Google Summer of Code 2013

GNU-Octave

Fem-fenics

Genaral Purpose Finite Element Library for GNU-Octave

Marco Vassallo

Version 0.0

December 20, 2013

Contents

Chapter 1

Introduction

Fem-fenics is an open source package (pkg) for the resolution of partial differential equations with Octave. The project has been developed during the Google Summer of Code 2013 with the help and the sustain of the GNU-Octave community under the supervision of prof. De Falco.

The report is structured as follows:

- in chapter 2 we provide a simple reference guide for beginners
- in chapter 3 is presented a detailed explanation of the relevant parts of the program. In this way, the interested reader can see what there is "behind" and expecially anyone interested in it can learn quickly how it is possible to extend the code and contribute to the project.
- in chapter 4 more examples are provided. For a lot of them, we present the octave script alongside the code for Fenics (in C++ and/or Python) in order to provide the user with a quick reference guide.

If you think that going inside the report could be boring, it is available a wiki at

`http://wiki.octave.org/Fem-fenics`

while if you want to see how the project has grown during the time you can give a look at

`http://gedeone-gsoc.blogspot.com/`

Finally, the API is available as Appendix but also at the following address

`http://octave.sourceforge.net/fem-fenics/overview.html`

Chapter 2

Introduction to Fem-fenics

2.1 Installation

Fem-fenics is an external package for Octave, which means that it can be installed only once that Octave has been successfully installed on the PC. Furthermore, as Fem-fenics is based on Fenics, it is also needed a running version of the latter. They can be easily installed following the guidelines provided on the official Octave [1] and Fenics [2] websites. Once that Octave and Fenics are correctly installed, to install Fem-fenics open Octave (which now is provided with a new amazing GUI) and type

```
>> pkg install fem-fenics -forge
```

That's all! For any problem during the installation don't hesitate to contact us. To be sure that everything is working fine, load the fem-fenics pkg and run one of the examples provided within the package:

```
>> pkg load fem-fenics
>> femfenics_examples()
```

For a description of the examples, look at chapter 4.

NOTE For completing the installation process successfully, the form compiler FFC and the header file dolfin.h should also be available on the machine. They are managed automatically by Fenics if it is installed as a binary package or with Dorsal. If it has been done manually, please be sure that they are available before starting the installation of Fem-fenics.

2.2 General layout and first example

A generic problem has to be solved in two steps:

1. a **.ufl file** where the abstract problem is described: this file has to be written in Unified Form Language (UFL), which is a domain specific language for defining discrete variational forms and functionals in a notation close to pen-and-paper formulation. UFL is easy to learn, and the User manual provides explanations and examples [3].

2. a script file **.m** where the abstract problem is imported and a specific problem is implemented and solved: this is the script file where the fem-fenics functions described in the following chapters are used.

We provide immediately a simple example in order to familiarize the user with the code.

The Poisson equation In this example, we show how it is possible to solve the Poisson equation with mixed Boundary Conditions. If we indicate with Ω the domain and with $\Gamma = \Gamma_N \cup \Gamma_D$ the boundaries, the problem can be expressed as

$$\begin{aligned} \Delta u &= f \qquad \text{on } \Omega \\ u &= 0 \qquad \text{on } \Gamma_D \\ \nabla u \cdot n &= g \qquad \text{on } \Gamma_N \end{aligned}$$

where f, g are data which represent the source and the flux of the scalar variable u. A possible variational formulation of the problem is:
find $u \in H^1_{0,\Gamma_D}$:

$$\begin{aligned} a(u,v) &= L(v) \qquad \forall v \in H^1_{0,\Gamma_D} \\ a(u,v) &= \int_\Omega \nabla u \cdot \nabla v \\ L(v) &= \int_\Omega fv + \int_{\Gamma_N} gv \end{aligned}$$

The abstract problem can thus be written in the `Poisson.ufl` file immediately. The only thing that has to be specified at this stage is the space of Finite Elements used for the discretization of H^1_{0,Γ_D}. In this example, we choose the space of continuous lagrangian polynomial of degree one

```
FiniteElement("Lagrange", triangle, 1)
```

but many more possibilities are available.

```
element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)

f = Coefficient(element)
g = Coefficient(element)

a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds
```

It is always a good idea to check if the ufl code is correctly written before importing it into Octave. Typing

```
>> ffc -l dolfin Poisson.ufl
```

in the shell shouldn't produce any error.

We can now implement and solve a specific instance of the Poisson problem with Octave. The parameters are set as follow

- $\Omega = [0,1] \times [0,1]$
- $\Gamma_D = (0,y) \cup (1,y) \subset \partial\Omega$
- $\Gamma_N = (x,0) \cup (x,1) \subset \partial\Omega$
- $f = 10 \exp \frac{(x-0.5)^2 + (y-0.5)^2}{0.02}$
- $g = \sin(5x)$

As a first thing we need to load into Octave the pkgs previously installed

```
pkg load fem-fenics msh
```

The ufl file can thus be imported inside Octave. For every specific element defined inside the ufl file there is a specific function which stores it for later use

- `ufl_import_FunctionSpace ('Poisson')` is a function which looks for the finite element space defined inside the file called Poisson.ufl; if everything is ok, it generates a function which we will use later
- `ufl_import_BilinearForm ('Poisson')` is a function which looks for the rhs of the equation, i.e. for the bilinear form defined inside Poisson.ufl
- `ufl_import_LinearForm ('Poisson')` is a function which looks for the linear form.

In some cases one could be interested in using these functions separately but if, as in our example, all the three elements are defined in the same ufl file (and only in this case), the `import_ufl_Problem ('Poisson')` can be used, which generates at once all the three functions described above

```
ufl_import_Problem ('Poisson');
```

To set the concrete elements which define the problem, the first things to do is to create a mesh. It can be managed easily using the msh pkg. For a structured squared mesh

```
x = y = linspace (0, 1, 33);
msho = msh2m_structured_mesh (x, y, 1, 1:4);
```

Once that the mesh is available, we can thus initialize the Fem-fenics mesh using the function `Mesh ()`:

```
mesh = Mesh (msho);
```

To initialize the functional space, we have to specify as argument only the fem-fenics mesh, because the finite element type and the polynomial degree have already been specified in the ufl file:

```
V = FunctionSpace('Poisson', mesh);
```

Essential BC can now be applied using `DirichletBC ()`; this function receives as argument the functional space, a function handle which specifies the value to set, and the label of the sides where the BC applies. In this case, homogenous boundary conditions hold on the left and right side of the square

```
bc = DirichletBC(V, @(x, y) 0.0, [2; 4]);
```

The last thing to do before solving the problem, is to set the coefficients specified in the ufl file. To set them, the function `Expression ()` can be used passing as argument a string which specifies the name of the coefficient (it is important that they are called in the same way as in the ufl file: the source term 'f' and the normal flux 'g'), and a function handle with the value prescribed:

```
ff = Expression ('f',
     @(x,y) 10*exp(-((x - 0.5)^2 + (y - 0.5)^2) / 0.02));
gg = Expression ('g', @(x,y) sin (5.0 * x));
```

Another possibility for dealing with the coefficients defined in the ufl file would be to use the function `Constant ()` or `Function ()`. The coefficients can thus be used together with the FunctionSpace to set the Bilinear and the Linear form

```
a = BilinearForm ('Poisson', V, V);
L = LinearForm ('Poisson', V, ff, gg);
```

The discretized representation of our operator is obtained using the functions `assemble ()` or `assemble_system ()`, which also allow to specify the BC(s) to apply

```
[A, b] = assemble_system (a, L, bc);
```

Here A is a sparse matrix and b is a column vector. All the functionalities available within Octave can now be exploited to solve the linear system. The easisest possibility is the backslash command:

```
u = A \ b;
```

Once that the solution has been obtained, the `u` vector is converted into a Fem-fenics function and plotted `plot ()` or saved `save ()` in the vtu format

```
u = Function ('u', V, sol);
save (u, 'poisson')
plot (u);
```

The complete code for the Poisson problem is reported below, while in figure 2.1 is presented the output.

```
#load the pkg and import the ufl problem
pkg load fem-fenics msh
import_ufl_Problem ('Poisson')

# Create the mesh and define function space
x = y = linspace (0, 1, 33);
mesh = Mesh(msh2m_structured_mesh (x, y, 1, 1:4));
```

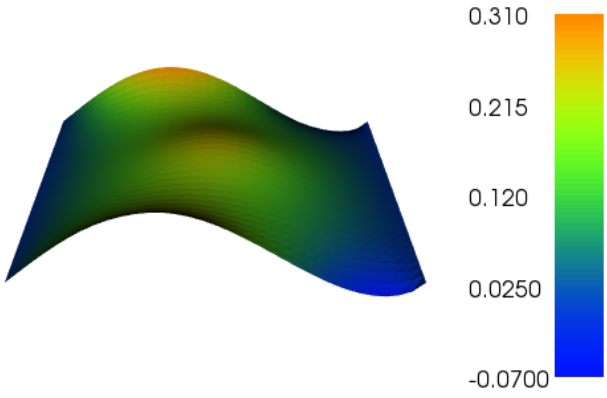

Figure 2.1: The result for the Poisson equation

```
V = FunctionSpace('Poisson', mesh);

# Define boundary condition and source term
bc = DirichletBC(V, @(x, y) 0.0, [2;4]);
ff = Expression ('f', @(x,y) 10*exp(-((x - 0.5)^2 + (y - 0.5)^2) /
    0.02));
gg = Expression ('g', @(x,y) sin (5.0 * x));

#Create the Bilinear and the Linear form
a = BilinearForm ('Poisson', V, V);
L = LinearForm ('Poisson', V, ff, gg);

#Extract the matrix and compute the solution
[A, b] = assemble_system (a, L, bc);
sol = A \ b;
u = Function ('u', V, sol);

# Save solution in VTK format and plot it
save (u, 'poisson')
plot (u);
```

Chapter 3

Implementation

Fem-fenics aims to fill a gap in Octave: even if there are packages for the creation of mesh [4], for the postprocessing of data [5] and for the resolution of some specific pde [6] [7], no general purpose finite element library is available.

