An Object Oriented Finite Element Library

User's Guide

Rachid Touzani
Laboratoire de Mathématiques Blaise Pascal
Université Clermont Auvergne, France
e-mail: rachid.touzani@uca.fr

Contents

We illustrate in this document various ways for using the library **OFELI**. The simplest level is to use an already prepared finite element code that makes use of **OFELI**. The most advanced level is the one that consists in developing specific classes for one's own problem and using utility classes in **OFELI**.

1 Generalities

It is useful to recall that **OFELI** is not itself a finite element code but a toolkit for developing finite element programs for specific applications. The **OFELI** package contains however some more sophisticated applications that help solving particular problems in several fields such as Thermal Analysis, Fluid Flow, Solid Mechanics and Electromagnetics. This makes possible for a beginner to start with simple codes or better to imitate what is already developed in order to develop new codes.

This user's guide is organized as follows : a first part consists of a tutorial divided into lessons of ascending complexity. Here various type of finite element codes are described as well as some useful aspects like using data files and file converters. The second part describes how to use **OFELI** as a toolbox to develop one's own codes. That is, we explain through a significant application the general methodology to write codes. The third part shows how to develop a new equation.

2 A Tutorial

We describe in this part some examples of finite element codes that make use of the **OFELI** library. We describe in detail the source files and associated user defined functions.

1. *Lesson 1*: A One–dimensional problem
2. *Lesson 2*: A Two–dimensional steady–state diffusion equation using a 3–Node triangle. The linear system of equations is solved by a direct method.
3. *Lesson 3*: The same problem using an iterative method.
4. *Lesson 4*: A Two–dimensional time dependent diffusion equation using a 3–Node triangle and the backward Euler scheme.
5. *Lesson 5*: A Two–dimensional steady–state diffusion equation solved as an unconstrained optimization problem.
6. *Lesson 6*: How to use a project file.
7. *Lesson 7*: How to use the 2–D mesh generator.
8. *Lesson 8*: How to use file converters.

As it can be seen, the lessons are progressive, *i.e.* it is preferable to learn them in increasing order.

2.1 A One Dimensional Problem

This lesson concerns a simple one–dimensional two–point boundary value problem.

2.1.1 The main code

Let us examine in detail the source file.

– We start by including the header file `OFELI.h` that itself includes all kernel class definitions.

```
#include "OFELI.h"
```

– The **OFELI** library is embedded in namespace called `OFELI`.

```
using namespace OFELI;
```

– Our program has arguments that will described later.

```
int main(int argc, char *argv[])
{
```

– `Lmin` and `Lmax` are the ends of the interval in which the problem is defined. Here we have fixed their respective values at `0` and `1`. `N` is the number of finite elements. Its default value is `10`.

```
   double L=1;
   int N=10;
```

– The **OFELI** function `banner` outputs the official banner of the library:

```
   banner();
```

– `N` is the program argument (if this one is present).

```
if (argc > 1)
   N = atoi(argv[1]);
```

– We now declare an instance of class `Mesh` with the appropriate constructor.

```
Mesh ms(L,N);
```

– We denote by `NbN` the number of unknowns, the solution being prescribed at `x=0` and then we print out the mesh.

```
int NbN = N+1;
cout << ms;
```

– We declare an instance of class `TrMatrix<double>` for the tridiagonal matrix, with size `NbN` and an instance of class `Vect<double>` for the right–hand side and the solution.

```
TrMatrix<double> A(NbN);
Vect<double> b(NbN);
```

– In order to test the code, we choose an exact solution: We set for right-hand side the function $f(x) = 20(1 - 20x^2)e^{-10x^2}$ and the boundary conditions $u(0) = u(1) = 0$. This yields the solution $u(x) = e^{-10x^2} + x(1 - e^{-10}) - 1$. To implement this without implementing additional functions we resort to the **OFELI**'s parser. For this, the member function set of class `Vect<double>` enables assigning regular expressions to nodes in function of their coordinates. The variables are `x`, `y` and `z`.

```
b.set(ms,"20*(1-20*x*x)*exp(-10*x*x)");
```

– `h` is the mesh size (length of an element). We can compute the right-hand side of the linear system by multiplying it by the mesh size.

```
double h = L/double(N);
b *= h;
```

– We now build up the matrix and the right-hand side. Note that we skip, for the moment, the first and the last lines for boundary condition treatment. Here `x` is the `i`–th node coordinate.

```
for (int i=2; i<NbN; i++) {
   A(i,i)   = 2./h;
   A(i,i+1) = -1./h;
   A(i,i-1) = -1./h;
}
```

– We modify the first and last equation in order to take account for boundary conditions.

```
A(1,1) = 1.; A(1,2) = 0.; b(1) = 0;
A(NbN,NbN) = 1.; A(NbN-1,NbN) = 0.; b(NbN) = 0;
```

– The linear system is solved.

```
A.solve(b);
```

– We finally output the solution, calculate the error at each node and output it. Note that the exact solution vector use also the parser.

```
cout << "\nSolution :\n" << b;
Vect<double> sol(NbN);
sol.set(ms,"exp(-10*x*x)+x*(1-exp(-10))-1");
cout << "Error = " << (b-sol).getNormMax() << endl;
```

– We can now end the program.

```
    return 0;
}
```

2.1.2 An Example

If you execute the program without any argument, you will obtain as output:

```
M E S H     D A T A
====================

Space Dimension         :       1
Number of nodes         :      11
Number of elements      :      10
Number of sides         :       0


Solution :
     1     0.00000000e+000
     2     1.71545445e-003
     3    -1.41343078e-001
     4    -3.11214412e-001
     5    -4.16034601e-001
     6    -4.32020322e-001
     7    -3.82338044e-001
     8    -2.98774350e-001
     9    -2.02104670e-001
    10    -1.01513714e-001
    11     0.00000000e+000

Error = 1.79492789e-002
```

2.2 A Two–dimensional steady–state diffusion equation

We consider here a 2–D steady state boundary value problem. We solve a Poisson equation (Diffusion) with "simple" data. Concerning boundary conditions, we impose a Dirichlet (essential) boundary condition on a portion of the domain and a homogeneous Neumann (natural) condition on the remaining boundary. Note that owing to the variational formulation the Neumann condition is implicit (*We have nothing to do for it*).

2.2.1 The Finite Element Code

Here is a description of the source code.

