Current and Future Plans for FEniCS

Anders Logg
logg@simula.no

Simula Research Laboratory

BIT Circus Stockholm, August 31 - September 1 2006

Outline

The FEniCS Project

- Initiated in 2003
- Develop free software for the Automation of CMM
- An international project with collaborators from Simula Research Laboratory, KTH, Chalmers, Delft University of Technology, Texas Tech, University of Chicago, and Argonne National Laboratory

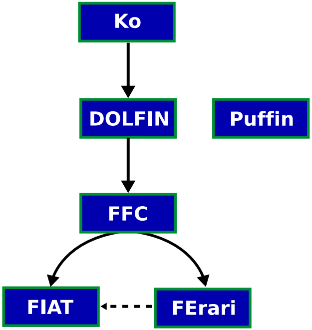
The Automation of CMM:

(i) The automation of discretization (done)
(ii) The automation of discrete solution
(iii) The automation of error control
(iv) The automation of modeling
(v) The automation of optimization

Key Features

- Simple and intuitive object-oriented API, C++ or Python
- Automatic and efficient evaluation of variational forms
- Automatic and efficient assembly of linear systems
- General families of finite elements, including arbitrary order continuous and discontinuous Lagrange elements
- Arbitrary mixed elements may be defined
- High-performance parallel linear algebra
- Triangular and tetrahedral meshes, adaptive mesh refinement
- Multi-adaptive mcG(q)/mdG(q) and mono-adaptive cG(q)/dG(q) ODE solvers
- Support for a range of output formats for post-processing, including DOLFIN XML, ParaView/Mayavi/VTK, OpenDX, Tecplot, Octave, MATLAB, GiD

Components



- **DOLFIN** is the C++/Python interface of FEniCS
- **FIAT** is the finite element backend of FEniCS
- **FFC** is a just-in-time compiler for variational forms
- **FErari** functions as an optimizing backend for FFC
- **Ko** is a special-purpose interface for simulation of mechanical systems
- **Puffin** is a light-weight version of FEniCS for Octave/MATLAB

Poisson's Equation

Find $U \in V_h$ such that $a(v, U) = L(v)$ for all $v \in \hat{V}_h$, where

$$
\begin{aligned}
a(v, U) &= \int_\Omega \nabla v \cdot \nabla U \,\mathrm{d}x \\
L(v) &= \int_\Omega v f \,\mathrm{d}x
\end{aligned}
$$

```
element = FiniteElement("Lagrange", ...)

v = TestFunction(element)
U = TrialFunction(element)
f = Function(element)

a = dot(grad(v), grad(U))*dx
L = v*f*dx
```

The Stokes equations

Differential equation:

$$\begin{aligned} -\Delta u + \nabla p &= f && \text{in } \Omega \\ \nabla \cdot u &= 0 && \text{in } \Omega \\ u &= u_0 && \text{on } \partial\Omega \end{aligned}$$

- Velocity $u = u(x)$
- Pressure $p = p(x)$

Stokes with Taylor–Hood elements

Find $(U, P) \in V_h = V_h^u \times V_h^p$ such that

$$\int_\Omega \nabla v : \nabla U - (\nabla \cdot v) P + q \nabla \cdot U \, \mathrm{d}x = \int_\Omega v \cdot f \, \mathrm{d}x$$

for all $(v, q) \in \hat{V}_h = \hat{V}_h^u \times \hat{V}_h^p$

- Approximating spaces $\hat{V}_h$ and V_h must satisfy the Babuška–Brezzi inf–sup condition
- Use Taylor–Hood elements:
  - P_q for velocity
  - P_{q-1} for pressure

Implementation

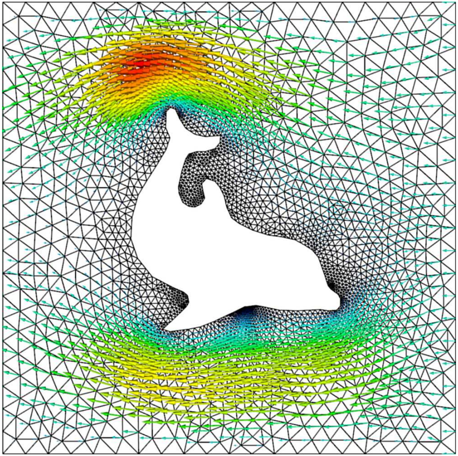
```
P2 = FiniteElement("Vector Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)
TH = P2 + P1

(v, q) = TestFunctions(TH)
(U, P) = TrialFunctions(TH)

f = Function(P2)

a = (dot(grad(v), grad(U)) - div(v)*P + q*div(U))*dx
L = dot(v, f)*dx
```