The goal of the project is thus to provide a package which can be used to solve user defined problems and which is able to exploit the functionality provided with Octave.

Instead of writing a library starting from scratch, an interface to one of the finite element library which are already available has been created. Among the many libraries taken into account, the one which was best suited for our purposes seemed to be the FEniCS project. It "is a collection of free, open source, software components with the common goal to enable automated solution of pde." In particular, Dolfin is the C++/Python interface of FEniCS, providing a consistent Problem Solving Environment for ODE and PDE. The idea has been to create wrappers in Octave for C++ Dolfin, in a similar way to what it has been done for Python. This is a very natural choice, because Octave is mainly written in script language and in C++. It is in fact possible to implement an Octave interpreter function in C++ through the native oct-file interface or, conversely, to use Octave's Matrix/Array Classes in a C++ application [8].

The works can be summarized as follows (fig. 3.1):

the elements already available in Octave for the resolution of PDE (Mesh and Linear Algebra) have been exploited, and wrappers to the other FEniCS functions added. To allow exchanges between these programs, the necessary functions for converting an Octave mesh/matrix into a FEniCS one and viceversa have been written.

Two main ideas have guided us throughout the realization of the pkg:

- keep the syntax as close as possible to the original one in Fenics (Python)
- make the interface as simple as possible.

3.1 General layout of a class

Seven new classes are implemented for dealing with FEniCS objects and for using them inside Octave:

- **boundarycondition** stores and builds a dolfin::DirichletBC

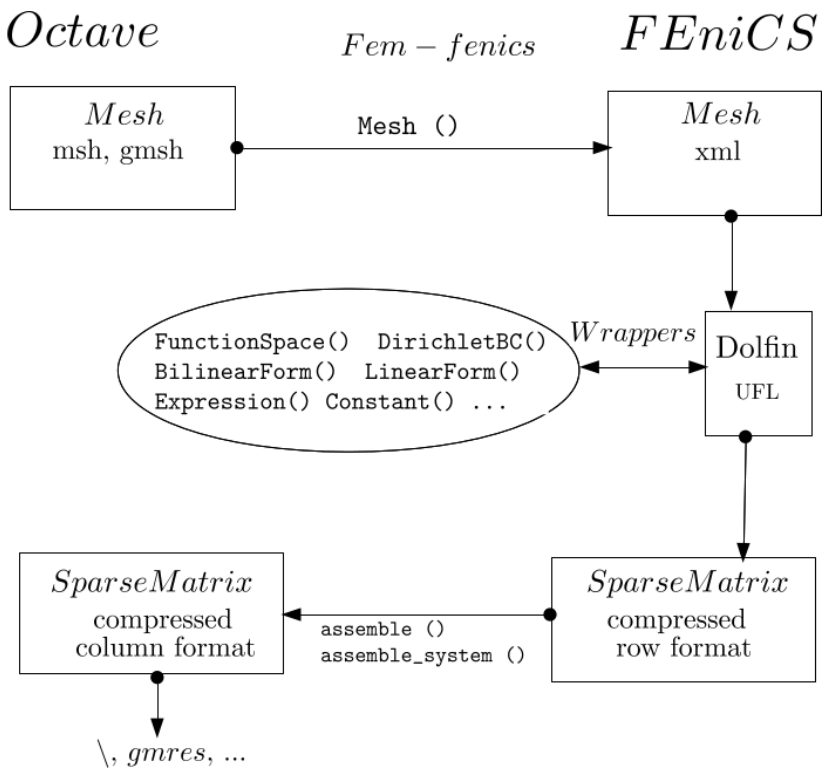

Figure 3.1: General layout of the package

- **coefficient** stores an expression object which is used for the evaluation of user defined values
- **expression** is needed for internal use only as explained below
- **form** stores a general dolfin::Form and can be used both for a dolfn::BilinearForm and for a dolfin::LinearForm
- **function** for the dolfin::Function objects
- **functionspace** stores the user defined FunctionSpace
- **mesh** converts a PDE-tool like mesh structure in a dolfin::Mesh

The classes are written with the "usual" C++ style, but they need to be derived publicly from octave_base_value and to be added to the Octave interpreter [8]. When a type is used for the first time during a session, it is also temporarily registered in the interpreter after all the other basic types (int, double, ...).

The general layout of a class can thus be kept simple and with the main purpose of storing the associated FEniCS objects, which is done throughout boost::shared_ptr< > to the corresponding FEniCS type. All the classes also implement at least two constructors: a default one which is necessary to register a type in the Octave interpreter, and a constructor which takes as argument the corresponding dolfin type.

As an example, the form class implementation follows, while classes which differ from the general layout are presented below in more details.

```
#ifndef _FORM_OCTAVE_
#define _FORM_OCTAVE_

#include <memory>
#include <vector>
#include <dolfin.h>
#include <octave/oct.h>

class form : public octave_base_value
{

 public:

  form () : octave_base_value () {}

  form (const dolfin::Form _frm)
    : octave_base_value (), frm (new dolfin::Form (_frm)) {}

  form (boost::shared_ptr <const dolfin::Form> _frm)
    : octave_base_value (), frm (_frm) {}

  void
  print (std::ostream& os, bool pr_as_read_syntax = false) const
    {
       os << "Form " << ": is a form of rank " << frm->rank ()
       << " with " << frm->num_coefficients ()
```

```
27        << " coefficients" << std::endl;
28      }
29
30    ~form(void) {}
31
32    bool is_defined (void) const { return true; }
33
34    const dolfin::Form & get_form (void) const { return (*frm); }
35
36    const boost::shared_ptr <const dolfin::Form> &
37    get_pform (void) const { return frm; }
38
39   private:
40
41    boost::shared_ptr <const dolfin::Form> frm;
42
43    DECLARE_OCTAVE_ALLOCATOR;
44    DECLARE_OV_TYPEID_FUNCTIONS_AND_DATA;
45
46  };
47
48  static bool form_type_loaded = false;
49
50  DEFINE_OCTAVE_ALLOCATOR (form);
51  DEFINE_OV_TYPEID_FUNCTIONS_AND_DATA (form, "form", "form");
52  #endif
```

3.1.1 Shared pointer

In all the classes presented above, the private members are stored using a boost::shared_ptr< > to the corresponding FEniCS type. This is done because we have to refer in several places to resources which are built dynamically and we want that they are destroyed only when the last reference is destroyed [9]. For example, if we have two different functional spaces in the same problem, like with Navier-Stokes for the velocity and the pressure, the mesh is shared between them and no one has its own copy. Furthermore, they are widely supported inside DOLFIN, and it can thus be avoided to have a copy of the same object for FEniCS and another one for DOLFIN: there is just one copy which is shared between DOLFIN and FEniCS.

3.1.2 The mesh class

In addition to usual methods, the mesh class implemens functionalities which allow to deal with meshes as they are currently available with the msh pkg, i.e. in the (p, e, t) format, and in Fenics, i.e. in the xml Dolfin format. It is therefore necessary to have two different constructors

```
mesh (Array<double>& p, Array<octave_idx_type>& e,
      Array<octave_idx_type>& t);

mesh (std::string _filename)
   : octave_base_value (), pmsh (new dolfin::Mesh(_filename)) {}
```

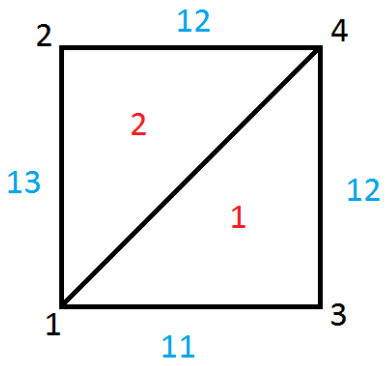

Figure 3.2: The (very) simple mesh for our example

where the first one accepts as input a mesh in (p, e, t) format and converts it into a xml one, while the latter loads the mesh stored in the _filename.xml file.

