
   Ph p,q;

   problem laplaceMixte([u1,u2,p],[v1,v2,q],solver=LU,eps=1.0e-30) =
      int2d(Th)( p*q*1e-10+ u1*v1 + u2*v2 + p*(dx(v1)+dy(v2)) + (dx(u1)+dy(u2))*q
)
    + int2d(Th) ( q)
    + int1d(Th)( (v1*N.x +v2*N.y)/-2);   //   int on gamma

    laplaceMixte;  //  the problem is now solved
    plot([u1,u2],coef=0.1,wait=1,ps="lapRTuv.eps",value=true); //  figure
2.29
    plot(p,fill=1,wait=1,ps="laRTp.eps",value=true);    //  figure 2.30
```

Remark 15 *To make more readable programs we stop now using blue color on dx,dy.*

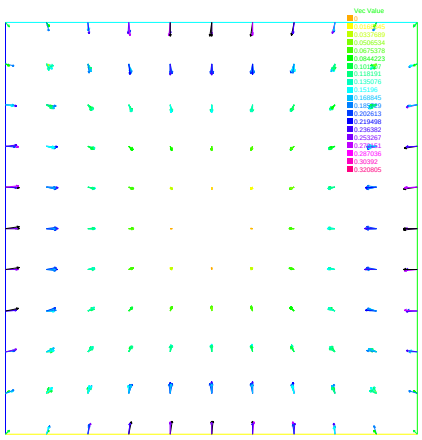


Figure 2.29: Flux (u_1, u_2)

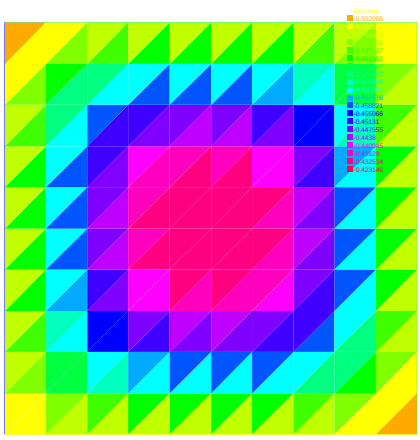


Figure 2.30: Isovalue of p

2.4.3 Parameter Description for `solve` and `problem`

The parameters are FE-function, the number n of FE-function is even ($n = 2 * k$), the k first function parameters are unknown, and the k last are test functions.

Remark 16 *If the functions are a part of vectorial FE then you must give all the functions of the vectorial FE in the same order (see laplaceMixte problem for example).*

Bug: 1 *The mixing of* `fespace` *with differents periodic boundary condition is not implemented. So all the finite element space use for test or unknow functions in a problem, must have the same type of periodic boundary condition or no periodic boundary condition. No clean message is given and the result is impredictible, Sorry.*

The named parameters are:

solver= `LU`, `CG`, `Crout`,`Cholesky`,`GMRES` ...

The default solver is `LU`. The storage mode of the matrix of the underlying linear system depends on the type of solver chosen; for `LU` the matrix is sky-line non symmetric, for `Crout` the matrix is sky-line symmetric, for `Cholesky` the matrix is sky-line symmetric positive definite, for `CG` the matrix is sparse symmetric positive, and for `GMRES` the matrix is just sparse.

eps= a real expression. ε sets the stopping test for the iterative methods like `CG`. Note that if ε is negative then the stopping test is:

$$||Ax - b|| < |\varepsilon|$$

if it is positive then the stopping test is

$$||Ax - b|| < \frac{|\varepsilon|}{||Ax_0 - b||}$$

init= boolean expression, if it is false or 0 the matrix is reconstructed. Note that if the mesh changes the matrix is reconstructed too.

precon= name of a function (for example `P`) to set the precondioner. The prototype for the function `P` must be

```
func real[int]  P(real[int] & xx) ;
```

tgv= Huge value, for lock boundary conditions

dimKrylov= dimension of the Krylov space.

2.4.4 Problem definition

Below `v` is the unknown function and `vv` is the test function.

After the "=" sign, one may find sums of:

- a name; this is the name given to the variational form (type `varf`) for possible reuse.
- A bilinear form `int1d(Th)( K*v*vv)` , `int1d(Th,2,5)( K*v*vv)` , a sparce matrix of type `matrix`
- A linear form `int1d(Th)( K*v)` , `int1d(Th,2,5)( K*v)` , a vector of type `real[int]`
- The boundary condition form
  - An "on" form (for Dirichlet) : `on(1, u = g )`
  - a linear form on Γ (for Neumann) `int1d(Th))( -f*vv)` or `int1d(Th,3))( -f*vv)`
  - a bilinear form on Γ or Γ_2 (for Robin) `int1d(Th))( K*v*vv)` or `int1d(Th,2))( K*v*vv)`.

If needed, the different kind of terms in the sum can appear more than once.

Remark: the integral mesh and the mesh associated to test function or unkwon fonction can be different in the case of linear form.

Remark 17 *`N.x` and `N.y` are the normal's components.*

Important: it is not possible to write in the same integral the linear part and the bilinear part such as in `int1d(Th)( K*v*vv - f*vv )`.

2.4.5 Integrals

Let Ω be a domain in $\mathbb{R}^2$ with boundary $\partial\Omega = \cup_{j=1}^{J}\Gamma_j$, where the curves are numbered in such a way that Γ_{j+1} follows Γ_j according to the positive orientation and $\Gamma_j \cap \Gamma_{j+1}$ is a point. If we have a mesh $\mathcal{T}_h$ of Ω, we then use three kinds of integrals:

- surface integral defined with the keyword `int2d`

$$\texttt{int2d}(\mathcal{T}_h)(\texttt{FE-function});$$

- integrals on the curve Γ_j,

$$\texttt{int1d}(\mathcal{T}_h, \Gamma_j)(\texttt{FE-function});$$

or on the curves $\Gamma_j, \cdots, \Gamma_\ell$

$$\texttt{int1d}(\mathcal{T}_h, \Gamma_j, \cdots, \Gamma_\ell)(\texttt{FE-function});$$

Integrals can be used to define the variational form, or to compute integrals proper. It is possible to choose the order of the integration formula by adding a parameter `qforder=` to define the order of the Gauss formula, or directly the name of the formula with `qft=name` in 2d integrals and `qfe=name` in 1d integrals.

The integration formulae on triangles are:

name (`qft=`)	on	order `qforder=`	exact	number of quadrature points
`qf1pT`	triangle	2	1	1
`qf2pT`	triangle	3	2	3
`qf3pT`	triangle	6	4	7

The integration formulae on edges are:

name (`qfe=`)	on	order `qforder=`	exact	number of quadrature points
`qf1pE`	segment	2	1	1
`qf2pE`	segment	3	2	2
`qf3pE`	segment	6	5	3

Integral on all edges $T_i \in \mathcal{T}_h$ $(i = 1, \cdots, n_t)$ of FE-function f

$$\sum_{i=1}^{n_t} \int_{\partial T_i} f$$

is calculated by

$$\texttt{intalledges}(\mathcal{T}_h)(f);$$

For example, we use this the mesh given in Fig. 2.4

```
mesh Th = square(1,2);
cout << intalledges(Th)(1) << endl;
```

The caluculated value is 10.4721 $(\sim 4 \times (1 + 0.5 + \sqrt{1 + 0.5^2})$.

2.4.6 Variational Form, Sparse Matrix, Right Hand Side Vector

It is possible to define variational forms:

```
mesh Th=square(10,10);
fespace Xh(Th,P2),Mh(Th,P1);
Xh u1,u2,v1,v2;
Mh p,q,ppp;
```

```
varf bx(u1,q) = int2d(Th)( (dx(u1)*q));
```

$$bx(u_1, q) = \int_{\Omega_h} \frac{\partial u_1}{\partial x} q$$

```
varf by(u1,q) = int2d(Th)( (dy(u1)*q));
```

$$by(u_1, q) = \int_{\Omega_h} \frac{\partial u_1}{\partial y} q$$

```
varf a(u1,u2)= int2d(Th)(  dx(u1)*dx(u2) + dy(u1)*dy(u2) )
                    +  on(1,2,4,u1=0)  +  on(3,u1=1) ;
```

$$a(u_1, v_2) = \int_{\Omega_h} \nabla u_1 . \nabla u_2; \qquad u_1 = 1 * g \text{ on } \Gamma_3, u_1 = 0 \text{ on } \Gamma_1 \cup \Gamma_2 \cup \Gamma_4$$

where f is defined later.
Later variational forms can be used to construct right hand side vectors, matrices associated to them, or to define a new problem;

```
Xh bc1; bc1[] = a(0,Xh);  //   right hand side for boundary condition
Xh b;

matrix A= a(Xh,Xh,solver=CG);   //  the Laplace matrix
matrix Bx= bx(Xh,Mh);    //   Bx = (Bx_ij) and  Bx_ij = bx(b_j^x, b_j^m)
//   where b_j^x is a basis of Xh, and b_j^m is a basis of Mh.
matrix By= by(Xh,Mh);    //   By = (By_ij) and  By_ij = by(b_j^x, b_j^m)
```

Remark 18 *The line of the matrix corresponding to test function on the bilinear form.*

Remark 19 *The vector bc1[] contains the contribution of the boundary condition* $u_1 = 1$.

Here we have three matrices A, Bx, By, and we can solve the problem:
find $u_1 \in X_h$ such that

$$a(v_1, u_1) = by(v_1, f), \forall v_1 \in X_{0h},$$
$$u_1 = g, \quad \text{on } \Gamma_1, \text{and} \quad u_1 = 0 \quad \text{on } \Gamma_1 \cup \Gamma_2 \cup \Gamma_4$$

with the following line (where $f = x$, and $g = sin(x)$)

```
Mh f=x;
Xh g=sin(x);
b = By*f[];  //
b += bc .* g[];    //  u1= g on Γ3 boundary see following remark
u1 = A^-1*b;   //  solve the linear system
```

Remark 20 *The boundary condition is implemented by penalization and the vector* `bc1[]` *contains the contribution of the boundary condition* $u_1 = 1$ *, so to change the boundary condition, we have just to multiply the vector bc1[] by the value f of the new boundary condition term by term with the operator* `.*`. *The* `StokesUzawa.edp` *3.6.2 gives a real example of using all this features.*

2.4.7 Plot

The command **plot** give the vizualization of results by FEM as follows:

Meshes $\mathcal{T}_h$ is visualized by "**plot**$(\mathcal{T}_h)$;"

Contour line of FE-function f is given by "**plot**(f);"

Vector fields $\vec{u} = (u_x, u_y)$ is shown by the arrows through the use of "**plot**$([u_x, u_y])$;"

Two array of double `Xcrd`(m) and `Ycrd`(m) draws a polygonal line through m points by

```
plot([Xcrd,Ycrd]);
```

where `Xcrd` expresses x-coordinate of these points and `Ycrd` y-coordinate.

Two or more visualization is possible at the same time as in

$$\textbf{plot}(\mathcal{T}_h, f);$$

Parameters of the plot command are usable on meshes, FE-functions, arrays of 2 FE-functions, arrays of two arrays of double, as in

$$\textbf{plot}(\mathcal{T}_h, f, \texttt{named parameters});$$

The named parameter are

wait= boolean expression to wait or not (by default no wait). If true we wait for a keyboard up event or mouse event, the character event can be

- **+** to zoom in around the mouse cursor,
- **-** to zoom out around the mouse cursor,
- **=** to restore de initial graphics state,
- **c** to decrease the vector arrow coef,
- **C** to increase the vector arrow coef,
- **r** to refresh the graphic window,
- **f** to toggle the filling between isovalues,
- **b** to toggle the black and white,
- **v** to toggle the plotting of value,
- **p** to save to a postscript file,
- **?** to show all actives keyboard char,

to redraw, otherwise we continue.

ps= string expression to save the plot on postscript file

coef= the vector arrow coef between arrow unit and domain unit.

fill= to fill between isovalues.

cmm= string expression to write in the graphic window

value= to plot the value of isoline and the value of vector arrow.

aspectratio= boolean to be sure that the aspect ratio of plot is preserved or not.

bb= array of 2 array (like `[[0.1,0.2],[0.5,0.6]]`), to set the bounding box and specify a partial view.

nbiso= (int) sets the number of isovalues (20 by default)

nbarrow= (int) sets the number of colors of arrow values (20 by default)

viso= sets the array value of isovalues (an array real[int])

varrow= sets the array value of color arrows (an array real[int])

bw= (bool) sets or not the plot in black and white color.

Example 47 (plot.edp)

```
real[int] xx(10),yy(10);
mesh Th=square(5,5);
fespace Vh(Th,P1);
Vh uh=x*x+y*y,vh=-y^2+x^2;
int i;
//   compute a cut
for (i=0;i<10;i++)
 {
   x=i/10.; y=i/10.;
   xx[i]=i;
   yy[i]=uh; //  value of uh at point (i/10. , i/10.)
 }
 plot(Th,uh,[uh,vh],value=true,ps="three.eps",wait=true); //  figure 2.31
 plot(Th,uh,[uh,vh],bb=[[0.1,0.2],[0.5,0.6]],wait=true); //  a zoom
 plot([xx,yy],ps="likegnu.eps",wait=true); //  figure 2.32
```

2.4.8 Convect

This operator performs one step of backward convection by the method of Characteristics-Galerkin. An equation like

$$\partial_t \phi + u\nabla\phi = 0, \quad \phi(x,0) = \phi^0(x)$$

is approximated by

$$\frac{1}{\delta t}(\phi^{n+1}(x) - \phi^n(X^n(x))) = 0$$

Roughly the term,$\phi^n o X^n$ is approximated by $\phi^n(x + u^n(x)\delta t)$. Up to quadrature errors the scheme is unconditionnally stable. The syntax is

