INTRODUCTION TO THE FINITE ELEMENT SOFTWARE FEniCS

ROLAND HERZOG

1 Introduction

FEniCS is free software for automated solution of differential equations. As a particular feature, it allows the formulation of the weak (variational) forms in a rather natural way. For instance, the bilinear and linear forms associated to the Poisson problem,

$$\text{Find} \quad u \in V \quad \text{such that} \quad a(u,v) = L(v) \quad \text{for all } v \in V$$

with

$$a(u,v) = \int_\Omega \nabla v \cdot \nabla u \, dx, \qquad L(v) = \int_\Omega v \, f \, dx$$

can be formulated as

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

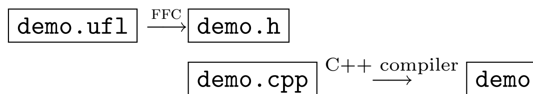
The language used here is the so-called Unified Form Language (UFL), which is an extension of the Python scripting language.

1.1. Components in the FEniCS Project. The interaction of components within the FEniCS project is reflected in Figure 1.1. The core library is Dolfin, which is a library written in C++ with interfaces to C++ and Python (as a module). It provides functionalities such as

- mesh handling (refinement, coarsening, smoothing, partitioning),
- mesh generation (currently only for simple shapes),
- assembly of linear and bilinear (in general: multilinear) forms,
- input/output routines

Linear algebra and visualization is handled by external software.

The typical usage of the library within a C++ program is like this:



In this case, the form compiler FFC (or its alternative, SyFi/SFC) generates a header file `demo.h` which defines a number of problem specific classes that may be instantiated by the user in her main program and passed to the Dolfin library. These provide the functionality needed in the innermost loop of FE assembly routines, e.g.,

- evaluation of basis functions and their derivatives,
- application of local degrees of freedom to a function,

Date: February 10, 2010.

- local-to-global mapping of the degrees of freedom,
- evaluation of the local element tensor (e.g., the local stiffness matrix).

The code which implements this functionality is automatically generated by the form compiler from the high-level description in the `.ufl` file. By contrast, from within a Python program, the form compiler is called automatically whenever necessary (just in time).

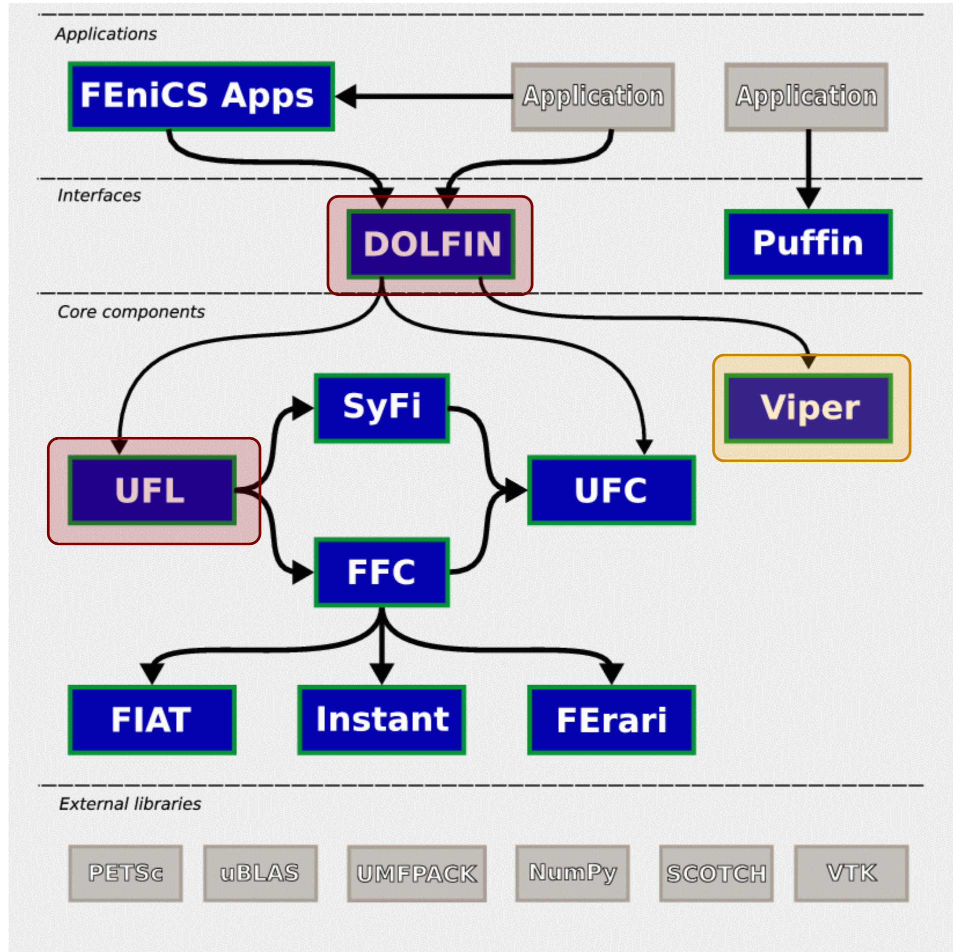


FIGURE 1.1. Interaction of components in FENICS. UFL = Unified Form Language, FFC = Unified Form Compiler, UFC = Unified Form-Assembly Code. Highlighted are the components we refer to this text.