The constructors are used within the Mesh () function, which therefore accepts as argument either a mesh generated within the msh pkg or a string with the name of the file where the dolfin mesh is stored.

Furthermore, if a mesh is stored in another different format, the program dolfin-convert can try to convert it to the dolfin xml format. For example, for a mesh generated with Metis:

```
Shell:
  >> dolfin-convert msh.gra msh.xml
```

and then inside the Octave script:

```
mesh = Mesh ('msh.xml');
```

Before exploring the code in more details, the main differences between the two storing formats are presented using the very simple, but rather instructive, example of a unit square mesh with just two elements, fig. 3.2.

pet A mesh is represented using the three matrices p, e, t, and, using msh, we can easily obtain the mesh for our example typing

```
mesh = msh2m_structured_mesh ([0 1], [0 1], 1, [11 12 12 13])
```

The matrix p stores information about the coordinates of the vertices

```
>> mesh.p

    0   0   1   1     x-coordinates
    0   1   0   1     y-coordinates
```

Thus the vertex in the n^{th} column is labelled as the vertex number n, and so on.

The matrix t stores information about the connectivity

```
>> mesh.t

    1   1     number of the first vertex of the element
    3   4     number of the second vertex of the element
    4   2     number of the third vertex of the element
    0   0
```

The first element is thus the one obtained connecting vertices 1-3-4 and so on.

The matrix e stores information related to every side edge, like the number of the vertices of the boundary elements, and the number of the geometrical border containing the edge, which is a convenient way to deal with boundary conditions in a problem.

```
>> mesh.e

   1    3    2    1     first vertex of the side edge
   3    4    4    2     second vertex of the side edge
   0    0    0    0
   0    0    0    0
  11   12   12   13     label of the geometrical border containing the edge
   0    0    0    0
   1    1    1    1
```

The side edge between vertex 1-3 is labelled 11, between 3-4 is 12...

dolfin xml A mesh is an object of the dolfin::Mesh class which stores information only about the coordinates of the vertices (like p) and the information about the connectivity (like t). A mesh can thus be manipulated using the functions and the methods of the class, which are presented below. Instead, the information about boundaries is not directly stored in the mesh. The mesh used in the example is stored as

```
<?xml version="1.0"?>
<dolfin xmlns:dolfin="http://fenicsproject.org">
  <mesh celltype="triangle" dim="2">
    <vertices size="4">
      <vertex index="0" x="0.000e+00" y="0.000e+00" />
      <vertex index="1" x="0.000e+00" y="1.000e+00" />
      <vertex index="2" x="1.000e+00" y="0.000e+00" />
      <vertex index="3" x="1.000e+00" y="1.000e+00" />
    </vertices>
    <cells size="2">
      <triangle index="0" v0="0" v1="2" v2="3" />
      <triangle index="1" v0="0" v1="1" v2="3" />
    </cells>
  </mesh>
</dolfin>
```

Conversion between the formats The first necessary step in our way to a package which links Octave and FEniCS is to convert a mesh from the (p, e, t)

format into the dolfin xml one. Furthermore, as dolfin provides methods and functions which allow to manipulate a mesh and which don't have a conterpart in the msh pkg, we have also created wrappers for them (specifically for mesh::refine).

As it has been shown above, the main difference between (p, e, t) and DOLFIN *xml* is the way in which the boundaries are distinguished. The former stores all the information in the e matrix, while the latter uses the functions and the methods of the dolfin::mesh class to set/get information about a mesh. The most useful classes available in dolfin are recalled

- **MeshIterator** To know whether an edge belongs or not to the boundary, we can iterate over all the edges of our mesh using the classes provided by DOLFIN:

```
for (dolfin::FacetIterator f (mesh); ! f.end (); ++f)
  {
    if ((*f).exterior () == true)
      {
        //do something with the boundary cells
      }
  }
```

- **MeshFunction** To store data related to a mesh, dolfin provides the template class MeshFunctions. "A MeshFunction is a function that can be evaluated at a set of mesh entities. A MeshFunction is discrete and is only defined at the set of mesh entities of a fixed topological dimension. A MeshFunction may for example be used to store a global numbering scheme for the entities of a (parallel) mesh, marking sub domains or boolean markers for mesh refinement." [10] For example, in the function `mshm_refine` of the msh package, the list of cells to be refined is stored as a MeshFunction, which for every cell says whether or not it has to be refined:

```
dolfin::CellFunction<bool> cell_markers (mesh);
cell_markers.set_all (false);

for (octave_idx_type i = 0;
     i < cells_to_refine.length (); ++i)
  cell_markers.set_value (cells_to_refine (i) , true);
```

- **MeshValueCollection** "It differs from the MeshFunction class in two ways. First, data do not need to be associated with all entities (only a subset). Second, data are associated with entities through the corresponding cell index and local entity number (relative to the cell), not by global entity index, which means that data may be stored robustly to file."[11] It is thus obvious that it is better to use the MeshValueCollection whenever saving or writing a mesh.

The container classes presented above can be used by their own, but to set/get data from a mesh it is better to use the methods provided by the classes:

- **MeshDomains** "The class MeshDomains stores the division of a Mesh into subdomains. For each topological dimension 0 <= d <= D, where

D is the topological dimension of the Mesh, a set of integer markers are stored for a subset of the entities of dimension d, indicating for each entity in the subset the number of the subdomain. It should be noted that the subset does not need to contain all entities of any given dimension; entities not contained in the subset are "unmarked"." [12]

- **MeshData** "The class MeshData is a container for auxiliary mesh data, represented either as MeshFunction over topological mesh entities, arrays or maps. Each dataset is identified by a unique user-specified string." [13]

Geometry from (p, e, t) to dolfin xml Converting the vertices and cells from (p, e, t) in the xml format can be done using the dolfin editor, while caution has to be taken for storing information associated with boundaries and subdomains, as presented in the next paragraph.

```
dolfin::MeshEditor editor;
boost::shared_ptr<dolfin::Mesh> msh (new dolfin::Mesh ());
editor.open (*msh, D, D);
editor.init_vertices (p.cols ());
editor.init_cells (t.cols ());

if (D == 2)
  {
    for (uint i = 0; i < p.cols (); ++i)
      editor.add_vertex (i,
                         p.xelem (0, i),
                         p.xelem (1, i));

    for (uint i = 0; i < t.cols (); ++i)
      editor.add_cell (i,
                       t.xelem (0, i) - 1,
                       t.xelem (1, i) - 1,
                       t.xelem (2, i) - 1);
  }

if (D == 3)
  {
    ...
  }

editor.close ();
```

Subdomain markers: from (p, e, t) to dolfin xml There are no fundamental differences between the 2D and 3D case, and they are thus treated together referring to the general dimension D. The subdomain information is contained in the t matrix, and it is temporarily copied to a MeshValueCollection. For every column of the t matrix, i.e. for every element of the mesh, we have to look for the corresponding element in the DOLFIN mesh. We use the class MeshIterator for moving around on the DOLFIN mesh:

```
dolfin::MeshValueCollection<uint> my_cell_marker (D);
```

```
for (uint i = 0; i < num_cells; ++i)
  dolfin::Vertex v (mesh, t(0, i));
    for (dolfin::CellIterator f (v); ! f.end (); ++f)
      {
        if ((*f) == all_vertices_in_the_ith_column)
          {
            my_cell_marker.set_value
              ((*f).index (), t(last_row, i), mesh);
            break;
          }
       }
```

The `all_vertices_in_the_ith_column` is just like a pseudo code: we have to be sure that the Cell pointed by f is the one corresponding to the i^{th} column of the matrix, checking the vertices one-by-one:

in $2D$ the cell is a triangle, and we thus have to check 3 vertices. As we don't know the order in which vertices are visited, we have to check all the $3! = 6$ different combinations:

```
        ...
        if ((*f).entities(0)[0] == t(0, i)
            && (*f).entities(0)[1] == t(1, i)
            && (*f).entities(0)[2] == t(2, i)
            || ... check the other 5 possibilities... )
       ....
```

where the `entities(std::size_t dim)` method returns an array with the indexes of the elements of dimension dim. Thus we use $dim = 0$ as we are looking for vertices.

In the $3D$ case, our cell is a tetrahedron, and we have to check all the $4! = 24$ possibilities, each of which is composed by 4 assertions; in total we have almost one hundred conditions!

