Fem-fenics

General purpose Finite Element library
for GNU-Octave

Work in progress
(help and remarks are welcome)

Marco Vassallo

November 11, 2013

Contents

Chapter 1

Introduction

Fem-Fenics is an open-source package 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 with 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 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. It means that you can install it only once that you have successfully installed Octave on your PC. Furthermore, as Fem-fenics is based on Fenics, you also need 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 you have got Octave and Fenics, you can just launch Octave (which now is provided with a new amazing GUI) and type

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

That's all! If you encounter any problem during the installation don't hesitate to contact us. To be sure that everything is working fine, you can 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, you are referred to chapter 4.

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

2.2 General layout

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 for 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 we have to specify at this stage is the space of Finite Elements which we want to use for the discretization of H^1_{0,Γ_D}. In our case, we choose the space of continuous lagrangian polynomial of degree one `FiniteElement("Lagrange", triangle, 1)`, but many more possibilities are available.

Poisson.ufl

```
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:

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

shouldn't produce any error. We can now implement and solve a specific instance of the Poisson problem in Octave. We choose to set the parameters 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 pkg that we have previously installed

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

We can thus import the ufl file inside Octave. From the ufl file, we have to generate the corresponding functions for fem-fenics. There is a specific function which seeks every specific element defined inside the ufl file:

- `ufl_import_FunctionSpace ('Poisson')` is a function which looks for the finite element space defined inside the file called Laplace.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 Laplace.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), we can just call `import_ufl_Problem ('Poisson')` which generates at once all the three functions described above.

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

To set the concrete elements which define our problem, the first things to do is to create a mesh. It can be managed easily using the msh pkg. In our case, we create a uniform 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 `fem_init_mesh()`:

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

If instead of an Octave mesh you have a mesh stored in a different format, you can try to convert it to the dolfin xml format using the program dolfin-convert. In fact, `Mesh ()` accepts as arguments also a string with the name of the dolfin xml file which contains your mesh. For example, if you have a mesh saved in the gmsh format, you can do as follows:

```
Shell:
  dolfin-convert msh.gmsh msh.xml
```

and then inside our Octave script:

```
mshd = fem_init_mesh ('msh.xml');
```

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 been specified in the ufl file:

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

We can now apply essential BC using `DirichletBC ()`. This function receives as argument the functional space, a function handle which specifies the value that we want to set, and the label of the sides where we want to apply the BC. In our case, we apply homogenous boundary condition on the left and right side of the square

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

The last thing that we have to do before solving the problem, is to set the coefficients specified in the ufl file. It is important that they are called in the same way as in the ufl file. In our case they are the source term 'f' and the normal flux 'g'. To set them, we can for example use the function `Expression ()` to which we have to pass as argument a string, which specifies the name of the coefficient, and a function handle with the value required:

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

We can now obtain the discretized representation of our operator using the functions `assemble ()` or `assemble_system ()`, which also allow us to specify the BC(s) that we want to apply. Whenever possible, it is better to use the `assemble_system ()` function because it keeps the symmetry of the matrix while setting the entries for the BC:

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

Here A is a sparse matrix and b is a column vector. We can thus use all the functionalities available within Octave to solve the linear system. For the moment we use the easisest possibility, i.e. the backslash command to solve the linear system:

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

Once that the solution has been obtained, we can transform the vector into a fem-fenics function and plot it `plot ()`, or save it `save ()` in the vtu format.

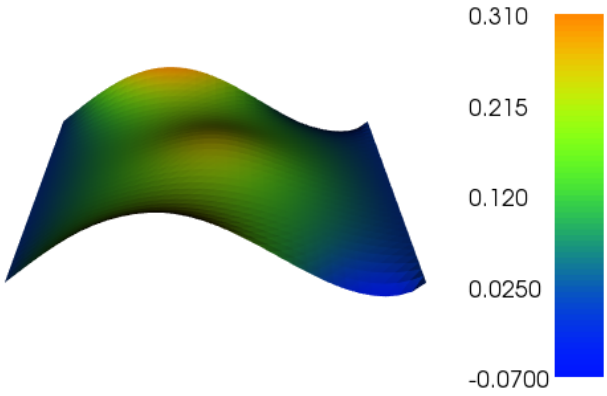

Figure 2.1: The result for the Poisson equation

```
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.