1.2. Some Features and Limitations. FENICS currently supports the following types of finite element function spaces (see [5, Section 4.3]):

- H^1 conforming FE spaces composed of:
  - Lagrange elements of arbitrary degree
- $H(\text{div})$ conforming FE spaces composed of:
  - Raviart-Thomas elements of arbitrary degree
  - Brezzi-Douglas-Marini elements of arbitrary degree

- Brezzi-Douglas-Fortin-Marini elements of arbitrary degree
  - Nédélec elements (1st kind) of arbitrary degree
  - Nédélec elements (2nd kind) of arbitrary degree
- $H(\text{curl})$ conforming FE spaces composed of:
  - Nédélec elements (1st kind) of arbitrary degree
  - Nédélec elements (2nd kind) of arbitrary degree
- L^2 conforming FE spaces composed of:
  - discontinuous Lagrange elements of arbitrary degree
  - Crouzeix-Raviart elements (degree one)

All of these can be used on simplicial meshes in 1D, 2D and 3D (intervals, triangles, tetrahedra), but currently not on other geometries (rectangles, cubes, prisms etc.).

The variational form language is described in more detail in Section 3. It supports in particular

- scalar, vector valued and matrix valued finite elements,
- integrals over cells, interior facets and exterior facets (but not over arbitrary lines or surfaces),
- jumps and averages over facets, which can be used for error estimation and in discontinuous Galerkin (DG) approaches.

The visualization of results is accomplished by external packages such as ParaView, MayaVi, or Viper, which is also part of the FENICS project.

2 Examples

The following examples conform to FFC version 0.7.0 (`ffc -v`) and the DOLFIN core library version 0.9.4 (`apt-cache -f search libdolfin0`), which came with the Ubuntu 9.10 (karmic koala) packages. Note that the interfaces to the DOLFIN library are still changing often.

2.1. The Poisson Equation.

Purpose of this example:

- solve Poisson equation with Lagrange FEs of various degrees
- present C++ and Python interfaces
- visualize the results
- implement various choices of boundary conditions and non-constant coefficients

We consider the Poisson equation in its variational form, i.e., find $u \in V$ such that

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

for all $v \in V = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\}$, where Γ_D is the Dirichlet part of the boundary $\Gamma = \partial\Omega$. In its strong form, this problem corresponds to $-\triangle u = f$ in Ω with boundary conditions $u = 0$ on Γ_D and $\partial u / \partial n = 0$ on the complement Γ_N.

The corresponding variational description in UFL looks like this (`Example_Poisson.ufl`):

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

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)
```

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

We compile it using FFC and obtain a header file `Example_Poisson.h`.

```
  ffc -l dolfin Example_Poisson.ufl
```

The following main program makes use of the general DOLFIN classes `Function`, `SubDomain`, `UnitSquare`, `Constant`, `DirichletBC`, `VariationalProblem`, and `File`. It also uses the classes `FunctionSpace`, `BilinearForm` and `LinearForm` which have been defined in the header file `Example_Poisson.h`, automatically generated by FFC.

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

using namespace dolfin;

// Source term
class Source : public Expression
{
public:

  Source() : Expression(2) {}

  void eval(double* values, const double* x) const
  {
    double dx = x[0] - 0.5;
    double dy = x[1] - 0.5;
    values[0] = 500.0*exp(-(dx*dx + dy*dy) / 0.02);
  }
};

// Subomain for Dirichlet boundary condition
class DirichletBoundary : public SubDomain
{
  bool inside(const double* x, bool on_boundary) const
  {
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS;
  }
};

int main()
{
  // Create mesh and function space
  UnitSquare mesh(32, 32);
  Example_Poisson::FunctionSpace V(mesh);

  // Define boundary condition
  Constant u0(mesh, 0.0);
  DirichletBoundary boundary;
  DirichletBC bc(V, u0, boundary);
```

```
  // Define variational problem
  Example_Poisson::BilinearForm a(V, V);
  Example_Poisson::LinearForm L(V);
  Source f;
  L.f = f;

  // Setup problem and compute solution
  VariationalProblem problem(a, L, bc);
  Function u(V);
  problem.parameters["linear_solver"] = "iterative";
  problem.solve(u);

  // Save solution in VTK format
  File solution_file("poisson_solution.pvd");
  solution_file << u;

  // Save the mesh also
  File mesh_file("poisson_mesh.xml");
  mesh_file << mesh;

  // Save the parameter file
  File parameter_file("poisson_parameters.xml");
  parameter_file << problem.parameters;

  //Plot the mesh and the solution (using viper)
  plot(mesh);
  plot(u);

  return 0;
}
```