```
<FE> = convect([<exp1>,<exp2>],<exp3>,<exp4>)
```

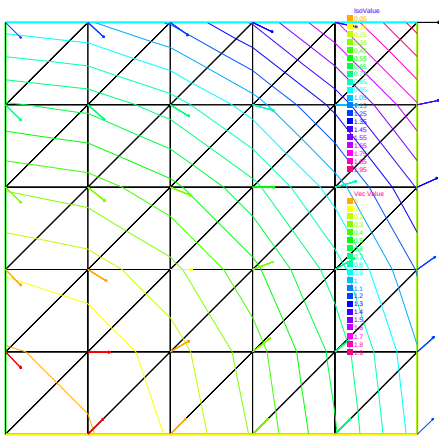


Figure 2.31: mesh, isovalue, and vector

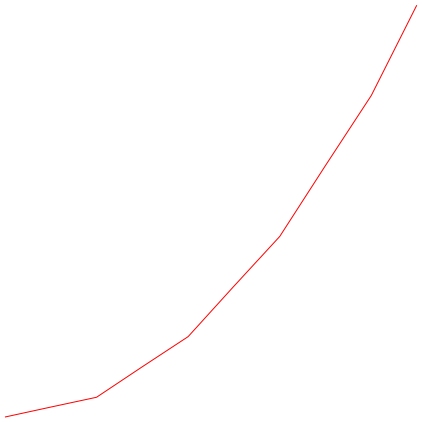


Figure 2.32: Plots a cut of uh. Note that a refinement of the same can be obtained in combination with gnuplot

<FE> is a name of finite element function to store the result $u \circ \chi$;

<exp1> is real expression of the x-velocity,

<exp2> is real expression of the y-velocity,

<exp3> is **minus**[2] the time step,

<exp4> is the name of the finite element function which is convected (u in the exemple above)

Warning `convect` is a non-local operator; in the instruction "phi=convec([u1,u2],-dt,phi0)" every values of phi0 are used to compute phi So `phi=convec([u1,u2],-dt,phi0)` won't work.

‘

[2]The minus is due to the backwark schema

Chapter 3

Mathematical models

3.1 Soap film

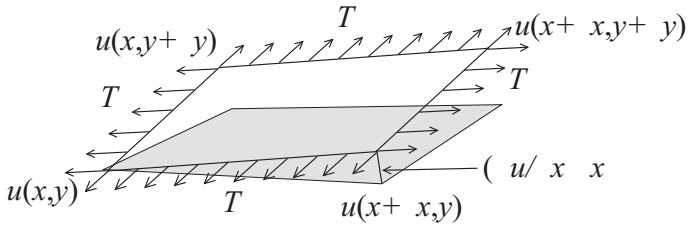
Our starting point here will be the mathematical model to find the shape of **soap film** which is glued to the ring on the $xy-$plane

$$C = \{(x, y);\ x = \cos t,\ y = \sin t,\ 0 \le t \le 2\pi\}.$$

We assume the shape of the film is descrined as the graph $(x, y, u(x, y))$ of the vertical displacement $u(x, y)$ $(x^2 + y^2 < 1)$ under a vertical pressure p in terms of force per unit area and an initial tension T in terms of force per unit length. Consider "small plane" ABCD, A:$(x, y, u(x, y))$, B:$(x, y, u(x + \delta x, y))$, C:$(x, y, u(x + \delta x, y + \delta y))$ and D:$(x, y, u(x, y + \delta y))$. Let us denote by $\nu(x, y) = (\nu_x(x, y), \nu_y(x, y), \nu_z(x, y))$ the normal vector of the surface $z = u(x, y)$. We see that the the vertical force due to the tension T acting along the edge AD is $-T\nu_x(x, y)\delta y$ and the the vertical force acting along the edge AD is

$$T\nu_x(x + \delta x, y)\delta y \simeq T\left(\nu_x(x, y) + \frac{\partial \nu_x}{\partial x}\delta x\right)(x, y)\delta y.$$

Similarly, for the edges AB and DC we have



$$-T\nu_y(x, y)\delta x, \qquad T\left(\nu_y(x, y) + \partial\nu_y/\partial y\right)(x, y)\delta x.$$

The force in the virtical direction on the surfece ABCD due to the tension T is given by the summension

$$T\left(\partial\nu_x/\partial x\right)\delta x\delta y + T\left(\partial\nu_y/\partial y\right)\delta y\delta x.$$

Assuming small displacements, we have

$$\begin{aligned}\nu_x &= (\partial u/\partial x)/\sqrt{1+(\partial u/\partial x)^2+(\partial u/\partial y)^2} \simeq \partial u/\partial x,\\ \nu_y &= (\partial u/\partial y)/\sqrt{1+(\partial u/\partial x)^2+(\partial u/\partial y)^2} \simeq \partial u/\partial y.\end{aligned}$$

Letting $\delta x \to dx$, $\delta y \to dy$, we have the equilibrium of the virtical displacement of soap film on ABCD by p

$$Tdxdy\partial^2 u/\partial x^2 + Tdxdy\partial^2 u/\partial y^2 + pdxdy = 0.$$

Using the Laplace operator $\Delta = \partial^2/\partial x^2 + \partial^2/\partial y^2$, we can find the virtual displacement write the following

$$-\Delta u = f \quad \text{in } \Omega \tag{3.1}$$

where $f = p/T$, $\Omega = \{(x,y);\ x^2+y^2<1\}$. Poisson's equation (3.1) appear also in **electrostatics** taking the form of $f = \rho/\epsilon$ where ρ is the charge density, ϵ the dielectric constant and u is named as electrostatic potential. The soap film is glued to the ring $\partial\Omega = C$, then we have the boundary condition

$$u = 0 \quad \text{on } \partial\Omega. \tag{3.2}$$

In electrostatic, (3.2) is also derived when $\partial\Omega$ is terminated to the earth. Like this, different physical problems lead similar mathematical models.

3.2 Electrostatics

We assume that there is no current and a time independent charge distribution. Then the electric field $\vec{E}$ satisfy

$$\mathrm{div}\vec{E} = \rho/\epsilon, \quad \mathrm{curl}\vec{E} = 0 \tag{3.3}$$

where ρ is the charge density and ϵ is called the permittivity of free space. From the second equation in (3.3), we can introduce the electrostatic potential such that $\vec{E} = -\nabla\phi$. Then we have Poisson equation $-\Delta\phi = f$, $f = -\rho/\epsilon$. We now obtain the equipotential line which is the level curve of ϕ, when there are no charges except conductors $\{C_i\}_{1,\cdots,K}$. Let us assume K conductors $C_1, \cdots, C_K$ within an enclosure C_0. Each one is held at an electrostatic potential φ_i. We assume that the enclosure $C0$ is held at potential 0. In order to know $\varphi(x)$ at any point x of the domain Ω, we must solve

$$-\Delta\varphi = 0 \quad \text{in}\Omega, \tag{3.4}$$

where Ω is the interior of C_0 minus the conductors C_i, and Γ is the boundary of Ω, that is $\sum_{i=0}^{N} C_i$. Here g is any function of x equal to φ_i on C_i and to 0 on C_0. The second equation is a reduced form for:

$$\varphi = \varphi_i \text{ on } C_i,\ i = 1...N, \varphi = 0 \text{ on } C_0. \tag{3.5}$$

3.2.1 The `freefem++` program

First we give the gemetical informations; $C_0 = \{(x,y);\ x^2+y^2 = 5^2\}$, $C_1 = \{(x,y) : \frac{1}{0.3^2}(x-2)^2 + \frac{1}{3^2}y^2 = 1\}$, $C_2 = \{(x,y) : \frac{1}{0.3^2}(x+2)^2 + \frac{1}{3^2}y^2 = 1\}$. Let Ω be the disk enclosed by C_0 with the elliptical holes enclosed by C_1 and C_2. Note that C_0 is described counter-clockwise, whereas the elliptical holes are described clockwise, because the boundary must be oriented so that the computational domain is to its left.

```
//  a circle with center at (0 ,0) and radius 5
border C0(t=0,2*pi) { x = 5 * cos(t); y = 5 * sin(t); }
border C1(t=0,2*pi) { x = 2+0.3 * cos(t); y = 3*sin(t); }
border C2(t=0,2*pi) { x = -2+0.3 * cos(t); y = 3*sin(t); }

mesh Th = buildmesh(C0(60)+C1(-50)+C2(-50));
plot(Th,ps="electroMesh"); //  figure 3.1
fespace Vh(Th,P1);     //  P1 FE-space
Vh uh,vh;              //  unkown and test function.
problem Electro(uh,vh) =  //   definion of  the problem
    int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) ) //   bilinear
    + on(C0,uh=0)       //   boundary condition on C0
    + on(C1,uh=1)       //   +1 volt on C1
    + on(C2,uh=-1) ;    //   -1 volt on C2

Electro; //  solve the problem, see figure 3.2 for the solution
plot(uh,ps="electro.eps",wait=true); //  figure 3.2
```

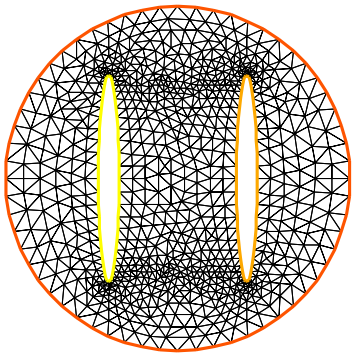


Figure 3.1: Disk with two elliptical holes

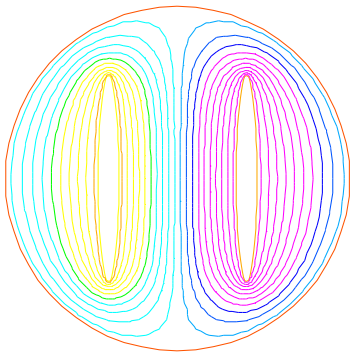


Figure 3.2: Equipotential lines, where C_1 is located in right hand side

3.3 Two-dimensional Black-Scholes equation

In mathematical finance, an option on two assets is modeled by a Black-Scholes equations in two space variables, (see for example Wilmott's book : a student introduction to mathe-

matical finance, Cambridge University Press).

$$\partial_t u + \frac{(\sigma_1 x_1)^2}{2}\frac{\partial^2 u}{\partial x_1^2} + \frac{(\sigma_2 x_2)^2}{2}\frac{\partial^2 u}{\partial x_2^2} + \rho x_1 x_2 \frac{\partial^2 u}{\partial x_1 \partial x_2} + rS_1\frac{\partial u}{\partial x_1} + rS_2\frac{\partial u}{\partial x_2} - rP = 0 \tag{3.6}$$

which is to be integrated in $(0,T)\times\mathbb{R}^+\times\mathbb{R}^+$ subject to, in the case of a put

$$u(x_1,x_2,T) = (K - \max(x_1,x_2))^+ . \tag{3.7}$$

Boundary conditions for this problem may not be so easy to device. As in the one dimensional case the PDE contains boundary conditions on the axis $x_1 = 0$ and on the axis $x_2 = 0$, namely two one dimensional Black-Scholes equations driven respectively by the data $u(0,+\infty,T)$ and $u(+\infty,0,T)$. These will be automatically accounted for because they are embedded in the PDE. So if we do nothing in the variational form (i.e. if we take a Neuman boundary condition at these two axis in the strong form) there will be no disturbance to these. At infinity in one of the variable, as in 1D, it makes sense to match the final condition:

$$u(x_1,x_2,t) \approx (K - \max(x_1,x_2))^+ e^{r(T-t)} \text{when } |\mathrm{x}| \to \infty \tag{3.8}$$

For an American put we will also have the constraint

$$u(x_1,x_2,t) \geq (K - \max(x_1,x_2))^+ e^{r(T-t)}. \tag{3.9}$$

We take

$$\sigma_1 = 0.3,\ \ \sigma_2 = 0.3,\ \ \rho = 0.3,\ \ r = 0.05,\ \ K = 40,\ \ T = 0.5 \tag{3.10}$$

An implicit Euler scheme with projection is used and a mesh adaptation is done every 10 time steps. The first order terms are treated by the Characteristic Galerkin method, which, roughly, approximates

$$\frac{\partial u}{\partial t} + a_1\frac{\partial u}{\partial x} + a_2\frac{\partial u}{\partial y} \approx \frac{1}{\delta t}\left(u^{n+1}(x) - u^n(x - \vec{a}\delta t)\right) \tag{3.11}$$

Example 48 (`blakschol.edp`)

```
//  verbosity=1;
int s=10; //  y-scale
int m=30;
int L=80;
int LL=80;
border aa(t=0,L){x=t;y=0;};
border bb(t=0,LL){x=L;y=t;};
border cc(t=L,0){x=t ;y=LL;};
border dd(t=LL,0){x = 0; y = t;};

mesh th = buildmesh(aa(m)+bb(m)+cc(m)+dd(m));
fespace Vh(th,P1);
```

```
   real sigmax=0.3;
   real sigmay=0.3;
   real rho=0.3;
   real r=0.05;
   real K=40;
   real dt=0.01;

   real eps=0.3;

   func f = max(K-max(x,y),0.);

   Vh u=f,v,w;

   func beta = 1;// (w<=f-eps)*eps + (w>=f) + (w<f)*(w>f-eps)*(eps+(w-f+eps)/eps)*
eps);

   plot(u,wait=1);

    th = adaptmesh(th,u,abserror=1,nbjacoby=2,
           err=0.004, nbvx=5000, omega=1.8,ratio=1.8, nbsmooth=3,
               splitpbedge=1, maxsubdiv=5,rescaling=1 );
    u=u;

   Vh xveloc = -x*r+x*sigmax^2+x*rho*sigmax*sigmay/2;
   Vh yveloc = -y*r+y*sigmay^2+y*rho*sigmax*sigmay/2;

   int j=0;
   int n;
   problem eq1(u,v,init=j,solver=LU) = int2d(th)(
                              u*v*(r+1/dt/beta)
                            + dx(u)*dx(v)*(x*sigmax)^2/2.
                            + dy(u)*dy(v)*(y*sigmay)^2/2.
         + dy(u)*dx(v)*rho*sigmax*sigmay*x*y/2.
                       + dx(u)*dy(v)*rho*sigmax*sigmay*x*y/2.    )
               + int2d(th)( -v*convect([xveloc,yveloc],dt,w)/dt/beta)
                     + on(bb,cc,u=f)// *exp(-r*t);
   ;
   int ww=1;
   for ( n=0; n*dt <= 1.0; n++)
   {
     cout <<" iteration " << n   <<  " j=" << j << endl;
     w=u;
     eq1;
     v = max(u-f,0.);
     plot(v,wait=ww);
     u = max(u,f);
     ww=0;

     if(j>10)  { cout << " adaptmesh " << endl;
```

```
      th = adaptmesh(th,u,verbosity=1,abserror=1,nbjacoby=2,
        err=0.001, nbvx=5000, omega=1.8, ratio=1.8, nbsmooth=3,
              splitpbedge=1, maxsubdiv=5,rescaling=1) ;
          j=-1;
       xveloc = -x*r+x*sigmax^2+x*rho*sigmax*sigmay/2;
       yveloc = -y*r+y*sigmay^2+y*rho*sigmax*sigmay/2;
       u=u;
       ww=1;
      };
    j=j+1;
    cout << " j = " <<  j << endl;
  };
  v = max(u-f,0.);
  plot(v,wait=1,value=1);
  plot(u,wait=1,value=1);
```

3.4 Periodic

Solve of the Laplace equation

$$-\Delta u = sin(x + \pi/4.) * cos(y + \pi/4.)$$

on a square $]0, 2\pi[^2$ with bi-periodic boundary condition.

Example 49 (`Periodic.edp`)

```
  mesh Th=square(10,10,[2*x*pi,2*y*pi]);
  //  defined the  fespace  with periodic condition
  //     label :  2 and 4  are left and right   side with y abcissa
  //              1 and 2  are bottom and upper side with x abcissa
  fespace Vh(Th,P2,periodic=[[2,y],[4,y],[1,x],[3,x]]);
   Vh uh,vh;              //  unkown and test function.
   func f=sin(x+pi/4.)*cos(y+pi/4.);      //   right hand side function

  problem laplace(uh,vh) =                      //   definion of  the prob-
lem
      int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) ) //   bilinear form
    + int2d(Th)( -f*vh )                         //   linear form
  ;

    laplace; //  solve the problem plot(uh); //  to see the result
    plot(uh,ps="period.eps",value=true);
```

3.5 Poisson on L-shape domain

Here we use more systematically the mesh adaptation to track the singularity at an obtuse angle of the domain.
Here `plot` has an extra parameter `ps="th.eps"`. Its effect is to create a postscript file

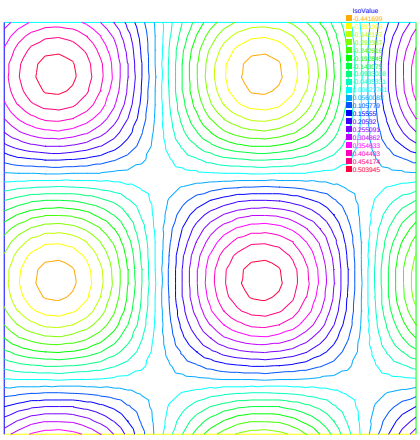


Figure 3.3: The isovalue of solution u with periodic boundary condition

named "th.eps" containing the triangulation `th` displayed during the execution of the program.
Then we write the triangulation data on disk with `savemesh` (see Example 24) and `th` for argument and a file name, here `th.msh`

```
savemesh(th,"th.msh");  //  saves mesh th in freefem format
```

There are several formats available to store the mesh.

Now we are going to solve the Laplace equation with Dirichlet boundary conditions. The problem is coercive and symmetric, so the linear system can be solved with the conjugate gradient method (parameter `solver=CG` with the stopping criteria on the residual, here `eps=1.0e-6`).

The domain is L-shaped and defined by a set of connecting segments a, b, c, d, e, f labeled $1, 2, 3, 4, 5, 6$.

