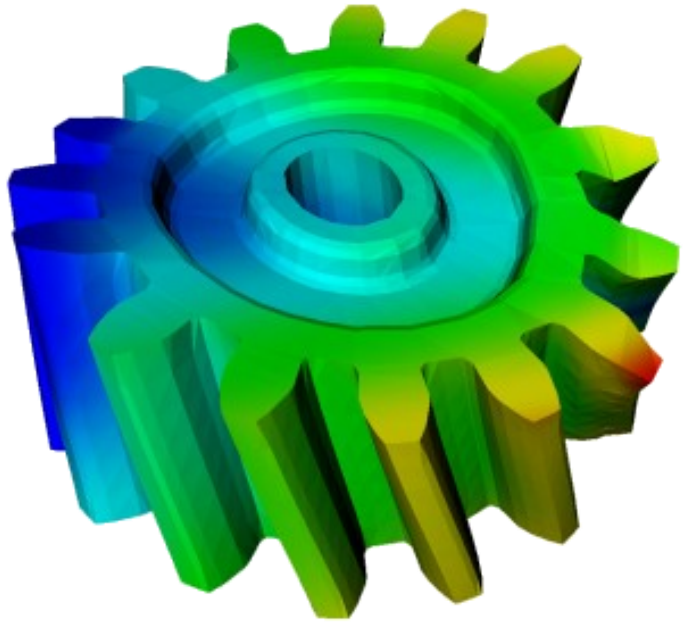
Automatic Code Generation and the FEniCS Project

Acknowledgments: Martin Sandve Alnæs, Johan Hake, Johan Hoffman, Johan Jansson, Claes Johnson, Dmitry Karpeev, Robert C. Kirby, Matthew G. Knepley, Hans Petter Langtangen, Kent-Andre Mardal, Kristian Oelgaard, Marie Rognes, L. Ridgway Scott, Ola Skavhaug, Andy R. Terrel, Garth N. Wells, Åsmund Ödegård, and Magnus Vikstrøm.



Anders Logg

Center for Biomedical Computing
Simula Research Laboratory

Department of Informatics
University of Oslo

Opportunities and Challenges in Computational Geodynamics
Caltech, March 30-31 2009

The FEniCS Project

Free software for the Automation of CMM

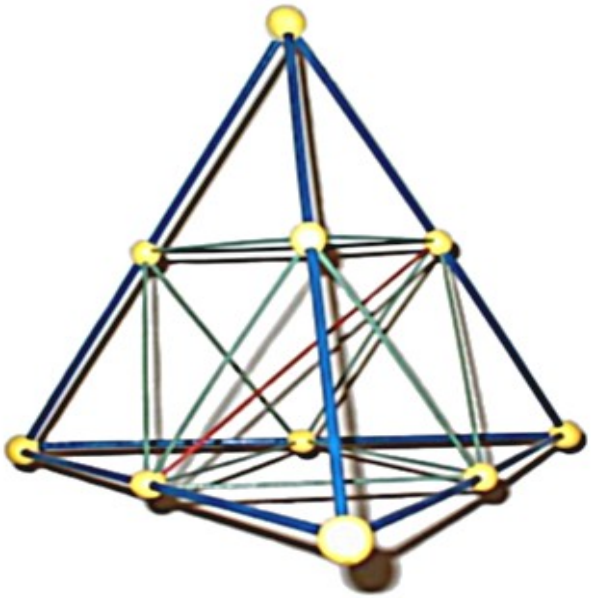
- Started 2003
- 12+ projects/components
- 15-20 active developers
- 1000+ monthly downloads
- 500+ monthly posts to mailing lists
- 10,000+ unique visitors each month
- 3500 changesets in 2008 (10 / day)

Project Team

- Chalmers University of Technology
- University of Chicago
- Argonne National Laboratory (ANL)
- Toyota Technological Institute at Chicago (TTI)
- Delft University of Technology
- Royal Institute of Technology (KTH)
- Simula Research Laboratory
- Texas Tech
- University of Cambridge
- Others

