The FEniCS Project
Philosophy, current status and future plans

Anders Logg
logg@simula.no

Simula Research Laboratory

FEniCS'06 in Delft, November 8-9 2006

Outline

Philosophy

Current status
- Overview
- Examples
- Efficiency

Future Plans
- Recent updates
- Future plans
- New projects?

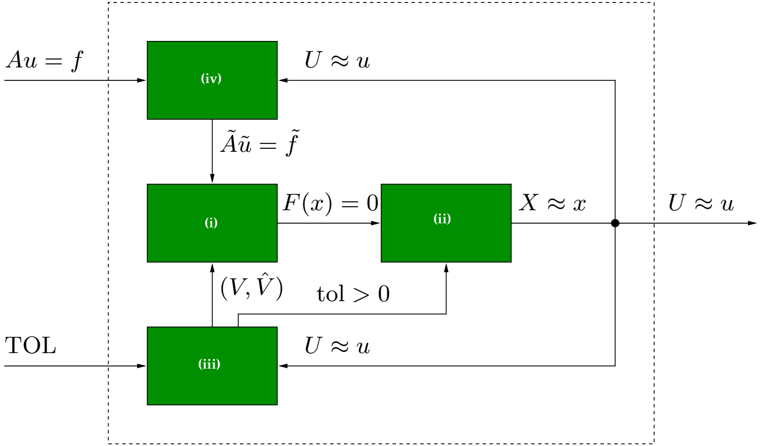
The FEniCS Project

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

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

Automation of CMM



Automating the finite element method

FEniCS automates (important aspects of) the finite element method:

- Automatic generation of finite elements (FIAT)

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

- Automatic evaluation of variational forms (FFC)

$$a(v, U) = \int_{\Omega} \nabla v \cdot \nabla U \, dx$$

- Automatic assembly of linear systems (DOLFIN)

$$\texttt{for all cells } K \in \mathcal{T}_{\Omega}\texttt{:} \ A \texttt{ += } A^K$$

Basic principles

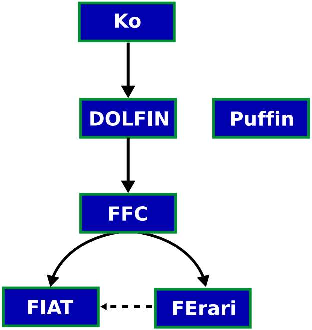
Basic principles:

- Generality (automation)
- Efficiency
- Simplicity
  - Methodology
  - Implementation
  - User interfaces
- Applications

Realization:

- Organized as a collection reusable components
- A rapid and open development process
- Modern programming techniques
- Novel algorithms

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

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
- High-performance parallel linear algebra
- General 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, Octave, MATLAB, GiD

Linear algebra

- 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")

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

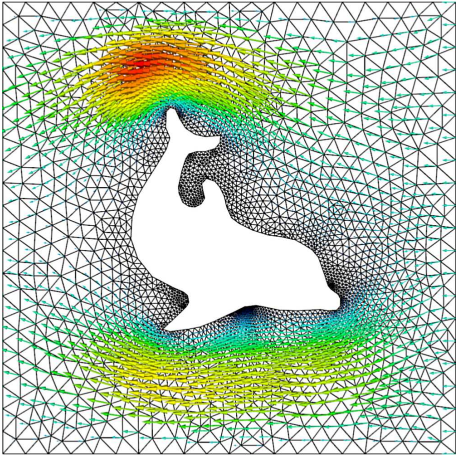
```
P2 = FiniteElement("Vector Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)
TH = P2 + P1

(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

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

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

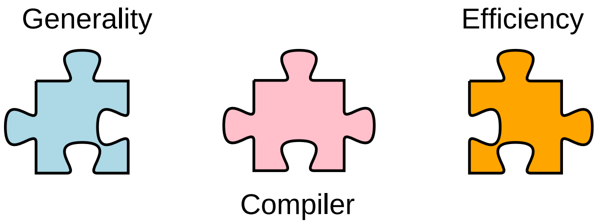
```
void eval(real block[], const AffineMap& map) const
{
  [...]

  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 0.6.3 / FFC 0.3.4)

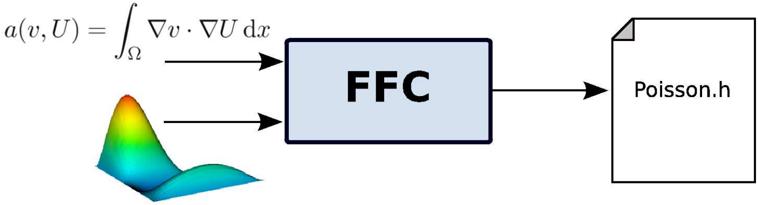
- Improved linear algebra supporting PETSc and uBlas
- A new improved mesh library
- FErari optimizations in FFC
- Evaluation of functionals
- Much improved ODE solvers
- Boundary integrals
- PyDOLFIN, the Python interface of DOLFIN
- Bugzilla database, pkg-config
- Improved manual, compiler support, demos, matrix factory, file formats, . . .

Highlights

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

A common framework: UFL/UFC

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



New FEniCS projects?

- UFC
- UFL

- Famms
- Instant
- PySE
- Swiginac
- SyFi

Famms: Automated code verification by MMS

Author: O. Skavhaug

```
from Famms import *
from Symbolic import *

f = Famms(nspacedim=2); (x, y) = f.x
v1 = sin(x); v2 = cos(y)
v = Vector((x,y), (v1,v2))
Lambda = 120; mu = 3

def F(u):
    return grad((Lambda+mu)*div(u)) + div(mu*grad(u))

f.assign(equation=F, solution=v, simulator=my_solver)
```

Instant: Inlining C/C++ in Python

Authors: M. Westlie and K.-A. Mardal

```
import Instant

code = 'double sum(int a, int b) { return a+b; }'

ext = Instant.Instant()
ext.create_extension(code=code, module='my_module')

from my_module import sum
print sum(3, 5)
```

PySE: Parallel FD in Python

Author: Å. Ödegård

```
from pyFDM import *

g = Grid(domain=([0,1], [0,1]), division=(100, 100))
u = Field(g)
t = 0
dt = T/n;
stencil = Identity(g.nsd) + dt*Laplace(g)
```

Swiginac: Symbolic mathematics in Python

Authors: O. Skavhaug, O. Certik

```
from swiginac import *

x = symbol('x')
y = symbol('y')

f = sin(x*x*y)
f.printc()

g = diff(f, x)
dfdx.printc()
```

SyFi: Symbolic FEM in Python

Author: K.-A. Mardal

```
from swiginac import *
from SyFi import *

triangle = ReferenceTriangle()
fe = LagrangeFE(triangle,3)

for i in range(0, fe.nbf()):
  for j in range(0, fe.nbf()):
    integrand = inner(grad(fe.N(i)),grad(fe.N(j)))
    Aij = triangle.integrate(integrand)
```