Solution (velocity field)



Stabilization

- Circumvent the Babuška–Brezzi condition by adding a stabilization term
- Modify the test function according to

$$(v, q) \rightarrow (v, q) + (\delta \nabla q, 0)$$

with $\delta = \beta h^2$

Find $(U, P) \in V_h = V_h^u \times V_h^p$ such that

$$\int_\Omega \nabla v : \nabla U - (\nabla \cdot v) P + q \nabla \cdot U + \delta \nabla q \cdot \nabla P \, \mathrm{d}x = \int_\Omega (v + \delta \nabla q) \cdot f \, \mathrm{d}x$$

for all $(v, q) \in \hat{V}_h = \hat{V}_h^u \times \hat{V}_h^q$

Implementation

```
vector = FiniteElement("Vector Lagrange", "triangle", 1)
scalar = FiniteElement("Lagrange", "triangle", 1)
system = vector + scalar

(v, q) = TestFunctions(system)
(U, P) = TrialFunctions(system)

f = Function(vector)
h = Function(scalar)

d = 0.2*h*h

a = (dot(grad(v), grad(U)) - div(v)*P + q*div(U) + \
     d*dot(grad(q), grad(P)))*dx
L = dot(v + mult(d, grad(q)), f)*dx
```

Benchmarks

- Measure CPU time for the evaluation of the element tensor (the "element stiffness matrix")
- Code automatically generated by the form compiler FFC
- Compute speedup compared to a standard quadrature-based approach with loops over quadrature points

Form	$q=1$	$q=2$	$q=3$	$q=4$	$q=5$	$q=6$	$q=7$	$q=8$
Mass 2D	12	31	50	78	108	147	183	232
Mass 3D	21	81	189	355	616	881	1442	1475
Poisson 2D	8	29	56	86	129	144	189	236
Poisson 3D	9	56	143	259	427	341	285	356
Navier–Stokes 2D	32	33	53	37	—	—	—	—
Navier–Stokes 3D	77	100	61	42	—	—	—	—
Elasticity 2D	10	43	67	97	—	—	—	—
Elasticity 3D	14	87	103	134	—	—	—	—

Compiling Poisson's equation: non-optimized, 16 ops

```
void eval(real block[], const AffineMap& map) const
{
  [...]

  block[0] = 0.5*G0_0_0 + 0.5*G0_0_1 +
             0.5*G0_1_0 + 0.5*G0_1_1;
  block[1] = -0.5*G0_0_0 - 0.5*G0_1_0;
  block[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  block[3] = -0.5*G0_0_0 - 0.5*G0_0_1;
  block[4] = 0.5*G0_0_0;
  block[5] = 0.5*G0_0_1;
  block[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  block[7] = 0.5*G0_1_0;
  block[8] = 0.5*G0_1_1;
}
```

Compiling Poisson's equation: ffc -O, 11 ops

```
void eval(real block[], const AffineMap& map) const
{
  [...]

  block[1] = -0.5*G0_0_0 + -0.5*G0_1_0;
  block[0] = -block[1] + 0.5*G0_0_1 + 0.5*G0_1_1;
  block[7] = -block[1] + -0.5*G0_0_0;
  block[6] = -block[7] + -0.5*G0_1_1;
  block[8] = -block[6] + -0.5*G0_1_0;
  block[2] = -block[8] + -0.5*G0_0_1;
  block[5] = -block[2] + -0.5*G0_1_1;
  block[3] = -block[5] + -0.5*G0_0_0;
  block[4] = -block[1] + -0.5*G0_1_0;
}
```

Compiling Poisson's equation: `ffc -f blas`, 36 ops

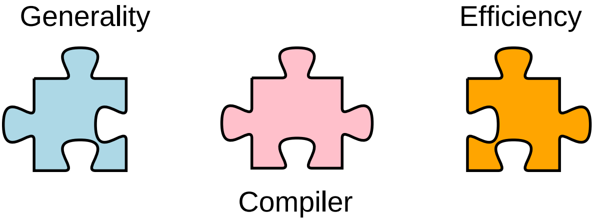
```
void eval(real block[], const AffineMap& map) const
{
  [...]

  cblas_dgemv(CblasRowMajor, CblasNoTrans,
              blas.mi, blas.ni, 1.0,
              blas.Ai, blas.ni, blas.Gi,
              1, 0.0, block, 1);
}
```

The compiler approach

- Any form
- Any element
- Maximum efficiency

Possible to combine generality with efficiency by using a compiler approach:



Recent updates (DOLFIN 0.6.2 / FFC 0.3.3)

- Release of DOLFIN 0.6.2 and FFC 0.3.3 (any day now)
- Improved linear algebra supporting PETSc and uBlas
- FErari optimization in FFC
- Much improved ODE solvers
- Boundary integrals
- PyDOLFIN, the Python interface of DOLFIN
- Bugzilla database
- Improved manual, compiler support, demos, matrix factory, file formats, ...