Now that the information is stored in our function, it can be associated to the mesh

```
   *(mesh.domains ().markers (D)) = my_marked_cell;
```

Subdomain markers: from dolfin xml to (p, e, t) In the DOLFIN .xml file, the information is stored like:

```
 ...
<mesh_value_collection name="m" type="uint" dim="2" size="2">
    <value cell_index="0" local_entity="0" value="1"/>
    <value cell_index="1" local_entity="0" value="2"/>
 ...
```

When the file is read using DOLFIN, the information is automatically associated with the mesh as a MeshValueCollection named `cell_domains`, which can be accessed to extract the information using the MeshDomains class. Obviously we have to be sure that the information is available within the file that we are reading, and that it is related to Cell, i.e. to elements of dimension D, before it is associated to the last row of the t matrix:

```
dolfin::MeshFunction<uint> my_cell_marker;
if (! mesh.domains ().is_empty ())
  if (mesh.domains ().num_marked (D) != 0)
    my_cell_marker = *(mesh.domains ().cell_domains ());

  for (j = 0; j < t.cols (); ++j)
    t(D + 1, j) = my_cell_marker[j];
```

Boundary Markers For boundary markers, things work in a similar way, as long as we remember that we are working with objects of dimension D - 1. In this case, the main difference is in the .xml file: it is no longer enough to say to what cell element the label is referred to, but we have to specify to which $D-1$ entity (a side or a face) the label is referred. For example:

```
....
mesh_value_collection name="m" type="uint" dim="1" size="4">
     <value cell_index="0" local_entity="0" value="12"/>
     <value cell_index="0" local_entity="2" value="11"/>
     <value cell_index="1" local_entity="0" value="12"/>
     <value cell_index="1" local_entity="2" value="13"/>
...
```

The cell number "0" is a triangle, and to the `local_entity` number "0", i.e. to the side number "0", is associated the label "12", while to the side number "2" is associated the label "11". To the side number "1", there are no labels associated. The number of the `local_entity` refers to the enumeration of the reference element. In any case, it is DOLFIN which takes care of the conversion of indeces from this format to the usual one, and we can thus use methods and functions as explained for the subdomain markers.

Mesh refine Now that it is possible to convert meshes between Octave and DOLFIN, the functions available in the dolfin::mesh class can be used to improve the functionality of the msh package. For the moment, it has been added the possibility of refining a mesh, either uniformly or specifying the list of the vertices we want to be refined. The function is now part of the msh pkg[4], and a more detailed desciption has been provided previously [14].

3.1.3 The functionspace class

A dolfin::FunctionSpace is defined by specifying a mesh and the type of the finite element which we want to use. The mesh is handled as presented above, while the FE are specified inside the .ufl file. Possible choices are [15]:

Finite Element Space	Symbol
Argyris	ARG *
Arnold–Winther	AW *
Brezzi–Douglas–Marini	BDM
Crouzeix–Raviart	CR
Discontinuous Lagrange	DG
Hermite	HER*
Lagrange	CG
Mardal–Tai–Winther	MTW *
Morley	MOR*
Nédélec 1st kind H (curl)	N1curl
Nédélec 2nd kind H (curl)	N2curl
Raviart–Thomas	RT

where the Finite Elements denoted with * are not yet fully supported inside FEniCS.

3.2 General layout of a function

There are three general kinds of functions in the code: functions which create an abstract problem (wrappers to UFL), functions which create the specific instance of a problem (wrapper to FEniCS) and functions which discretize the problem and generate the matrices.

3.2.1 Wrappers to UFL

As stated in section 2.2, a problem is divided in two files:a *.ufl* file where the abstract problem is described in Unified Form Language (UFL), and a script file *.m* where a specific problem is implemented and solved. We suppose that they are called Poisson.ufl and Poisson.m . In order to use the information stored in the UFL file, i.e. the bilinear and the linear form, they have to be imported inside Octave. When the UFL file is compiled using the ffc compiler, a header file *Poisson.h* is generated. In this header file, it is defined the Poisson class, which derives from dolfin::Form, and the constructor for the bilinear and linear form are set. This file is thus available only at compilation time, but it has to be included somehow in the wrapper function for the Bilinear and the Linear form. An easy solution would have been to write a set of pre established problems where the user could only change the values of the coefficient for a specific problem; but, as we want to let the user free to write his own variational problem, a different approach has been adopted. The ufl file is compiled at run time and generates a header file. Then, a Poisson.cc file is written from a template which takes as input the name of the header file and is compiled including the Poisson.h file; now the corresponding octave functions for the specific problem is available and will be used from BilinearForm, LinearForm, FunctionSpace, ... As an example it is presented the import_ufl_BilinearForm function.

```
1 function import_ufl_BilinearForm (var_prob)
2
3 ...
```

```

  %the function which writes the var-prob.cc file
  generate_rhs (var_prob);

  %the function which writes the makefile
  generate_makefile (var_prob, private);

  % the makefile is executed in a terminal:
  % 1) generate the header file from ufl
  %  ffc -l dolfin var_prob.ufl
  % 2) compile the var_prob.cc
  %  mkoctfile var_prob.cc -I.
  system (sprintf ("make -f Makefile_%s rhs", var_prob));

  ...

endfunction
```

```
function output = generate_rhs (ufl_name)

  STRING ="
  #include "@@UFL_NAME@@.h"

  ...

  DEFUN_DLD (@@UFL_NAME@@_BilinearForm, args, , ""A =
      fem_rhs_@@UFL_NAME@@ (FUNCTIONAL SPACE, COEFF)"")
  {
    ...

    const functionspace & fspo1
      = static_cast<const functionspace&> (args(0).get_rep ());
    const functionspace & fspo2
      = static_cast<const functionspace&> (args(1).get_rep ());

    const dolfin::FunctionSpace & U = fspo1.get_fsp ();
    const dolfin::FunctionSpace & V = fspo2.get_fsp ();
    @@UFL_NAME@@::BilinearForm a (U, V);

    ...


  }";

  STRING = strrep (STRING, "@@UFL_NAME@@", ufl_name);

  fid = fopen (sprintf ("%s_BilinearForm.cc", ufl_name), 'w');
  fputs (fid, STRING);
  output = fclose (fid);

endfunction
```

3.2.2 Wrappers to DOLFIN

The general layout of a function is very simple and it is composed of 4 steps which we describe using an example:

```
DEFUN_DLD (fem_fs, args, , "initialize a fs from a mesh")
{
         // 1 read data
         const mesh & msho = static_cast<const mesh&> (args(0).get_rep
             ());
         // 2 convert the data from octave to dolfin
         const dolfin::Mesh & mshd = msho.get_msh ();
         // 3 build the new object using dolfin
         boost::shared_ptr <const dolfin::FunctionSpace> g (new
             Laplace::FunctionSpace (mshd));
         // 4 convert the new object from dolfin to Octave and return it
         octave_value retval = new functionspace(g);
          return retval;
}
```

All the functions presented above follow this general structure, and thus here we present in detail only functions which present some differences.

Sparse Matrices

Polymorphism

DirichletBC and Coefficient

These two functions take as input a function handle which cannot be directly evaluated by a dolfin function to set, respectively, the value on the boundary or the value of the coefficient. It has thus been derived from dolfin::Expression a class "expression" which has as private member an octave function handle and which overloads the function eval(). In this way, an object of the class expression can be initialized throughout a function handle and can be used inside dolfin because "it is" a dolfin::Expression

```
class expression : public dolfin::Expression
{
  ...

  void
  eval (dolfin::Array<double>& values,
        const dolfin::Array<double>& x) const
    {
      octave_value_list b;
      b.resize (x.size ());
      for (std::size_t i = 0; i < x.size (); ++i)
        b(i) = x[i];
      octave_value_list tmp = feval (f->function_value (), b);
      Array<double> res = tmp(0).array_value ();

      for (std::size_t i = 0; i < values.size (); ++i)
        values[i] = res(i);
    }
```

```
19
20  private:
21   octave_fcn_handle * f;
22 };
```

DirichletBC The BC are imposed directly to the mesh setting to zero all the off diagonal elements in the corresponding line. This means that we could loose the symmetry of the matrix, if any. To avoid this problem, instead of the method `apply()` it is possible to use the function `assemble_system()` , which preserves the symmetry of the system but which needs to build together the lhs and the rhs.

Coefficient The coefficient of the variational problem can be specified using either a Coefficient or a Function. They are different objects which behave in different ways: a Coefficient, as exlained above, overloads the `eval()` method of the dolfin::Expression class and it is evaluated at run time using the octave function `feval()`. A Function instead doesn't need to be evaluated because it is assembled copying element-by-element the values contained in the input vector.

3.2.3 Wrapper to FEniCS

3.2.4 Code on the fly

Chapter 4

More Advanced Examples

In this chapter more examples are provided. At the beginning of each section, the problem is briefly presented and then the Octave script for the resolution of the problem using Fem-fenics is presented alongside the code written in C++ and/or the Python. For each problem, we refer the reader to the complete desciption on the FEniCS website.