The we invoke the build tool SCons to obtain the executable `Example_Poisson`, run it and visualize the solution, once with ParaView, once with MayaVi.

```
  scons
  ./Example_Poisson
  paraview --data=poisson_solution.pvd
  mayavi -d poisson_solution000000.vtu
```

Here is a file with the same functionality in Python:

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquare(32, 32)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define Dirichlet boundary (x = 0 or x = 1)
class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (x[0]<DOLFIN_EPS or x
            [0]>1.0-DOLFIN_EPS)

# Define boundary condition
```

```
u0 = Constant(mesh, 0.0)
bc = DirichletBC(V, u0, DirichletBoundary())

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
f = Expression("500.0*exp(-(pow(x[0]-0.5,2)+pow(x
    [1]-0.5,2))/0.02)", V = V)
a = inner(grad(v), grad(u))*dx
L = v*f*dx

# Setup problem and compute solution
problem = VariationalProblem(a, L, bc)
problem.parameters["linear_solver"] = "iterative"
u = problem.solve()

# Save solution in VTK format
solution_file = File("poisson_solution.pvd")
solution_file << u

# Save the mesh also
mesh_file = File("poisson_mesh.xml")
mesh_file << mesh

# Save the parameter file
parameter_file = File("poisson_parameters.xml")
parameter_file << problem.parameters

# Plot the mesh and the solution (using viper)
plot(mesh)
plot(u)
interactive()
```

Execute it by saying

```
python -i Example_Poisson.py
```

where the switch `-i` stands for interactive mode after the script has been executed.

Changing the Boundary Conditions. We now change the problem by modifying the boundary conditions. We use $u = \sin(\pi\, x_2)$ on Γ_D, $\partial u/\partial n + \alpha\, u = g$ with $\alpha(x) \equiv 1$ and $g(x) = -10\, x_1$ on the complement Γ_N. Moreover, we replace $-\triangle u$ by $-\operatorname{div}(A\nabla u)$ with $A(x) = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$.

The variational problem then becomes: Find $u \in u_0 + V$ such that

$$\int_\Omega \nabla v \cdot A \cdot \nabla u \, dx + \int_{\Gamma_N} \alpha\, v\, u \, ds = \int_\Omega v\, f \, dx + \int_{\Gamma_N} v\, g \, ds$$

for all $v \in V = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\}$.

Here are the relevant changes to the Python code above:

```
# Define boundary condition
u0 = Expression("sin(x[1]*pi)", V = V)
```

```
bc = DirichletBC(V, u0, DirichletBoundary())

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
f = Expression("500.0*exp(-(pow(x[0]-0.5,2)+pow(x
    [1]-0.5,2))/0.02)", V = V)
g = Expression("-10.0*x[0]", V = V)
A = as_matrix([[2.0, 1.0], [1.0, 2.0]])
alpha = Constant(mesh, 1.0)
a = inner(grad(v), A*grad(u))*dx + (alpha * v * u)*ds
L = v*f*dx + v*g*ds
```

Suggestions. Change the order of the finite element. Change the setting to 3D. Try

```
help(u)
print(u); u(0.5,0.5); u.cell(); u.element(); u.vector()
    .array()[0:10]

help(V)
print(V); V.dim(); type(V)

