An (incomplete) introduction to FEniCS

Mauro Bonafini

Verona, June 6, 2014

An overview

- The FEniCS Project is a collection of free software with an extensive list of features for automated, efficient solution of differential equations
- Initiated 2003 in Chicago
- C++/Python library (#include <dolfin.h> / form dolfin import *)
- Part of Debian and Ubuntu, licensed under the GNU LGPL
- Automatic and efficient evaluation of variational forms (1D, 2D, 3D) and assembly of linear systems for Finite Elements Method
- General families of finite elements, including arbitrary order continuous and discontinuous Lagrange elements
- Arbitrary mixed elements and built-in plotting
- Good documentation and examples

Mesh generation

Meshes for rectangular, circular and ellipsoidal domains are available as build-in components. The same can be done for domains based on these geometries (e.g. a rectangle with a circular hole).

Mesh for polygonal domains: first define a vector of points representing the vertices and then generate the mesh

```
vertices = [Point(x_1,y_1), ... ,Point(x_n,y_n),Point(x_1,y_1)]
Th = mesh()
PolygonalMeshGenerator.generate(Th, vertices, cell_size)
```

For more general meshes we need to build them using an external software (Triangle, Gmsh, TetGen, ...), convert the resulting file (.ele/.node, .msh/.gmsh, .mesh, ...) into the FEniCS mesh format .xml using

```
dolfin-convert fileID.* fileID.xml
```

and then load the mesh

```
Th = Mesh("fileID.xml")
```

Note: for P1 elements it is not in general true that the i-th coefficient is the value of the function at the i-th node.

An example using Python interface

We want to solve the simple problem

$$\begin{cases} -\Delta u(x,y) = f(x,y) & \text{in } \Omega = [0,1]^2 \\ u(x,y) = u_B(x,y) & \text{on } \Gamma_D \\ -\dfrac{\partial u}{\partial \mathbf{n}} = g(x,y) & \text{on } \Gamma_N \end{cases}$$

where $\Gamma_D = \{(x,y) \in \Omega : x \in \{0,1\}\}$, $\Gamma_N = \{(x,y) \in \Omega : y \in \{0,1\}\}$, $f(x,y) = -6$, $u_B(x,y) = 1 + x + 2y^2$ and $g(x,y) = -4y$. The exact solution is given by

$$u(x,y) = 1 + x^2 + 2y^2$$

and the associated bilinear and linear form are

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} \qquad L(v) = \int_\Omega fv \, d\mathbf{x} - \int_{\Gamma_N} gv \, ds.$$

Python code

The previous problem can be solved using FEniCS as follow:

```
from dolfin import *              # import the software library
set_log_level(PROGRESS)           # suppress some outputs

# Create mesh and define function space
Th = UnitSquareMesh(15,15)        # regular mesh over [0,1]^2
Vh = FunctionSpace(Th, "CG", 1) # piecewise linear basis functions

# Define variational problem
g = Expression("-4*x[1]")         # Neumann boundary condition (x[1] is y)
f = Constant(-6)                  # right-hand side

u = TrialFunction(Vh)             # find u s.t.
v = TestFunction(Vh)              # for all v
a = inner(grad(u), grad(v))*dx  # bilinear form
L = f*v*dx - g*v*ds               # linear form
```

```python
# Define boundary conditions
def boundary(x, on_boundary):   # python function: identify boundary
    tol = 1E-14
    return on_boundary and (abs(x[0]) < tol or abs(x[0] - 1) < tol)

uB = Expression("1 + x[0] + 2*x[1]*x[1]")   # Dirichlet boundary condition
bc = DirichletBC(Vh, uB, boundary)          # set a DirichletBC object

# Compute solution
u = Function(Vh) # the solution is a function

problem = LinearVariationalProblem(a, L, u, bc) # set problem
solver = LinearVariationalSolver(problem)       # create solver
solver.parameters["linear_solver"] = "gmres"    # set solver
solver.parameters["preconditioner"] = "ilu"     # set preconditioner
gmres_prm = solver.parameters["krylov_solver"]  # set some parameters
gmres_prm["absolute_tolerance"] = 1e-7
gmres_prm["relative_tolerance"] = 1e-4
gmres_prm["maximum_iterations"] = 1000
solver.solve()                                  # solve

# Here ends the main part of our code.
```

```
# L2 error and H1 error
uexact = Expression("1+x[0]*x[0]+2*x[1]*x[1]")
L2 = errornorm(uexact, u, norm_type="L2")
H1 = L2 + errornorm(uexact, u, norm_type="H10")
print "\nL2 error:", L2
print "\nH1 error:", H1, "\n"

# Plot solution and gradient
grad_u = project(grad(u), VectorFunctionSpace(Th, "CG", 1))
plot(u, axes=True)
plot(grad_u)

# Dump solution to file in VTK format (or XML format)
file = File("mixedPoisson.pvd")
file << u
file = File("mixedPoisson.xml")
file << u

# Hold plot
interactive()
```

Run in a terminal window "python mixedPoisson2d.py" for the results.