Main Features

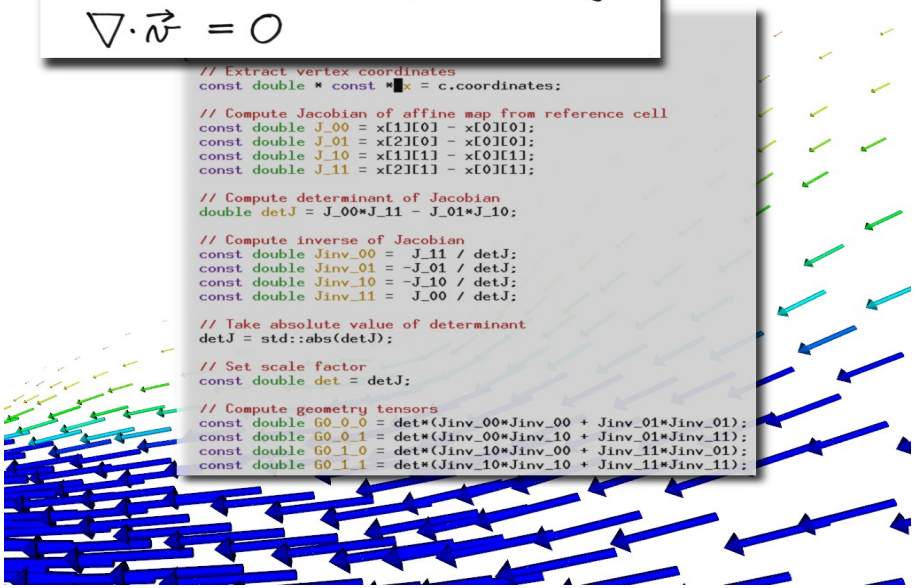
- **Generality**
  - **“Any” equation (expressive form language)**
  - **“Any” finite element method (CG, DG, BDM, Nedelec, ...)**
  - **“Any” mesh (simplex meshes in 1D, 2D, 3D)**
  - **“Any” linear algebra backend (PETSc, Epetra, uBLAS, MTL4)**
- **Efficiency**
  - **High performance assembly**
  - **High performance linear algebra**
  - **Automated domain-specific optimization**



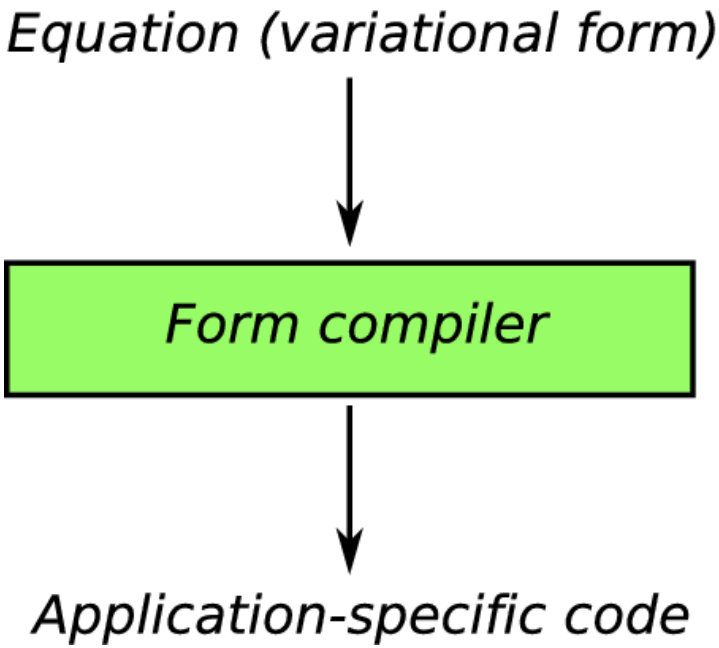
Automatic Code Generation

- Combining generality and efficiency
- Input: equation
- Output: efficient application-specific code