help(mesh)
print(mesh); mesh.coordinates()[0:10]; mesh.cells()
mesh.hmax(); mesh.num_entities(0)
```

in the interactive Python shell. Try also the command line tool `ufl-convert`.

2.2. The Stokes System.

Purpose of this example:

- introduce vector valued and compound finite elements (velocity/pressure)
- implement different boundary conditions for different components of a system
- export finite element matrices
- import finite element meshes

We consider the Stokes system in its strong form:

$$\begin{aligned} -\mu \triangle \boldsymbol{u} + \nabla p &= \boldsymbol{f} \quad \text{in } \Omega, \\ \operatorname{div} \boldsymbol{u} &= 0 \quad \text{in } \Omega, \end{aligned}$$

where $\boldsymbol{u}$ represents the velocity of a fluid with dynamic viscosity μ, and p is its pressure. Suitable boundary conditions are for example $\boldsymbol{u} = \boldsymbol{u}_0$ on Γ, which have to satisfy the compatibility condition $\int_\Gamma \boldsymbol{u}_0 \cdot \boldsymbol{n} = 0$. Note that the pressure is only determined up to a constant factor in this setting. We take as a first example $\Omega = (0,1)^2$ with $\boldsymbol{u}_0 = (1,0)$ on the upper edge, $\boldsymbol{u}_0 = (0,0)$ elsewhere and $\boldsymbol{f} = (0,0)$. This problem is known as a lid-driven cavity flow. The weak formulation of the Stokes problem is: Find $(\boldsymbol{u}, p) \in (\boldsymbol{u}_0 + V) \times Q$ such that

$$\begin{aligned} \mu \int_\Omega \nabla \boldsymbol{v} : \nabla \boldsymbol{u} \, dx - \int_\Omega \operatorname{div} \boldsymbol{v} \, p \, dx + \int_\Gamma p \underbrace{\boldsymbol{v} \cdot \vec{n}}_{=0} \, ds &= \int_\Omega \boldsymbol{v} \cdot \boldsymbol{f} \, dx \\ \int_\Omega q \operatorname{div} \boldsymbol{u} \, dx &= 0 \end{aligned}$$

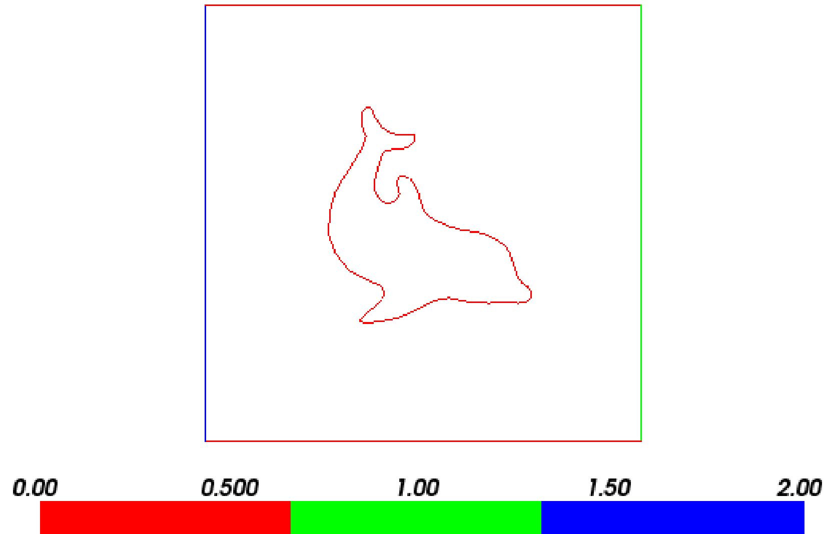
for all $\boldsymbol{v} \in V = \{\boldsymbol{v} \in [H^1(\Omega)]^2 : \boldsymbol{v} = \mathbf{0} \text{ on } \Gamma\}$ and all $q \in Q = \{q \in L^2(\Omega) : \int_\Omega q \, dx = 0\} \cong L^2(\Omega)/\mathbb{R}$. The problem has a saddle point structure.

```
from dolfin import *

# Create mesh and define function spaces (Taylor-Hood FE)
mesh = UnitSquare(16, 16)
V = VectorFunctionSpace(mesh, "Lagrange", 2)
Q = FunctionSpace(mesh, "Lagrange", 1)
W = V + Q

# Define Dirichlet boundary (everywhere)
class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

# Define Dirichlet boundary condition everywhere
u0 = Expression(("x[1]>1.0-"+str(DOLFIN_EPS),"0"), V = V)
bc = DirichletBC(W.sub(0), u0, DirichletBoundary())

# Define variational problem
(v, q) = TestFunctions(W) # same as vq = TestFunction(W);
    (v,q) = split(vq)
(u, p) = TrialFunctions(W)
f = Constant(mesh, (0.0, 0.0))
mu = Constant(mesh, 1.0e-3)
n = FacetNormal("triangle")
a = (mu*inner(grad(v), grad(u)) - div(v)*p + q*div(u))*dx
    + (p*dot(v,n))*ds
L = inner(v, f)*dx

# Setup problem and compute solution
problem = VariationalProblem(a, L, bc)
problem.parameters["linear_solver"] = "direct"
U = problem.solve()

# Split the mixed solution using a deep copy
# (copy data instead of sharing, needed to compute norms
   below)
(u, p) = U.split(True)

# The average value of the pressure is currently random
# since the condition \int p dx = 0 has not yet been
# enforced. We adjust the average of p now as a post
# processing step.
int_p = p*dx
average_p = assemble(int_p, mesh = mesh)
p_array = p.vector().array() - average_p
p.vector()[:] = p_array

# Save solution in VTK format
ufile_pvd = File("stokes_velocity.pvd")
ufile_pvd << u
```

```
pfile_pvd = File("stokes_pressure.pvd")
pfile_pvd << p

# Plot solution
plot(u, title="Velocity")
plot(p, title="Pressure")

# Alternatively, we can assemble the linear system
# manually, which offers the possibility of exporting it
A = assemble(a)
b = assemble(L)
mfile = File("A_before_bc.m")
mfile << A
mfile = File("b_before_bc.m")
mfile << b