– As usual we start by including the principal header file and the file `Therm.h` that includes all classes related to heat transfer problems.

```
#include "OFELI.h"
#include "Therm.h"
using namespace OFELI;
```

– Our program will have as argument the mesh file name.

```
int main(int argc, char *argv[])
{
```

– We output the **OFELI** banner and get the program argument

```
banner();
if (argc <= 1) {
   cout << "Usage : lesson2 <mesh_file>" << endl;
   exit(1);
}
```

– We construct an instance of class `Mesh` by giving the name of mesh file. Note that

```
Mesh ms(argv[1]);
```

– The problem matrix is symmetric and will be stored in skyline format (thus using class `SkSMatrix<double>`)

```
SkSMatrix<double> A(ms);
```

– Vectors `b` and `bc` (as instances of class `Vect<double>`) will contain the right-hand side and the solution, and imposed boundary conditions at nodes:

```
Vect<double> b(ms.getNbDOF()), bc(ms.getNbDOF());
```

– We assign imposed boundary conditions to vector `bc` by using member function `setNodeBC` which allows using an interpreted function of node coordinates: We prescribe the function y to nodes with code `1`.

```
bc.setNodeBC(ms,1,"y");
```

– We now start building the linear system. For this, we have to implement a loop over all element meshes by using member functions `TopElement()` and `getElement()`. The returned pointer `theElement` enables access to current element data. The **OFELI** library defines the macro `MeshElements(ms)` that stands for a shorthand for the line

```
for (ms.topElement(); (theElement=ms.getElement());)
```

For each element, we construct an instance of class `DC2DT3` for diffusion-convection problems in 2-D using 3-node triangles. The member function `Diffusion` calculates the contribution to element matrix diffusion term. We then assemble matrix and right-hand side (which is `0` here).

```
MeshElements(ms) {
   DC2DT3 eq(theElement);
   eq.Diffusion();
   eq.ElementAssembly(A);
   eq.ElementAssembly(b);
}
```

Of course, in the present case, assembling the right-hand side is actually useless.

– Once the linear system is assembled, Dirichlet boundary conditions are imposed by a penalty technique. This is implemented via the function `Prescribe`, member of all matrix classes.

```
A.Prescribe(ms,b,bc);
```

– Solution is obtained by factorizing and backsubstituting:

```
A.solve(b);
```

Vector `b` contains now the solution.

– We finally output the solution and end the program.

```
    cout << "\nSolution:\n" << b;
    return 0;
}
```

2.2.2 A finite element mesh

To test this program we use a finite element mesh of a rectangle `[0,3]x[0,1]`. The imposed boundary conditions are

$$u(x,0) = 0, u(x,1) = 1, \qquad 0 < x < 3.$$

Homogeneous Neumann boundary conditions are "imposed" on the portions $x = 0$ and $x = 3$. The solution is then

$$u(x,y) = y.$$

The mesh file is called `test.xml` (included in the package). Note, in this file, that a code equal to `0` is associated to nodes with y=0 and `1` is associated to nodes with `y=1`. The lines starting with `BC` give the associated values to these codes, the case of value `0` is by default.

You can now execute the code to obtain the exact solution.

2.3 Using an iterative solver

We consider the same example as in *Lesson 2* with the following modifications :

1. We solve the linear system using the Conjugate Gradient method.
2. In view of an iterative method we prefer to use, to handle boundary conditions, a classical substitution method rather than the penalty formulation.
3. We use a defined material by giving its name in the mesh file.

2.3.1 The Finite Element Code

The main program

Here is a description of the source code.

– We start like in *Lesson 2*.

```
#include "OFELI.h"
#include "Therm.h"
using namespace OFELI;

int main(int argc, char *argv[])
{
```

– As usual, we declare an instance of class `Mesh`.

```
    Mesh ms(argv[1]);
    banner();
```

– We expand the argument of the program :

```
    if (argc <= 1) {
      cout << "Usage: lesson3 <mesh_file>" << endl;
      exit(1);
    }
```

– After reading mesh data we note that handling boundary conditions by elimination requires renumbering the equations. For this, we invoke the member class `NumberEquations` of class `Mesh`.

```
Mesh ms(argv[1]);
ms.NumberEquations();
```

– We store the matrix in a sparse format , thus using class `SpMatrix<double>`.

```
SpMatrix<double> A(ms);
```

– Vectors `b` and `x` will store respectively the right-hand side and the solution. Note that, since imposed degrees of freedom are eliminated from the equations, the vectors have as sizes the actual number of equations.

```
Vect<double> b(ms.getNbEq()), x(ms.getNbEq());
```

– The vector `bc`, instance of class `Vect<double>` will store imposed boundary conditions. It is constructed in the same way as in Lesson 2.

```
BCVect<double> bc(ms.getNbDOF());
```

– We construct the linear system of equations just as in *Lesson 2*. The difference here is that element right-hand sides need to be updated to take into account imposed boundary conditions at element level. This is necessary when using an elimination technique for boundary conditions. The member function `UpdateBC` is then used before assembly.

```
MeshElements(ms) {
   DC2DT3 eq(theElement);
   eq.Diffusion();
   eq.updateBC(bc);
   eq.ElementAssembly(A);
   eq.ElementAssembly(b);
}
```

– We will use a preconditioned Conjugate Gradient . As an example, we use here the ILU (Incomplete LU factorization) preconditioner . This is realized by calling the function `CG` to run the conjugate gradient. We impose a tolerance of 10^{-8}.

```
double toler = 1.e-8;
int nb_it = CG(a,Prec<double>(A,ILU_PREC),b,x,1000,toler,2);
```

Note that `CG` returns the number of performed iterations.

– We print out this number of iterations.

```
cout << "Nb. of iterations: " << nb_it << endl;
```

– We can incorporate boundary conditions into the solution vector.

```
Vect<double> u(ms.getNbDOF());
u.insertBC(ms,x,bc);
```

– We finally end the program.

```
return 0;
}
```

How to declare a material ?

This is very simple: in the mesh data file, all elements have, in the present example, the code `1`. If we say nothing, then a generic material (precisely called `GenericMaterial` with default properties is used. Otherwise, we can assign, in the mesh file a material, here Aluminium to this code, by the line

```
<Material>1  Aluminium</Material>
```

This line must be given after all elements lines. Note that the file `Aluminium.md` must be present in the material's directory. We are now ready to test the package.

2.3.2 A test

We use here a finer mesh than in *Lesson 2* and add the line defining the material. We obtain the same solution as *Lesson 2* after 15 iterations.

2.4 A time dependent problem

We introduce, in this lesson, new aspects of **OFELI** programming :

1. We consider a time-dependent heat transfer problem that we solve by Backward Euler time stepping scheme.
2. We consider the case of Neumann Boundary conditions.
3. Data (problem parameters) are introduced by a data file using `IPF`.

2.4.1 The Finite Element Code

The main program

Here is a description of the source code.

– We start, as usual, by including required headers and naming the appropriate namespace.

```
#include "OFELI.h"
#include "Therm.h"
#include "User.h"
using namespace OFELI;
```

– The program will have as argument the name of the parameter data file :

```
int main(int argc, char *argv[])
{
```

– We expand program arguments and declare an instance of class `IPF` for parameter file :

```
   if (argc <= 1) {
      cout << "Usage: lesson4 <parameter_file>" << endl;
      exit(1)
   }
   IPF data(argv[1]);
```

– Parameters `max_time` (maximum time value) and `deltat` (time step) are retrieved as `IPF` class members.

```
   double max_time = data.getMaxTime();
   double deltat = data.getTimeStep();
```

– The mesh instance is constructed by giving the mesh file.

```
   Mesh ms(data.getMeshFile());
```

– In the present example, we introduce boundary conditions through a user defined class. This may be optional for Dirichlet conditions but necessary for Neumann ones.

```
   User ud(ms);
```

Implementation of class `User` will be given later.