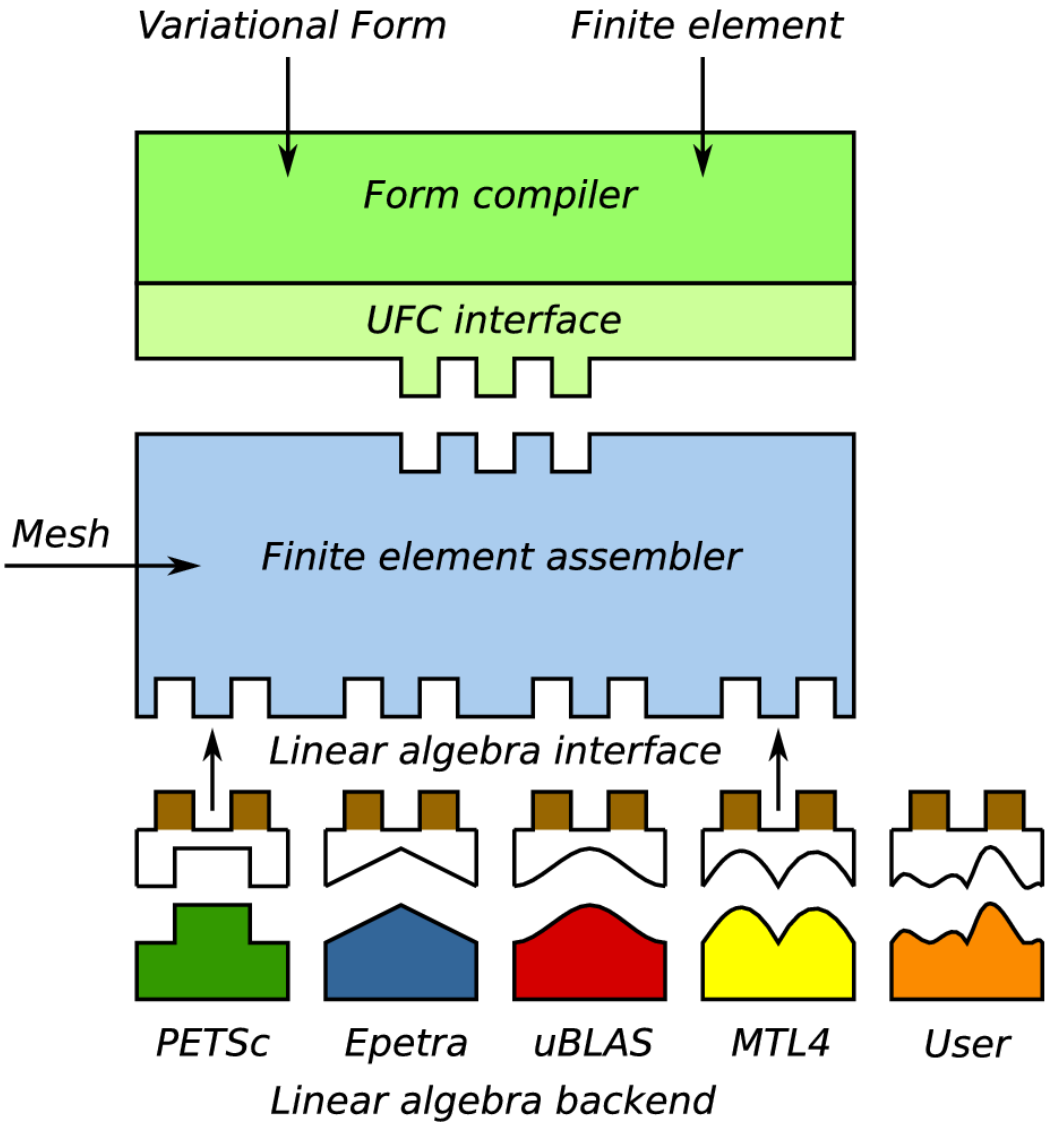
$$\rho\left(\frac{\partial \vec{u}}{\partial t} + \vec{u}\cdot\nabla\vec{u}\right) = -\nabla p + \mu\nabla^2\vec{v} + \rho\vec{b}$$
$$\nabla\cdot\vec{v} = 0$$