4.1 Mixed Formulation for the Poisson Equation

In this example the Poisson equation is solved with a "mixed approach": it is usedthe stable FE space obtained using Brezzi-Douglas-Marini polynomial of order 1 and Dicontinuos element of order 0.

$$\begin{aligned} -\text{div}\ (\sigma(x,y))) &= f(x,y) && \text{in } \Omega \\ \sigma(x,y) &= \nabla u(x,y) && \text{in } \Omega \\ u(x,y) &= 0 && \text{on } \Gamma_D \\ (\sigma(x,y)) \cdot \mathbf{n} &= \sin(5x) && \text{on } \Gamma_N \end{aligned}$$

A complete description of the problem is avilable on the Fenics website [16].

Listing 4.1: Fem-fenics

```
pkg load fem-fenics msh
import_ufl_Problem ('MixedPoisson')

# Create mesh
x = y = linspace (0, 1, 33);
mesh = Mesh(msh2m_structured_mesh (x, y, 1,
    1:4));

# File MixedPoisson.ufl
#  BDM = FiniteElement("BDM", triangle, 1)
#  DG  = FiniteElement("DG", triangle, 0)
#  W = BDM * DG
```

Listing 4.2: Python

```
from dolfin import *


# Create mesh
mesh = UnitSquareMesh(32, 32)


# Define function spaces and mixed (product)
    space
BDM = FunctionSpace(mesh, "BDM", 1)
DG = FunctionSpace(mesh, "DG", 0)
W = BDM * DG
```

```
V = FunctionSpace('MixedPoisson', mesh);

# Define trial and test function
# File MixedPoisson.ufl
#  (sigma, u) = TrialFunctions(W)
#  (tau, v)  = TestFunctions(W)
#  CG = FiniteElement("CG", triangle, 1)
#  f = Coefficient(CG)
f = Expression ('f',
     @(x,y) 10*exp(-((x - 0.5)^2 + (y -
         0.5)^2) / 0.02));

# Define variational form
# File MixedPoisson.ufl
#  a = (dot(sigma, tau) + div(tau)*u +
    div(sigma)*v)*dx
#  L = - f*v*dx
a = BilinearForm ('MixedPoisson', V, V);
L = LinearForm ('MixedPoisson', V, f);









# Define essential boundary

bc1 = DirichletBC (SubSpace (V, 1), @(x,y) [0;
    -sin(5.0*x)], 1);
bc2 = DirichletBC (SubSpace (V, 1), @(x,y) [0;
    sin(5.0*x)], 3);

# Compute solution
[A, b] = assemble_system (a, L, bc1, bc2);
sol = A \ b;
func = Function ('func', V, sol);

sigma = Function ('sigma', func, 1);
u = Function ('u', func, 2);
```

```
# Define trial and test functions
(sigma, u) = TrialFunctions(W)
(tau, v) = TestFunctions(W)


f = Expression
    ("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] -
        0.5, 2)) / 0.02)")

# Define variational form

a = (dot(sigma, tau) + div(tau)*u +
    div(sigma)*v)*dx
L = - f*v*dx



# Define function G such that G \cdot n = g
class BoundarySource(Expression):
    def __init__(self, mesh):
        self.mesh = mesh
    def eval_cell(self, values, x, ufc_cell):
        cell = Cell(self.mesh, ufc_cell.index)
        n = cell.normal(ufc_cell.local_facet)
        g = sin(5*x[0])
        values[0] = g*n[0]
        values[1] = g*n[1]
    def value_shape(self):
        return (2,)

G = BoundarySource(mesh)

# Define essential boundary
def boundary(x):
    return x[1] < DOLFIN_EPS or x[1] > 1.0 -
        DOLFIN_EPS

bc = DirichletBC(W.sub(0), G, boundary)

# Compute solution
w = Function(W)
solve(a == L, w, bc)


(sigma, u) = w.split()
```

```
# Plot solution
plot (sigma);
plot (u);
```

```
# Plot sigma and u
plot(sigma)
plot(u)
interactive()
```

```
#
```

```
# Copyright 2011, The FEniCS Project
```

4.2 Incompressible Navier-Stokes equation

In this example the incompressible Navier-Stokes equation

$$\frac{\partial u}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} - \nu \Delta \mathbf{u} + \nabla p = f \qquad \text{in } \Omega$$
$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega$$

are solved using the Chorin-Temam algorithm. The L-shaped domain Ω can be obtained using the msh pkg.

```
name = [tmpnam ".geo"];
fid = fopen (name, "w");
fputs (fid,"Point (1) = {0, 0, 0, 0.1};\n");
fputs (fid,"Point (2) = {1, 0, 0, 0.1};\n");
fputs (fid,"Point (3) = {1, 0.5, 0, 0.1};\n");
fputs (fid,"Point (4) = {0.5, 0.5, 0, 0.1};\n");
fputs (fid,"Point (5) = {0.5, 1, 0, 0.1};\n");
fputs (fid,"Point (6) = {0, 1, 0,0.1};\n");

fputs (fid,"Line (1) = {5, 6};\n");
fputs (fid,"Line (2) = {2, 3};\n");

fputs (fid,"Line(3) = {6,1,2};\n");
fputs (fid,"Line(4) = {5,4,3};\n");
fputs (fid,"Line Loop(7) = {3,2,-4,1};\n");
fputs (fid,"Plane Surface(8) = {7};\n");
fclose (fid);
msho = msh2m_gmsh (canonicalize_file_name (name)(1:end-4),...
                   "scale", 1,"clscale", .2);
unlink (canonicalize_file_name (name));
```

The flow is driven by an oscillating pressure $p_{in}(t) = \sin 3t$ at the inflow while the pressure is kept constant $p_{out} = 0$ at the outflow. A complete description of the problem is avilable on the Fenics website [17].

Listing 4.3: Fem-fenics

```
pkg load fem-fenics msh
import_ufl_Problem ("TentativeVelocity");
import_ufl_Problem ("VelocityUpdate");
```