Poisson.m

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

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

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

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

```

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

Chapter 3

Implementation

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
- make the interface as simple as possible.

3.1 General layout of a class

All these classes derive from `octave_base_value`.

3.2 General layout of a function

3.2.1 Mesh generation and conversion

3.2.2 Sparse Matrices

3.2.3 Shared pointer

3.2.4 Polymorphism

3.2.5 Code release

3.2.6 Code on the fly

3.2.7 Autoconf

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

We want to speak about it because, even if it is not strictly related to the femfenics library, I hope it could be helpful for someone else because some solutions which could seem right at a first sight are definitely wrong.

As stated above, if we want to generate automatically a Makefile, 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;
```

```
13    }
```

Warning If in the Makefile.in you write something like

```
1     HAVE_DOLFIN_H = @HAVE_DOLFIN_H@
2     ifdef HAVE_DOLFIN_H
3       CPPFLAGS += -DHAVE_DOLFIN_H
4       LIBS += -ldolfin
5     endif
```

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

Chapter 4

More Advanced Examples

4.1 Navier-Stokes equation with Chorin-Temam projection algorithm

TentativeVelocity.ufl

```
# Copyright (C) 2010 Anders Logg
# Define function spaces (P2-P1)
V = VectorElement("CG", triangle, 2)
Q = FiniteElement("CG", triangle, 1)

# Define trial and test functions
u = TrialFunction(V)
v = TestFunction(V)

# Define coefficients
k  = Constant(triangle)
u0 = Coefficient(V)
f  = Coefficient(V)
nu = 0.01

# Define bilinear and linear forms
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
a  = lhs(eq)
L  = rhs(eq)
```

PressureUpdate.ufl

```
 # Copyright (C) 2010 Anders Logg
 # Define function spaces (P2-P1)
V = VectorElement("CG", triangle, 2)
Q = FiniteElement("CG", triangle, 1)

# Define trial and test functions
p = TrialFunction(Q)
q = TestFunction(Q)

```

```
# Define coefficients
k  = Constant(triangle)
u1 = Coefficient(V)

# Define bilinear and linear forms
a = inner(grad(p), grad(q))*dx
L = -(1/k)*div(u1)*q*dx
```

VelocityUpdate.ufl

```
 # Copyright (C) 2010 Anders Logg
# Define function spaces (P2-P1)
V = VectorElement("CG", triangle, 2)
Q = FiniteElement("CG", triangle, 1)

# Define trial and test functions
u = TrialFunction(V)
v = TestFunction(V)

# Define coefficients
k  = Constant(triangle)
u1 = Coefficient(V)
p1 = Coefficient(Q)

# Define bilinear and linear forms
a = inner(u, v)*dx
L = inner(u1, v)*dx - k*inner(grad(p1), v)*dx
```

NS.m

```
pkg load fem-fenics msh
import_ufl_Problem ("TentativeVelocity");
import_ufl_Problem ("VelocityUpdate");
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
L-shape-domain;
mesh = Mesh (msho);

# Define function spaces (P2-P1).
V = FunctionSpace ('VelocityUpdate', mesh);
Q = FunctionSpace ('PressureUpdate', mesh);

# Set parameter values and define coefficients
dt = 0.01;
T = 3.;
k = Constant ('k', dt);
f = Constant ('f', [0; 0]);
u0 = Expression ('u0', @(x,y) [0; 0]);

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

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

# Assemble matrices
a1 = BilinearForm ('TentativeVelocity', V, V, k);
a2 = BilinearForm ('PressureUpdate', Q, Q);
a3 = BilinearForm ('VelocityUpdate', V, V);
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
  L1 = LinearForm ('TentativeVelocity', V, k, u0, f);
  b1 = assemble (L1, noslip);
  utmp = A1 \ b1;
  u1 = Function ('u1', V, utmp);

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

  # 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
```

L-shape-domain.m

```
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");
```

```
9
10 fputs (fid,"Line (1) = {5, 6};\n");
11 fputs (fid,"Line (2) = {2, 3};\n");
12
13 fputs (fid,"Line(3) = {6,1,2};\n");
14 fputs (fid,"Line(4) = {5,4,3};\n");
15 fputs (fid,"Line Loop(7) = {3,2,-4,1};\n");
16 fputs (fid,"Plane Surface(8) = {7};\n");
17 fclose (fid);
18 msho = msh2m_gmsh (canonicalize_file_name (name)(1:end-4),...
19                   "scale", 1,"clscale", .2);
20 unlink (canonicalize_file_name (name));
```

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

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.