– We declare matrix and vector data: first, the matrix `A` is declared as instance of class `SkSMatrix` . The vectors `b`, `u` and `bc` will contain respectively, alternatively the right-hand side and the current solution, the previous solution and prescribed Dirichlet boundary conditions.

```
SkSMatrix<double> A(ms);
Vect<double> b(ms.getNbDOF()), u(ms.getNbDOF()), bc(ms.getNbDOF());
```

– Since, we are dealing with a transient problem, we need initial data. This is retrieved from class member `setInitialData` of class `User`:

```
ud.setInitialData(u);
```

– Before starting time stepping loop, we calculate the number of time steps and initialize time:

```
int nb_step = int(max_time/deltat);
double time = 0;
```

– We start a loop over time steps :

```
for (int step=1; step<=nb_step; step++) {
```

– The first thing to do here is to update time value and initialize the right-hand side to zero since this one will be assembled.

```
    time += deltat;
    b = 0;
```

– We next transmit the user data class instance `ud` the time value:

```
    ud.setTime(time);
```

– In order to deal with a problem with time-dependent boundary condition we re-fill vector `bc` at this level.

```
    ud.setDBC(bc);
```

– We write a loop over finite elements as in the previous lessons:

```
    MeshElements(ms) {
```

– We use here class `DC2DT3` with the constructor that involves time. Instance `eq` will then be used to build matrix and right-hand side.

```
        DC2DT3 eq(theElement,u,time);
```

– The element matrix is constructed with capacity term (chosen here to be lumped) and diffusion term :

```
        eq.LCapacity(1./deltat);
        eq.Diffusion();
```

Note that capacity matrix is multiplied by the inverse of time step. This is necessary to implement the backward Euler scheme.

– We assemble matrix and right-hand side (useless for the present example). Note that, since the matrix does not depend on time, it is assembled once and factorized once.

```
        if (step==1)
           eq.ElementAssembly(A);
        eq.ElementAssembly(b);
    }
```

The loop on elements is closed.

– To deal with Neumann boundary conditions (involving boundary integrals), we have to loop over given sides. The loop looks like the one over elements :

```
MeshSides(ms) {
```

– For each side (pointed by `eq`) we invoke a constructor that involves sides.

```
   DC2DT3 eq(theSide,u,time);
```

– We fill the side vector using the instance `ud` of class `User`. The function `BoundaryRHS` calculates the side integral.

```
eq.BoundaryRHS(ud);
```

– We assemble side vectors just like for elements and close the loop.

```
   eq.SideAssembly(b);
}
```

– Once the linear system is assembled, we impose Dirichlet boundary conditions by a penalty techniques implemented in member function `Prescribe`:

```
A.Prescribe(ms,b,bc,step-1);
```

– As said before, factorization is carried out at the first time step only. Obviously, solution is called each time step.

```
A.solve(b);
```

– Now, vector `b` contains the solution. We copy it to `u` to store it as a previous solution.

```
u = b;
```

– We may want to output the solution each time step:

```
   cout << "\nSolution for time: " << time << endl << u;
}
return 0;
}
```

and then close the time stepping loop and the program.

A User defined class

We have now to implement class `User` that defines boundary conditions, initial conditions, . . . The class is defined in file `User.h`.

– Of course, we start by including file **OFELI**. and invoking the namespace :

```
#include "OFELI.h"
using namespace OFELI;
```

– Class `User` inherits from abstract class `UserData`.

```
class User : public UserData<double> {
```

– This class has only public members and no attributes.

```
public :
```

– We have a constructor that provides the mesh to the class : nothing to do, the parent class does the job for you.

```
User(Mesh &mesh) : UserData<double>(mesh) { ; }
```

– We define member function to give a value to prescribe for boundary condition in function of node code, node coordinates, time value and degree of freedom : Here, we impose that a code 2 imposes the value 1.0. Any other code will impose the default value 0.0.

```
double BoundaryCondition(const Point<double> &x, int code,
                         double time=0., size_t dof=1)
{
   double ret = 0.0;
   if (code == 2)
      ret = 1.0;
   return ret;
}
```

– The same scheme works for Neumann boundary condition :

```
double SurfaceForce(const Point<double> &x, int code,
                    double time, size_t dof)
{
   if (code)
      return 1.0;
   else
      return 0.0;
}
```

– The class definition ends here.

```
};
```

– Let us finally note that since no implementation is given for initial condition, the default one is 0.0 for each degree of freedom.

2.4.2 A test

We use here exactly the same mesh file as in the previous lesson. Of course, we obtain the same solution. The convergence is obtained after 3 iterations.

2.5 An optimization problem

The present lesson demonstrates how to use an optimization problem solver. We solve the same problem as in Lesson 2 as an optimization problem where Dirichlet boundary conditions are considered as equality constraints. The optimization algorithm is the Truncated Newton algorithm described in function OptimTN. This is not the best method to solve a Laplace equation, but our purpose here is to learn how to use this class.

2.5.1 The Finite Element Code

The main program

Here is a description of the source code.

- We start, as usual, by including required headers and naming the appropriate namespace. We furthermore include the header file of the optimization definition class called Opt.

```
#include "OFELI.h"
#include "Opt.h"
#include "User.h"
using namespace OFELI;
```

- The program will have as argument the name of the parameter data file :

```
int main(int argc, char *argv[ ])
{
```

- We next declare a pointer to class Element that will be used later.

```
Element *el;
```

- We expand program arguments and declare an instance of class IPF for parameter file :

```
if (argc <= 1) {
   cout << "Usage : ex5 <parameter_file>" << endl;
   exit(1)
}
IPF data(argv[1]);
```

- The mesh data file is obtained from a member function of instance `data`.

```
Mesh ms(data.getMeshFile());
```

- We introduce boundary conditions through a user defined class (See Lesson 4).

```
User ud(ms);
```

  Implementation of class **`User`** will be given later.

- **`n`** is the number of degrees of freedom.

```
int n = ms.getNbDOF();
```

- We declare some vectors: `x` is the solution vector, vectors `low` and `up` will contain for each degree of freedom upper and lower bound respectively to prescribe and `pivot` is a vector that will contains (after optimization) flags to indicate which constraint is reached.

```
Vect<double> x(n), low(n), up(n);
Vect<int> pivot(n);
```

- Dirichlet boundary conditions are taken into account as before :

```
Vect<double> bc(n);
ud.setDBC(bc);
```

- We start now to properly define the optimization problem. For this, we declare an instance of a class `Opt` that is defined separately. The constructor of this class invokes the mesh instance and the instance **`ud`** that will be useful to transmit to the class body or boundary sources.

```
Opt theOpt(ms,ud);
```

- We also initialize the solution to zero :

```
x = 0;
```

- As said before, Dirichlet boundary conditions are considered here as equality constraints and are then incorporated into vectors low and up via a utility function called `BCAsConstraint` contained in the mentioned file `OptimAux.h`.

```
BCAsConstraint(ms,bc,up,low);
```

- The function `OptimTN` can now be invoked to run the optimization algorithm.

```
OptimTN<Opt>(theOpt,x,low,up,pivot,100,1.e-12,1);
```

  The reader can refer to the function `OptimTN` to understand the meaning of each argument of the function.