Coming updates (DOLFIN 0.6.3)

- A new mesh library!

Linear algebra backends

- Complete support for PETSc
  - High-performance parallel linear algebra
  - Krylov solvers, preconditioners
- Complete support for uBlas
  - BLAS level 1, 2 and 3
  - Dense, packed and sparse matrices
  - C++ operator overloading and expression templates
  - Krylov solvers, preconditioners added by DOLFIN
- Uniform interface to both linear algebra backends
- LU factorization by UMFPACK for uBlas matrix types
- Eigenvalue problems solved by SLEPc for PETSc matrix types
- Matrix-free solvers ("virtual matrices")

Matrices and vectors

```
Matrix A(M, N);
Vector x(N);

A(5, 5) = 1.0;
x(3) = 2.0;
```

- Default data types: `Matrix`, `Vector`
- Additional data types: `SparseMatrix`, `DenseMatrix`, `PETScMatrix`, `uBlasMatrix`
- Common interface: `GenericMatrix`, `GenericVector`

Solving linear systems (simple)

Direct solution by LU factorization:

```
LU::solve(A, x, b);
```

Iterative solution by ILU-preconditioned GMRES:

```
GMRES::solve(A, x, b);
```

Solving linear systems (contd.)

Specify Krylov method and preconditioner:

```
KrylovSolver solver(gmres, ilu);
solver.solve(A, x, b);
```

- Krylov methods: `cg`, `gmres`, `bicgstab`
- Preconditioners: `jacobi`, `sor`, `ilu`, `icc`, `amg`

Key features

- Dimension-independent interface
- Efficient (close to optimal) storage
- Automatic computation of connectivity
- Parallel

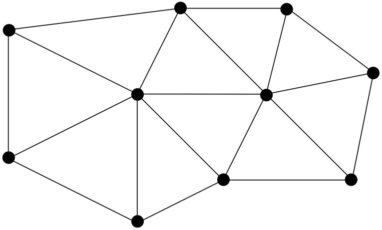
Benchmarks

Initial results for some random mesh:

Task	Old mesh	New mesh
Reading and initializing 1000 times	0.9 s	0.21 s
Refining mesh uniformly 8 times	27.2 s	2.14 s
Iterating over connectivity 100 times	18.2 s	1.86 s
Memory usage	281 MB	43 MB

Mesh abstractions

- Mesh = (Topology, Geometry)
- Topology = ({ Mesh entities }, Connectivity)
- Mesh entity = (dim, index)
- Connectivity = { Incidence relations $d - d'$ }



Mesh entities

Entity	Dimension	Codimension
Vertex	0	–
Edge	1	–
Face	2	–
Facet	–	1
Cell	–	0

- Mesh entity defined by (dim, index)
- Named mesh entities: `Vertex`, `Edge`, `Face`, `Facet`, `Cell`

Mesh iterators

Basic iteration:

```
Mesh mesh;
for (MeshEntityIterator e(mesh, d); !e.end(); ++e)
  for (MeshEntityIterator f(e, 0); !f.end(); ++f)
    f->foo();
```

Iteration with named iterators:

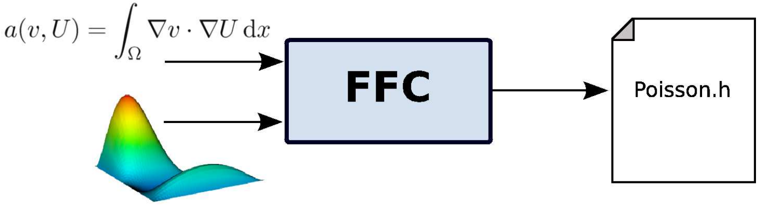
```
for (CellIterator c(mesh); !c.end(); ++c)
  for (VertexIterator v(c); !v.end(); ++v)
    v->foo();
```

Highlights

- UFL/UFC
- Automation of error control
  - Automatic generation of dual problems
  - Automatic generation of a posteriori error estimates
- Discontinuous Galerkin methods
- Mesh algorithms
  - Adaptive mesh refinement
  - Mesh algorithms for ALE methods
- Improved geometry support
- Finite element exterior calculus

A common framework

- UFL - Unified Form Language
- UFC - Unified Form-assembly Code
- Unify, standardize, extend
- Working prototypes: FFC (Logg), SyFi (Mardal)



FEniCS'06 in Delft November 8–9

http://www.fenics.org/