```
border a(t=0,1.0){x=t;   y=0;  label=1;};
border b(t=0,0.5){x=1;   y=t;  label=2;};
border c(t=0,0.5){x=1-t; y=0.5;label=3;};
border d(t=0.5,1){x=0.5; y=t;  label=4;};
border e(t=0.5,1){x=1-t; y=1;  label=5;};
border f(t=0.0,1){x=0;   y=1-t;label=6;};
mesh Th = buildmesh (a(6) + b(4) + c(4) +d(4) + e(4) + f(6));
fespace Vh(Th,P1);
plot(Th,ps="th.eps"); //  Section 2.4.7
\\ Next we solve  he same problem on an adapted (and finer) mesh 4 times:
fespace Vh(Th,P1);  //  set FE-space
Vh u,v;             //  set unknown and test function
real error=0.1;        //  level of error
problem Probem1(u,v,solver=CG,eps=1.0e-6) =
    int2d(Th)(  dx(u)*dx(v) + dy(u)*dy(v))
  - int2d(Th) ( v*1 )
  + on(1,2,3,4,5,6,u=0)  ;
```

```
int i;                  //    declare loop index
for (i=0;i< 4;i++)
{
  Probem1;
  Th=adaptmesh(Th,u,err=error);
  error = error/2;
} ;
```

after each solve a new mesh adapted to u is computed. To speed up the adaptation we change by hand a default parameter of `adaptmesh:err`, which specifies the required precision, so as to make the new mesh finer.

In practice the program is more complex for two reasons

- We must use a dynamic name for files if we want to keep track of all iterations. This is done with the concatenation operator $+$. for instance

  ```
  for(i = 0; i< 4;i++)
      savemesh("th"+i+".msh",th);
  ```

  saves mesh th four times in files `th1.msh`,`th2.msh`,`th3.msh`, `th3.msh`.

- There are many default parameters which can be redefined either throughout the rest of the program or locally within `adaptmesh`. The list with their default value is in section 2.3.3.

3.6 Stokes and Navier-Stokes

The Stokes equations are:

$$\left.\begin{array}{rl} -\Delta u + \nabla p & = 0 \\ \nabla \cdot u & = 0 \end{array}\right\} \quad \text{in } \Omega \tag{3.12}$$

where u is the velocity vector and p the pressure. For simplicity, let us choose Dirichlet boundary conditions on the velocity, $u = u_\Gamma$ on Γ.

A classical way to discretize the Stokes equation with a mixed formulation, is to solve the variational problem and then discretize it:

Find $(u_h, p_h) \in X_h^2 \times M_h$ such that $u_h = u_{\Gamma h}$, and such that

$$\begin{array}{rl} \displaystyle\int_{\Omega_h} \nabla u_h \cdot \nabla v_h + \int \nabla p_h \cdot v_h & = 0, \quad \forall v_h \in X_{0h} \\ \displaystyle\int_{\Omega_h} \nabla . u_h q_h & = 0, \quad \forall q_h \in M_h \end{array} \tag{3.13}$$

where X_{0h} is the space of functions of X_h which are zero on Γ. The velocity space is approximated by X_h space, and the pressure space is approximated by M_h space.

3.6.1 Cavity Flow

The driven cavity flow problem is solved first at zero Reynolds number (Stokes flow) and then at Reynolds 100. The velocity pressure formulation is used first and then the calculation is repeated with the stream function vorticity formulation.

The driven cavity problem is the problem (3.12) where $u_\Gamma \cdot n = 0$ and $u_\Gamma \cdot s = 1$ on the top boundary and zero elsewhere (n is the Γ normal, and s is the Γ tangent).
The mesh is constructed by

```
mesh Th=square(8,8);
```

The labels assigned by `square` to the bottom,right,up and left edges are respectively $1, 2, 3, 4$.

We use a classical Taylor-Hood element technic to solve the problem:

The velocity is approximated with the P_2 FE (X_h space), and the the pressure is approximated with the P_1 FE (M_h space),

where

$$X_h = \{v \in H^1(]0,1[^2) / \forall K \in \mathcal{T}_h \quad v_{|K} \in P_2\}$$

and

$$M_h = \{v \in H^1(]0,1[^2) / \forall K \in \mathcal{T}_h \quad v_{|K} \in P_1\}$$

The FE-spaces and functions are constructed by

```
fespace Xh(Th,P2);
fespace Mh(Th,P1);
Xh u2,v2;
Xh u1,v1;
Xh p,q;
```

The Stokes operator is implemented as a system-solve for the velocity $(u1, u2)$ and the pressure p. The test function for the velocity is $(v1, v2)$ and q for the pressure, so the variational form (3.13) in freefem language is:

```
solve Stokes (u1,u2,p,v1,v2,q,solver=Crout) =
    int2d(Th)( ( dx(u1)*dx(v1) + dy(u1)*dy(v1)
            +  dx(u2)*dx(v2) + dy(u2)*dy(v2) )
            + p*q*(0.000001)
            + p*dx(v1)+ p*dy(v2)
            + dx(u1)*q+ dy(u2)*q
           )
  + on(3,u1=1,u2=0)
  + on(1,2,4,u1=0,u2=0);
```

Each unknown has its own boundary conditions.

Technical Remark There is some arbitrary decision here as to where to affect the boundary condition within the linear system. Basically the Dirichlet operator (`on`) should be associated with the unknown which contains it so that the penalization appears on the diagonal of the matrix of the underlying discrete linear system, otherwise it will be ill conditioned.

Remark 21 *Notice the term* `p*q*(0.000001)` *is added, because the solver Crout needs it: all the sub-matrices must be invertible.*

If the streamlines are required, they can be computed by finding ψ such that rot$\psi = u$ or better,

$$-\Delta\psi = \nabla \times u$$

```
Xh psi,phi;

solve streamlines(psi,phi) =
      int2d(Th)( dx(psi)*dx(phi) + dy(psi)*dy(phi))
   +  int2d(Th)( -phi*(dy(u1)-dx(u2)))
   +  on(1,2,3,4,psi=0);
```

Now the Navier-Stokes equations are solved

$$\frac{\partial u}{\partial t} + u \cdot \nabla u - \Delta u + \nabla p = 0, \quad \nabla \cdot u = 0$$

with the same boundary conditions and with initial conditions $u = 0$.

This is implemented by using the convection operator `convect` for the term $\frac{\partial u}{\partial t} + u \cdot \nabla u$, giving a discretization in time

$$\begin{array}{ll} \frac{1}{\delta t}(u^{n+1} - u^n \circ X^n) - \nu\Delta u^{n+1} + \nabla p^{n+1} & = 0, \\ \nabla \cdot u^{n+1} & = 0 \end{array} \tag{3.14}$$

The term,$u^n \circ X^n(x) \approx u^n(x - u^n(x)\delta t)$ will be computed by the operator "convect" , so we obtain

```
int i=0;
real  nu=1./100.;
real dt=0.1;
real alpha=1/dt;

Xh up1,up2;

problem  NS (u1,u2,p,v1,v2,q,solver=Crout,init=i) =
    int2d(Th)(
              alpha*( u1*v1 + u2*v2)
             + nu * ( dx(u1)*dx(v1) + dy(u1)*dy(v1)
             +  dx(u2)*dx(v2) + dy(u2)*dy(v2) )
             + p*q*(0.000001)
             + p*dx(v1)+ p*dy(v2)
             + dx(u1)*q+ dy(u2)*q
            )
  + int2d(Th) ( -alpha*
       convect([up1,up2],-dt,up1)*v1 -alpha*convect([up1,up2],-dt,up2)*v2
)
  + on(3,u1=1,u2=0)
  + on(1,2,4,u1=0,u2=0)
;
```

```
for (i=0;i<=10;i++)
 {
   up1=u1;
   up2=u2;
   NS;
   if ( !(i % 10))  //  plot every 10 iteration
    plot(coef=0.2,cmm=" [u1,u2] et p  ",p,[u1,u2]);
 } ;
```

Notice that the matrices are reused (keyword `init=i`)

3.6.2 Stokes with Uzawa

In this example we have a full Stokes problem, solve also the cavity problem, with the classical Uzawa conjugate gradient.

The idea of the algorithm is very simple, in the first equation of the Stokes problem, if we know the pressure, when we can compute the velocity $u(p)$, and to solve the problem is to find p, such that $\nabla.u(p) = 0$. The last problem is linear, symmetric negative, so we can use the conjugate gradient algoritm .

First we define mesh, and the Taylor-Hood approximation. So X_h is the velocity space, and M_h is the pressure space.

```
mesh Th=square(10,10);
fespace Xh(Th,P2),Mh(Th,P1);
Xh u1,u2,v1,v2;
Mh p,q,ppp;  //   ppp is a working pressure


varf bx(u1,q) = int2d(Th)( -(dx(u1)*q));
varf by(u1,q) = int2d(Th)( -(dy(u1)*q));
varf a(u1,u2)= int2d(Th)(  dx(u1)*dx(u2) + dy(u1)*dy(u2) )
                    +  on(3,u1=1)  +  on(1,2,4,u1=0) ;
//   remark:  put the on(3,u1=1) before  on(1,2,4,u1=0)
//   because we want zero on intersection

matrix A= a(Xh,Xh,solver=CG);
matrix Bx= bx(Xh,Mh);
matrix By= by(Xh,Mh);

Xh bc1; bc1[] = a(0,Xh);  //   boundary condition contribution  on u1
Xh bc2; bc2   = 0 ;       //   no boundary condition contribution on u2
Xh b;
```

Construct the function `divup` $p \longrightarrow \nabla.u(p)$.

```
func real[int] divup(real[int] & pp)
{
   //   compute u1(pp)
   b[]  = Bx'*pp; b[] += bc1[] ;    u1[] = A^-1*b[];
```

```
    //   compute u2(pp)
    b[]  = By'*pp; b[] += bc2[] ;    u2[] = A^-1*b[];
    //   div(u1,u2) = Bx'*u1[] + By'*u2[];
    ppp[] =   Bx*u1[];   //    ppp = t Bx u1
    ppp[] +=  By*u2[];   //        + t By u2
    return ppp[] ;
};
```

Call now the conjugate gradient algorithm:

```
p=0;q=0;
LinearCG(divup,p[],q[],eps=1.e-6,nbiter=50);
divup(p[]); //  compute the final solution

plot([u1,u2],p,wait=1,value=true,coef=0.1);
```

3.6.3 Navier Stokes with Uzawa

In this example we solve the Navier-Stokes equation, in the driven-cavity, with the Uzawa algorithm preconditioned by the Cahouet-Chabart method.

The idea of the precondioner is that in a periodic domain, all differential operators commute and the Uzawa algorithm comes to solving the linear operator $\nabla.((\alpha Id + \nu\Delta)^{-1}\nabla$, where Id is the identity operator. So the preconditioner suggested is $\alpha\Delta^{-1} + \nu Id$.

To implement this, we reuse the previous example, by including a file. Then we define the time step Δt, viscosity, and new variational form, and matrix.

```
include "StokesUzawa.edp" //  include the Stokes part
real dt=0.05, alpha=1/dt;  //   Δ t

cout << " alpha = " << alpha;
real xnu=1./400; //  viscosity  ν  =  Reynolds number^-1

//   the new variational form with mass term
varf at(u1,u2)= int2d(Th)( xnu*dx(u1)*dx(u2)
                         + xnu*dy(u1)*dy(u2) + u1*u2*alpha  )
                         +  on(1,2,4,u1=0)  + on(3,u1=1) ;

A = at(Xh,Xh,solver=CG);  //   change the matrix

//   set the 2 convect variational form
varf  vfconv1(uu,vv)=int2d(Th,qforder=5)(convect([u1,u2],-dt,u1)*vv*alpha);
varf  vfconv2(v2,v1)=int2d(Th,qforder=5)(convect([u1,u2],-dt,u2)*v1*alpha);

int idt;       //  index of of time set
real temps=0;  //  current time

Mh pprec,prhs;
varf vfMass(p,q) = int2d(Th)(p*q);
```

```
matrix MassMh=vfMass(Mh,Mh,solver=CG);

varf vfLap(p,q)=int2d(Th)(dx(pprec)*dx(q)+dy(pprec)*dy(q) + pprec*q*1e-
10);
matrix LapMh= vfLap(Mh,Mh,solver=Cholesky);
```

The function to define the preconditioner

```
func real[int]  CahouetChabart(real[int] & xx)
{  //   xx =  ∫ (div u) w_i
   //      α LapMh^-1  + ν MassMh^-1
   pprec[]= LapMh^-1* xx;
   prhs[] =  MassMh^-1*xx;
   pprec[] = alpha*pprec[]+xnu* prhs[];
   return pprec[];
};
```

The loop in time. Warning with the stop test of the conjuguate gradient, because we start from the previous solution and the end the previous solution is close to the final solution, don't take a relative stop test to the first residual, take an absolue stop test (negative here)

```
for (idt = 1; idt < 50; idt++)
 {
   temps += dt;
   cout << " --------- temps " << temps << " \n ";
   b1[] =  vfconv1(0,Xh);
   b2[] =  vfconv2(0,Xh);
   cout << "  min b1 b2  " << b1[].min << " " << b2[].min << endl;
   cout << "  max b1 b2  " << b1[].max << " " << b2[].max << endl;
   //  call Conjugued Gradient with preconditioner '
   //   warning eps < 0 => absolue stop test
   LinearCG(divup,p[],q[],eps=-1.e-6,nbiter=50,precon=CahouetChabart);
   divup(p[]);   //   computed the velocity

   plot([u1,u2],p,wait=!(idt%10),value= 1,coef=0.1);
 }
```

3.7 Readmesh

Freefem can read and write files which can be reused once read but the names of the borders are lost and they have to be replaced by the number which corresponds to their order of appearance in the program, unless the number is forced by the keyword "label".