- If the algorithm has succeeded (which is the case in this example) we can output the solution and close the program.

```
   cout << "\nSolution :\n" << x;
}
```

A user defined class

We have now to implement class `Opt` that defines the problem to solve and provides to the objective function and its gradient. The class is defined in file `Opt.h` that we study here below.

- We start by including files `OFELI.h` and `Therm.h` since we are going to solve a heat transfer problem. We also invoke the namespace `OFELI`:

```
#include "OFELI.h"
#include "Therm.h"
#include "User.h"
using namespace OFELI;
```

  This class will have a constructor that acquires the mesh and the user data instance and stores pointers to these objects:

```
class Opt {

public:
   Opt(Mesh &ms, User &ud) { _ms = &ms; _ud = &ud; }
```

- The other public member is the objective function.

```
void Objective(Vect<double> &x, double &f, Vect<double> &g) {
   f = 0.;
   g = 0.;
   MeshElements(*_ms) {
      DC2DT3 eq(theElement);
      Vect<double> ge(3);
      Vect<double> xe(theElement,x);
      f += eq.Energy(xe,*_ud);
      eq.EnerGrad(xe,*_ud,ge);
      g.Assembly(theElement,ge);
   }
}
```

  Note that the arguments of the objective function are necessarily the optimization variable vector, the objective function to compute and the gradient vector to compute. Here the class `DC2DT3` provides necessary material for optimization purposes. Namely, member function `Energy` return the energy value and function `EnerGrad` calculates the element energy gradient that needs be assembled into the global vector `g`.

- It remains to declare the mesh and user data pointers as private attributes of the class and end the class definition.

```
   private:
      Mesh *_ms;
      User *_ud;
};
```

2.5.2 A test

We use here exactly the same mesh file as in the previous lesson. Of course, we obtain the same solution. The output of the program is the following:

```
NIT  NF   CG          F                  GTG

 0    1    0     6.00000000e+000     4.00000000e+001
```

```
    1    2    3     1.67852990e+000     4.09059524e+001
    2    3    5     1.50042385e+000     4.09106651e+001
    3    4    8     1.50001179e+000     4.09109024e+001
    4    5   10     1.50000062e+000     4.09109074e+001
    5    6   13     1.50000000e+000     4.09109074e+001
    6    7   15     1.50000000e+000     4.09109074e+001
    7    8   17     1.50000000e+000     4.09109074e+001
    8    9   19     1.50000000e+000     4.09109074e+001


Optimal Function Value = 1.5

Solution :
     1     1.00000000e+000
     2     7.49999986e-001
     3     4.99999972e-001
     4     2.49999994e-001
     5     0.00000000e+000
     6     1.00000000e+000
     7     7.49999988e-001
     8     4.99999982e-001
     9     2.49999978e-001
    10     0.00000000e+000
    11     1.00000000e+000
    12     7.49999989e-001
    13     4.99999982e-001
    14     2.49999978e-001
    15     0.00000000e+000
    16     1.00000000e+000
    17     7.49999986e-001
    18     4.99999972e-001
    19     2.49999994e-001
    20     0.00000000e+000
```

2.6 Using Project File

We present here an example of use of the **Project File**. This powerful tool simplifies for a user the introduction of input data specific to **OFELI**. The idea is very simple : you have to write a text file following some simple rules and declare, in your code, an instance of class **`IPF`**. Each parameter introduced in your file is then recovered from a specific member function from your instance. We shall illustrate hereafter this through an example of a fluid flow code.

Consider the following text file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<OFELI_File>
<info>
   <title></title>
   <date>January 1, 2000</date>
   <author></author>
</info>
<Project name="test">
   <max_time>1000.</max_time>
   <time_step>0.01</max_time>
   <verbose value="1"/>
   <output value="0"/>
```

```
   <save value="0"/>
   <tolerance value="1.e-5"/>
   <plot value="1000"/>
   <init value="1"/>
   <parameter label="density">1.0</parameter>
   <parameter label="viscosity">0.01</parameter>
   <mesh_file value="test.m"/>
   <parameter label="output_file" value="test.v"/>
   <save_file value="test.sav"/>
</Project>
</OFELI_File>
```

In the program that uses such a file, we have the following lines:

```
   IPF proj(argv[1]);
   double max_time = proj.getMaxTime();
   double deltat = proj.getTimeStep();
   int verbose = proj.getVerbose();
   int output_flag = proj.getOutput();
   int save_flag = proj.getSave();
   double tol = data.getTolerance();
   int plot_flag = proj.getPlot();
   double dens = proj.getString("density");
   double visc = data.getString("viscosity");
   Mesh ms(proj.getMeshFile());
   IOField vf(proj.getMeshFile(),data.getString("output_file"),ms,XML_WRITE);
   int init_flag = proj.getInit();
```

In this way, all these parameters are retrieved in a finite element program without any explicit i/o operation.

2.7 Using mesh generator

The current release of **OFELI** is not provided within a native mesh generator. For this, we prefer to add to the package a public domain mesh generator that we have interfaced with **OFELI** classes. The included mesh generator is called BAMG. It was developed by an INRIA team.

To generate a 2-D finite element mesh you have to use the class Domain to create a domain and then call the function BAMG that generates a mesh file in the OFELI XML format. The following example contained in the **OFELI** package illustrates a typical usage of the mesh generator. It uses the program g2m created while you install the utilities

Let us give an example of OFELI XML to generate a domain and then a mesh:

```
s

<?xml version="1.0" encoding="ISO-8859-1" ?>
<OFELI_File>
<info>
    <title>Definition of a domain with a hole</title>
    <date>January 1, 2010</date>
    <author>R. Touzani</author>
</info>
<Domain dim="2">
   <vertex>   0.0     0.0   2    0.10</vertex>
   <vertex>   1.0     0.0   2    0.10</vertex>
   <vertex>   1.0     1.0   2    0.10</vertex>
   <vertex>   0.0     1.0   2    0.10</vertex>
   <vertex>   0.4     0.4   1    0.01</vertex>
```

```
   <vertex>   0.6     0.4   1    0.01</vertex>
   <vertex>   0.6     0.6   1    0.01</vertex>
   <vertex>   0.4     0.6   1    0.01</vertex>
   <vertex>   0.1     0.2   1    0.01</vertex>
   <vertex>   0.2     0.2   1    0.01</vertex>
   <line>   1     2     2 </line>
   <line>   2     3     2 </line>
   <line>   3     4     2 </line>
   <line>   4     1     2 </line>
   <line>   5     6     1 </line>
   <line>   6     7     1 </line>
   <line>   7     8     1 </line>
   <line>   8     5     1 </line>
   <circle>   9     9   10    1 </circle>
   <subdomain>    1    1   10</subdomain>
</Domain>
</OFELI_File>
```

The above text file enables generating a rectangular domain containing a circular hole. Once this file is created and called `test.dom`, we can generate the file `hole.m` by typing