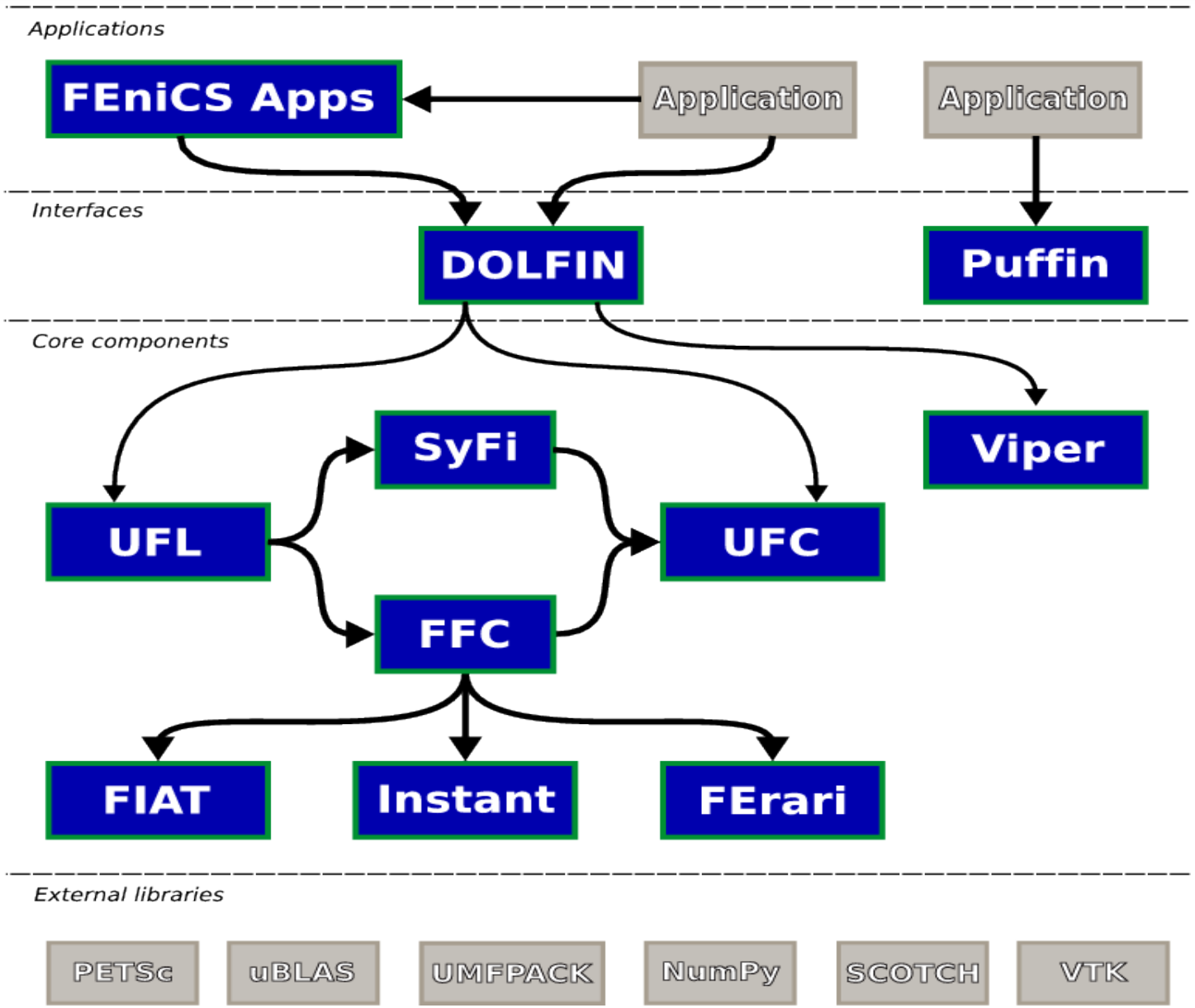
```
// Extract vertex coordinates
const double * const *x = c.coordinates;

// Compute Jacobian of affine map from reference cell
const double J_00 = x[1][0] - x[0][0];
const double J_01 = x[2][0] - x[0][0];
const double J_10 = x[1][1] - x[0][1];
const double J_11 = x[2][1] - x[0][1];

// Compute determinant of Jacobian
double detJ = J_00*J_11 - J_01*J_10;

// Compute inverse of Jacobian
const double Jinv_00 =  J_11 / detJ;
const double Jinv_01 = -J_01 / detJ;
const double Jinv_10 = -J_10 / detJ;
const double Jinv_11 =  J_00 / detJ;

// Take absolute value of determinant
detJ = std::abs(detJ);

// Set scale factor
const double det = detJ;

// Compute geometry tensors
const double G0_0_0 = det*(Jinv_00*Jinv_00 + Jinv_01*Jinv_01);
const double G0_0_1 = det*(Jinv_00*Jinv_10 + Jinv_01*Jinv_11);
const double G0_1_0 = det*(Jinv_10*Jinv_00 + Jinv_11*Jinv_01);
const double G0_1_1 = det*(Jinv_10*Jinv_10 + Jinv_11*Jinv_11);
```