```
border floor(t=0,1){ x=t; y=0; label=1;}; //  the unit square
border right(t=0,1){ x=1; y=t; label=5;};
border ceiling(t=1,0){ x=t; y=1; label=5;};
border left(t=1,0){ x=0; y=t; label=5;};
int n=10;
```

```
mesh th= buildmesh(floor(n)+right(n)+ceiling(n)+left(n));
savemesh(th,"toto.am_fmt");  //  format "formated Marrocco"
savemesh(th,"toto.Th");      //  format database "bamg"
savemesh(th,"toto.msh");     //  format freefem
mesh th2 = readmesh("toto.msh");
fespace femp1(th,P1);
femp1 f = sin(x)*cos(y),g;
{ //  save solution
ofstream file("f.txt");
file << f[] << endl;
}  //  close the file (end block)
{  //  read
ifstream file("f.txt");
file >> g[] ;
} //  close reading file (end block)
fespace Vh2(th2,P1);
Vh2 u,v;
plot(g);
solve pb(u,v) =
    int2d(th)( u*v - dx(u)*dx(v)-dy(u)*dy(v) )
  + int2d(th)(-g*v)
  + int1d(th,5)( g*v)
  + on(1,u=0) ;
plot (th2,u);
```

There are many formats of mesh files available for communication with other tools such as emc2, modulef..., the suffix gives the chosen type. More details can be found in the article by F. Hecht "bamg : a bidimentional anisotropic mesh generator" available from the freefem web page.
Note also the wrong sign in the Laplace equation, but freefem can handle it as long as it is not a resonance mode (i.e. the matrix of the linear system should be non-singular).

3.7.1 Domain decomposition

We present, three classique exemples, of domain decomposition technique: first, Schwarz algorithm with overlapping, second Schwarz algorithm without overlapping (also call Shur complement), and last we show to use the conjuguate gradient to solve the boundary problem of the Shur complement.

3.7.2 Schwarz algorithm with overlapping

To solve

$$-\Delta u = f, \quad \text{in } \Omega = \Omega_1 \cup \Omega_2 \quad u|_\Gamma = 0$$

the Schwarz algorithm runs like this

$$-\Delta u_1^{m+1} = f \text{ in } \Omega_1 \quad u_1^{m+1}|_{\Gamma_1} = u_2^m$$

$$-\Delta u_2^{m+1} = f \text{ in } \Omega_2 \quad u_2^{m+1}|_{\Gamma_2} = u_1^m$$

where Γ_i is the boundary of Ω_i and on the condition that $\Omega_1 \cap \Omega_2 \neq \emptyset$ and that u_i are zero at iteration 1.

Here we take Ω_1 to be a quadrangle, Ω_2 a disk and we apply the algorithm starting from zero (see Fig. 2.15).

Example 50 (schwarz-overlap.edp)

```
int inside = 2;  //   inside boundary
int outside = 1; //   outside boundary
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
border e(t=0, pi/2){ x= cos(t); y = sin(t);label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=outside;};
int n=4;
mesh th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh( e(5*n) + e1(25*n) );
plot(th,TH,wait=1);  //   to see the 2 meshes
```

The space and problem definition is :

```
fespace vh(th,P1);
fespace VH(TH,P1);
vh u=0,v; VH U,V;
int i=0;

problem PB(U,V,init=i,solver=Cholesky) =
    int2d(TH)( dx(U)*dx(V)+dy(U)*dy(V) )
  + int2d(TH)( -V) + on(inside,U = u)  + on(outside,U= 0 ) ;
problem pb(u,v,init=i,solver=Cholesky) =
    int2d(th)( dx(u)*dx(v)+dy(u)*dy(v) )
  + int2d(th)( -v) + on(inside ,u = U) + on(outside,u = 0 ) ;
```

The calculation loop:

```
for ( i=0 ;i< 10; i++)
{
   PB;
   pb;
   plot(U,u,wait=true);
};
```

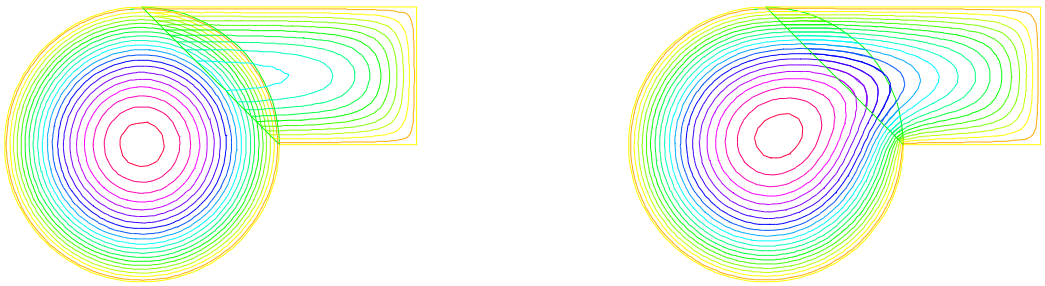


Figure 3.4: Isovalues of the solution at iteration 0 and iteration 9

3.7.3 Schwarz algorithm without overlapping

To solve

$$-\Delta u = f \text{ in } \Omega = \Omega_1 \cup \Omega_2 \quad u|_\Gamma = 0,$$

the Schwarz algorithm for domain decomposition without overlapping runs like the following.

Let introduce Γ_i is common the boundary of Ω_1 and Ω_2 and $\Gamma_e^i = \partial\Omega_i \setminus \Gamma_i$.

The probem find λ such that $(u_1|_{\Gamma_i} = u_2|_{\Gamma_i})$ where u_i is solution of the following Laplace problem:

$$-\Delta u_i = f \text{ in } \Omega_i \quad u_i|_{\Gamma_i} = \lambda \quad u_i|_{\Gamma_e^i} = 0$$

To solve this problem we just make a loop with upgradingλ with

$$\lambda = \lambda \pm \frac{(u_1 - u_2)}{2}$$

where the sign $+$ or $-$ of $\pm$ is choose to have convergence.

Example 51 (schwarz-no-overlap.edp)

```
//  schwarz1 without overlapping (see Fig. 2.16)
int inside = 2;
int outside = 1;
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
border e(t=0, 1){ x= 1-t; y = t;label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=outside;};
int n=4;
mesh th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh ( e(5*n) + e1(25*n) );
plot(th,TH,wait=1,ps="schwarz-no-u.eps");
```

```
fespace vh(th,P1);
fespace VH(TH,P1);
vh u=0,v; VH U,V;
vh lambda=0;
int i=0;

problem PB(U,V,init=i,solver=Cholesky) =
    int2d(TH)( dx(U)*dx(V)+dy(U)*dy(V) )
  + int2d(TH)( -V)
  + int1d(TH,inside)(-lambda*V) +    on(outside,U= 0 ) ;
problem pb(u,v,init=i,solver=Cholesky) =
    int2d(th)( dx(u)*dx(v)+dy(u)*dy(v) )
  + int2d(th)( -v)
  + int1d(th,inside)(+lambda*v) +    on(outside,u = 0 ) ;

for ( i=0 ;i< 10; i++)
{
   PB;
   pb;
   lambda = lambda - (u-U)/2;
   plot(U,u,wait=true);
};

plot(U,u,ps="schwarz-no-u.eps");
```

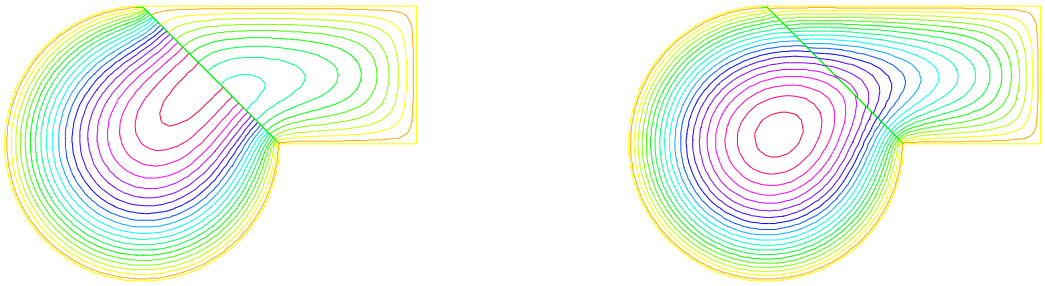


Figure 3.5: Isovalues of the solution at iteration 0 and iteration 9 without overlapping

Example 52 (Schwarz-gc.edp) *The version of this example for Shur componant. The border problem is solve with conjuguate gradient.*

First, we construct the two domain (see Fig. 2.16, `a=Gamma1, b=Gamma2, c=Gamma3, d=e=GammaInside, e1=GammaArc`*).*

```
//  Schwarz without overlapping (Shur complenement Neumann -> Dirichet)
real cpu=clock();
```

```
int inside = 2;
int outside = 1;

border Gamma1(t=1,2){x=t;y=0;label=outside;};
border Gamma2(t=0,1){x=2;y=t;label=outside;};
border Gamma3(t=2,0){x=t ;y=1;label=outside;};

border GammaInside(t=1,0){x = 1-t; y = t;label=inside;};

border GammaArc(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=outside;};
int n=4;
//   build the mesh of Ω1 and Ω2
mesh Th1 = buildmesh( Gamma1(5*n) + Gamma2(5*n) + GammaInside(5*n) + Gamma3(5*n
mesh Th2 = buildmesh ( GammaInside(-5*n) + GammaArc(25*n) );
plot(Th1,Th2);

//  defined the 2 FE-space
fespace Vh1(Th1,P1),      Vh2(Th2,P1);
```

Remark, to day is not possible to defined a function just on a border, so the lambda function is defined on the all domain Ω_1 by:

```
Vh1 lambda=0;  //  take λ ∈ Vh1
```

The two Laplace problem:

```
Vh1 u1,v1;              Vh2 u2,v2;
int i=0;  //  for factorization optimization
problem Pb2(u2,v2,init=i,solver=Cholesky) =
    int2d(Th2)( dx(u2)*dx(v2)+dy(u2)*dy(v2) )
  + int2d(Th2)( -v2)
  + int1d(Th2,inside)(-lambda*v2) +    on(outside,u2= 0 ) ;
problem Pb1(u1,v1,init=i,solver=Cholesky) =
    int2d(Th1)( dx(u1)*dx(v1)+dy(u1)*dy(v1) )
  + int2d(Th1)( -v1)
  + int1d(Th1,inside)(+lambda*v1) +    on(outside,u1 = 0 ) ;
```

Now, we define a border matrix , because the lambda function is none zero inside the domain Ω_1:

```
varf b(u2,v2,solver=CG) =int1d(Th1,inside)(u2*v2);
matrix B= b(Vh1,Vh1,solver=CG);
```

The boundary problem function,

$$\lambda \longrightarrow \int_{\Gamma_i} (u_1 - u_2)v_1$$

```
func real[int] BoundaryProblem(real[int] &l)
{
```

```
    lambda[]=l; //  make FE-function form l
    Pb1;     Pb2;
    i++;  //   no  refactorization i !=0
    v1=-(u1-u2);
    lambda[]=B*v1[];
    return lambda[] ;
};
```

Remark, the difference between the two notations `v1` and `v1[]` is: `v1` is the finite element function and `v1[]` is the vector in the canonical basis of the finite element function `v1` .

```
Vh1 p=0,q=0;
//   solve the problem with Conjugue Gradient
LinearCG(BoundaryProblem,p[],q[],eps=1.e-6,nbiter=100);
//   compute the final solution, because CG works with increment
BoundaryProblem(p[]); //  solve again  to have right u1,u2

cout << " -- CPU time  schwarz-gc:" <<  clock()-cpu << endl;
plot(u1,u2); //  plot
```

3.8 Elasticity

Consider an elastic plate with undeformed shape $\Omega\times]-h,h[$ in $\mathbb{R}^3$, $\Omega\subset\mathbb{R}^2$. By the deformation of the plate, we assume that a point $P(x_1,x_2,x_3)$ moves to $\mathcal{P}(\xi_1,\xi_2,\xi_3)$. The vector $\vec{u}=(u_1,u_2,u_3)=(\xi_1-x_1,\xi_2-x_2,\xi_3-x_3)$ is called *displacement vector*. By the deformation, the line segment $\mathbf{x},\mathbf{x}+\tau\Delta\mathbf{x}$ moves approximately to $\mathbf{x}+u(\mathbf{x}),\mathbf{x}+\tau\Delta\mathbf{x}+u(\mathbf{x}+\tau\Delta\mathbf{x})$ for small τ, where $\mathbf{x}=(x_1,x_2,x_3)$, $\Delta\mathbf{x}=(\Delta x_1,\Delta x_2,\Delta x_3)$. We now calculate the ratio between two segments

$$\eta(\tau)=\tau^{-1}|\Delta\mathbf{x}|^{-1}\left(|u(\mathbf{x}+\tau\Delta\mathbf{x})-u(\mathbf{x})+\tau\Delta\mathbf{x}|-\tau|\Delta\mathbf{x}|\right)$$

then we have (see e.g. [16, p.32])

$$\lim_{\tau\to 0}\eta(\tau)=(1+2e_{ij}\nu_i\nu_j)^{1/2}-1,\quad 2e_{ij}=\frac{\partial u_k}{\partial x_i}\frac{\partial u_k}{\partial x_j}+\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)$$

where $\nu_i=\Delta x_i|\Delta\mathbf{x}|^{-1}$. If the deformation is *small*, then we may consider that

$$(\partial u_k/\partial x_i)(\partial u_k/\partial x_i)\approx 0$$

and the following is called *small* strain tensor

$$\varepsilon_{ij}(u)=\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)$$

The tensor e_{ij} is called *finite strain tensor*.

Consider the small plane $\Delta\Pi(\mathbf{x})$ centered at $\mathbf{x}$ with the unit normal direction $\vec{n}=(n_1,n_2,n_3)$, then the surface on $\Delta\Pi(\mathbf{x})$ at $\mathbf{x}$ is

$$(\sigma_{1j}(\mathbf{x})n_j,\sigma_{2j}(\mathbf{x})n_j,\sigma_{3j}(\mathbf{x})n_j)$$

where $\sigma_{ij}(\mathbf{x})$ is called *stress tensor* at $\mathbf{x}$. Hooke's law is the assumption of a linear relation between σ_{ij} and ε_{ij} such as

$$\sigma_{ij}(\mathbf{x}) = c_{ijkl}(\mathbf{x})\varepsilon_{ij}(\mathbf{x})$$

with the symmetry $c_{ijkl} = c_{jikl}, c_{ijkl} = c_{ijlk}, c_{ijkl} = c_{klij}$.

If Hooke's tensor $c_{ijkl}(\mathbf{x})$ do not depend on the choice of coordinate system, the material is called *isotropic* at $\mathbf{x}$. If c_{ijkl} is constant, the material is called *homogeneous*. In homogeneous isotropic case, there is *Lamé constants* λ, μ (see e.g. [16, p.43]) satisfying

$$\sigma_{ij} = \lambda\delta_{ij}\mathrm{div}u + 2\mu\varepsilon_{ij} \tag{3.15}$$

where δ is the Kronecker symbol. We assume that the elastic plate is fixed on $\Gamma_D\times]-h,h[$, $\Gamma_D \subset \partial\Omega$. If the body force $f = (f_1, f_2, f_3)$ is given in $\Omega\times]-h,h[$ and surface force g is given in $\Gamma_N\times]-h,h[, \Gamma_N = \partial\Omega \setminus \overline{\Gamma_D}$, then the equation of equilibrium is given as follows:

$$-\partial_j\sigma_{ij} = f_i \text{ in } \Omega\times]-h,h[, \quad i = 1,2,3 \tag{3.16}$$

$$\sigma_{ij}n_j = g_i \text{ on } \Gamma_N\times]-h,h[, \quad u_i = 0 \text{ on } \Gamma_D\times]-h,h[, \quad i = 1,2,3 \tag{3.17}$$

We now explain the plain elasticity.

Plain strain: On the end of plate, the contact condition $u_3 = 0$, $g_3 =$ is satisfied. In this case, we can suppose that $f_3 = g_3 = u_3 = 0$ and $\vec{u}(x_1, x_2, x_3) = \overline{u}(x_1, x_2)$ for all $-h < x_3 < h$.

Plain stress: The cylinder is assumed to be very thin and subjected to no load on the ends $x_3 = \pm h$, that is,

$$\sigma_{3i} = 0, \quad x_3 = \pm h, \quad i\ 1,2,3$$

The assumption leads that $\sigma_{3i} = 0$ in $\Omega\times]-h,h[$ and $\vec{u}(x_1, x_2, x_3) = \overline{u}(x_1, x_2)$ for all $-h < x_3 < h$.

Generalized plain stress: The cylinder is subjected to no load on the ends $x_3 = \pm h$. Introducing the mean values with respect to thickness,

$$\overline{u}_i(x_1, x_2) = \frac{1}{2h}\int_{-h}^{h} u(x_1, x_2, x_3)dx_3$$

and we derinde $\overline{u}_3 \equiv 0$. Similary we define the mean values $\overline{f}, \overline{g}$ of the body force and surface force as well as the mean values $\overline{\varepsilon}_{ij}$ and $\overline{\sigma}_{ij}$ of the components of stress and strain, respectively.

In what follows we omit the overlines of $\overline{u}, \overline{f}, \overline{g}, \overline{\varepsilon}_{ij}$ and $\overline{\varepsilon}_{ij}$. Then we obtain similar equation of equilibrium given in (3.16) replacing $\Omega\times]-h,h[$ with Ω and changing $i = 1,2$. In the case of plane stress, $\sigma_{ij} = \lambda^*\delta_{ij}\mathrm{div}u + 3\mu\varepsilon_{ij}, \lambda^* = (2\lambda\mu)/(\lambda+\mu)$.

The equations of elasticity are naturally written in variational form for the displacement vector $u(x) \in V$ as

$$\int_\Omega [\mu\epsilon_{ij}(\vec{u})\epsilon_{ij}(\vec{v}) + \lambda\epsilon_{ii}(u)\epsilon_{jj}(\vec{v})] = \int_\Omega \vec{f}\cdot\vec{v} + \int_\Gamma \vec{g}\cdot\vec{v}, \forall\vec{v} \in V$$

where V is the linear closed subspace of $H^1(\Omega)^2$.

3.9 Deformation of Beam

Elastic solids subject to forces deform: a point in the solid, originaly at (x,y) goes to (X,Y) after. When the displacement vector $\vec{v} = (v_1, v_2) = (X - x, Y - y)$ is small, Hooke's law relates the stress tensor σ inside the solid to the deformation tensor ϵ:

$$\sigma_{ij} = \lambda\delta_{ij}\nabla.\vec{v} + \mu\epsilon_{ij},\ \epsilon_{ij} = \frac{1}{2}(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i})$$

where δ is the Kronecker symbol and where λ, μ are two constants describing the material mechanical properties in terms of the modulus of elasticity, and Young's modulus.

The equations of elasticity are naturally written in variational form for the displacement vector $v(x) \in V$ as

$$\int_\Omega [\mu\epsilon_{ij}(\vec{v})\epsilon_{ij}(\vec{w}) + \lambda\epsilon_{ii}(v)\epsilon_{jj}(\vec{w})] = \int_\Omega \vec{g}\cdot\vec{w} + \int_\Gamma \vec{h}\cdot\vec{w}, \forall\vec{w} \in V$$

Example 53 (Deformation of Beam) *We can find the deformation of the rectangle beam* $[0, 10] \times [0, 2]$ *under the gravity force* $\vec{f}$ *and the boundary stress* $i\vec{g}$ *is zero on lower and upper side, and on the two vertical sides of the beam are locked.*