# Apply the Dirichlet boundary conditions and export the
# linear system again
bc.apply(A, b)
mfile = File("A_after_bc.m")
mfile << A
mfile = File("b_after_bc.m")
mfile << b

# Solve the manually assembled problem
U2 = Function(W)
solve(A, U2.vector(), b)

# Compute the difference (should be zero) of velocities
(u2, p2) = U2.split(True)
diff = u.vector() - u2.vector()
print "Norm of velocity difference: %.4e" % diff.norm("l2
   ")
```

Note that we have not enforced the condition $\int_\Omega p\,dx = 0$, and thus the system matrix has a kernel of dimension one. (This is handled by the linear algebra subsystem.)

Changing the Boundary Conditions. We now consider a different possibility of specifying Dirichlet boundary conditions. Instead of using boolean indicator functions like `DirichletBoundary` above, one may use a `MeshFunction` to distinguish various parts of the boundary. In general, a `MeshFunction` is a function which is defined on the entities of a given mesh, e.g., on its cells, facets, or vertices. It can be used for many purposes such as to represent subdomains of different material, to mark cells for refinement, or to identify different types of boundary conditions.



In the present situation, we follow one of the examples that came with DOLFIN and consider the following boundary conditions for the Stokes system:

$$\begin{aligned} \boldsymbol{u} &= 0 && \text{on } \Gamma_0 && \text{no slip boundary} \\ \boldsymbol{u} &= \boldsymbol{u}_0 && \text{on } \Gamma_1 && \text{inflow boundary} \\ p &= 0 && \text{on } \Gamma_2 && \text{outflow boundary.} \end{aligned}$$

These three parts of the boundary are specified by the values $\{0, 1, 2\}$ of a mesh function. In this setting, the value of the pressure is uniquely defined, i.e., the FE stiffness matrix is invertible.

```
from dolfin import *

# Load mesh and subdomains
mesh = Mesh("dolfin-2.xml.gz")
subdomains = MeshFunction("uint", mesh, "subdomains.xml.
    gz")

# Define function spaces
V = VectorFunctionSpace(mesh, "Lagrange", 2)
Q = FunctionSpace(mesh, "Lagrange", 1)
W = V + Q

# No-slip boundary condition for velocity
bc_noslip = Constant(mesh, (0, 0))
bc0 = DirichletBC(W.sub(0), bc_noslip, subdomains, 0)

# Inflow boundary condition for velocity
bc_inflow = Expression(("-sin(x[1]*pi)", "0.0"), V = V)
bc1 = DirichletBC(W.sub(0), bc_inflow, subdomains, 1)

# Boundary condition for pressure at outflow
bc_zero = Constant(mesh, 0.0)
```

```
bc2 = DirichletBC(W.sub(1), bc_zero, subdomains, 2)

# Collect boundary conditions
bcs = [bc0, bc1, bc2]

# Define variational problem
(v, q) = TestFunctions(W)
(u, p) = TrialFunctions(W)
f = Constant(mesh, (0, 0))
mu = Constant(mesh, 1.0e-3)
n = FacetNormal("triangle")
a = (mu*inner(grad(v), grad(u)) - div(v)*p + q*div(u))*dx
    + (p*dot(v,n))*ds
L = inner(v, f)*dx

# Setup problem and compute solution
problem = VariationalProblem(a, L, bcs)
problem.parameters["linear_solver"] = "direct"
U = problem.solve()

# Split the mixed solution using deep copy
# (needed for further computation on coefficient vector)
(u, p) = U.split(True)
u_plot = plot(u)
p_plot = plot(p)

print "Norm of velocity coefficient vector: %.4e" % u.
   vector().norm("l2")
print "Norm of pressure coefficient vector: %.4e" % p.
   vector().norm("l2")


# Alternative assembly process with more than one
# Dirichlet boundary condition: simply use:
#   A, b = assemble_system(a, L, bcs)
# or the following:
A = assemble(a)
b = assemble(L)
for condition in bcs:
    condition.apply(A, b)

# Solve the manually assembled problem
U2 = Function(W)
solve(A, U2.vector(), b)

# Compute the difference (should be zero) of velocities
(u2, p2) = U2.split(True)
diff = u.vector() - u2.vector()
print "Norm of velocity difference: %.4e" % diff.norm("l2
   ")
```

```
# Extract the boundary mesh
boundary_mesh = BoundaryMesh(mesh)

# When the boundary mesh is created, we also get two maps
# (attached to the boundary mesh as mesh data) named
# "cell map" and "vertex map". They are mesh functions
# which map the cells and vertices of the boundary mesh
# to their counterparts in the original mesh.
cell_map = boundary_mesh.data().mesh_function("cell map")