System Design



Software Components



Expressive Form Language

Mixed finite element methods for linear elasticity with weakly imposed symmetry, Arnold, Falk, Winther (2007)

```
S = VectorElement('BDM', triangle, k)
V = VectorElement('Discontinuous Lagrange', triangle, k - 1)
Q = TensorElement('Discontinuous Lagrange', triangle, k - 1)

element = MixedElement(S, V, Q)

(tau, v, eta) = TestFunctions(element)
(sigma, u, gamma) = TrialFunctions(element)

a = (sigma, tau) + (-tr(sigma), tr(tau)) \
  + (div(tau), u) + (-div(sigma), v) \
  + (tau, gamma) + (eta, sigma)
```

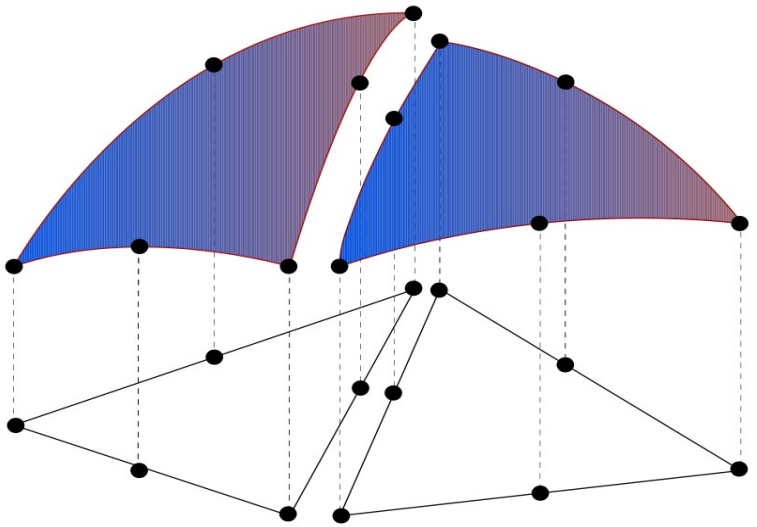
$$a((\tau, v, \eta), (\sigma, u, \gamma)) = (\sigma, \tau) - (\operatorname{tr}\sigma, \operatorname{tr}\tau)$$

$$+(\operatorname{div}\tau, u) - (\operatorname{div}\sigma, v) + (\tau, \gamma) + (\eta, \sigma)$$

Expressive Form Language

Discontinuous Galerkin formulation for Poisson (interior penalty method)

```
element = FiniteElement('Discontinuous Lagrange', triangle, 1)

v = TestFunction(element)
u = TrialFunction(element)

n = FacetNormal(triangle)
h = MeshSize(triangle)

alpha = 4.0
gamma = 8.0

a = (grad(v), grad(u)) \
  + (-avg(grad(v)), jump(u, n), dS) \
  + (-jump(v, n), avg(grad(u)), dS) \
  + (alpha/h('+')*jump(v, n), jump(u, n), dS) \
  + (-grad(v), u*n, ds) \
  + (-v*n, grad(u), ds) \
  + (gamma/h*v, u, ds)
```