```
g2m  -d  test.dom
```

Note that all parameters of the command `g2m` can be obtained by typing

```
g2m  --help
```

2.8 Using file converters

The package **OFELI** contains two utility programs:

- `cmesh`: to convert various mesh files.
- `cfield`: to convert various output files.

These conversions allow to use commercial (or public) mesh generators and post-processors.

2.8.1 Using `cmesh`

The program `cmesh` converts mesh files to and from the native **OFELI** format. The command line of the program is:

```
cmesh  --from <ofeli|bin|em|amd|bamg|emc2|gambit|gmsh|
               netgen|tetgen|matlab|triangle>
       --to <ofeli|amd|gpl|gmsh|tec|vtk|matlab>
       -i <string> [-o <string>] [-n <int>] [--] [--version] [-h]
```

Where:

```
  --from <ofeli|em|amd|bamg|emc2|gambit|gmsh|netgen|tetgen|matlab|triangle>
         (required)
```

Available input formats:

ofeli	OFELI XML mesh file	*.m
em	EasyMesh file	*.n, *.e and *.s
bamg	BAMG file	*.bamg
gambit	Gambit neutral file	*.neu
gmsh	Gmsh file	*.msh
netgen	Netgen files	*.vol
tetgen	Tetgen files	*.node and *.ele
matlab	Matlab file	*-matlab.m
triangle	Triangle files	*.node and *.ele

`--to <ofeli|gpl|gmsh|tec|vtk|matlab> (required)`

Available output formats:

gambit	Gambit Neutral file	*.neu
gmsh	Gmsh file	*.msh
netgen	Netgen file	*.vol
tetgen	Tetgen files	*.node and *.ele
matlab	Matlab file	*-matlab.m
triangle	Triangle files	*.node and *.ele

`-i <string>, --input <string>`
`(required)` Mesh Input File

`-o <string>, --output <string>`
Mesh Output File

`-n <int>, --nb_dof <int>`
Nb. of degrees of freedom per node

`--version`
Displays version information and exits.

`-h, --help`
Displays usage information and exits.

Here above, the invoked programs and file formats are the following:

- ofeli is the actual **OFELI** XML format defined in **OFELI**.
- EasyMesh is a free program that generates two dimensional, unstructured, Delaunay and constrained Delaunay triangulations in general domains. It can be downloaded from the site http://www-dinma.univ.trieste.it/nirftc/research/easymesh
- Gnuplot is a command-line driven program for producing 2D and 3D plots. It is under the GNU General Public License. Available information can be found in the site http://www.gnuplot.info/
- BAMG is a 2–D mesh generator. It allows to construct adapted meshes from a given metric. It was developed at INRIA, France. Available information can be found in the site http://www-rocq1.inria.fr/gamma/cdrom/www/bamg/eng.htm
- Gambit is a commercial mesh generator. Available information can be found in the site http://fluent.com/software/gambit/
- Gmsh is a free three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. It can be downloaded from the site http://www.geuz.org/gmsh/
- Tecplot is high quality post graphical commercial processing program developed by AMTEC. Available information can be found in the site http://www.tecplot.com

- Tetgen is a free three-dimensional Delaunay tetrahedral mesh generator developed by Hang Si (E-mail: si@wias-berlin.de). Available information can be found in the site http://tetgen.berlios.de/
- Triangle is a powerful two-dimensional Delaunay mesh generator developed by Jonathan Richard Shewchuk (E-mail: jrs@cs.berkeley.edu). Available information can be found in the site http://www.cs.cmu.edu/~quake/triangle.html
- Matlab is a language of scientific computing including visualization. It is developed by MathWorks. Available information can be found in the site http://www.mathworks.com/products/matlab/
- VTK is an open source library of graphical processing. It is developed by Kitware. Graphical postprocessing can be obtained by software Paraview. Available information on Paraview can be downloaded from the site http://www.paraview.org
- The optional argument is an integer that defines the (constant) number of degrees of freedom per node. This information is indeed not always available from mesh generators and is needed in ofeli format. Its default value is 1.

2.8.2 Using cfield

The program cfield converts **OFELI** XML field files to other formats. The command line of the program is:

```
cfield -f<gmsh|gpl|tec|vtk> -m <string> -i <string> [-o <string>] [--]
       [--version] [-h]
```

where

`-f <gmsh|gpl|tec|vtk>, --format <gmsh|gpl|tec|vtk>`
`(required)`

Available output formats:

gmsh	Gmsh Postprocessing File	*.pos
gpl	Gnuplot File	*-gnuplot.dat
tec	Tecplot file	*-tecplot.dat
vtk	vtk file	*.vtk

`-m <string>, --mesh <string>`
(required) Mesh file name

`-i <string>, --input <string>`
(required) Input field file name

`-o <string>, --output <string>`
Output field file name

`--version`
Displays version information and exits.

`-h, --help`
Displays usage information and exits.

3 File Formats

OFELI data files use the XML syntax. They are valid XML documents. Input files can be given separately or gathered in one or more files Note that old data file of **OFELI** can be converted by using one of the utility programs **mdf2xml** or **fdf2xml** contained in the package.

A typical set of header lines of **OFELI** XML files is the following lines:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<OFELI_File>
<info>
   <title></title>
   <date></date>
   <author></author>
</info>
  ...
  ...
</OFELI_File>
```

The tags **title**, **date** and **author** can be filled in order to keep useful information for a user.

After the preamble given by the element **<info>**, the XML file can contain any of the following elements in any order:

Project	To describe project data: parameters, input and output files, ...
	This information enables constructing the class **IPF**
Domain	To describe domain geometry
Mesh	To describe mesh data
Prescription	To describe prescription of boundary conditions, body and boundary forces, ...
Material	To describe material data
Field	To describe input and output field data.

3.1 Element: Project

The element **Project** enables giving various parameters to control program execution as well as various file names. All acquired data are used to construct the class **IPF**. When invoking this element, one must supply the attribute that gives the projects name as follows:

```
<Project name="project_name">
   ...
   ...
</Project>
```

The element **Project** has a large choice of subelements. Each subelement is a parameter that can be retrieved by calling a member function of class **IPF**. These parameters either have a predefined name, *e.g.* **max_time** that clearly chooses the maximal time for computations and whose is retrieved in the class **IPF** by the member function **getMaxTime**, or by a generic parameter for which a user can define a label. For instance, in the line

```
<parameter label="deltat" value="0.1"/>
```

the read parameter is retrieved by the code line

```
dt = ipf.getDouble("deltat");
```

or equivalently

```
ipf.get("deltat",dt);
```

the read parameter is retrieved by the code line

```
double dt = ipf.getDouble("deltat");
```

or equivalently

```
ipf.get("deltat",dt);
```

where **ipf** is an instance of class **IPF**.