```
//    a weighting beam sitting on a

int bottombeam = 2;
border a(t=2,0)  { x=0; y=t ;label=1;};        //   left beam
border b(t=0,10) { x=t; y=0 ;label=bottombeam;};   //   bottom of beam
border c(t=0,2)  { x=10; y=t ;label=1;};       //   rigth beam
border d(t=0,10) { x=10-t; y=2; label=3;};     //   top beam
real E = 21.5; //  Young's modulus
real sigma = 0.29;  //  Poisson's ratio
real mu = E/(2*(1+sigma)); //  Lame constants
real lambda = E*sigma/((1+sigma)*(1-2*sigma));
real gravity = -0.05;
mesh th = buildmesh( b(20)+c(5)+d(20)+a(5));
fespace Vh(th,[P1,P1]);
Vh [uu,vv], [w,s];
cout << "lambda,mu,gravity ="<<lambda<<" "<< mu <<" "<< gravity << endl;
//  deformation of a beam under its own weight
solve  bb([uu,vv],[w,s])  =
    int2d(th)(
           2*mu*(dx(uu)*dx(w)+ ((dx(vv)+dy(uu))*(dx(s)+dy(w)))/4 )
           + lambda*(dx(uu)+dy(vv))*(dx(w)+dy(s))/2
             )
  + int2d(th) (-gravity*s)
  + on(1,uu=0,vv=0)
 ;

plot([uu,vv],wait=1);
```

```
plot([uu,vv],wait=1,bb=[[-0.5,2.5],[2.5,-0.5]]);
mesh th1 = movemesh(th, [x+uu, y+vv]);
plot(th1,wait=1);
```

3.10 Fracture Mechanics

Consider the plate with the crack whose undeformed shape is a curve Σ with the two edges γ_1, γ_2. We assume the stress tensor σ_{ij} is the state of plate stress regarding $(x,y) \in \Omega_\Sigma = \Omega \setminus \Sigma$. Here Ω stands for the undeformed shape of elastic plate without crack. If the part Γ_N of the boundary $\partial\Omega$ is fixed and a load $\mathcal{L} = (\vec{f}, \vec{g}) \in L^2(\Omega)^2 \times L^2(\Gamma_N)^2$ is given, then the displacement $\vec{u}$ is the minimizer of the potential energy functional

$$\mathcal{E}(\vec{v}; \mathcal{L}, \Omega_\Sigma) = \int_{\Omega_\Sigma} \{w(x,\vec{v}) - \vec{f}\cdot\vec{v}\} - \int_{\Gamma_N} \vec{g}\cdot\vec{v}$$

over the functional space $V(\Omega_\Sigma)$,

$$V(\Omega_\Sigma) = \left\{\vec{v} \in H^1(\Omega_\Sigma)^2;\ \vec{v} = 0 \quad \text{on } \Gamma_D = \partial\Omega \setminus \overline{\Gamma_N}\right\},$$

where $w(x,\vec{v}) = \sigma_{ij}(\vec{v})\varepsilon_{ij}(\vec{v})/2$,

$$\sigma_{ij}(\vec{v}) = C_{ijkl}(x)\varepsilon_{kl}(\vec{v}), \quad \varepsilon_{ij}(\vec{v}) = (\partial v_i/\partial x_j + \partial v_j/\partial x_i)/2, \qquad (C_{ijkl}: \quad \text{Hooke's tensor}).$$

If the elasticity is homogeneous isotropic, then the displacement $\vec{u}(x)$ is decomposed in an open neighborhood U_k of γ_k as in (see e.g. [17])

$$\vec{u}(x) = \sum_{l=1}^{2} K_l(\gamma_k) r_k^{1/2} S_{kl}^C(\theta_k) + \vec{u}_{k,R}(x) \quad \text{for } x \in \Omega_\Sigma \cap U_k,\ k = 1,2 \tag{3.18}$$

with $\vec{u}_{k,R} \in H^2(\Omega_\Sigma \cap U_k)^2$, where U_k, $k = 1,2$ are open neighborhoods of γ_k such that $\partial L_1 \cap U_1 = \gamma_1$, $\partial L_m \cap U_2 = \gamma_2$, and

$$\begin{aligned} S_{k1}^C(\theta_k) &= \frac{1}{4\mu}\frac{1}{(2\pi)^{1/2}} \begin{bmatrix} [2\kappa-1]\cos(\theta_k/2) - \cos(3\theta_k/2) \\ -[2\kappa+1]\sin(\theta_k/2) + \sin(3\theta_k/2) \end{bmatrix}, \\ S_{k2}^C(\theta_k) &= \frac{1}{4\mu}\frac{1}{(2\pi)^{1/2}} \begin{bmatrix} -[2\kappa-1]\sin(\theta_k/2) + 3\sin(3\theta_k/2) \\ -[2\kappa+1]\cos(\theta_k/2) + \cos(3\theta_k/2) \end{bmatrix}. \end{aligned} \tag{3.19}$$

where μ is the shear modulus of elasticity, $\kappa = 3 - 4\nu$ (ν is the Poisson's ratio) for plane strain and $\kappa = \frac{3-\nu}{1+\nu}$ for plane stress.

The coefficients $K_1(\gamma_i)$ and $K_2(\gamma_i)$, which are important parameters in fracture mechanics, are called stress intensity factors of the opening mode (mode I) and the sliding mode (mode II), respectively.

For simlicity, we consider the following simple crack

$$\Omega = \{(x,y):\ -1 < x < 1, -1 < y < 1\}, \qquad \Sigma = \{(x,y):\ -1 \le x \le 0, y = 0\}$$

with only one crack tip $\gamma = (0,0)$. Unfortunately, `freefem++` cannot treat crack, so we use the modification of the domain with U-shape channel (see Fig. 2.24) with $d = 0.0001$. The undeformed crack Σ is approximated by

$$\begin{aligned} \Sigma_d &= \{(x,y):\ -1 \le x \le -10*d, -d \le y \le d\} \\ &\quad \cup \{(x,y):\ -10*d \le x \le 0, -d + 0.1*x \le y \le d - 0.1*x\} \end{aligned}$$

and $\Gamma_D =$ R in Fig. 2.24. In this example, we use three technique:

- Fast Finite Element Interpolator from the mesh `Th` to `Zoom` for the scale-up of near γ (see Section 2.4.1).
- After obtaining the displacement vector $\vec{u} = (u, v)$, we shall watch the deformation of the crack near γ as follows,

```
mesh Plate = movemesh(Zoom,[x+u,y+v]);
plot(Plate);
```

- Important technique is adaptive mesh, because the large singularity occur at γ as shown in (3.18).

First example create mode I deformation by the opposed surface force on `B` and `T` in the vertical direction of Σ, and the displacement is fixed on `R`.

In a laboratory, fracture engineer use photoelasticity to make stress field visible, which shows the principal stress difference

$$\sigma_1 - \sigma_2 = \sqrt{(\sigma_{11} - \sigma_{22})^2 + 4 * \sigma_{12}^2} \tag{3.20}$$

where σ_1 and σ_2 are the principal stresses. In opening mode, the photoelasticity make symmetric pattern concentrated at γ.

Example 54 (Crack Opening, $K_2(\gamma) = 0$)

```
real d = 0.0001;
int n = 5;
real cb=1, ca=1, tip=0.0;
border L1(t=0,ca-d) { x=-cb; y=-d-t; }
border L2(t=0,ca-d) { x=-cb; y=ca-t; }
border B(t=0,2) { x=cb*(t-1); y=-ca; }
border C1(t=0,1) { x=-ca*(1-t)+(tip-10*d)*t; y=d; }
border C21(t=0,1) { x=(tip-10*d)*(1-t)+tip*t; y=d*(1-t); }
border C22(t=0,1) { x=(tip-10*d)*t+tip*(1-t); y=-d*t; }
border C3(t=0,1) { x=(tip-10*d)*(1-t)-ca*t; y=-d; }
border C4(t=0,2*d) { x=-ca; y=-d+t; }
border R(t=0,2) { x=cb; y=cb*(t-1); }
border T(t=0,2) { x=cb*(1-t); y=ca; }
mesh Th = buildmesh (L1(n/2)+L2(n/2)+B(n)
                        +C1(n)+C21(3)+C22(3)+C3(n)+R(n)+T(n));
cb=0.1; ca=0.1;
plot(Th,wait=1);
mesh Zoom = buildmesh (L1(n/2)+L2(n/2)+B(n)+C1(n)
                          +C21(3)+C22(3)+C3(n)+R(n)+T(n));
plot(Zoom,wait=1);
real E = 21.5;
real sigma = 0.29;
real mu = E/(2*(1+sigma));
real lambda = E*sigma/((1+sigma)*(1-2*sigma));
fespace Vh(Th,[P2,P2]);
fespace zVh(Zoom,P2);
Vh [u,v], [w,s];
```

```
solve  Problem([u,v],[w,s])  =
    int2d(Th)(
            2*mu*(dx(u)*dx(w)+ ((dx(v)+dy(u))*(dx(s)+dy(w)))/4 )
            + lambda*(dx(u)+dy(v))*(dx(w)+dy(s))/2
            )
    -int1d(Th,T)(0.1*(4-x)*s)+int1d(Th,B)(0.1*(4-x)*s)
    +on(R,u=0)+on(R,v=0);                  //  fixed
;

zVh Sx, Sy, Sxy, N;
for (int i=1; i<=5; i++)
{
  mesh Plate = movemesh(Zoom,[x+u,y+v]); //  deformation near γ
  Sx = lambda*(dx(u)+dy(v)) + 2*mu*dx(u);
  Sy = lambda*(dx(u)+dy(v)) + 2*mu*dy(v);
  Sxy = mu*(dy(u) + dx(v));
  N = 0.1*1*sqrt((Sx-Sy)^2+4*Sxy^2); // principal stress difference
  if (i==1) {
     plot(Plate,ps="1stCOD.eps",bw=1); //  Fig. 3.6
     plot(N,ps="1stPhoto.eps",bw=1);   //  Fig. 3.6
  } else if (i==5) {
     plot(Plate,ps="LastCOD.eps",bw=1); //  Fig. 3.7
     plot(N,ps="LastPhoto.eps",bw=1);   //  Fig. 3.7
     break;
  }
  Th=adaptmesh(Th,[u,v]);
  Problem;
}
```

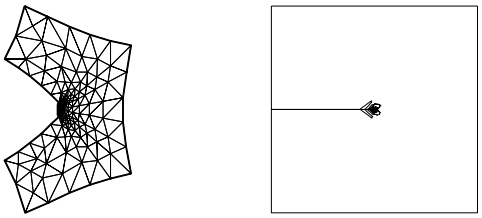


Figure 3.6: Crack open displacement (COD) and Principal stress difference in the first mesh

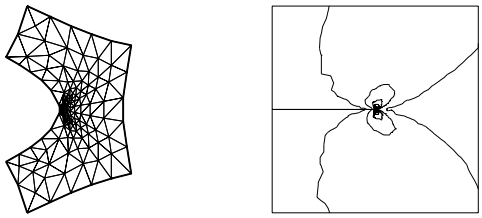


Figure 3.7: COD and Principal stress difference in the last adaptive mesh

It is difficult to create mode II deformation by the opposed shear force on B and T that is observed in a laboratory. So we use the body shear force along Σ, that is, the x-component f_1 of the body force $\vec{f}$ is given by

$$f_1(x,y) = H(y-0.001) * H(0.1-y) - H(-y-0.001) * H(y+0.1)$$

where $H(t) = 1$ if $t > 0$; $= 0$ if $t < 0$.

Example 55 (Crack Sliding, $K_2(\gamma) = 0$)

```
(use the same mesh Th)
cb=0.01; ca=0.01;
mesh Zoom = buildmesh (L1(n/2)+L2(n/2)+B(n)+C1(n)
                       +C21(3)+C22(3)+C3(n)+R(n)+T(n));
(use same FE-space Vh and elastic modulus)
fespace Vh1(Th,P1);
Vh1 fx = ((y>0.001)*(y<0.1))-((y<-0.001)*(y>-0.1)) ;

solve  Problem([u,v],[w,s])  =
    int2d(Th)(
             2*mu*(dx(u)*dx(w)+ ((dx(v)+dy(u))*(dx(s)+dy(w)))/4 )
             + lambda*(dx(u)+dy(v))*(dx(w)+dy(s))/2
             )
    -int2d(Th)(fx*w)
    +on(R,u=0)+on(R,v=0);        //  fixed
;

for (int i=1; i<=3; i++)
{
  mesh Plate = movemesh(Zoom,[x+u,y+v]); //  deformation near γ
  Sx = lambda*(dx(u)+dy(v)) + 2*mu*dx(u);
  Sy = lambda*(dx(u)+dy(v)) + 2*mu*dy(v);
  Sxy = mu*(dy(u) + dx(v));
  N = 0.1*1*sqrt((Sx-Sy)^2+4*Sxy^2); // principal stress difference
  if (i==1) {
     plot(Plate,ps="1stCOD2.eps",bw=1); //  Fig. 3.9
     plot(N,ps="1stPhoto2.eps",bw=1);   //  Fig. 3.8
  } else if (i==3) {
     plot(Plate,ps="LastCOD2.eps",bw=1); //  Fig. 3.9
     plot(N,ps="LastPhoto2.eps",bw=1);   //  Fig. 3.9
     break;
  }
  Th=adaptmesh(Th,[u,v]);
  Problem;
}
```

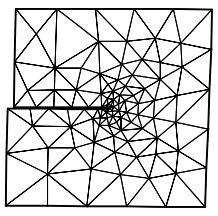


Figure 3.8: (COD) and Principal stress difference in the first mesh

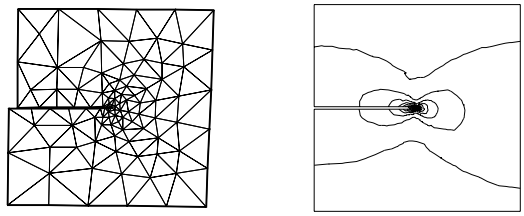


Figure 3.9: COD and Principal stress difference in the last adaptive mesh

3.11 Fluid-structure Interaction

This problem involves the Lamé system of elasticity and the Stokes system for viscous fluids with velocity $\vec{u}$ and pressure p: In our example the Lamé system and the Stokes system are coupled by a common boundary on which the fluid stress creates a displacement of the boundary and hence changes the shape of the domain where the Stokes problem is integrated. The geometry is that of a vertical driven cavity with an elastic lid. The lid is a beam with weight so it will be deformed by its own weight and by the normal stress due to the fluid reaction. The cavity is the 10×10 square and the lid is a rectangle of height $l = 2$.

A beam (see Example 53) sits on a box full of fluid rotating because the left vertical side has velocity one. The beam is bent by its own weight, but the pressure of the fluid modifies the bending.
The bending displacement of the beam is given by (uu,vv) solution of