Listing 4.4: Python

```
from dolfin import *
```

```
import_ufl_Problem ("PressureUpdate");

# We can either load the mesh from the file as
    in Dolfin but
# we can also use the msh pkg to generate the
    L-shape domain
# as showed above

mesh = Mesh ('lshape.xml');

# Define function spaces (P2-P1). From ufl file
#  V = VectorElement("CG", triangle, 2)
#  Q = FiniteElement("CG", triangle, 1)
V = FunctionSpace ('VelocityUpdate', mesh);
Q = FunctionSpace ('PressureUpdate', mesh);

# Define trial and test functions. From ufl
    file
#  u = TrialFunction(V)
#  p = TrialFunction(Q)
#  v = TestFunction(V)
#  q = TestFunction(Q)

# Set parameter values. From ufl file
#  nu = 0.01
dt = 0.01;
T = 3.;




# Define boundary conditions
noslip = DirichletBC (V, @(x,y) [0; 0], [3,
    4]);









outflow = DirichletBC (Q, @(x,y) 0, 2);




# Create functions
u0 = Expression ('u0', @(x,y) [0; 0]);
```

```
# Load mesh from file


mesh = Mesh("lshape.xml")

# Define function spaces (P2-P1)


V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)

# Define trial and test functions

u = TrialFunction(V)
p = TrialFunction(Q)
v = TestFunction(V)
q = TestFunction(Q)

# Set parameter values
dt = 0.01
T = 3
nu = 0.01

# Define time-dependent pressure BC
p_in = Expression("sin(3.0*t)", t=0.0)

# Define boundary conditions
noslip = DirichletBC(V, (0, 0),
          "on_boundary && \
          (x[0] < DOLFIN_EPS | x[1] <
              DOLFIN_EPS | \
          (x[0] > 0.5 - DOLFIN_EPS && x[1] >
              0.5 - DOLFIN_EPS))")
inflow = DirichletBC(Q, p_in, "x[1] > 1.0 -
    DOLFIN_EPS")
outflow = DirichletBC(Q, 0, "x[0] > 1.0 -
    DOLFIN_EPS")
bcu = [noslip]
bcp = [inflow, outflow]

# Create functions
u0 = Function(V)
u1 = Function(V)
p1 = Function(Q)
```

```
# Define coefficients
k = Constant ('k', dt);
f = Constant ('f', [0; 0]);

# Tentative velocity step. From ufl file
#  eq = (1/k)*inner(u - u0, v)*dx +
    inner(grad(u0)*u0, v)*dx \
#       + nu*inner(grad(u), grad(v))*dx -
    inner(f, v)*dx
a1 = BilinearForm ('TentativeVelocity', V, V,
    k);

# Pressure update. From ufl file
#  a = inner(grad(p), grad(q))*dx
#  L = -(1/k)*div(u1)*q*dx
a2 = BilinearForm ('PressureUpdate', Q, Q);

# Velocity update
#  a = inner(u, v)*dx
#  L = inner(u1, v)*dx - k*inner(grad(p1),
    v)*dx
a3 = BilinearForm ('VelocityUpdate', V, V);

# Assemble matrices
A1 = assemble (a1, noslip);

A3 = assemble (a3, noslip);









# Time-stepping
t = dt; i = 0;
while t < T

  # Update pressure boundary condition
  inflow = DirichletBC (Q, @(x,y) sin(3.0*t),
      1);

  # Compute tentative velocity step
  "Computing tentative velocity"
  L1 = LinearForm ('TentativeVelocity', V, k,
      u0, f);
  b1 = assemble (L1, noslip);
```

```
# Define coefficients
k = Constant(dt)
f = Constant((0, 0))

# Tentative velocity step
F1 = (1/k)*inner(u - u0, v)*dx +
    inner(grad(u0)*u0, v)*dx \
    + nu*inner(grad(u), grad(v))*dx -
        inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Pressure update
a2 = inner(grad(p), grad(q))*dx
L2 = -(1/k)*div(u1)*q*dx


# Velocity update
a3 = inner(u, v)*dx
L3 = inner(u1, v)*dx - k*inner(grad(p1), v)*dx



# Assemble matrices
A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)

# Use amg preconditioner if available
prec = "amg" if
    has_krylov_solver_preconditioner("amg")
            else "default"

# Create files for storing solution
ufile = File("results/velocity.pvd")
pfile = File("results/pressure.pvd")

# Time-stepping
t = dt
while t < T + DOLFIN_EPS:

    # Update pressure boundary condition
    p_in.t = t


    # Compute tentative velocity step
    begin("Computing tentative velocity")
    b1 = assemble(L1)
    [bc.apply(A1, b1) for bc in bcu]
    solve(A1, u1.vector(), b1, "gmres",
```

```
  utmp = A1 \ b1;
  u1 = Function ('u1', V, utmp);

  # Pressure correction
  "Computing pressure correction"
  L2 = LinearForm ('PressureUpdate', Q, u1, k);
  [A2, b2] = assemble_system (a2, L2, inflow,
      outflow);
  ptmp = A2 \ b2;
  p1 = Function ('p1', Q, ptmp);

  # Velocity correction
  "Computing velocity correction"
  L3 = LinearForm ('VelocityUpdate', V, k, u1,
      p1);
  b3 = assemble (L3, noslip);
  ut = A3 \ b3;
  u1 = Function ('u0', V, ut);

  # Plot solution
  plot (p1);
  plot (u1);

  # Save to file
  save (p1, sprintf ("p_%3.3d", ++i));
  save (u1, sprintf ("u_%3.3d", i));

  # Move to next time step
  u0 = u1;
  t += dt

end


#
```

```
        "default")
    end()

    # Pressure correction
    begin("Computing pressure correction")
    b2 = assemble(L2)
    [bc.apply(A2, b2) for bc in bcp]
    solve(A2, p1.vector(), b2, "gmres", prec)
    end()


    # Velocity correction
    begin("Computing velocity correction")
    b3 = assemble(L3)
    [bc.apply(A3, b3) for bc in bcu]
    solve(A3, u1.vector(), b3, "gmres",
        "default")
    end()

    # Plot solution
    plot(p1, title="Pressure", rescale=True)
    plot(u1, title="Velocity", rescale=True)

    # Save to file
    ufile << u1
    pfile << p1

    # Move to next time step
    u0.assign(u1)
    t += dt
    print "t =", t

# Hold plot
interactive()

# Copyright 2011, The FEniCS Project
```

4.3 HyperElasticity

This time we compare the code with the c++ version of DOLFIN. The problem for an elastic material can be expressed as a minimization problem

$$\min_{u \in V} \Pi$$

$$\Pi = \int_\Omega \psi(u)\,\mathrm{d}x - \int_\Omega B \cdot u\,\mathrm{d}x - \int_{\partial\Omega} T \cdot u\,\mathrm{d}s$$

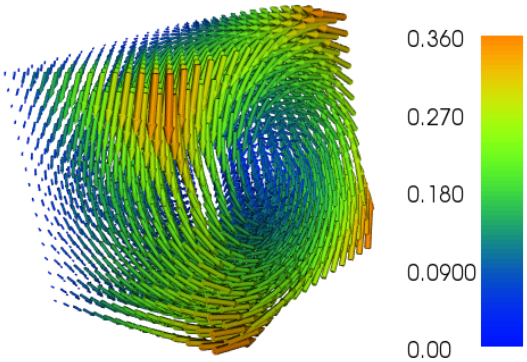

Figure 4.1: Solution of the HyperElasticity problem

where Π is the total potential energy, ψ is the elastic stored energy, B is a body force and T is a traction force.

A complete description of the problem is avilable on the Fenics website [18]. The final solution will look like in figure 4.1.

Listing 4.5: UFL code

```
# Function spaces
element = VectorElement("Lagrange", tetrahedron, 1)

# Trial and test functions
du = TrialFunction(element)  # Incremental displacement
v  = TestFunction(element)   # Test function

# Functions
u = Coefficient(element)     # Displacement from previous iteration
B = Coefficient(element)     # Body force per unit volume
T = Coefficient(element)     # Traction force on the boundary

# Kinematics
I = Identity(element.cell().d) # Identity tensor
F = I + grad(u)              # Deformation gradient
C = F.T*F                    # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic = tr(C)
J  = det(F)

# Elasticity parameters
mu    = Constant(tetrahedron)
lmbda = Constant(tetrahedron)

# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2
```

```
# Total potential energy
Pi = psi*dx - inner(B, u)*dx - inner(T, u)*ds

# First variation of Pi (directional derivative about u in the direction
    of v)
F = derivative(Pi, u, v)

# Compute Jacobian of F
J = derivative(F, u, du)

# Copyright 2011, The FEniCS Project
```

Listing 4.6: Fem-fenics

```
pkg load fem-fenics msh
problem = 'HyperElasticity';
import_ufl_Problem (problem);
```

Listing 4.7: C++

```
#include <dolfin.h>
#include "HyperElasticity.h"

using namespace dolfin;

// Sub domain for clamp at left end
class Left : public SubDomain
{
  bool inside(const Array<double>& x, bool
      on_boundary) const
  {
    return (std::abs(x[0]) < DOLFIN_EPS) &&
        on_boundary;
  }
};

// Sub domain for rotation at right end
class Right : public SubDomain
{
  bool inside(const Array<double>& x, bool
      on_boundary) const
  {
    return (std::abs(x[0] - 1.0) < DOLFIN_EPS)
        && on_boundary;
  }
};

// Dirichlet boundary condition for clamp at
    left end
class Clamp : public Expression
{
public:
```

```
Rotation = @(x,y,z) ...
 [0; ...
 0.5*(0.5 + (y - 0.5)*cos(pi/3) -
     (z-0.5)*sin(pi/3) - y);...
 0.5*(0.5 + (y - 0.5)*sin(pi/3) +
     (z-0.5)*cos(pi/3) - z)];
```

```
# Create mesh and define function space
x = y = z = linspace (0, 1, 17);
```

```
  Clamp() : Expression(3) {}

  void eval(Array<double>& values, const
      Array<double>& x) const
  {
    values[0] = 0.0;
    values[1] = 0.0;
    values[2] = 0.0;
  }

};

// Dirichlet boundary condition for rotation
    at right end
class Rotation : public Expression
{
public:

  Rotation() : Expression(3) {}

  void eval(Array<double>& values, const
      Array<double>& x) const
  {
    const double scale = 0.5;

    // Center of rotation
    const double y0 = 0.5;
    const double z0 = 0.5;

    // Large angle of rotation (60 degrees)
    double theta = 1.04719755;

    // New coordinates
    double y = y0 + (x[1]-y0)*cos(theta) -
        (x[2]-z0)*sin(theta);
    double z = z0 + (x[1]-y0)*sin(theta) +
        (x[2]-z0)*cos(theta);

    // Rotate at right end
    values[0] = 0.0;
    values[1] = scale*(y - x[1]);
    values[2] = scale*(z - x[2]);
  }

};

int main()
{
  // Create mesh and define function space
  UnitCubeMesh mesh (16, 16, 16);
```

```
mshd = Mesh (msh3m_structured_mesh (x, y, z,
    1, 1:6));
V  = FunctionSpace (problem, mshd);




# Create Dirichlet boundary conditions
bcl = DirichletBC (V, @(x,y,z) [0; 0; 0], 1);
bcr = DirichletBC (V, Rotation, 2);
bcs = {bcl, bcr};


# Define source and boundary traction functions

B = Constant ('B', [0.0; -0.5; 0.0]);
T = Constant ('T', [0.1; 0.0; 0.0]);




# Set material parameters
E = 10.0;
nu = 0.3;
mu = Constant ('mu', E./(2*(1+nu)));
lmbda = Constant ('lmbda',
    E*nu./((1+nu)*(1-2*nu)));
u = Expression ('u', @(x,y,z) [0; 0; 0]);

# Create (linear) form defining (nonlinear)
    variational problem
L = ResidualForm (problem, V, mu, lmbda, B, T,
    u);




# Solve nonlinear variational problem F(u; v)
    = 0
u0 = assemble (L, bcs{:});
# Create function for the resolution of the NL
    problem
function [y, jac] = f (problem, xx, V, bc1,
    bc2, B, T, mu, lmbda)
```

```
HyperElasticity::FunctionSpace V(mesh);

// Define Dirichlet boundaries
Left left;
Right right;

// Define Dirichlet boundary functions
Clamp c;
Rotation r;

// Create Dirichlet boundary conditions
DirichletBC bcl(V, c, left);
DirichletBC bcr(V, r, right);
std::vector<const BoundaryCondition*> bcs;
bcs.push_back(&bcl); bcs.push_back(&bcr);

// Define source and boundary traction
    functions
Constant B(0.0, -0.5, 0.0);
Constant T(0.1, 0.0, 0.0);

// Define solution function
Function u(V);

// Set material parameters
const double E = 10.0;
const double nu = 0.3;
Constant mu(E/(2*(1 + nu)));
Constant lambda(E*nu/((1 + nu)*(1 - 2*nu)));



// Create (linear) form defining (nonlinear)
    variational problem
HyperElasticity::ResidualForm F(V);
F.mu = mu; F.lmbda = lambda; F.B = B; F.T =
    T; F.u = u;

// Create jacobian dF = F' (for use in
    nonlinear solver).
HyperElasticity::JacobianForm J(V, V);
J.mu = mu; J.lmbda = lambda; J.u = u;

// Solve nonlinear variational problem F(u;
    v) = 0
solve(F == 0, u, bcs, J);
```

```
  u = Function ('u', V, xx);
  a = JacobianForm (problem, V, mu, lmbda, u);
  L = ResidualForm (problem, V, mu, lmbda, B,
      T, u);
  if (nargout == 1)
    [y, xx] = assemble (L, xx, bc1, bc2);
  elseif (nargout == 2)
    [jac, y, xx] = assemble_system (a, L, xx,
        bc1, bc2);
  endif
endfunction

fs = @(xx) f (problem, xx, V, bcl, bcr, B, T,
    mu, lmbda);
[x, fval, info] = fsolve (fs, u0, optimset
    ("jacobian", "on"));
func = Function ('u', V, x);

# Save solution in VTK format
save (func, 'displacement');


# Plot solution
plot (func);


#
```

```
  // Save solution in VTK format
  File file("displacement.pvd");
  file << u;

  // Plot solution
  plot(u);
  interactive();

  return 0;
}
# Copyright 2011, The FEniCS Project
```