Interfaces

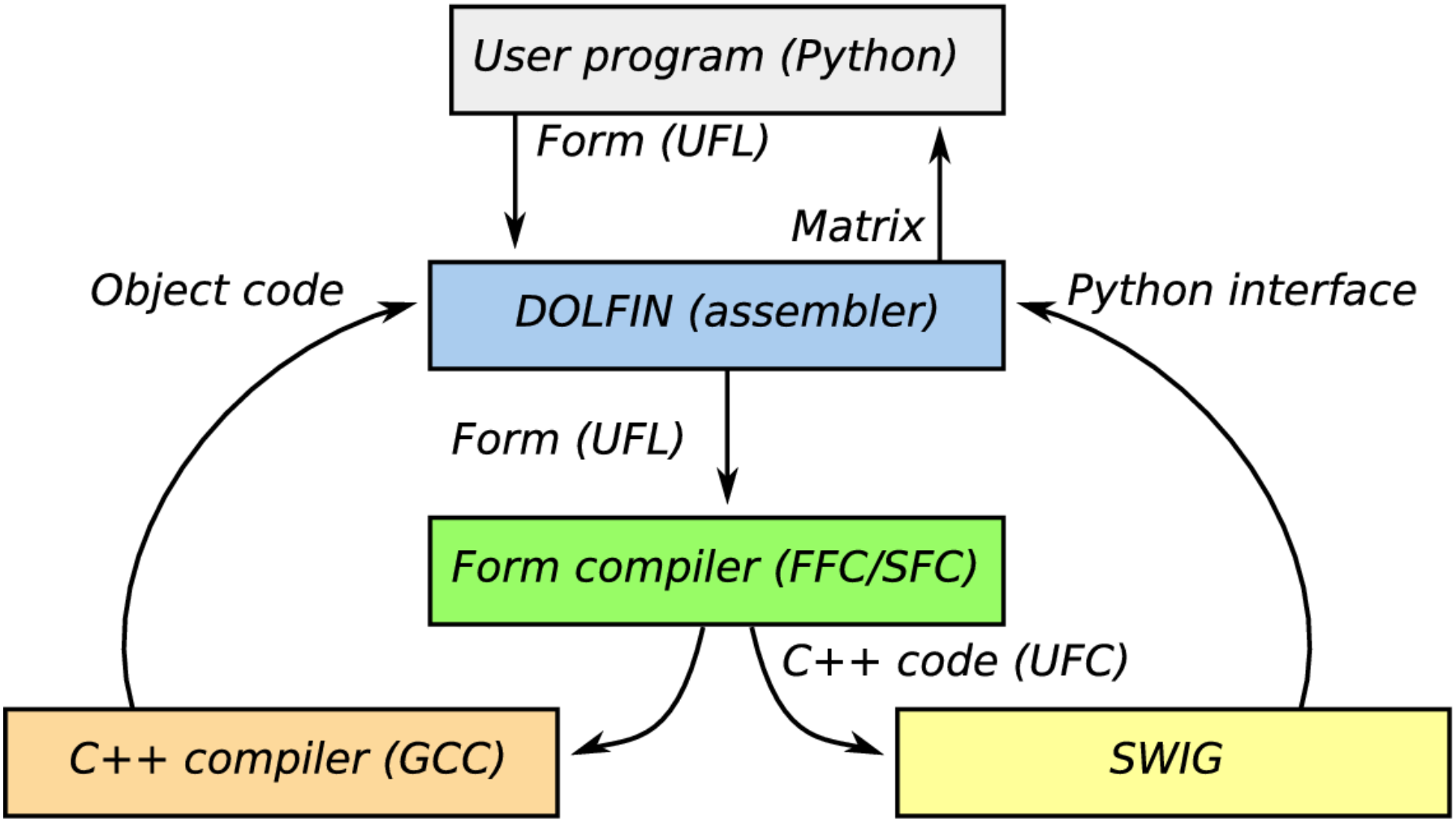
- C++ interface
  - C++ class library (`Matrix, Vector, Mesh, Function, ...`)
  - Forms expressed in separate form files (`.ufl`)
  - Call form compiler on command line (`ffc Poisson.ufl`)
  - Include generated code (`#include ''Poisson.h''`)

- Python interface
  - Python class library (`Matrix, Vector, Mesh, Function, ...`)
  - Integrated form language:

```
A = assemble(dot(grad(v), grad(u)))
```