```
   //   Fluid-structure interaction for a weighting beam sitting on a
   //  square cavity filled with a fluid.

   int bottombeam = 2; //  label of bottombeam
   border a(t=2,0)  { x=0; y=t ;label=1;};        //   left beam
   border b(t=0,10) { x=t; y=0 ;label=bottombeam;};        //   bottom of
beam
   border c(t=0,2)  { x=10; y=t ;label=1;};       //   rigth beam
   border d(t=0,10) { x=10-t; y=2; label=3;};     //   top beam
   real E = 21.5;
   real sigma = 0.29;
   real mu = E/(2*(1+sigma));
   real lambda = E*sigma/((1+sigma)*(1-2*sigma));
   real gravity = -0.05;
   mesh th = buildmesh( b(20)+c(5)+d(20)+a(5));
   fespace Vh(th,P1);
   Vh uu,w,vv,s,fluidforce=0;
   cout << "lambda,mu,gravity ="<<lambda<< " " << mu << " " << gravity <<
endl;
   //  deformation of a beam under its own weight
   solve  bb([uu,vv],[w,s])  =
       int2d(th)(
                    2*mu*(dx(uu)*dx(w)+ ((dx(vv)+dy(uu))*(dx(s)+dy(w)))/4
)
                  + lambda*(dx(uu)+dy(vv))*(dx(w)+dy(s))/2
                )
     + int2d(th) (-gravity*s)
     + on(1,uu=0,vv=0)
     + fluidforce[];
    ;

    plot([uu,vv],wait=1);
    mesh th1 = movemesh(th, [x+uu, y+vv]);
    plot(th1,wait=1);
```

Then Stokes equation for fluids ast low speed are solved in the box below the beam, but the beam has deformed the box (see border h):

```
// Stokes on square  b,e,f,g  driven cavite on left side g
border e(t=0,10) { x=t; y=-10; label= 1; };      //   bottom
border f(t=0,10) { x=10; y=-10+t ; label= 1; };   //   right
border g(t=0,10) { x=0; y=-t ;label= 2;};       //   left
border h(t=0,10) { x=t; y=vv(t,0)*( t>=0.001 )*(t <= 9.999);
                   label=3;};   //   top of cavity deforme

mesh sh = buildmesh(h(-20)+f(10)+e(10)+g(10));
plot(sh,wait=1);
```

We use the Uzawa conjuguate gradient to solve the Stokes problem like in example 3.6.2

```
fespace Xh(sh,P2),Mh(sh,P1);
Xh u1,u2,v1,v2;
Mh p,q,ppp;


varf bx(u1,q) = int2d(sh)( -(dx(u1)*q));

varf by(u1,q) = int2d(sh)( -(dy(u1)*q));

varf Lap(u1,u2)= int2d(sh)(  dx(u1)*dx(u2) + dy(u1)*dy(u2) )
                   +  on(2,u1=1) +  on(1,3,u1=0)  ;

Xh bc1; bc1[] = Lap(0,Xh);
Xh brhs;

matrix A= Lap(Xh,Xh,solver=CG);
matrix Bx= bx(Xh,Mh);
matrix By= by(Xh,Mh);
Xh bcx=0,bcy=1;

func real[int] divup(real[int] & pp)
{
  int verb=verbosity;
   verbosity=0;
   brhs[]  = Bx'*pp; brhs[] += bc1[] .*bcx[];
   u1[] = A^-1*brhs[];
   brhs[]  = By'*pp; brhs[] += bc1[] .*bcy[];
   u2[] = A^-1*brhs[];
   ppp[] =   Bx*u1[];
   ppp[] +=  By*u2[];
   verbosity=verb;
   return ppp[] ;
};

 p=0;q=0;u1=0;v1=0;
```

```
 LinearCG(divup,p[],q[],eps=1.e-3,nbiter=50);
 divup(p[]);
```

Now the beam will feel the stress constraint from the fluid:

```
  Vh sigma11,sigma22,sigma12;
  Vh uu1=uu,vv1=vv;

  sigma11([x+uu,y+vv]) = (2*dx(u1)-p);
  sigma22([x+uu,y+vv]) = (2*dy(u2)-p);
  sigma12([x+uu,y+vv]) = (dx(u1)+dy(u2));
```

which comes as a boundary condition to the PDE of the beam:

```
 varf fluidf([uu,vv],[w,s]) fluidforce =
 solve  bbst([uu,vv],[w,s],init=i)  =
     int2d(th)(
                  2*mu*(dx(uu)*dx(w)+ ((dx(vv)+dy(uu))*(dx(s)+dy(w)))/4
)
                + lambda*(dx(uu)+dy(vv))*(dx(w)+dy(s))/2
               )
   + int2d(th) (-gravity*s)
   + int1d(th,bottombeam)( -coef*(   sigma11*N.x*w + sigma22*N.y*s
                                  + sigma12*(N.y*w+N.x*s) )  )
   + on(1,uu=0,vv=0);
  plot([uu,vv],wait=1);
  real  err = sqrt(int2d(th)( (uu-uu1)^2 + (vv-vv1)^2 ));
  cout <<  " Erreur L2 = " << err << "----------\n";
```

Notice that the matrix generated by bbst is reused (see `init=i`). Finally we deform the beam

```
  th1 = movemesh(th, [x+0.2*uu, y+0.2*vv]);
  plot(th1,wait=1);
```

3.12 Region

Examples!Region.edp This example explains the definition and manipulation of *region*, i.e. subdomains of the whole domain.

Consider this L-shaped domain with 3 diagonals as internal boundaries, defining 4 subdomains:

Example 56

```
 //    example using region keywork
 //  construct a mesh with 4 regions (sub-domains)
 border a(t=0,1){x=t;y=0;};
 border b(t=0,0.5){x=1;y=t;};
 border c(t=0,0.5){x=1-t;y=0.5;};
```

```
border d(t=0.5,1){x=0.5;y=t;};
border e(t=0.5,1){x=1-t;y=1;};
border f(t=0,1){x=0;y=1-t;};
//   internal boundary
border i1(t=0,0.5){x=t;y=1-t;};
border i2(t=0,0.5){x=t;y=t;};
border i3(t=0,0.5){x=1-t;y=t;};

mesh th = buildmesh (a(6) + b(4) + c(4) +d(4) + e(4) +
    f(6)+i1(6)+i2(6)+i3(6));
fespace Ph(th,P0);  //  constant discontinuous functions / element
fespace Vh(th,P1);  //  P1 ontinuous functions / element

Ph reg=region; //   defined the R0 function  associed to region number
plot(reg,fill=1,wait=1,value=1);
```

`region` is a keyword of freefem++ which is in fact a variable depending of the current position (is not a function today, use `Ph reg=region;` to set a function). This variable value returned is the number of the subdomain of the current position. This number is defined by "buildmesh" which scans while building the mesh all its connected component. So to get the number of a region containing a particular point one does:

```
int nupper=reg(0.4,0.9); //  get the region number of point (0.4,0.9)
int nlower=reg(0.9,0.1);  //  get the region number of point (0.4,0.1)
cout << " nlower " <<  nlower << ", nupper = " << nupper<< endl;
//   defined the characteristics fonctions of upper and lower region
Ph nu=1+5*(region==nlower) + 10*(region==nupper);
plot(nu,fill=1,wait=1);
```

This is particularly useful to define discontinuous functions such as might occur when one part of the domain is copper and the other one is iron, for example.
We this in mind we proceed to solve a Laplace equation with discontinuous coefficients (ν is 1, 6 and 11 below).

```
Ph nu=1+5*(region==nlower) + 10*(region==nupper);
plot(nu,fill=1,wait=1);
problem lap(u,v) =   int2d(th)( dx(u)*dx(v)*dy(u)*dy(v)) + int2d(-1*v)
+ on(a,b,c,d,e,f,u=0);
plot(u);
```

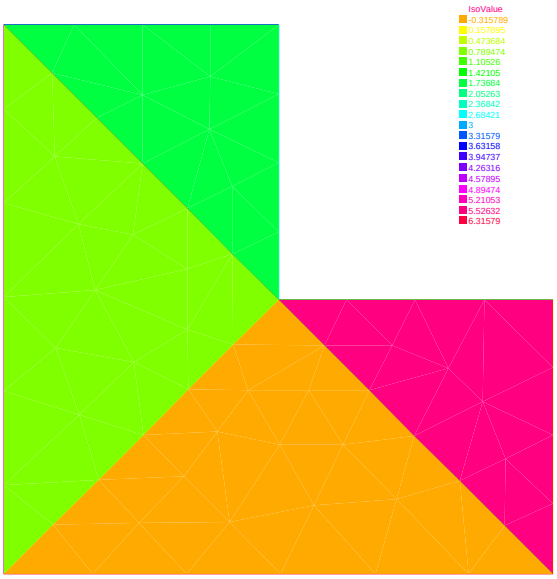


Figure 3.10: the function `reg`

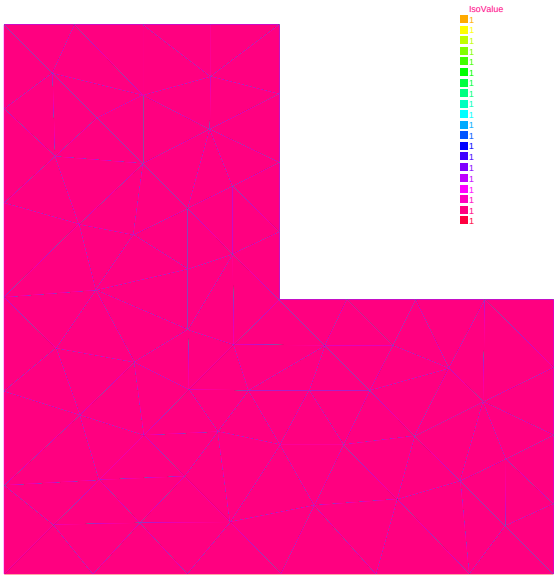


Figure 3.11: the function `nu`

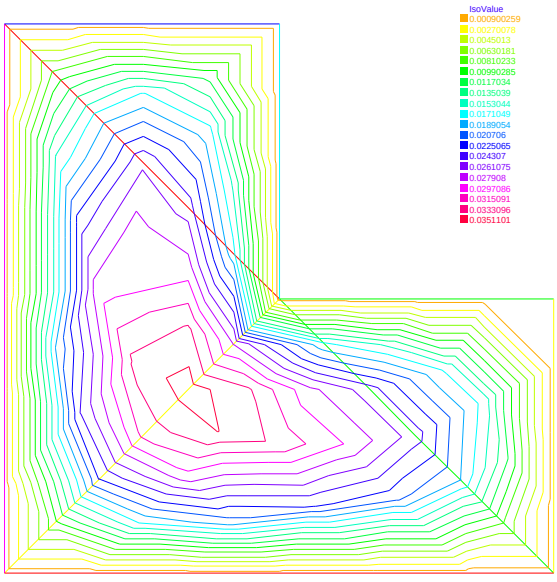


Figure 3.12: the isovalue of the solution u

Chapter 4

Algorithms

4.1 Conjugate Gradient

If we want to solve the euler problem: find $x \in \mathbb{R}^n$ such that

$$\frac{\partial J}{\partial x_i}(x) = 0$$

where J is a functional to minimize from $\mathbb{R}^n$ to $\mathbb{R}$.

if the function is convexe we can use the conjugate gradient to solve the problem, an we just need the function (named `dJ` for example) which compute $\frac{\partial J}{\partial x_i}$, so the two parameters are the name of the function with prototype `func real[int] dJ(real[int] & xx)` which compute $\frac{\partial J}{\partial x_i}$, a vector `x` of type `real[int]` to initialize the process and get the result.

Two version are available:

linearCG linear case , the functional J is quadratic.

NLCG non linear case (the function is just convex).

The named parameter of this two function are:

nbiter= set the number of iteration (by default 100)

precon= set the preconditionner function (`P` for exemple) by default it is the identity, remark the prototype is `func real[int] P(real[int] &x)`.

eps= set the value of the stop test ε ($= 10^{-6}$ by default) if positive then relative test $||dJ(x)||_P \leq \varepsilon * ||dJ(x_0)||_P$, otherwise the absolute test is $||dJ(x)||_P^2 \leq |\varepsilon|$.

veps= set et return the value of the stop test, if positive then relative test $||dJ(x)||_P \leq \varepsilon * ||dJ(x_0)||_P$, otherwise the absolute test is $||dJ(x)||_P^2 \leq |\varepsilon|$. The return value is minus the real stop test (remark: it is usefull in loop).

Example 57

```
real[int] matx(10),b(10),x(10);

func real[int] mat(real[int] &x)
{
```

```
    for (int i=0;i<x.n;i++)
      matx[i]=(i+1)*x[i];
    matx -= b; //  sub the right hand side
    return matx;  //  return of global variable
  };

  func real[int] matId(real[int] &x) { return x;};

  b=1; x=0; //  set  right hand side and initial gest
  LinearCG(mat,x,eps=1.e-6,nbiter=20,precon=matId);
  cout << x;
  //  verification
  for (int i=0;i<x.n;i++)
      assert(abs(x[i]*(i+1) - b[i]) < 1e-5);
  //
  b=1; x=0; //  set  right hand side and initial gest
  NLCG(mat,x,eps=1.e-6,nbiter=20,precon=matId);
```

4.2 Optimization

Two algorithms of COOOL a package [15] are interfaced. the Newton Raphson method (call `Newtow`) and the `BFGS` method. Be careful this algorithm implementation use full matrix.

Example 58 (algo.edp)

```
    func real J(real[int] & x)
      {
        real s=0;
        for (int i=0;i<x.n;i++)
           s +=(i+1)*x[i]*x[i]*0.5 - b[i]*x[i];
           cout << "J ="<< s << " x =" <<  x[0] << " " << x[1] << "...\n"
;
        return s;
      }
    b=1; x=2; //  set  right hand side and initial gest
    BFGS(J,mat,x,eps=1.e-6,nbiter=20,nbiterline=20);
    cout << "BFGS: J(x) = " << J(x) << " err=" << error(x,b) << endl;
```

Chapter 5

Parallel computing

5.1 Parallel version experimental

A first test of parallisation of `FreeFem++` is make under mpi. We add three word in the language:

`mpisize` The total number of processes

`mpirank` the number of my curent process in $\{0, ..., mpisize - 1\}$.

`processor` a function to set the possessor to send or receive data

```
processor(10) << a ; //  send to the process 10 the data a;
processor(10) >> a ; //  receive from the process 10 the data a;
```

5.2 Algorithms written by `freefem++`

We propose the solve the following non-linear academic problem of minimization of a functional

$$J(u) = \int_\Omega f(|\nabla u|^2) - u * b$$

where u is function of $H_0^1(\Omega)$. and where f is defined by

$$f(x) = a * x + x - ln(1 + x), \quad f'(x) = a + \frac{x}{1+x}, \quad f''(x) = \frac{1}{(1+x)^2}$$

5.2.1 Non linear conjugate gradient algorithm

```
mesh Th=square(10,10);  //  mesh definition of Ω
fespace Vh(Th,P1);      //  finite element space
fespace Ph(Th,P0);      //  make optimization
```

A small hack to construct a function

$$Cl = \begin{cases} 1 & \text{on interior degree of freedom} \\ 0 & \text{on boundary degree of freedom} \end{cases}$$

```
//  Hack to construct an array :
//   1 on interior nodes and 0 on boundary nodes
varf vCl(u,v) = on(1,2,3,4,u=1);
Vh Cl;
Cl[]= vCl(0,Vh,tgv=1);  //   0 and tgv
real tgv=Cl[].max;     //
Cl[] = -Cl[];  Cl[] += tgv; Cl[] /=tgv;
```

the definition of f, f', f'' and b

```
//   J(u) = ∫Ω f(|∇ u|^2) - ∫Ω u b
//   f(x) = a*x + x - ln(1+x),   f'(x) = a + x/(1+x),   f''(x) = 1/(1+x)^2
real a=0.001;

func real f(real u) { return u*a+u-log(1+u); }
func real df(real u) { return a+u/(1+u);}
func real ddf(real u) { return 1/((1+u)*(1+u));}
Vh b=1;  //  to defined b
```

the routine to compute the functional J

```
func real J(real[int] & x)
  {
    Vh u;u[]=x;
    real r=int2d(Th)(f( dx(u)*dx(u) + dy(u)*dy(u) ) - b*u) ;
    cout << "J(x) =" << r << " " << x.min <<  " " << x.max << endl;
    return r;
  }
```

The function to compute DJ, where u is the current solution.