# Now we can create a plottable version of the subdomains
# function (only vertex and cell valued meshfunctions can
# be plotted in viper).
boundary_subdomains = MeshFunction("uint", boundary_mesh,
     1)
boundary_subdomains.values()[:] = subdomains.values()[
    cell_map.values()]

# Plot the mesh and boundary mesh
mesh_plot = plot (mesh, title="Mesh")
boundary_mesh_plot = plot(boundary_mesh, title="Boundary
    Mesh")
boundary_subdomains_plot = plot(boundary_subdomains,
    title="Boundary Subdomains")

# Save the boundary subdomains plot as a file
boundary_subdomains_plot.write_png("stokes_subdomains.png
    ")

# Save the boundary mesh (this includes the "cell map"
    and "vertex map"
# mesh data)
mesh_file = File("stokes_boundary_mesh.xml")
mesh_file << boundary_mesh
```

2.3. A Nonlinear Equation. Nonlinear equations can be solved through the use of Newton's method. We denote the nonlinear residual by $F(u)$. Every Newton step requires the solution of

$$J(u)\,\delta u = -F(u),$$

where $J(u) = F'(u)$ is the Jacobian of F. In the context of PDEs, $F(u)$ is a vector whose components are generated from a linear form $L(u; v)$, as v ranges over all test functions: $[F(u)]_i = L(v_i; u)$. And thus the Jacobian matrix corresponds to a bilinear form $a(\cdot, \cdot)$ in the following sense:

$$[J(u)\,\delta u]_i = a(v_i, \delta u; u).$$

(Recall that test functions are always the first argument in a bilinear form, and trial functions are second.) It is a particular feature of FEniCS that the Jacobian can be generated automatically from differentiating L. This is achieved by the statement `a = derivative(L, u, du)` in the code below, which solves the semilinear problem

$$\begin{aligned} -\triangle u + u^3 &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma. \end{aligned}$$

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquare(32, 32)
V = FunctionSpace(mesh, "Lagrange", 2)

# Define Dirichlet boundary (everywhere)
class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

# Define boundary condition
u0 = Constant(mesh, 0.0)
bc = DirichletBC(V, u0, DirichletBoundary())

# Define the source and solution functions
f = Expression("500.0*exp(-(pow(x[0]-0.5,2)+pow(x
   [1]-0.5,2))/0.02)", V = V)
u = Function(V)

# Define the variational problem which represents the
   Newton step
v = TestFunction(V)
du = TrialFunction(V)

# We need to define only the nonlinear residual
L = inner(grad(v), grad(u))*dx + v*u**3*dx - v*f*dx

# since the bilinear form can be created automatically
# as the derivative of the residual L at u
a = derivative(L, u, du)

# Setup the nonlinear problem L(u) = 0
problem = VariationalProblem(a, L, bc, nonlinear=True)

# and solve it; NOTE that u = problem.solve()
# DOES NOT WORK (no progress in iterations)
problem.solve(u)
```

Note that the high level command `problem.solve(u)` calls the Newton solver, and the assembly of the Jacobian and the residual is triggered automatically by the solver. The following code shows how to achieve the same with lower level code. We manually create the problem class `SemilinearEquation`, derived from the `NonlinearProblem` class. Note that we need to implement the evaluation routine of the residual $F(v;u)$ and of the Jacobian $J(v,\delta u;u)$. These consist in calling the `assemble` routine and then applying the Dirichlet boundary conditions for the Newton step. We also need to manually create the Newton solver instance.

```
# We now solve the same problem again, but we manually
# create the NonlinearProblem instance and the solver.
# See also demo/pde/cahn-hilliard/python/demo.py
class SemilinearEquation(NonlinearProblem):
```

```
    # The constructor is defined to take all arguments we
        need
    def __init__(self, a, L, bc):
        NonlinearProblem.__init__(self)
        self.L = L
        self.a = a
        self.bc = bc
        self.reset_sparsity = True
    # evaluation of residual F(x)
    def F(self, b, x):
        assemble(self.L, tensor=b)
        self.bc.apply(b)
    # evaluation of Jacobian F'(x)
    def J(self, A, x):
        assemble(self.a, tensor=A, reset_sparsity=self.
            reset_sparsity)
        self.bc.apply(A)
        self.reset_sparsity = False

# Define another solution function
u2 = Function(V)
v = TestFunction(V)
du2 = TrialFunction(V)

# This means that we also need to define the variational
    problem again
L2 = inner(grad(v), grad(u2))*dx + v*u2**3*dx - v*f*dx
a2 = derivative(L2, u2, du2)

# Create nonlinear problem and Newton solver
problem2 = SemilinearEquation(a2, L2, bc)
solver = NewtonSolver("lu")
solver.parameters["convergence_criterion"] = "incremental
    "
solver.parameters["relative_tolerance"] = 1e-6

# Solve the problem
solver.solve(problem2, u2.vector())

# Compare the solutions
diff = u.vector() - u2.vector()
print "Norm of solution difference: %.4e" % diff.norm("l2
    ")
```

Suggestions. Modify the problem to include non-homogeneous boundary conditions $u = g$ on Γ, and mixed boundary conditions.