Order of convergence

We check the right order of convergence of our implementation by solving a simpler Poisson equation $-\Delta u = f$ on $\Omega = [-1, 1]^2$, where f and the boundary conditions are chosen s.t.

$$u(x, y) = \left(\sqrt{x^2 + y^2}\right)^{2n+1} \in H^{2n+1}(\Omega)$$

turns out to be the exact solution (FEniCS version of order.edp).

The FreeFem code

```
cout << "number of sites: " << Th.nv << endl;
cout << "number of triangles: " << Th.nt << endl;
cout << "degree of freedom: " << Vh.ndof << endl;
```

is substituted by

```
print "number of sites: ", Th.num_vertices()
print "number of triangles: ", Th.num_cells()
print "degree of freedom: ", u.vector().array().size
```

Run in a terminal window "python orderPoisson2d.py *n r*" for the results.

Assembly of the linear system and its resolution

There are several ways for generating the stiffness matrix A, the load vector b and solve the corresponding linear system, for example

```
problem = LinearVariationalProblem(a, L, u, bc)
solver = LinearVariationalSolver(problem)
solver.solve()
# or
solve(a == L, u, bc)
# or
A = assemble(a)
b = assemble(L)
bc.apply(A, b)
solve(A, u.vector(), b)
# or
A, b = assemble_system(a, L, bc)
solve(A, u.vector(), b)
```

In every case we can control the resolution process as we have seen in the previous code. For a complete list of parameters affecting the solution of a linear system run in a terminal window

```
python LinearAlgebra.py
```

C++ code

First of all we have to define the variational problem in an external file mixed.ufl

```
# The bilinear form a(u, v) and linear form L(v) for our example
element = FiniteElement("CG", 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
```

and then compile this file

```
ffc -l dolfin mixed.ufl
```

to get a C++ header file, mixed.h, which we will include in our main.cpp file, which reads as follows:

```cpp
#include <dolfin.h>
#include "mixed.h"
using namespace dolfin;

// boundary value
class functionUB : public Expression {
    void eval(Array<double>& values, const Array<double>& x) const {
        values[0] = 1 + x[0] + 2*x[1]*x[1];
    }
};
// Neumann condition
class functionG : public Expression {
    void eval(Array<double>& values, const Array<double>& x) const {
        values[0] = -4*x[1];
    }
};
// Dirichlet boundary
class OurDirichletBoundary : public SubDomain {
    bool inside(const Array<double>& x, bool on_boundary) const {
        return on_boundary && (x[0] < DOLFIN_EPS || x[0] > 1-DOLFIN_EPS);
    }
};
```

```cpp
int main(void) {
    // Create mesh and function space
    UnitSquareMesh Th(32, 32);       // regular mesh over [0,1]^2
    mixed::FunctionSpace Vh(Th);     // function space defined in mixed.h

    // Define boundary conditions
    functionUB uB;                       // Dirichlet boundary condition
    OurDirichletBoundary boundary;       // identify boundary
    DirichletBC bc(Vh, uB, boundary);    // set a DirichletBC object

    // Define variational forms
    Constant f(-6); functionG g;
    mixed::BilinearForm a(Vh, Vh);  // the bilinear form from mixed.h
    mixed::LinearForm L(Vh);        // the linear form from mixed.h
    L.f = f; L.g = g;               // initialize parameters of L

    // Compute solution
    Function u(Vh);
    solve(a==L, u, bc);

    /* ... */
    return 0;
}
```

FEniCS vs FreeFem++

The same problem can be solved using FreeFem++

```
/* Create mesh and define function space */
mesh Th = square(32,32);
fespace Vh(Th, P1);

/* Define the various functions */
func uB = 1+x+2*y^2;
func f = -6;
func g = -4*y;

/* Define the problem and compute solution */
Vh u, v;
solve mixed(u,v) =
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
    - int2d(Th)(f*v)
    - int1d(Th,1,3)(-g*v)
    + on(2,4,u=uB);
```

It seems that for this particular example FreeFem++ outperforms FEniCS in both cases (C++ and Python implementation).