```
Vh u=0; //   the current value of the solution
Vh alpha; //  of store  f(|∇ u|^2)
int iter=0;
alpha=df( dx(u)*dx(u) + dy(u)*dy(u) ); //  optimization

func real[int] dJ(real[int] & x)
  {
    int verb=verbosity; verbosity=0;
    Vh u;u[]=x;
    alpha=df( dx(u)*dx(u) + dy(u)*dy(u) ); //  optimization
```

```
      varf au(uh,vh) = int2d(Th)( alpha*( dx(u)*dx(vh) + dy(u)*dy(vh) )
- b*vh);
      x= au(0,Vh);
      x = x.* Cl[]; //   the grad in 0 on boundary
      verbosity=verb;
      return x; //  warning no return of local array
    }
```

We want to construct also a preconditionner function C with solving the problem: find $u_h \in V_{0h}$ such that

$$\forall v_h \in V_{0h}, \quad \int_\Omega \alpha \nabla u_h . \nabla vh = \int_\Omega b v_h$$

where $\alpha = f(|\nabla u|^2)$.

```
varf alap(uh,vh,solver=Cholesky,init=iter)=
   int2d(Th)( alpha *( dx(uh)*dx(vh) + dy(uh)*dy(vh) ))   + on(1,2,3,4,uh=0);

varf amass(uh,vh,solver=Cholesky,init=iter)=  int2d(Th)( uh*vh)  + on(1,2,3,4,u

matrix Amass = alap(Vh,Vh,solver=CG); //

matrix Alap=  alap(Vh,Vh,solver=Cholesky,factorize=1);   //

//  the preconditionner function
func real[int] C(real[int] & x)
{
   real[int] u(x.n);
   u=Amass*x;
   x = Alap^-1*u;
   x = x .* Cl[];
   return x; //  no return of local array  variable
}
```

A good idea solving the problem is make 10 iteration of the conjugate gradient, recompute de preconditonner and restart the conjugate gradient:

```
   verbosity=5;
   int conv=0;
   real eps=1e-6;
   for(int i=0;i<20;i++)
   {
     conv=NLCG(dJ,u[],nbiter=10,precon=C,veps=eps); //
     if (conv) break;  //  if converge break loop

     alpha=df( dx(u)*dx(u) + dy(u)*dy(u) ); //  recompute alpha opti-
mization
     Alap = alap(Vh,Vh,solver=Cholesky,factorize=1);
     cout << " restart with new preconditionner " << conv << " eps ="
<< eps << endl;
   }
```

```
plot (u,wait=1,cmm="solution with NLCG");
```

Remark 22 The keycode `veps=eps` change the value of the current `eps`, this is usefule in this case, because at the first iteration the value of `eps` is change to $-$ the absolute stop test and we save this initial stop test of for the all process. We remove the problem of the relative stop test in iterative procedure, because we start close to the solution and the relative stop test become very hard to reach.

5.2.2 Newtow Ralphson algorithm

Now, we solve the problem with Newtow Ralphson algorithm, to solve the Euler problem $\nabla J(u) = 0$ the algorithme is

$$u^{n+1} = u^n - (\nabla^2 J(u^n)^{-1} * dJ(u^n)$$

First we introduice the two variational form `vdJ` and `vhJ` to compute respectively ∇J and $\nabla^2 J$

```
//    methode of  Newtow Ralphson to solve dJ(u)=0;
//
                    u^{n+1} = u^n - (∂ dJ/∂ u_i)^{-1} * dJ(u^n)

//    ------------------------------------------
  Ph dalpha ; // to store = f''( |∇ u|^2)   optimisation


  //  the variational form of evaluate  dJ =  ∇ J
  //  ------------------------------------
  //   dJ =  f'()*( dx(u)*dx(vh) + dy(u)*dy(vh)
  varf vdJ(uh,vh) =  int2d(Th)( alpha*( dx(u)*dx(vh) + dy(u)*dy(vh) )
- b*vh)
  + on(1,2,3,4, uh=0);


  //  the variational form of evaluate  ddJ   = ∇^2 J
  //  hJ(uh,vh) =  f'()*( dx(uh)*dx(vh) + dy(uh)*dy(vh)
  //             + f''()( dx(u)*dx(uh) + dy(u)*dy(uh) ) * (dx(u)*dx(vh)
+ dy(u)*dy(vh))
  varf vhJ(uh,vh) = int2d(Th)( alpha*( dx(uh)*dx(vh) + dy(uh)*dy(vh) )
   +  dalpha*( dx(u)*dx(vh) + dy(u)*dy(vh)  )*( dx(u)*dx(uh) + dy(u)*dy(uh)
) )
   + on(1,2,3,4, uh=0);

 //  the Newtow algorithm
  Vh v,w;
  u=0;
  for (int i=0;i<100;i++)
```

```
{
 alpha = df( dx(u)*dx(u) + dy(u)*dy(u) ) ; //  optimization
 dalpha = ddf( dx(u)*dx(u) + dy(u)*dy(u) ) ; //  optimization
 v[]= vdJ(0,Vh);  //   v = ∇ J(u)
 real res= v[]'*v[]; //  the dot product
 cout << i <<  " residu^2 = " <<  res  << endl;
 if( res< 1e-12) break;
 matrix H= vhJ(Vh,Vh,factorize=1,solver=LU);
 w[]=H^-1*v[];
 u[] -= w[];
}
plot (u,wait=1,cmm="solution with Newtow Ralphson");
```

5.3 Schwarz in parallel

If exemple is just the rewritting of exemple `schwarz-overlap` see Section 3.7.3

Example 59

```
if ( mpisize != 2 ) {
  cout << " sorry number of processeur !=2 " << endl;
  exit(1);}
verbosity=3;
real pi=4*atan(1);
int inside = 2;
int outside = 1;
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
border e(t=0, pi/2){ x= cos(t); y = sin(t);label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=outside;};
int n=4;
mesh th,TH;

if (mpirank == 0)
 {
   th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
   processor(1) << th ;
   processor(1) >> TH;
}
else
 {
   TH = buildmesh ( e(5*n) + e1(25*n) );
   cout << " end TH  " << endl;
   processor(0) << TH ;
   processor(0) >> th;
 }
```

```
fespace vh(th,P1);
fespace VH(TH,P1);
vh u=0,v; VH U,V;
int i=0;

problem PB(U,V,init=i,solver=Cholesky) =
    int2d(TH)( dx(U)*dx(V)+dy(U)*dy(V) )
  + int2d(TH)( -V) + on(inside,U = u)  +    on(outside,U= 0 ) ;
problem pb(u,v,init=i,solver=Cholesky) =
    int2d(th)( dx(u)*dx(v)+dy(u)*dy(v) )
  + int2d(th)( -v) + on(inside ,u = U)  +    on(outside,u = 0 ) ;


for ( i=0 ;i< 10; i++)
{
  if (mpirank == 0)
  {
   PB;                      //  compute U
   processor(1) << U[];  //  send    U   to 1
   processor(1) >> u[];  //  receive u from 1
  }
  else
  {
   pb;                     //  compute u
   processor(0) << u[]; //  send    u     to 0
   processor(0) >> U[]; //  recieve U from 0
  }
};
 if (mpirank==0)
    plot(U,u,ps="Uu.eps");
```

Chapter 6

Grammar

6.1 Keywords

```
R3
bool
border
break
complex
continue
else
end
fespace
for
func
if
ifstream
include
int
matrix
mesh
ofstream
problem
real
return
solve
string
varf
while
```

6.2 The bison grammar

```
start:   input ENDOFFILE;
```

```
input:   instructions ;

instructions:  instruction
        | instructions  instruction   ;

list_of_id_args:
            | id
            | id '=' no_comma_expr
            | FESPACE id
            | type_of_dcl id
            | type_of_dcl '&' id
            | '[' list_of_id_args ']'
            | list_of_id_args ',' id
            | list_of_id_args ',' '[' list_of_id_args ']'
            | list_of_id_args ',' id '=' no_comma_expr
            | list_of_id_args ',' FESPACE id
            | list_of_id_args ',' type_of_dcl id
            | list_of_id_args ',' type_of_dcl '&' id ;

list_of_id1:  id
            | list_of_id1 ',' id   ;

id: ID | FESPACE ;

list_of_dcls:    ID
              |  ID '='   no_comma_expr
              |  ID  '(' parameters_list ')'
              |  list_of_dcls ',' list_of_dcls  ;


parameters_list:
          no_set_expr
        |  FESPACE  ID
        |  ID '=' no_set_expr
        | parameters_list ',' no_set_expr
        | parameters_list ',' id '=' no_set_expr ;

type_of_dcl:   TYPE
            | TYPE '[' TYPE ']' ;

ID_space:
    ID
 |  ID '[' no_set_expr ']'
 |  ID '=' no_set_expr
 |  '[' list_of_id1 ']'
 |  '[' list_of_id1 ']' '[' no_set_expr ']'
 |  '[' list_of_id1 ']' '=' no_set_expr ;

ID_array_space:
```

```
      ID '(' no_set_expr ')'
    |  '[' list_of_id1 ']' '(' no_set_expr ')' ;

   fespace: FESPACE ;

   spaceIDa  :      ID_array_space
              |    spaceIDa ',' ID_array_space  ;

   spaceIDb  :      ID_space
              |    spaceIDb ',' ID_space ;

   spaceIDs :    fespace               spaceIDb
              |  fespace '[' TYPE ']'  spaceIDa    ;

   fespace_def: ID '(' parameters_list ')' ;

   fespace_def_list:  fespace_def
                   | fespace_def_list ',' fespace_def ;


   declaration:   type_of_dcl list_of_dcls ';'
                | 'fespace' fespace_def_list    ';'
                | spaceIDs ';'
                | FUNCTION ID '=' Expr ';'
                | FUNCTION type_of_dcl ID  '(' list_of_id_args ')'  '{'
instructions'}'
                | FUNCTION ID '(' list_of_id_args ')'   '='   no_comma_expr
';'     ;

   begin: '{'  ;
   end:   '}'  ;

   for_loop:    'for'   ;
   while_loop:  'while' ;

   instruction:   ';'
            | 'include'  STRING
            |  Expr  ';'
            |  declaration
            |  for_loop  '(' Expr ';' Expr ';' Expr ')' instruction
            |  while_loop '(' Expr ')' instruction
            |  'if' '(' Expr ')'   instruction
            |  'if' '(' Expr ')'   instruction  ELSE instruction
            |  begin  instructions end
            |  'border'  ID   border_expr
            |  'border'   ID   '['  array ']' ';'
            |  'break'  ';'
            |  'continue'  ';'
            |  'return'  Expr ';'  ;
```

```
bornes: '(' ID '=' Expr ',' Expr ')' ;

border_expr:   bornes instruction  ;

Expr:    no_comma_expr
       | Expr ',' Expr ;


unop:     '-'
        | '+'
        | '!'
        | '++'
        | '--'  ;

no_comma_expr:
          no_set_expr
        | no_set_expr '=' no_comma_expr
        | no_set_expr '+=' no_comma_expr
        | no_set_expr '-=' no_comma_expr
        | no_set_expr '*=' no_comma_expr
        | no_set_expr '/=' no_comma_expr ;

no_set_expr:
          unary_expr
        | no_set_expr '*' no_set_expr
        | no_set_expr '.*' no_set_expr
        | no_set_expr './' no_set_expr
        | no_set_expr '/' no_set_expr
        | no_set_expr '%' no_set_expr
        | no_set_expr '+' no_set_expr
        | no_set_expr '-' no_set_expr
        | no_set_expr '<<' no_set_expr
        | no_set_expr '>>' no_set_expr
        | no_set_expr '&' no_set_expr
        | no_set_expr '&&' no_set_expr
        | no_set_expr '|' no_set_expr
        | no_set_expr '||' no_set_expr
        | no_set_expr '<' no_set_expr
        | no_set_expr '<=' no_set_expr
        | no_set_expr '>' no_set_expr
        | no_set_expr '>=' no_set_expr
        | no_set_expr '==' no_set_expr
        | no_set_expr '!=' no_set_expr ;


parameters:
        |   no_set_expr
        |   FESPACE
        |   id '=' no_set_expr
```

```
          |   parameters ',' FESPACE
          |   parameters ',' no_set_expr
          |   parameters ',' id '=' no_set_expr ;

  array:   no_comma_expr
         | array ',' no_comma_expr ;


  unary_expr:
      pow_expr
    | unop  pow_expr %prec UNARY ;

  pow_expr: primary
    |      primary  '^' unary_expr
    |      primary  '_' unary_expr
    |      primary ''  ;    //   transpose

  primary:
             ID
    |        LNUM
    |        DNUM
    |        CNUM
    |        STRING
    |        primary '('  parameters ')'
    |        primary '[' Expr ']'
    |        primary '['  ']'
    |        primary '.'  ID
    |        primary '++'
    |        primary '--'
    |        TYPE '('  Expr ')' ;
    |        '(' Expr ')'
    |        '[' array  ']' ;
```

6.3 The Types of the languages, and cast

```
--Add_KN_<double> =  <Add_KN_<double>>

--Add_Mulc_KN_<double> * =  <Add_Mulc_KN_<double>>

--AnyTypeWithOutCheck =  <AnyTypeWithOutCheck>

--C_F0 =  <C_F0>

--DotStar_KN_<double> =  <DotStar_KN_<double>>

--E_Array =  <E_Array>

--FEbase<double> * =  <FEbase<double>>
   <FEbase<double>> :   <FEbase<double>>
```

```
--FEbase<double> ** =  <FEbase<double> **>

--FEbaseArray<double> * =  <FEbaseArray<double>>

--FEbaseArray<double> ** =  <FEbaseArray<double> **>
   []  type :<Polymorphic>   operator :
  (    <FEbase<double> **> :   <FEbaseArray<double> **>, <long> )