2.4. Magnetostatic Example. In this section we consider the problem of finding the magnetic field induced by a given static current. The Maxwell's equations reduce to the system

$$\nabla \times \boldsymbol{H} = \boldsymbol{J}, \qquad \nabla \cdot \boldsymbol{B} = 0$$

in this case, where $\vec{J}$ is the given current density and μ is the magnetic permeability which may vary in space. The unknown quantities are the magnetic induction $\boldsymbol{B}$ and the magnetic field $\boldsymbol{H}$. In materials with linear constitutive laws, they are related by $\boldsymbol{B} = \mu \boldsymbol{H}$. We thus write the system above as

$$\nabla \times (\mu^{-1}\boldsymbol{B}) = \boldsymbol{J}, \qquad \nabla \cdot \boldsymbol{B} = 0$$

entirely in terms of the magnetic induction $\boldsymbol{B}$. These equations hold on a certain domain Ω, which is shielded from the rest of $\mathbb{R}^3$ through certain boundary conditions. In view of $\boldsymbol{B}$ being divergence-free, we can find a vector potential $\boldsymbol{A}$ such that $\boldsymbol{B} = \nabla \times \boldsymbol{A}$. Plugging in, we see that

$$\nabla \times (\mu^{-1}\nabla \times \boldsymbol{A}) = \boldsymbol{J}$$

holds on Ω. However, the vector potential is clearly not unique, since $\boldsymbol{B} = \nabla \times (\boldsymbol{A} + \nabla\phi)$ holds for all scalar fields ϕ. Uniqueness of $\boldsymbol{A}$ can be obtained through a gauging condition. We use the Coulomb gauge which demands that $\nabla \cdot \boldsymbol{A} = 0$.

In order to introduce the weak formulation, we set

$$H(\mathrm{curl};\Omega) = \{\boldsymbol{v} \in L^2(\Omega)^3 : \nabla \times \boldsymbol{v} \in L^2(\Omega)^3\}$$
$$H_0(\mathrm{curl};\Omega) = \{\boldsymbol{v} \in L^2(\Omega)^3 : \nabla \times \boldsymbol{v} \in L^2(\Omega)^3,\ \boldsymbol{v} \times \boldsymbol{n} = 0 \text{ on } \Gamma\}.$$

The subspace $H_0(\mathrm{curl};\Omega)$ of $H(\mathrm{curl};\Omega)$ consists of those vector fields which have vanishing tangential trace on the boundary Γ of Ω. This is the appropriate form of essential (Dirichlet-type) boundary conditions for functions in $H(\mathrm{curl};\Omega)$. From a technical point of view, $\boldsymbol{A} \times \boldsymbol{n} = 0$ on Γ represent that Ω has perfect electrically conducting walls.

The weak form of our equation becomes: Find $(\boldsymbol{A}, \varphi) \in H_0(\mathrm{curl};\Omega) \times H^1(\Omega)$ such that

$$\int_\Omega (\nabla \times \boldsymbol{v}) : (\mu^{-1}\nabla \times \boldsymbol{A})\,dx + \int_\Omega \boldsymbol{v} \cdot \nabla\varphi\,dx = \int_\Omega \boldsymbol{v} \cdot \boldsymbol{J}\,dx$$
$$\int_\Omega \nabla\psi \cdot \boldsymbol{A}\,dx = 0$$

for all $(\boldsymbol{v}, \psi) \in H_0(\mathrm{curl};\Omega) \times H^1(\Omega)$. Note that the given current density $\boldsymbol{J}$ has to satisfy the compability condition $\nabla \cdot \boldsymbol{J} = 0$ in Ω.

We can implement this variational problem in FEniCS using the vector valued $H(\mathrm{curl};\Omega)$-conforming Nédélec elements (of the first kind). As an example, we present the code where $\boldsymbol{J}$ is the current density in a cylindrical coil (and zero elsewhere). The magnetic permeability is $\mu =$??? in the conductive region of the coil and $\mu =$ (as in vacuum) elsewhere.

In a postprocessing step, we also recover $\boldsymbol{B} = \nabla \times \boldsymbol{A}$ by projecting the expression $\nabla \times \boldsymbol{A}$ onto the space of piecewise constant vector functions.