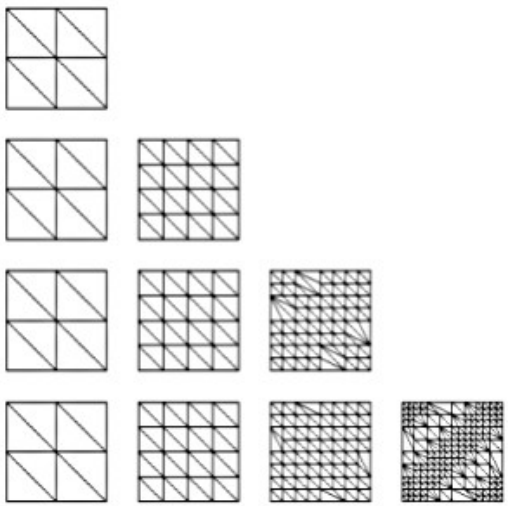
  - Automated code generation
  - Dynamic loading of generated code (JIT compilation)

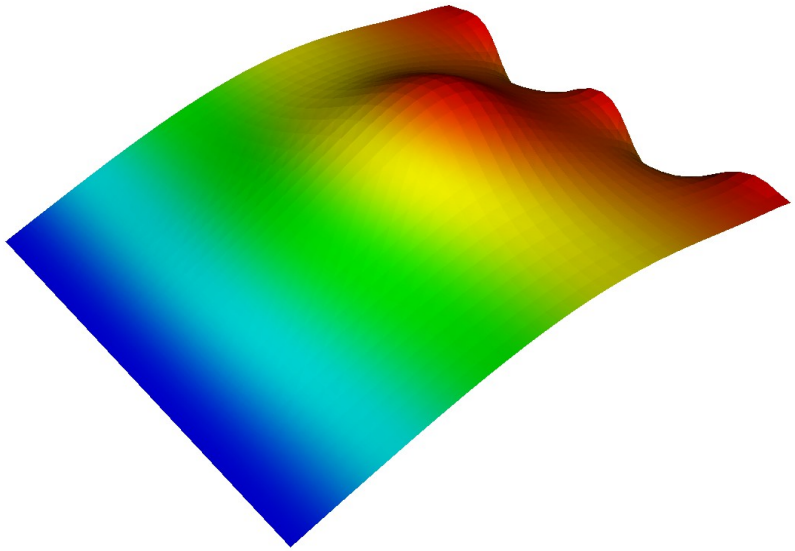
Just-In-Time Compilation (JIT)

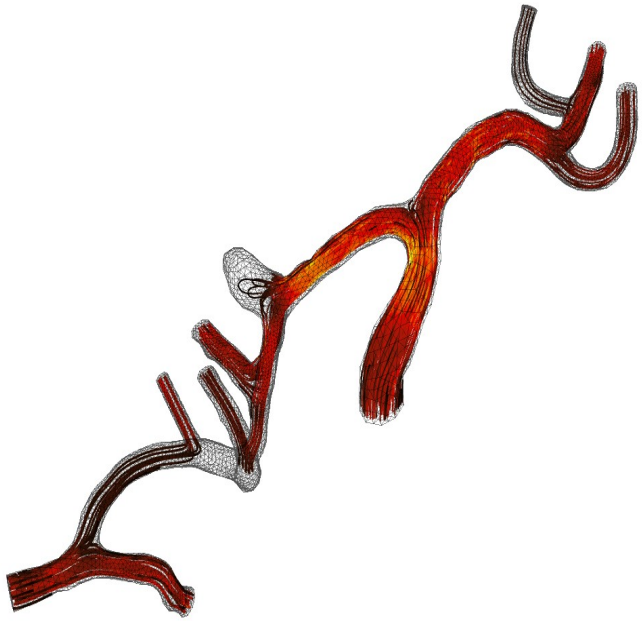


Examples

- Matrix assembly
- Mesh refinement
- Solving Poisson's equation
- User-defined operators
- Solving the Navier-Stokes equations







Matrix Assembly

```
mesh = UnitCube(16, 16, 16)
V = FunctionSpace(mesh, 'CG', 5)

v = TestFunction(V)
u = TrialFunction(V)
f = Function(V, 'sin(x[0]*x[1]*x[2])')
a = (grad(v), grad(u))
L = (v, f)

A = assemble(a)
b = assemble(L)
```