--Fem2D::Mesh * =  <Fem2D::Mesh>
   <Fem2D::Mesh> :   <Fem2D::Mesh **>
--Fem2D::Mesh ** =  <Fem2D::Mesh **>
   <-,  type :<Polymorphic>
  (    <Fem2D::Mesh> :   <string> )
  (    <long> :   <Fem2D::Mesh **>, <double>, <double> )

   area,  type :<Polymorphic>   operator. :
  (    <double> :   <Fem2D::Mesh **> )

   nt,  type :<Polymorphic>
  operator. :
  (    <long> :   <Fem2D::Mesh **> )

   nv,  type :<Polymorphic>   operator. :
  (    <long> :   <Fem2D::Mesh **> )


--Fem2D::MeshPoint * =  <Fem2D::MeshPoint>
   N,  type :<Polymorphic>   operator. :
  (    <Fem2D::R3> :   <Fem2D::MeshPoint> )

   P,  type :<Polymorphic>   operator. :
  (    <Fem2D::R3> :   <Fem2D::MeshPoint> )


--Fem2D::R2 * =  <Fem2D::R2>

--Fem2D::R3 * =  <Fem2D::R3>
   x,  type :<Polymorphic>   operator. :
  (    <double *> :   <Fem2D::R3> )

   y,  type :<Polymorphic>   operator. :
  (    <double *> :   <Fem2D::R3> )

   z,  type :<Polymorphic>   operator. :
  (    <double *> :   <Fem2D::R3> )


--Fem2D::TypeOfFE * =  <Fem2D::TypeOfFE>

--KN<double> =  <KN<double>>
   []  type :<Polymorphic>   operator :
  (    <double *> :   <KN<double>>, <long> )


--KN<double> * =  <KN<double> *>
```

```
   <-,  type :<Polymorphic>
  (    <KN<double> *> :   <KN<double> *>, <long> )

   []  type :<Polymorphic>   operator :
  (    <double *> :   <KN<double> *>, <long> )

   max,  type :<Polymorphic>   operator. :
  (    <double> :   <KN<double> *> )

   min,  type :<Polymorphic>   operator. :
  (    <double> :   <KN<double> *> )

   n,  type :<Polymorphic>
  operator. :
  (    <long> :   <KN<double> *> )

   sum,  type :<Polymorphic>   operator. :
  (    <double> :   <KN<double> *> )


--KN_<double> =  <KN_<double>>

--KN_<double> * =  <KN_<double> *>

--Matrice_Creuse<double> * =  <Matrice_Creuse<double>>
   <Matrice_Creuse<double>> :   <Problem>
--Matrice_Creuse_Transpose<double> =  <Matrice_Creuse_Transpose<double>>

--Matrice_Creuse_inv<double> =  <Matrice_Creuse_inv<double>>

--Mulc_KN_<double> =  <Mulc_KN_<double>>

--MyMap<String, double> * =  <MyMap<String, double>>
   []  type :<Polymorphic>   operator :
  (    <double *> :   <MyMap<String, double>>, <string> )


--Polymorphic * =  <Polymorphic>

--Sub_KN_<double> =  <Sub_KN_<double>>

--Transpose<KN<double>> =  <Transpose<KN<double>>>

--TypeSolveMat * =  <TypeSolveMat>

--VirtualMatrice<double>::plusAtx =  <VirtualMatrice<double>::plusAtx>

--VirtualMatrice<double>::plusAx =  <VirtualMatrice<double>::plusAx>

--VirtualMatrice<double>::solveAxeqb =  <VirtualMatrice<double>::solveAxeqb>

--bool =  <bool>
   <bool> :   <bool *>
--bool * =  <bool *>

--char * =  <char>
```

```
--const BC_set<double> * =  <BC_set<double>>

--const CDomainOfIntegration * =  <CDomainOfIntegration>
   ()  type :<Polymorphic>   operator :
  (    <FormBilinear> :   <CDomainOfIntegration>, <LinearComb<std::pair<MGauche, MDroit>,
  (    <double> :   <CDomainOfIntegration>, <double> )
  (    <FormLinear> :   <CDomainOfIntegration>, <LinearComb<MDroit, C_F0>> )


--const C_args * =  <C_args>
   <C_args> :   <FormBilinear>     ()  type :<Polymorphic>   operator :
  (    <Call_FormLinear> :   <C_args>, <long>, <v_fes **> )
  (    <Call_FormBilinear> :   <C_args>, <v_fes **>, <v_fes **> )


--const Call_FormBilinear * =  <Call_FormBilinear>

--const Call_FormLinear * =  <Call_FormLinear>

--const E_Border * =  <E_Border>

--const E_BorderN * =  <E_BorderN>

--const Fem2D::QuadratureFormular * =  <Fem2D::QuadratureFormular>

--const Fem2D::QuadratureFormular1d * =  <Fem2D::QuadratureFormular1d>

--const FormBilinear * =  <FormBilinear>
   ()  type :<Polymorphic>   operator :
  (    <Call_FormBilinear> :   <FormBilinear>, <v_fes **>, <v_fes **> )
  (    <Call_FormLinear> :   <FormBilinear>, <long>, <v_fes **> )


--const FormLinear * =  <FormLinear>
   ()  type :<Polymorphic>   operator :
  (    <Call_FormLinear> :   <FormLinear>, <v_fes **> )


--const IntFunction * =  <IntFunction>

--const LinearComb<MDroit, C_F0> * =  <LinearComb<MDroit, C_F0>>

--const LinearComb<MGauche, C_F0> * =  <LinearComb<MGauche, C_F0>>

--const LinearComb<std::pair<MGauche, MDroit>, C_F0> * =  <LinearComb<std::pair<MGauche, M

--const Problem * =  <Problem>

--const Solve * =  <Solve>

--const char * =  <char>

--double =  <double>
   <double> :   <double *>     ()  type :<Polymorphic>   operator :
  (    <double> :   <double>, <double>, <double> )
```

```
--double * =  <double *>

--interpolate_f_X_1<double>::type =  <interpolate_f_X_1<double>::type>

--long =  <long>
   <long> :   <long *>
--long * =  <long *>

--istream * =  <istream>
   <istream> :   <istream **>
--istream ** =  <istream **>

--ostream * =  <ostream>
   <ostream> :   <ostream **>
--ostream ** =  <ostream **>
   <-,  type :<Polymorphic>   operator( ):
  (    <ostream> :   <string> )


--string * =  <string>
   <string> :   <string **>
--string ** =  <string **>

--std::complex<double> =  <complex>
   <complex> :   <complex *>
--std::complex<double> * =  <complex *>

--std::ios_base::openmode =  <std::ios_base::openmode>

--std::pair<FEbase<double> *, int> =  <std::pair<FEbase<double> *, int>>
   (),  type :<Polymorphic>   operator :
  (    <double> :   <std::pair<FEbase<double> *, int>>, <double>, <double> )
  (    <interpolate_f_X_1<double>::type> :   <std::pair<FEbase<double> *, int>>, <E_Array>

   [],  type :<Polymorphic>  operator. :
  (    <KN<double> *> :   <std::pair<FEbase<double> *, int>> )

   n,  type :<Polymorphic>   operator. :
  (    <long> :   <std::pair<FEbase<double> *, int>> )
--std::pair<FEbaseArray<double> *, int> =  <std::pair<FEbaseArray<double> *, int>>
   []  type :<Polymorphic>
  operator :
  (    <std::pair<FEbase<double> *, int>> :   <std::pair<FEbaseArray<double> *, int>>, <lo
--std::pair<Fem2D::Mesh **, int> * =  <std::pair<Fem2D::Mesh **, int>>
--v_fes * =  <v_fes>
   <v_fes> :   <v_fes **>
--v_fes ** =  <v_fes **>

--void =  <void>
```

6.4 All the operators

```
 - CG,  type :<TypeSolveMat>
```

```
- Cholesky,  type :<TypeSolveMat>
- Crout,  type :<TypeSolveMat>
- GMRES,  type :<TypeSolveMat>
- LU,  type :<TypeSolveMat>
- LinearCG,  type :<Polymorphic>   operator() :
 (    <long> :   <Polymorphic>, <KN<double> *>, <KN<double> *> )

- N,  type :<Fem2D::R3>
- NoUseOfWait,  type :<bool *>
- P,  type :<Fem2D::R3>
- P0,  type :<Fem2D::TypeOfFE>
- P1,  type :<Fem2D::TypeOfFE>
- P1nc,  type :<Fem2D::TypeOfFE>
- P2,  type :<Fem2D::TypeOfFE>
- RT0,  type :<Fem2D::TypeOfFE>
- RTmodif,  type :<Fem2D::TypeOfFE>
- abs,  type :<Polymorphic>  operator() :
 (    <double> :   <double> )

- acos,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )

- acosh,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )

- adaptmesh,  type :<Polymorphic>   operator() :
 (    <Fem2D::Mesh> :   <Fem2D::Mesh>... )

- append,  type :<std::ios_base::openmode>
- asin,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )

- asinh,  type :<Polymorphic>  operator() :
 (    <double> :   <double> )

- atan,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
 (    <double> :   <double>, <double> )

- atan2,  type :<Polymorphic>   operator() :
 (    <double> :   <double>, <double> )

- atanh,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )

- buildmesh,  type :<Polymorphic>   operator() :
 (    <Fem2D::Mesh> :   <E_BorderN> )

- buildmeshborder,  type :<Polymorphic>   operator() :
 (    <Fem2D::Mesh> :   <E_BorderN> )

- cin,  type :<istream>
- clock,  type :<Polymorphic>
 (    <double> :   )

- conj,  type :<Polymorphic>   operator() :
```

```
 (    <complex> :   <complex> )

- convect,  type :<Polymorphic>   operator() :
 (    <double> :   <E_Array>, <double>, <double> )

- cos,  type :<Polymorphic>  operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- cosh,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- cout,  type :<ostream>
- dumptable,  type :<Polymorphic>   operator() :
 (    <ostream> :   <ostream> )

- dx,  type :<Polymorphic>   operator() :
 (    <LinearComb<MDroit, C_F0>> :   <LinearComb<MDroit, C_F0>> )
 (    <double> :   <std::pair<FEbase<double> *, int>> )
 (    <LinearComb<MGauche, C_F0>> :   <LinearComb<MGauche, C_F0>> )

- dy,  type :<Polymorphic>   operator() :
 (    <LinearComb<MDroit, C_F0>> :   <LinearComb<MDroit, C_F0>> )
 (    <double> :   <std::pair<FEbase<double> *, int>> )
 (    <LinearComb<MGauche, C_F0>> :   <LinearComb<MGauche, C_F0>> )

- endl,  type :<char>
- exec,  type :<Polymorphic>   operator() :
 (    <long> :   <string> )

- exit,  type :<Polymorphic>  operator() :
 (    <long> :   <long> )

- exp,  type :<Polymorphic>  operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- false,  type :<bool>
- imag,  type :<Polymorphic>   operator() :
 (    <double> :   <complex> )

- int1d,  type :<Polymorphic>   operator() :
 (    <CDomainOfIntegration> :   <Fem2D::Mesh>... )

- int2d,  type :<Polymorphic>   operator() :
 (    <CDomainOfIntegration> :   <Fem2D::Mesh>... )

- label,  type :<long *>
- log,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- log10,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
```

```
- max,  type :<Polymorphic>   operator() :
 (    <double> :   <double>, <double> )
 (    <long> :   <long>, <long> )

- min,  type :<Polymorphic>  operator() :
 (    <double> :   <double>, <double> )
 (    <long> :   <long>, <long> )

- movemesh,  type :<Polymorphic>   operator() :
 (    <Fem2D::Mesh> :   <Fem2D::Mesh>, <E_Array>... )

- nu_triangle,  type :<long *>
- on,  type :<Polymorphic>   operator() :
 (    <BC_set<double>> :   <long>... )

- pi,  type :<double>
- plot,  type :<Polymorphic>   operator() :
 (    <long> :  ... )

- pow,  type :<Polymorphic>   operator() :
 (    <double> :   <double>, <double> )
 (    <complex> :   <complex>, <complex> )

- qf1pE,  type :<Fem2D::QuadratureFormular1d>
- qf1pT,  type :<Fem2D::QuadratureFormular>
- qf2pE,  type :<Fem2D::QuadratureFormular1d>
- qf2pT,  type :<Fem2D::QuadratureFormular>
- qf3pE,  type :<Fem2D::QuadratureFormular1d>
- qf5pT,  type :<Fem2D::QuadratureFormular>
- readmesh,  type :<Polymorphic>   operator() :
 (    <Fem2D::Mesh> :   <string> )

- real,  type :<Polymorphic>   operator() :
 (    <double> :   <complex> )

- region,  type :<long *>
- savemesh,  type :<Polymorphic>  operator() :
 (    <Fem2D::Mesh> :   <Fem2D::Mesh>, <string>... )

- sin,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- sinh,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- sqrt,  type :<Polymorphic>   operator() :
 (    <double> :   <double> )
 (    <complex> :   <complex> )

- square,  type :<Polymorphic>    operator() :
 (    <Fem2D::Mesh> :   <long>, <long> )
 (    <Fem2D::Mesh> :   <long>, <long>, <E_Array> )

- tan,  type :<Polymorphic>   operator() :
```

```
 (    <double> :   <double> )

- true,  type :<bool>
- trunc,  type :<Polymorphic>   operator() :
 (    <Fem2D::Mesh> :   <Fem2D::Mesh>, <bool> )

- verbosity,  type :<long *>
- wait,  type :<bool *>
- x,  type :<double *>
- y,  type :<double *>
- z,  type :<double *>
```

Bibliography

[1] D. Bernardi, F.Hecht, K. Ohtsuka, O. Pironneau: *freefem+ documentation*, on the web at ftp://www.freefem.org/freefemplus.

[2] D. Bernardi, F.Hecht, O. Pironneau, C. Prud'homme: *freefem documentation*, on the web at http://www.asci.fr

[3] P.L. George: *Automatic triangulation*, Wiley 1996.

[4] F. Hecht: The mesh adapting software: bamg. INRIA report 1998.

[5] R. Dautray and J.L.Lions, Mathematical Analysis and Numerical Methods for Science and Technology, Vol. 1 "Physical Origins and Classical Methods", Springer-Verlag, 1990.

[6] J.L. Lions, O. Pironneau: Parallel Algorithms for boundary value problems, Note CRAS. Dec 1998. Also : Superpositions for composite domains (to appear)

[7] B. Lucquin, O. Pironneau: *Scientific Computing for Engineers* Wiley 1998.

[8] F. Preparata, M. Shamos; *Computational Geometry* Springer series in Computer sciences, 1984.

[9] R. Rannacher: On Chorin's projection method for the incompressible Navier-Stokes equations, in "Navier-Stokes Equations: Theory and Numerical Methods" (R. Rautmann, et al., eds.), Proc. Oberwolfach Conf., August 19-23, 1991, Springer, 1992

[10] J.L. Steger: The Chimera method of flow simulation, Workshop on applied CFD, Univ of Tennessee Space Institute, August 1991.

[11] N. Wirth: *Algorthims + Data Structures = Programs*, Prentice Hall, 1976.

[12] The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. Visit `http://www.sgi.com/tech/stl/` for more informations.

[13] Bison is a general-purpose parser generator that converts a grammar description for an LALR context-free grammar into a C program to parse that grammar.
Visit `http://www.gnu.org/manual/bison/index.html`, for more information.

[14] Bjarne Stroustrup, The `C++` programming language, Third edition, Addison-Wesley, 1997.

[15] COOOL: a package of tools for writing optimization code and solving optimization problems. Visit `http://coool.mines.edu/` for more informations.

[16] J. Nečas and L/ Hlaváček, Mathematical theory of elastic and elasto-plastic bodies: An introduction, Elsevier, 1981.

[17] K. Ohtsuka, O. Pironneau and F. Hecht, Theoretical and Numerical analysis of energy release rate in 2D fracture, INFORMATION **3** (2000), 303–315.

Index