Parallel workflows for computational science and engineering: FEniCS (draft)
Architecture, interfaces and syntax

Johan Hoffman, Johan Jansson
Rodrigo Vilela de Abreu, Niclas Jansson

December 5, 2013
Aims/Vision of FEniCS

- Research setting:
  - Adaptive general FEM/HPC software for PDE: FEniCS
  - Focus on challenging problems in turbulence/fluid-structure interaction

- Automated computational modeling: weak form of PDE as input + tolerance on output quantity ⇒ automated generation of discretization and solution satisfying tolerance
  - Automated weak form evaluation/tensor assembly
  - Automated duality-based error control/adaptivity
  - Automated modeling: no explicit turbulence model, turbulent dissipation comes only from numerical stabilization
  - Optimal strong scaling up to thousands of cores for full framework on massively parallel hardware
FEniCS development

FEniCS:
fenicsproject.org Open source software project for automated solution of PDE with many involved universities/individuals. (see fenicsproject.org for developer info/papers). Python and C++ programming interface.

FEniCS-HPC:
Branch with focus on good scaling on massively parallel architectures, targeting open/challenging problems in turbulent flow/FSI. Developed mainly by CTL group. Only the C++ programming interface.

fenicsproject.org
ctl.csc.kth.se
The Finite Element Method (notation)

Want to solve differential equation:

\[ R(u) = 0 \text{ or } \]
\[ (R(u), v) = \int_\Omega R(u)v \, dx = 0, \forall v \in V \text{ (weak/variational form)} \]

We seek a solution \( U \) in finite element vector space \( V^h \) defined by the mesh and basis functions \( \phi_j \) of the form:

\[
U(x) = \sum_{j=1}^{M} \xi_j \phi_j(x)
\]

We require the residual to be orthogonal to \( V^h \):

\[
(R(U), v) = 0, \forall v \in V^h
\]

Typical notation for linear problems:

\[
(R(U), v) = a(U, v) - L(v) = 0
\]

Ex. Poisson:

\[
R(u) = \Delta u - f = 0, \quad a(U, v) - L(v) = (\nabla U, \nabla v) - (f, v) = 0
\]
FEniCS main concepts

1. Functions and function spaces
2. Form language (expressions of Functions)
3. Forms (weak formulation of PDE)
4. Assembling tensors (matrix/vector/scalars)
5. Solving linear systems
6. Boundary conditions
7. Automated linearization
Architecture of FEniCS

Component structure:

▶ Automated generation of finite elements/basis functions (FIAT)

\[ e = (K, P, \mathcal{N}) \]

▶ Automated evaluation of variational forms on one cell based on code generation (FFC+UFL)

\[ A^K = a_K(v, U) = \int_K \nabla v \cdot \nabla U \, dx \]

▶ Automated assembly of discrete systems on a mesh \( T_\Omega \) (DOLFIN-HPC)

\[ A = 0 \]
\[ for \ all \ elements \ K \in T_\Omega \]
\[ A += A^K \]

▶ Automated turbulent Unified Continuum modeling (Unicorn)

\[ r_{UC}(v, W) = (v, \rho(\partial_t u + (u \cdot \nabla)u) + \nabla \cdot \sigma - g) + SD(v, W) \]
FEniCS form compilation/code generation

- Automates a key step in the implementation of finite element methods for partial differential equations
- Input: a variational form and a finite element
- Output: C/C++ function for element tensor

Input form in (ASCII) mathematical notation:

\[ a(v, u) = \int_{\Omega} \nabla u(x) \cdot \nabla v(x) \, dx \]

Compiler:

\[
\text{>> ffc [-l language]} \, \text{poisson.form}
\]
virtual void tabulate_tensor(double∗ A, const double∗ const w,  
const ufc::cell& c) const  
{
    // Quadrature weight
    const static double W0 = 0.5;

    // Tabulated basis functions and arrays of non-zero columns
    const static double Psi_w[1][3] = 
        {{0.33333333333, 0.33333333333, 0.33333333333}};
    static const unsigned int nzc0[2] = {0, 1};
    static const unsigned int nzc1[2] = {0, 2};

    // Geometry constants
    const double G0 = Jinv_00 * Jinv_10 * W0 * det;
    const double G1 = Jinv_00 * Jinv_11 * W0 * det;
    const double G2 = Jinv_01 * Jinv_00 * W0 * det;
    const double G3 = Jinv_01 * Jinv_10 * W0 * det;
    const double G4 = Jinv_10 * Jinv_00 * W0 * det;
    const double G5 = Jinv_11 * Jinv_00 * W0 * det;

    // Loop integration points
    for (unsigned int ip = 0; ip < 1; ip++)  
    {
        // Compute function value
        double F0 = 0;
        for (unsigned int r = 0; r < 3; r++)
        {
            F0 += Psi_w[ip][r] * w[0][r];
        }

        const double Gip0 = (G0 + G1) * F0;
        const double Gip1 = (G2 + G3) * F0;
        const double Gip2 = (G4 + G5) * F0;
        for (unsigned int i = 0; i < 2; i++)  
        {
            for (unsigned int j = 0; j < 2; j++)  
            {
                A[nzc0[i]*3 + nzc0[j]] += Psi_vu[ip][i] * Psi_vu[ip][j] * Gip1;
                A[nzc0[i]*3 + nzc1[j]] += Psi_vu[ip][i] * Psi_vu[ip][j] * Gip0;
                A[nzc1[i]*3 + nzc0[j]] += Psi_vu[ip][i] * Psi_vu[ip][j] * Gip0;
            }
        }
    }
Stokes equation: form language