Mesh Refinement

```
mesh = Mesh('mesh.xml')

cell_markers = MeshFunction('bool', mesh, 3)
for cell in cells(mesh):
    if condition:
        cell_markers.set(cell, True)

mesh.refine(cell_markers)
mesh.smooth()
plot(mesh)
```

Solving Poisson's Equation (I)

```
a = (grad(v), grad(u))
L = (v, f)

A = assemble(a)
b = assemble(L)
bc.apply(A, b)

u = Function(V)
solve(A, u.vector(), b)

# solve(A, u.vector(), b, cg, amg_hypre)
```

Solving Poisson's Equation (II)

```
a = (grad(v), grad(u))
L = (v, f)

problem = VariationalProblem(a, L, bc)
u = problem.solve()

plot(u)
plot(grad(u))
```


Linear Elasticity

Variational formulation (mathematical notation)

$$\epsilon(v) = \frac{1}{2}(\text{grad}(v) + (\text{grad}(v))^\top)$$

$$\sigma(v) = 2\mu\epsilon(v) + \lambda(\text{tr}\,\epsilon)I$$

$$a(v,u) = (\epsilon(v), \sigma(u))$$

$$L(v) = (v, f)$$

Linear Elasticity

Variational formulation (implementation)

```
def epsilon(v):
    return 0.5*(grad(v) + grad(v).T)

def sigma(v):
    return 2*mu*epsilon(v) + l*tr(epsilon(v))*I

a = (epsilon(v), sigma(u))
L = (v, f)
```


Solving Navier-Stokes

```
def solve(self, problem, options=OPTIONS):

    # Get problem parameters
    mesh = problem.mesh
    shape = self.shape(mesh)

    # Set time step
    h = MeshSize(shape, mesh)
    dt = problem.dt or 0.25*h.min()

    # Create finite element spaces
    V = VectorElement("CG", shape, 1)
    Q = FiniteElement("CG", shape, 1)
    DG = FiniteElement("DG", shape, 0)

    # Test and trial functions
    v = TestFunction(V)
    q = TestFunction(Q)
    u = TrialFunction(V)
    p = TrialFunction(Q)

    # Functions
    u0 = Function(V, mesh, Vector())
    us = Function(V, mesh, Vector())
    u1 = Function(V, mesh, Vector())
    p1 = Function(Q, mesh, Vector())
    nu = Function(DG, mesh, problem.nu)
    k  = Function(DG, mesh, dt)
    f  = problem.f

    # Tentative velocity step
    a0 = dot(v, u)*dx + k*nu*dot(grad(v), grad(u))*dx
    L0 = dot(v, u0)*dx + k*dot(v, f)*dx - k*dot(v, mult(grad(u0), u0))*dx

    # Poisson problem for the pressure
    a1 = 1.0e-6*p*q*dx + dot(grad(q), grad(p))*dx
    L1 = -(1.0/k)*q*div(us)*dx

    # Velocity update
    a2 = dot(v, u)*dx
    L2 = dot(v, us)*dx - k*dot(v, grad(p1))*dx

    # Assemble matrices
    A0 = assemble(a0, mesh)
    A1 = assemble(a1, mesh)
    A2 = assemble(a2, mesh)
```

```
# Time loop
t = 0.0
while t < problem.T:

    # Propagate values to next time step
    t += dt
    u0.assign(u1)

    # Compute tentative velocity
    b = assemble(L0, mesh)
    [bc.apply(A0, b, a0) for bc in problem.bcv]
    solve(A0, us.vector(), b, gmres, ilu)

    # Compute p1
    b = assemble(L1, mesh)
    [bc.apply(A1, b, a1) for bc in problem.bcp]
    solve(A1, p1.vector(), b, gmres, amg)

    # Compute u1
    b = assemble(L2, mesh)
    [bc.apply(A2, b, a2) for bc in problem.bcv]
    solve(A2, u1.vector(), b, gmres, ilu)
```

Current Activities

- UFL: Unified Form Language
- Parallel assembly/solve
- Higher order meshes
- Finite element exterior calculus
- Automated error control / adaptivity

- FEniCS book project (Springer Verlag 2009)
- FEniCS'09 workshop June 11-12 2009

www.fenics.org