2.5. Further Reading.

- FEniCS web site
- FEniCS tutorial [3]
- UFC specifications [1]

3 Overview of Unified Form Language (UFL)

According to the UFL Specification and User Manual [1], UFL is designed to express variational forms of the following kind:

$$
\begin{aligned}
a(v_1,\dots,v_r;w_1,\dots,w_n) = &\sum_{k=1}^{n_c}\int_{\Omega_k} I_k^c(v_1,\dots,v_r;w_1,\dots,w_n)\,dx \\
&+\sum_{k=1}^{n_e}\int_{\partial\Omega_k} I_k^e(v_1,\dots,v_r;w_1,\dots,w_n)\,ds \\
&+\sum_{k=1}^{n_i}\int_{\Gamma_k} I_k^i(v_1,\dots,v_r;w_1,\dots,w_n)\,dS,
\end{aligned}
$$

where $v_1,\dots,v_r$ are form arguments (basis functions), and $w_1,\dots,w_n$ are form coefficients. Moreover, dx, ds and dS refer to integration over a domain Ω_k, its boundary (experior facets) $\partial\Omega_k$, or its interior facets Γ_k respectively.

The number r of basis functions represents the arity of the form: For instance, bilinear forms have arity $r = 2$, linear forms have arity $r = 1$. Basis functions are declared by the keyword `BasisFunction` in the UFL.

Note: When defining a multivariate form, the arguments $v_1,\dots,v_r$ are assigned in their order of declaration, not in the order in which they appear in the form. This means that—in a bilinear form—one should always define the test function first.

```
v = BasisFunction("Lagrange", "triangle", 2)
u = BasisFunction("Lagrange", "triangle", 2)
a = inner(grad(v),grad(u))
```

However, `TestFunction` and `TrialFunction` are special instances of `BasisFunction` with the property that a `TestFunction` will always be the first argument in a form and a `TrialFunction` will always be the second argument.

Coefficients are declared by the keyword `Function` or the special cases `Constant`, `VectorConstant` or `TensorConstant` in UFL. In Python, they can also be generated by the `Expression` constructor.

In UFL one can make extensive use of index notation and tensor algebraic concepts, but we don't go into details here. We only mention the following operators which act on tensors of appropriate ranks: `transpose`, `tr` (trace), `dot` (collapse the last axis of the left and the first axis of the right argument), `inner` (collapse all axes), `outer` (outer product), `cross` (defined only for two vectors of length), `det` (determinant), `dev` (deviatoric part of a matrix), `sym` and `skew` (symmetric and skey symmetric parts of a matrix), `cofac` (cofactor of a matrix), `inv` (inverse of a matrix).

Forms defining PDEs need spatial derivatives of basis functions. One may use

```
f = Dx(v, 2) # or equivalently
f = v.dx(2)
```

to denote $\partial v/\partial x_2$ for example. Useful abbreviations exist for often used compound derivatives such as `grad`, `div` and `curl`. Jump and average operators for interior facets are also defined, which are needed to solve problems with discontinuous Galerkin (DG) function spaces.

UFL also supports differentiation with respect to user defined variables (see the demo file `HyperElasticity.ufl` for an example) and differentiation with respect to basis functions. The latter can be used, for instance to automatically generate linearized equations (e.g., for Newton iterations).

A Setup Environment

Currently, FEniCS is installed on host `leibniz.mathematik.tu-chemnitz.de` in the following versions:

- FFC Version 0.6.2
- Dolfin Version 0.9.2

Here is how to setup the environment on `leibniz`:

```
source /LOCAL/Dolfin/setup-variables

# Expand the PATH variable
export PATH=$PATH:/LOCAL/Dolfin/fenics/ffc/scripts:\
/LOCAL/Dolfin/paraview-3.6.1-Linux-x86_64/bin

export D_DEMO=/LOCAL/Dolfin/fenics/dolfin/demo
```

References

[1] M. S. Alnaes, H.-P. Langtangen, A.Logg, K.-A. Mardal, and O. Skavhaug. UFC specification and user manual. Technical report, 2008. http//www.fenics.org/ufc/.

[2] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and L. R. Scott. The FEniCS project. Technical Report 2003–21, Chalmers Finite Element Center Preprint Series, 2003.

[3] H.-P. Langtangen. A FEniCS tutorial. Technical report, 2009. http://www.ima.umn.edu/~arnold/8445.f09/fenics/fenics-tutorial.pdf.

[4] A. Logg. Automating the finite element method. *Archives of Computational Methods in Engineering*, 14(2):93–138, 2007.

[5] A. Logg and G.N. Wells. Dolfin: Automated finite element computing. *submitted*, 2009.

Chemnitz University of Technology, Faculty of Mathematics, D–09107 Chemnitz, Germany

E-mail address: `roland.herzog@mathematik.tu-chemnitz.de`

URL: `http://www.tu-chemnitz.de/herzog`