4.4 Fictitious Domain

A penalization method to take into account obstacles in incompressible viscous flows

Appendix A

API reference

A.1 Import problem defined with ufl

import_ufl_BilinearForm

Function File: = *import_ufl_BilinearForm* (*myproblem*)

Import a BilinearForm from a ufl file.

myproblem is the name of the ufl file where the BilinearForm is defined.

This function creates in the pwd a file called *myproblem_BilinearForm.oct*.

See also: import_ufl_Problem, FunctionSpace, BilinearForm, LinearForm, Functional.

import_ufl_LinearForm

Function File: = *import_ufl_LinearForm* (*myproblem*)

Import a LinearForm from a ufl file.

myproblem is the name of the ufl file where the LinearForm is defined. This function creates in the pwd a file called *myproblem_LinearForm.oct*.

See also: import_ufl_Problem, FunctionSpace, BilinearForm, LinearForm, Functional.

import_ufl_Functional

Function File: = *import_ufl_Functional* (*myproblem*)

Import a Functional from a ufl file.

myproblem is the name of the ufl file where the Functional is defined. This function creates in the pwd a file called *myproblem_Functional.oct*.

See also: import_ufl_Problem, FunctionSpace, BilinearForm, LinearForm, Functional.

import_ufl_FunctionSpace

Function File: = *import_ufl_FunctionSpace* (*myproblem*)

Import a FunctionSpace from a ufl file.

myproblem is the name of the ufl file where the FunctionSpace is defined. This function creates in the pwd a file called *myproblem_FunctionSpace.oct*.

See also: import_ufl_Problem, FunctionSpace, BilinearForm, LinearForm, Functional.

import_ufl_Problem

Function File: = *import_ufl_Problem* (*myproblem*)

Import a Variational Problem from a ufl file.

myproblem is the name of the ufl file where the BilinearForm, the LinearForm and the FunctionSpace are defined.

See also: import_ufl_BilinearForm, FunctionSpace, BilinearForm, LinearForm, Functional.

A.2 Problem geometry and FE space

Mesh

Function File: [*mesh_out*] = *Mesh* (*mesh_in*)

Construct a mesh from file or from (p, e, t) format. The *mesh_in* should be either

- a string containing the name of the file where the mesh is stored in .xml file If the file is not a .xml file you can try to use the command dolfin-convert directly from the terminal.
- a PDE-tool like structure with matrix fields (p,e,t)

The output *mesh_out* is a representation of the *mesh_in* which is compatible with fem-fenics. The easiest way for dealing with meshes is using the msh pkg.

See also: FunctionSpace.

FunctionSpace

Function File: *V* = *FunctionSpace* (*myproblem, mesh*)

Generate a FunctionSpace on a specific mesh.

This function takes as input the name *myproblem* of the ufl file where the FunctionSpace is defined and the *mesh* where it has to be created.

See also: FunctionSpace, SubSpace, import_ufl_FunctionSpace.

SubSpace

Function File: [*V1*] = *SubSpace* (*V, index*)

Extract a SubSpace from an object of type FunctionSpace. The input arguments are

- *V* which is a FunctionalSpace
- *index* is a positive integer number which represents the SubSpace which has to be extracted.

The output *V1* is the SubSpace needed.

See also: FunctionSpace.

A.3 Problem variables

Constant

Function File: [*c*] = *Constant* (*name, value*)

Create a constatnt object over all the mesh elements with the value specified.

This function takes as input the *name* of the Constant that has to be created and its *value*, which can be either a scalar or a vector.

See also: Expression, Function.

Expression

Function File: [*f*] = *Expression* (*name, Function_handle*)

Create an object with the value specified as a function handle. The input parameters are

- *name* is the name of the coefficient as it is declared in the ufl file
- *Function_handle* is a function handle which specify the expression to apply for our coefficient

The output *f* is an object which contains a representation of the function

See also: Constant, Function.

Function

Function File: [*func*] = *Function* (*name, FunctionSpace* (*or Function*),*Vector* (*or index*))

Initialize an object with the values specified in a vector or extracting a component from a vectorial field. This function can be used in two different ways

- To create a function from a vector. In this case, the arguments are:
  - *name* is a string representing the name of the function
  - *FunctionSpace* is the fem-fenics function space where the vector is defined
  - *Vector* specifies the values of the coefficients for each basic function of the *FunctioSpace*
- To extract a scalar field from a vectorial one
  - *name* is a string representing the name of the function
  - *Function* is the vector valued Function
  - *Index* contains the index of the scalar field to extract.Index starts from 1.

The output *func* is an object which contains a representation of the function *Vector* which can be plotted or saved or passed as argument for a variational problem.

See also: Constant, Expression, plot, save.

DirichletBC

Function File: [*bc*] = *DirichletBC* (*FunctionSpace, Boundary_Label, Function_handle*)

Specify essential boundary condition on a specific side. The input parameters are

- *FunctionSpace* is the fem-fenics space where we want to apply the BC
- *Function_handle* is a function handle which contains the expression that we want to apply as a BC. If we have a Vector field, we can just use a vector of function handles: *Function handle* = [@(x, y) f1, @(x, y) f2, ...]
- *Boundary_Label* is an Array which contains the label(s) of the side(s) where the BC has to be applied.

The output *bc* is an object which contains the boundary conditions

See also: Mesh, FunctionSpace.

A.4 Definition of the abstract Variational problem

BilinearForm

Function File: [*a*] = *BilinearForm* (*my_problem, U, V, coefficient_1, coefficient_2,...*)

Construct a BilinearForm previously imported from ufl.
The compulsory arguments are:

- *my_problem* the name of the problem to solve.
- the FunctionSpace U and V where the problem is defined.

The optional arguments are the *coefficient_1*, *coefficient_2* which specify the parameters for the BilinearForm as stated in the ufl file. They can be either a Constant, a Function or an Expression.

See also: import_ufl_BilinearForm, import_ufl_Problem, FunctionSpace, LinearForm, ResidualForm.

LinearForm

Function File: $[L]$ = *LinearForm* (*my_problem, U, coefficient_1, coefficient_2,...*)

Construct a Functional previously imported from a ufl file.

The compulsory arguments are:

- *my_problem* the name of the problem to solve.
- the FunctionSpace U where the problem is defined.