Nonlinear problems in FEniCS

Let us consider the advection–reaction nonlinear problem

$$-\mu \Delta u(x,y) + \rho u^2(x,y) = 1 \quad (x,y) \in [0,1]^2$$

with homogeneous Dirichlet boundary conditions. The weak formulation requires us to find $u \in H_0^1$ such that

$$F(u) = \mu \int \nabla u \cdot \nabla v + \rho \int u^2 v - \int v = 0 \quad \forall v \in H_0^1.$$

FEniCS has the capability to treat directly this kind of problems.
Note: we have already seen during our lectures the FreeFem++ code for solving this problem.

```python
from dolfin import *
Th = UnitSquareMesh(20,20)
Vh = FunctionSpace(Th, "CG", 1)
mu = Constant(0.01)
rho = Constant(1.0)

# Define boundary conditions
def Boundary(x, on_boundary):
    return on_boundary
bc = DirichletBC(Vh, Constant("0.0"), on_boundary)

# Define variational problem
du = TrialFunction(Vh)
v  = TestFunction(Vh)
u = Function(Vh)
F  = (mu*inner(grad(u), grad(v))+rho*(u*u*v)-v)*dx
J = derivative(F, u, du)    # automatic computation of the Jacobian

# Compute solution
problem = NonlinearVariationalProblem(F, u, bc, J) # set nonlinear problem
solver  = NonlinearVariationalSolver(problem)      # set nonlinear solver
solver.solve()                                     # solve
```

Discontinuous Galerkin approach in FEniCS

For the Poisson problem it is possible to derive the following DG-N formulation: find $u_\delta \in W_\delta$ such that

$$\sum_{m=1}^{M} (\nabla u_\delta, \nabla v_\delta)_{K_m} - \sum_{e \in \mathcal{E}_\delta} \int_e [v_\delta] \cdot \{\{\nabla u_\delta\}\} - \tau \sum_{e \in \mathcal{E}_\delta} \int_e [u_\delta] \cdot \{\{\nabla v_\delta\}\}$$

$$+ \sum_{e \in \mathcal{E}_\delta} \frac{\gamma}{|e|} \int_e [u_\delta] \cdot [v_\delta] - \sum_{e \subset \partial\Omega} \int_e v_\delta \frac{\partial u_\delta}{\partial \mathbf{n}} - \tau \sum_{e \subset \partial\Omega} \int_e u_\delta \frac{\partial v_\delta}{\partial \mathbf{n}}$$

$$+ \sum_{e \subset \partial\Omega} \frac{\gamma}{|e|} \int_e u_\delta v_\delta = \sum_{m=1}^{M} (f, v_\delta)_{K_m} - \tau \sum_{e \subset \partial\Omega} \int_e g_\delta \frac{\partial v_\delta}{\partial \mathbf{n}} + \sum_{e \subset \partial\Omega} \frac{\gamma}{|e|} \int_e g_\delta v_\delta$$

$\forall v_\delta \in W_\delta$. We solve $-\Delta u = f$ with homogeneous Dirichlet b.c. and f chosen s.t. the exact solution is $u(x,y) = (x - x^2)\exp(3x)\sin(2\pi y)$ (section 11.1.1 in Quarteroni's book).

Note: $|e|$ should be replaced with something closer to it.

```python
from dolfin import *
r = 1
tau = 1.0
gamma = 10.0*r*r
f = Expression("-exp(3*x[0])*(-9*pow(x[0],2)-3*x[0]+4)*sin(2*pi*x[1]) \
                +(x[0]-pow(x[0],2))*exp(3*x[0])*4*pow(pi,2)*sin(2*pi*x[1])"
g = Constant(0.0)
Th = UnitSquareMesh(30,30)
Vh = FunctionSpace(Th, "DG", r)

u = TrialFunction(Vh)
v = TestFunction(Vh)
n = FacetNormal(Th)
h = CellSize(Th)

a = dot(grad(u), grad(v))*dx - dot(jump(v, n), avg(grad(u)))*dS \
  - tau*dot(jump(u, n), avg(grad(v)))*dS \
  + gamma/avg(h)*dot(jump(u, n), jump(v, n))*dS \
  - v*dot(grad(u), n)*ds - tau*u*dot(grad(v), n)*ds + gamma/h*u*v*ds
L = f*v*dx - tau*g*v*ds + gamma/h*g*v*ds

u = Function(Vh)
solve(a==L, u)
```