The following table describes the list of parameters in the **Prescription** file:

- **verbose**: Level for information output. Typically, the integer number must be between **1** and **10**. Its default value is **1**.
- **output**: Level for solution output. Its default value is **0**.
- **save**: Level for solution saving in file. Its default value is **0**.
- **Prescription**: To describe prescription of boundary conditions, body and boundary forces, ...
- **Material**: To describe material data.
- **Field**: To describe input and output field data.
- **save**: Level for solution saving in file. Its default value is **0**.
- **plot**: An integer that defines a level for solution saving in plot file. Its default value is **0**.
- **bc**: Flag for boundary condition (Dirichlet) handling.
  - **= 1**: Boundary condition is described in a prescription XML file (Default value).
  - **= 2**: Boundary condition vector is in an XML field file.
- **bf**: Flag for body force handling.
  - **= 1**, Body force is described in a prescription XML file (Default value).
  - **= 2**, Body force vector is in a XML field file.
- **sf**: Flag for surface force (Neumann boundary condition) handling.
  - **= 1**, Surface force vector in a prescription XML file,
  - **= 2**, Read surface force vector in a XML field file.
- **max_time**, A real number that defines maximal time for a time dependent calculation. Its default value is **1.0**.
- **time_step**: Time step for a time dependent calculation. Its default value is **0.1**.
- **nb_steps**: Number of time steps for a time dependent calculation. Its default value is **10**.
- **nb_iter**: Maximum number of iterations for an iterative scheme. Its default value is **100**.
- **tolerance**: Tolerance for convergence for an iterative scheme. Its default value is **1.e-6**.
- **integer**: An integer parameter that can be retrieved by the member function **getIntPar(i)**.

- **double**: A double precision parameter that can be retrieved by the member function. **getDoublePar(i)** where **i** is the rank of appearance of this keyword. Up to **10** double precision parameters can be contained in the file. This maximal number is defined by the constant **MAX_NB_PAR** in the **OFELI** constants.
- **complex**: A complex parameter that can be retrieved by the member function. **getComplexPar(i)** where **i** is the rank of appearance of this keyword. Up to **10** complex parameters can be contained in the file. This maximal number is defined by the constant **MAX_NB_PAR** in the **OFELI** constants.
- **mesh_file**: Name of file that contains mesh data.
- **init_file**: Name of file that contains initial data in XML format.
- **restart_file**: Name of file that contains restarting field file in XML format. This file is useful when an iteration process (or time stepping procedure) is used and the programs stops to restart later
- **bc_file**: Name of file that contains (Dirichlet) boundary condition data in XML format.
- **bf_file**: Name of file that contains body force initial data in XML format.
- **sf_file**: Name of file that contains surface force data in XML format.
- **save_file**: Name of file that fields to save in XML format.
- **plot_file**: Name of file that contains fields to plot in XML format.
- **data_file**: Name of file that contains various data in XML format.
- **aux_file**: Name of file that contains any other data in any format. Any occurrence of this keyword will define a new file name that can be retrieved through the member function **getAuxFile(i)** where **i** is the rank of the appearance of this keyword. Up to 10 occurrences can be contained in the file. This maximal number is defined by the constant **MAX_NB_PAR** in the **OFELI** constants.
- **parameter**: As explained in the example above, this subelement must contain an option called label that identifies the parameters and then optionally the option value to specify a value. If this option is not present, a value must be given before closing the subelement

Note that the argument of each subelement can be given either through the attribute **value** or through a value that given between the opening and the closing of the subelement.

Let us give a simple example of XML file using the **Project** element, where we have used both possibilities of defining subelements.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<OFELI_File>
<info>
   <title>Project file</title>
   <date>August 18, 2008</date>
   <author>R. Touzani</author>
</info>
<Project name="beam">
   <mesh_file>beam.m</mesh_file>
   <data_file>beam.pr</data_file>
   <parameter label="d-file">beam.d</parameter>
   <parameter label="density" value="1.2"/>
   <nb_iter>100</nb_iter>
   <tolerance value="1.e-5"></tolerance>
   <verbose>1</verbose>
   <output>1</output>
   <save value="1"/>
```

```
<Project>
</OFELI_File>
```

3.2 Element: Domain

The element **Domain** enables defining a domain geometry. At the current stage of development of **OFELI**, a domain definition is necessary to generate meshes in the 2-D configurations. This element has 2 attributes:

- The attribute **dim** defines the space dimension. Typically, **1**, **2** or **3**. Its default value is **2**.
- The attribute **nb_dof** defines the number of degrees of freedom on any unknown support. For instance, if unknowns are supported by nodes, one can specify that each node supports 2 degrees of freedom for a planar elasticity problem. The default value of this parameter is **1**.

An example of use of this element is:

```
<Domain dim="2">
   <vertex>  0.   0.   1   0.1  </vertex>
   <vertex>  1.   0.   1   0.1  </vertex>
   <vertex>  1.   1.   1   0.1  </vertex>
   <vertex>  0.   1.   1   0.1  </vertex>
   <vertex>  0. 5  0.5   2   0.1  </vertex>
   <vertex>  0. 6  0.5   2   0.1  </vertex>
   <line>   1    2    -2  </line>
   <line>   2    3    -2  </line>
   <line>   3    4    -2  </line>
   <line>   4    1    -2  </line>
   <circle>   6   6   5  1 </circle>
   <subdomain>   1   1  10  </subdomain>
</Domain>
```

Let us describe the subelements of element **Domain**:

– **vertex**: To describe a vertex in the domain. This subelement describes a vertex by the following data:

```
x  y  h  c
```

where **x** and **y** are the vertex coordinates, **h** is the mesh size around the vertex and **c** is the code to assign to the vertex. This code will be transferred to the vertex once a mesh is generated.

– **line**: To describe a straight line that joins 2 vertices. This subelement describes a straight line by the following data:

```
n1  n2  cc
```

where the line goes from vertex **n1** to vertex **n2** and **cc** is the code to assign to the nodes generated. Note that line is actually oriented from **n1** to **n2**.

– **circle**: To describe a circular arc. This subelement describes a circular arc by the following data:

```
n1  n2  n3 cc
```

where the arc goes from vertex **n1** to vertex **n2**. Note that we can have **n1=n2** which in this case generates an entire circle. The center of the circle is located at vertex **n3**. The integer **cc** stands for the code to assign to nodes generated on the line (Dirichlet) if **cc>0** and to sides generated on the line (Neumann) if **cc<0**.

– **contour**: To describe a contour (a closed connection of lines). This subelement describes a contour arc by the following data:

```
l1  l2  ...  ln
```

Here the contour is given by the consecutive lines **l1** to **ln**. These lines must be given in the direct orientation (counter clockwise).

– **hole**: To describe a hole (an internal contour). This subelement describes a hole by the following data:

```
l1  l2  ...  ln
```

Here the hole is a contour given by the consecutive lines **l1** to **ln**. These lines must be given in the clockwise orientation.

– **subdomain**: To describe a subdomain with a specific code. This subelement describes a circular arc by the following data:

```
n  c
```

where **n** is the label of the contour that describes the subdomain and **c** is a code (integer number) to associate to the subdomain.

3.3 Element: Mesh

The element **Mesh** enables providing data that describe a finite element mesh. It has 2 optional attributes:

- The attribute **dim** defines the space dimension. Typically, **1**, **2** or **3**. Its default value is **2**.
- The attribute **nb_dof** defines the number of degrees of freedom on any unknown support. For instance, if unknowns are supported by nodes, one can specify that each node supports 2 degrees of freedom for a planar elasticity problem. The default value of this parameter is **1**.

An example of use of this element is:

```
<Mesh dim="3" nb_dof="2">
  ...
  ...
</Mesh>
```

This element has the following subelements:

– **Nodes**: To describe nodes. This subelement enables defining each node data. Typically, it can be used as follows:

```
<Nodes>
   x1   y1   z1   c1   x2   y2   z2   c2   x3   y3   z3   c3
   x4   y4   z4   c4        ...
   ...
   ...      ...        xn   yn   zn   cn
 </Nodes>
```

More precisely, each node is given by its coordinates. In this example, a 3-D problem requires three coordinates. For a 2-D problem only x and y–coordinates are required. The coordinates are followed by an integer number that describes a code to associate to the node. This code is used to prescribe boundary conditions. It is important to mention that any nonzero code enforces a boundary condition of a given DOF (Degree Of Freedom). By convention, this code is chosen such that it has as many digits as the number of DOF for the node. For instance if the number of DOF of a node is 3, then a code of **231** yields a code **1** for the first DOF, **3** for the second DOF and **2** for the third one.

Another important thing to note is that the nodes are given in a free format one after the other. Moreover, the number of nodes doesn't have to be specified. The parser deduces it from the list size.

– **Elements**: To describe elements. This subelement enables defining the finite elements. It has the following attributes:

  - The attribute **shape** specifies the shape of the finite element. It must take one of the following values: **line**, **triangle** or **tria**, **quadrilateral** or **quad**, **tetrahedron** or **tetra**, and **hexahedron** or **hexa**. The default value is **line** for 1-D, **triangle** for 2-D and **tetrahedron** for 3-D.
  - The attribute **nodes** is the number of element nodes. Its default value is **2** for 1-D, **3** for 2-D, and **4** for 3-D.

  A typical example of subelement **Elements** is the following:

```
<Elements shape="triangle" nodes="3">
   1   2   5   1     2   3   5   1
   3   4   5   1     4   1   5   1
</Elements>
<Elements shape="quadrilateral" nodes="4">
   2   6   7   3   2
</Elements>
```

  Note that the elements are grouped shape by shape.

  More precisely, for each element are given:

  - The list of its nodes. Their number is given by the attribute **nodes** or by its default value.
  - An integer number that stands for its code. This code is helpful to specify the material in which lies the element. It can also be used for any other purpose to select lists of elements.

  Note that the number of elements doesn't have to be specified. The parser deduces it from the list size.

– **Sides**: To describe sides. This subelement enables defining sides (edges in 2-D, faces in 3-D) in a finite element mesh. It has the following attributes:

  - The attribute **shape** specifies the shape of the side. It must take one of the following values: **line**, **triangle** or **tria**, **quadrilateral** or **quad**. The default value is **line** for 2-D and **triangle** for 3-D.
  - The attribute **nodes** is the number of side nodes. Its default value is **2** for 2-D and **3** for 3-D.

  A typical example of subelement **Sides** is the following:

```
<Sides shape="line" nodes="2">
   1   2   1     2   3   1
</Sides>
```

  Note that the sides are grouped shape by shape.

  More precisely, for each side are given:

  - The list of its nodes. Their number is given by the attribute **nodes** or by its default value.
  - An integer number that stands for its code. This code plays the same role as for nodes.

  Note that the number of sides doesn't need be specified. The parser deduces it from the list size.

– **Material**: To describe materials for elements. This subelement **Material** enables attributing a material to each element code. Element codes are given as integers in the **Elements** section. If no material is associated to a code, the library assigns a so-called **Generic** material with default physical properties. This is to be used for testing purposes. For a realistic use of the library, each material is defined through its properties by an XML file. For instance, the material **Iron** is defined in the file **Iron.md**. Depending on the stage of development of the library, number of material files are already present. The element **Material** enables defining a user's material.

A typical example of subelement **Material** is the following:

```
<Material>
   1   Rubber
   5   Copper
</Material>
```

More precisely, each material is given by an integer that is the code and a string that is the material name. Either the material file exists in the given list of **OFELI** materials (here files **Rubber.md** and **Copper.md**), or the user provides in his own directory the required material file.

3.4 Element: Prescription

This element encloses information on conditions to prescribe for the numerical solution by the **OFELI** library. We mean here by prescription, enforcement of boundary conditions (Dirichlet), Boundary forces (Neumann boundary conditions, Body forces (right-hand side of equations, initial condition, ...) To each type of prescription corresponds a subelement. Moreover, prescription of variable (time and/or space dependent) conditions are allowable through algebraic equations.

The element **Prescription** doesn't have any attribute. It has the following subelements:

– **BoundaryCondition**: To prescribe (essential or Dirichlet) boundary conditions. This subelement enables prescribing a Dirichlet boundary condition. A typical example of its use is:

```
<BoundaryCondition code="1" dof="2">x*exp(t)</BoundaryCondition>
```

More precisely, this subelement has the following attributes:

- The attribute **code** specifies the code for which the boundary condition is assigned. For example, if the degrees of freedom are supported by nodes, this code is the one associated to nodes.
- The attribute **dof** specifies the degree of freedom index to which the boundary condition is assigned. If this attribute is not present, the condition is enforced to all dofs'.

– **BodyForce**: To prescribe body forces or sources, ... This subelement enables prescribing the volume right-hand side of the partial differential equation (Domain integral in the variational formulation). Depending on the problem origin, this one can be called *Body Force*, *Load*, *Source*, ...

A typical example of its use is:

```
<BodyForce dof="2">1.0</BodyForce>
```

As it can be remarked, this subelement works like **BoundaryCondition** except the attribute **code** which has no meaning in this context.

– **Source**: Identical to **BodyForce**.

– **BoundaryForce**: To prescribe boundary forces (Neumann boundary conditions), like tractions, fluxes, ... This subelement enables prescribing the surface right-hand side of the partial

differential equation (Boundary integral in the variational formulation or Neumann condition). Depending on the problem origin, this one can be called *Boundary Force*, *Traction*, *Flux*, ...

A typical example of its use is:

```
<BoundaryForce code ="5" dof="2">x-y</BoundaryForce>
```

As it can be remarked, this subelement works like **BoundaryCondition**, The difference being that this condition is generally applied to sides (edges or faces) whereas the Dirichlet boundary condition applies generally to nodes.

- **Traction**: Identical to **BoundaryForce**.
- **Flux**: Identical to **BoundaryForce**.
- **Initial**: To prescribe an initial condition. This subelement enables prescribing an initial condition for a time-dependent problem or an initial solution for an iterative process.

  A typical example of its use is:

  ```
  <Initial dof="1">(1.0+sin(x))*exp(-t)</Initial>
  ```

  As it can be remarked, this subelement works like **BodyForce** for instance.

3.5 Element: Material

Material data are stored in specific XML files. Each file corresponds to a given material. The **OFELI** library contains a collection of material files that will be enriched in the forthcoming releases.

In **OFELI**, the material named **Mat** is described in the XML file: **Mat.md** Let us give as example the material file for the material **Copper**. Here is the listing of the file **Copper.md** the

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<OFELI_File>
<info>
   <title>Material data for Copper</title>
   <date></date>
   <author></author>
</info>
<Material name="Copper">
   <Density>1.</Density>
   <SpecificHeat>8920.</SpecificHeat>
   <ThermalConductivity>401.</ThermalConductivity>
   <ElectricConductivity>5.9302e07</ElectricConductivity>
   <ElectricResistivity>1.6863e-8</ElectricResistivity>
   <MagneticPermeability>12.566371e-7</MagneticPermeability>
   <PoissonRatio>0.34</PoissonRatio>
   <YoungModulus>15.e10</YoungModulus>
</Material>
</OFELI_File>
```

The structure of this file doesn't need any additional explanation. We shall however give hereafter the list of properties that can be stored in the XML file:

- ◇ **Density**: Density of material (Heat and Mass Transfer).
- ◇ **SpecificHeat**: Specific Heat (Heat Transfer).
- ◇ **ThermalConductivity**: Thermal Conductivity (Heat Transfer).
- ◇ **MeltingTemperature**: Melting Temperature (Heat Transfer).

◇ **EvaporationTemperature**: Evaporation Temperature (Heat Transfer).

◇ **ThermalExpansion**: Thermal Expansion (Heat and Mass Transfer).

◇ **LatentHeatMelting**: Latent Heat for Melting (Heat Transfer).

◇ **LatentHeatEvaporation**: Latent Heat for Evaporation (Heat Transfer).

◇ **DielectricConstant**: Dielectric Constant (Electromagnetism).

◇ **ElectricConductivity**: Electric Conductivity (Electromagnetism).

◇ **ElectricResistivity**: Electric Resistivity: Inverse of Conductivity.

◇ **MagneticPermeability**: Magnetic Permeability (Electromagnetism).

◇ **Viscosity**: Kinematic Viscosity (Fuid Dynamics).

◇ **YoungModulus**: Young Modulus (Solid Mechanics).

◇ **PoissonRatio**: Poisson Ratio (Solid Mechanics).

3.6 Element: `Field`

The element **Field** is useful to store vectors, such as input vectors, results. We have grouped all these vectors under the term **Field**. The XML field file that contains these vectors can be transformed via conversion programs to various file formats for well known free and commercial graphical postprocessors.

Fields can be divided into 3 types depending on the degree of freedom support: Fields can be given by nodes, elements or sides. In addition, in view of handling time-dependent problems, the XML file can contain as many vectors as necessary, each one corresponding to a given time step.

A typical XML file containing fields looks like this

```
<OFELI_File>
  ...
<Field name="Temperature" type="Node" nb_dof="1">
   <Step time="0.1">
     ... ...
     ... ...
   </Step>
   <Step time="0.2">
     ... ...
     ... ...
   </Step>
</Field>
<Field name="Displacement" type="Element" nb_dof="2">
   <Step time="0.1">
      <constant dof="1">1.0</constant>
      <expression dof="2">x*exp(t)</expression>
   </Step>
</Field>
</OFELI_File>
```

More precisely, the element **Field** has the attributes:

- The attribute **name** specifies the name to give to the field. This attribute is optional.
- The attribute **type** specifies the type of the field. It must take one of the values: **Node**, **Element** or **Side**. The default value is **Node**.
- The attribute **nb_dof** gives the number of degrees of freedom for one support, *e.g.* if the type is **Node**, there are **nb_dof** values per node. The default value of this attribute is **1**.

– The element **Field** has only one subelement: **Step**.

– The subelement **Step** gives the vector entries for the specified value of the attribute **time**. It owns two subelements:

- The element **constant** enables assigning a constant value to all vector components for one given dof or all dofs. It has the attribute **dof** that can specify the dof to be assigned. By default, all dofs are assigned this constant value.
- The element **expression** enables assigning an algebraic expression that may involve the coordinates **x**, **y**, **z** and the time **t**, to all vector components for one given dof or all dofs. It has the attribute **dof** that can specify the dof to be assigned. By default, all dofs are assigned this expression.

3.7 Element: Function

The element **Function** defines a tabulated function of one, two or three variables. In order to minimize computational cost, each variable is defined by a uniform partitioning given by its minimal value, its maximal values and the number of grid points. This element has as unique attribute the name of the function.

A typical usage of this element is:

```
<Function name="Density">
       <Variable label="x" nb_pts="5" min="0" max="1"/>
       <Variable label="y" nb_pts="4" min="10" max="12"/>
       <Data>
          1.0   2.0   5.0   7.0
          2.0   3.0   5.0   8.0
          7.0   2.0   5.0   9.0
          0.0   2.0   8.0  10.0
         11.0  20.0  25.0  30.0
       </Data>
    </Function>
```

Let us describe the subelements of element **Function**:

◇ **Variable**: To describe a variable

◇ **Data**: To give list of function values?

The subelement Variable describes a variable. Its attributes are:

- The attribute label gives a name to the variable. This name has no particular usage. Only the order of the variables is important for a function evaluation.
- The attribute **nb_pts** gives the number of grid points for this variable, *i.e.* This is the number of grid intervals plus one.
- The attribute **min** gives the minimal value of the variable.
- The attribute **max** gives the maximal value of the variable.
- The subelement **Data** gives the function values ordered as follows (This is an example of a function of 2 variables):

```
val(1,1)   val(1,2)   ...  val(1,n2)
val(2,1)   val(2,2)   ...  val(2,n2)
...
val(n1,1)  val(n1,2)  ...  val(n1,n2)
```

where **n1** and **n2** are the number of points for the first and second variable respectively.

4 Debugging

The **OFELI** library is equipped with some simple tools to help debugging and tracking errors. For a large size finite element code using **OFELI** this may not be sufficient to detect all troubles but may help find some ones. The simplest method to track errors is to use compiler directives to check some inconsistencies.

4.1 Debugging directives

Two macros are defined to check array bounds :

- Activating the macro `_OFELI_DEBUG` enables outputting a message each time a class constructor or destructor is called. The file name and the line number in the class implementation file are also given.
- Activating the macro `_BOUNDS_DEBUG` enables checking the bounds of each vector each time these ones are invoked.