The optional arguments are the *coefficient_1*, *coefficient_2* which specify the parameters for the LinearForm with the same name which was used in the ufl file. They can be either a Constant, a Function or an Expression.

See also: import_ufl_LinearForm, import_ufl_Problem, BilinearForm, ResidualForm, BilinearForm.

ResidualForm

Function File: $[L]$ = *LinearForm* (*my_problem, U, coefficient_1, coefficient_2,...*)

Construct a ResidualForm previously imported from a ufl file with the function import_ufl_LinearForm.

The compulsory arguments are:

- *my_problem* the name of the problem to solve.
- the FunctionSpace U where the problem is defined.

The optional arguments are the *coefficient_1*, *coefficient_2* which specify the parameters for the ResidualForm with the same name which was used in the ufl file. They can be either a Constant, a Function or an Expression.

See also: import_ufl_LinearForm, import_ufl_Problem, BilinearForm, ResidualForm, BilinearForm.

JacobianForm

Function File: $[J]$ = *Functional* (*my_problem, U, V, coefficient_1, coefficient_2,...*)

Construct a JacobianForm previously imported from a ufl file with the function import_ufl_BilinearForm.

The compulsory arguments are:

- *my_problem* the name of the problem to solve.
- the FunctionSpace U and V where the problem is defined.

The optional arguments are the *coefficient_1*, *coefficient_2* which specify the parameters for the JacobianForm with the same name which was used in the ufl file. They can be either a Constant, a Function or an Expression.

See also: import_ufl_BilinearForm, LinearForm, ResidualForm, BilinearForm.

Functional

Function File: $[L]$ = *Functional* (*my_problem, U, coefficient_1, coefficient_2,...*)

Construct a Functional previously imported from a ufl file.

The compulsory arguments are:

- *my_problem* the name of the problem to solve.
- the FunctionSpace U where the problem is defined.

The optional arguments are the *coefficient_1*, *coefficient_2* which specify the parameters for the Functional with the same name which was used in the ufl file. They can be either a Constant, a Function or an Expression.

See also: import_ufl_Functional, LinearForm, ResidualForm, BilinearForm.

A.5 Creation of the discretized problem

assemble

Function File: $[A]$, $[x\,(\text{Optional})]$ = *assemble* (*form_a, DirichletBC*)

Construct the discretization of a Form and apply essential BC. The input arguments are

- *form_a* which is the form to assemble. It can be a form of rank 2 (BilinearForm or JacobianForm), a form of rank 1 (LinearForm or ResidualForm) or a form of rank 0 (Functional).
- *DirichletBC* represents the optional BC applied to the system.

The output A is a discretized representation of the *form_a*:

- A is a sparse Matrix if *form_a* is a bilinear form
- A is a Vector if *form_a* is a linear form
- A is a Double if *form_a* is a functional

If a boundary condition has to be applied to a vector for a nonlinear problem then it should be provided as 2nd argument and it will be given back as the second output argument. For an example of this situation, please refer to the HyperElasticity example.

See also: BilinearForm, LinearForm, ResidualForm, JacobianForm, Functional.

assemble_system

Function File: [A], [b], [x(Optional)] = *assemble_system* (*form_a, form_L, DirichletBC*)

Construct the discretization of a system and apply essential BC. The input arguments are

- *form_a* which is the BilinearForm to assemble.
- *form_L* which is the LinearForm to assemble.
- *DirichletBC* represents the optional BC applied to the system.

The output A is a matrix representing the *form_a* while b represents *form_L*. If boundary conditions have to be applied to a vector for a nonlinear problem then it should be provide as 3rd argument and it will be given back as the 3rd output argument. For an example of this situation, please refer to the HyperElasticity example.

See also: BilinearForm, LinearForm, ResidualForm, JacobianForm, Functional.

A.6 Post processing

@function/save

Function File: fem_save (*Function, Name*)

Save a function in vtu format. The input parameters are

- *Function* is the function that you want to save
- *Name* is a string for the output name

The output is a file in format .vtu

See also: plot, Function.

@function/plot

Function File: plot (*Function*)

Plot a Function.

See also: Function, Save.

@mesh/plot

Function File: plot (*Mesh, Nodal_ Values*(*OPTIONAL*))

Plot a Mesh. The input parameter is the Mesh and optionally also a vector representing the values of a function at each node.

See also: Mesh, save.

@function/feval

Function File: [*value*] = *feval* (*function_ name, Coordinate*)

Evaluate a function at a specific point of the domain and return the value. The input parameters are the function and the point where it has to be evaluated.

See also: Function.

Appendix B

Autoconf and Automake

In this section we want to discuss how we can write a config.ac and a Makefile.in files which:

- check if a program is available and stop if it is not
- check if a header file is available and issue a warning if not, but go ahead with the compilation

To reach this goal, we need two components:

configure.ac Is a file which checks whether the program/header is available or not and sets consequently the values of some variables.

```
# Checks if the program mkoctfile is available and sets the variable HAVE_MKOCTFILE consequently
AC_CHECK_PROG([HAVE_MKOCTFILE], [mkoctfile], [yes], [no])
# if mkoctfile is not available, it issues an error and stops the compilation
if [test $HAVE_MKOCTFILE = "no"]; then
  AC_MSG_ERROR([mkoctfile required to install $PACKAGE_NAME])
fi

#Checks if the header dolfin.h is available; if it is available, the value of the ac_dolfin_cpp_flags is substituted with -DHAVE_DOLFIN_H, otherwise it is left empty and a warning message is printed
AC_CHECK_HEADER([dolfin.h],
  [AC_SUBST(ac_dolfin_cpp_flags,-DHAVE_DOLFIN_H)
      AC_SUBST(ac_dolfin_ld_flags,-ldolfin)],
  [AC_MSG_WARN([dolfin headers could not be found, some functionalities will be disabled, don't worry your package will still be working, though.])] ).

# It generates the Makefile, using the template described below
AC_CONFIG_FILES([Makefile])
```

Makefile.ac This file is a template for the Makefile, which will be automatically generated when the configure.ac file is executed. The values of the variable `ac_dolfin_cpp_flags` and `ac_dolfin_ld_flags` are substituted with the results obtained above:

```
CPPFLAGS += @ac_dolfin_cpp_flags@
LDFLAGS += @ac_dolfin_ld_flags@
```

In this way, if dolfin.h is available, CPPFLAGS contains also the flag -DHAVE_DOLFIN_H.

program.cc Our .cc program, should thus include the header dolfin.h only if `-DHAVE_DOLFIN_H` is defined at compilation time. For example

```
#ifdef HAVE_DOLFIN_H
#include <dolfin.h>
#endif
int main ()
{

#ifndef HAVE_DOLFIN_H
    error("program: the program was built without support for
        dolfin");
#else
  /* Body of your function */
#endif
 return 0;
}
```

Warning If in the Makefile.in you write something like

```
HAVE_DOLFIN_H = @HAVE_DOLFIN_H@
ifdef HAVE_DOLFIN_H
  CPPFLAGS += -DHAVE_DOLFIN_H
  LIBS += -ldolfin
endif
```

it doesn't work because the variable `HAVE_DOLFIN_H` seems to be always defined, even if the header is not available.

Bibliography

[1] http://www.gnu.org/software/octave/download.html.

[2] http://fenicsproject.org/download/.

[3] http://fenicsproject.org/documentation/ufl/1.2.0/user/user_manual.html.

[4] http://octave.sourceforge.net/msh/index.html.

[5] http://octave.sourceforge.net/fpl/index.html.

[6] http://octave.sourceforge.net/secs1d/index.html.

[7] http://octave.sourceforge.net/bim/index.html.

[8] http://jordi.platinum.linux.pl/octave/what-is-octave.pdf.

[9] Luca Formaggia. Advanced programming for scientific computing. lecture title: Smart pointers. 2012.

[10] http://fenicsproject.org/documentation/dolfin/1.2.0/cpp/programmers-reference/mesh/MeshFunction.html.

[11] http://fenicsproject.org/documentation/dolfin/1.2.0/cpp/programmers-reference/mesh/MeshValueCollection.html.

[12] http://fenicsproject.org/documentation/dolfin/1.2.0/cpp/programmers-reference/mesh/MeshDomain.html.

[13] http://fenicsproject.org/documentation/dolfin/1.2.0/cpp/programmers-reference/mesh/MeshData.html.

[14] http://gedeone-gsoc.blogspot.co.uk/2013/06/update-4.html.

[15] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The fenics book*, volume 84. Springer, 2012.

[16] http://fenicsproject.org/documentation/dolfin/1.2.0/python/demo/pde/mixed-poisson/python/documentation.html.

[17] http://fenicsproject.org/documentation/dolfin/1.2.0/python/demo/pde/navier-stokes/python/documentation.html.

[18] http://fenicsproject.org/documentation/dolfin/1.2.0/python/demo/pde/hyperelasticity/python/documentation.html.