\[-\Delta u + \nabla p = f\]
\[\nabla \cdot u = 0\]

```python
# Create mixed space (Taylor-Hood)
V = VectorElement('Lagrange', 'tetrahedron', 2)
Q = FiniteElement('Lagrange', 'tetrahedron', 1)
TH = V * Q

# Create trial and test functions
(u, p) = TrialFunctions(TH)
(v, q) = TestFunctions(TH)

# Coefficient function appearing in L
f = Coefficient(V)

# Define forms
a = inner(grad(u), grad(v))*dx - p*div(v)*dx + div(u)*q*dx
L = dot(f, v)*dx

# Compute solution w = (u, p) with BCs given in 'bcs'
w = Function(TH)
solve(a == L, w, bcs)
```
Functions and function spaces (Python)

Import the DOLFIN Python interface:

```python
from dolfin import *
```

Load mesh and create cG(1) function space (continuous piecewise linear functions):

```python
mesh = UnitSquare(5, 5)
V = FunctionSpace(mesh, "CG", 1)
```

Create a function $U \in V$ and inspect the vector of coefficients $\xi$:

```python
U = Function(V)
print U.vector().array()
```

Alternatively load a mesh from file:

```python
mesh = Mesh("mesh.xml")
```
Represent the known coefficient \( f(x) = \cos(3) \times x_0 \) with \( x = [x_0, x_1] \) in 2D.

\[
f = \text{Expression}("\cos(3)\times x[0] ")
\]
Function expressions

Differential and algebraic operators in text-version of mathematical notation (example: $q = \nabla U \cdot \nabla U - fU$):

$q = \text{inner} (\text{grad}(U), \text{grad}(U)) - f*U$

See the UFL manual for full syntax.
Form notation (weak formulation of PDE)

Recall notation of inner product:
\[ M = \| \nabla U \|^2 = (\nabla U, \nabla U) = \int_{\Omega} \nabla U \cdot \nabla U \, dx \]

In FEniCS:
\[ M = \text{inner}(\text{grad}(U), \text{grad}(U)) \star dx \]

Also possible to express boundary terms, here \( M = \int_{\Gamma} U ds \):
\[ M = U \star ds \]
Recall how we defined the weak form of a PDE:
\[ a(U, v) = L(v), \forall v \in V_h \]

For Poisson’s equation for example:
\[ (\nabla U, \nabla v) = (f, v), \forall v \in V_h \]

Introduce two special functions:

\[ U = \text{TrialFunction}(V) \] # Unknown function we seek
\[ v = \text{TestFunction}(V) \] # All functions in $V$

We can now express the above bilinear form $a(U, v)$ and linear form $L(v)$ like so:

\[
\begin{align*}
a &= \text{inner}(\text{grad}(U), \text{grad}(v)) \cdot dx \\
L &= f \cdot v \cdot dx
\end{align*}
\]
Assembling tensors (matrix/vector/scalar)

A bilinear form \( a(U, v) \) corresponds to a matrix \( A_{ij} = a(\phi_i, \phi_j) \).
A linear form \( L(v) \) corresponds to a vector \( b_i = L(\phi_i) \).
A functional \( M \) corresponds to a scalar \( q = M \).

We can assemble a form into the corresponding tensor by:

\[
\begin{align*}
A &= \text{assemble}(a) \\
b &= \text{assemble}(L) \\
q &= \text{assemble}(M)
\end{align*}
\]

For example, assembling the \( L_2 \)-norm of a function: \( \| f \| = \sqrt{(f, f)} \)

\[
\begin{align*}
M &= \text{inner}(f, f) \cdot dx \\
f_{L2norm} &= \text{assemble}(M) \\
f_{L2norm} &= \sqrt{f_{L2norm}}
\end{align*}
\]
Solving linear systems

FEniCS has a high-level interface to state-of-the-art linear algebra packages (PETSc, Trilinos, etc.)

To solve a linear system with the default linear solver:

\[ x = \text{Vector}() \]
\[ \text{solve}(A, x, b) \]

We can also use the compact solve interface for the PDE:

\[ U = \text{Function}(V) \]
\[ \text{solve}(a \equiv L, U) \]

with the possibility of choosing different solvers and options:

\[
\text{solve}(a \equiv L, U, \text{solver_parameters}=
\{\text{\{'linear_solver': 'bicgstab',}
\text{\{'preconditioner': 'ilu'}\})
\]
Boundary conditions...
from dolfin import *

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

# Define Dirichlet boundary (x = 0 or x = 1)
def boundary(x):
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS

# Define boundary condition
u0 = Constant(0.0)
bcs = DirichletBC(V, u0, boundary)

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
f = Expression("10*exp(-pow(x[0]-0.5, 2) + pow(x[1]-0.5, 2))/0.02")
g = Expression("sin(5*x[0])")
a = inner(grad(v), grad(u))*dx
L = v*f*dx - v*g*ds

# Compute solution (assemble matrix/vector, solve linear system)
U = Function(Vh)
solve(a == L, U, bcs=bcs)

# Plot solution
plot(u, interactive=True)
Automated linearization

...
Hands-on demonstration

Complete goal-oriented adaptive solver
Exercise

Worksheet: Adaptive stabilized Navier-Stokes