5. FEniCS and the Automation of CMM

Anders Logg
logg@tti-c.org

Toyota Technological Institute at Chicago

Sixth Winter School in Computational Mathematics
Geilo, March 5-10 2006

The Automation of CMM

- Basic principles
- Key steps towards the Automation of CMM

The FEniCS project

- FIAT
- FFC
- DOLFIN

- Chapters 8, 9 and 11 in lecture notes

The principles of automation

For given input satisfying a fixed set of conditions (the *input conditions*), produce output satisfying a given set of conditions (the *output conditions*)

- Need to have an *algorithm*
- Need *feed-back* control
  - Measurement, evaluation, action
  - Error estimator, stopping criterion, modification strategy
- Need modularization (abstraction)

Computational Mathematical Modeling (CMM)

Given model (input):

$$A(u) = f$$

Given output condition:

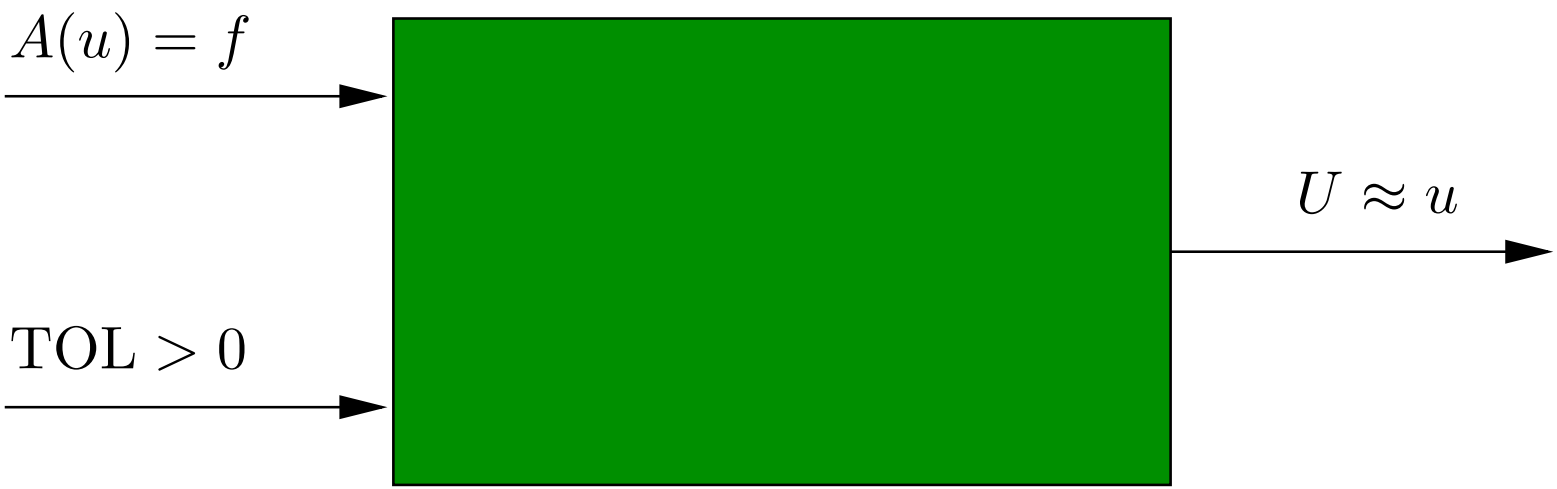
$$|M(U) - M(u)| \leq \mathrm{TOL}$$

where M is a given functional

Problems:

- Algorithm: computing $U \approx u$ (FEM)
- Feed-back: meeting the output condition (dual problem)
- Modularization: designing the system (FEniCS)

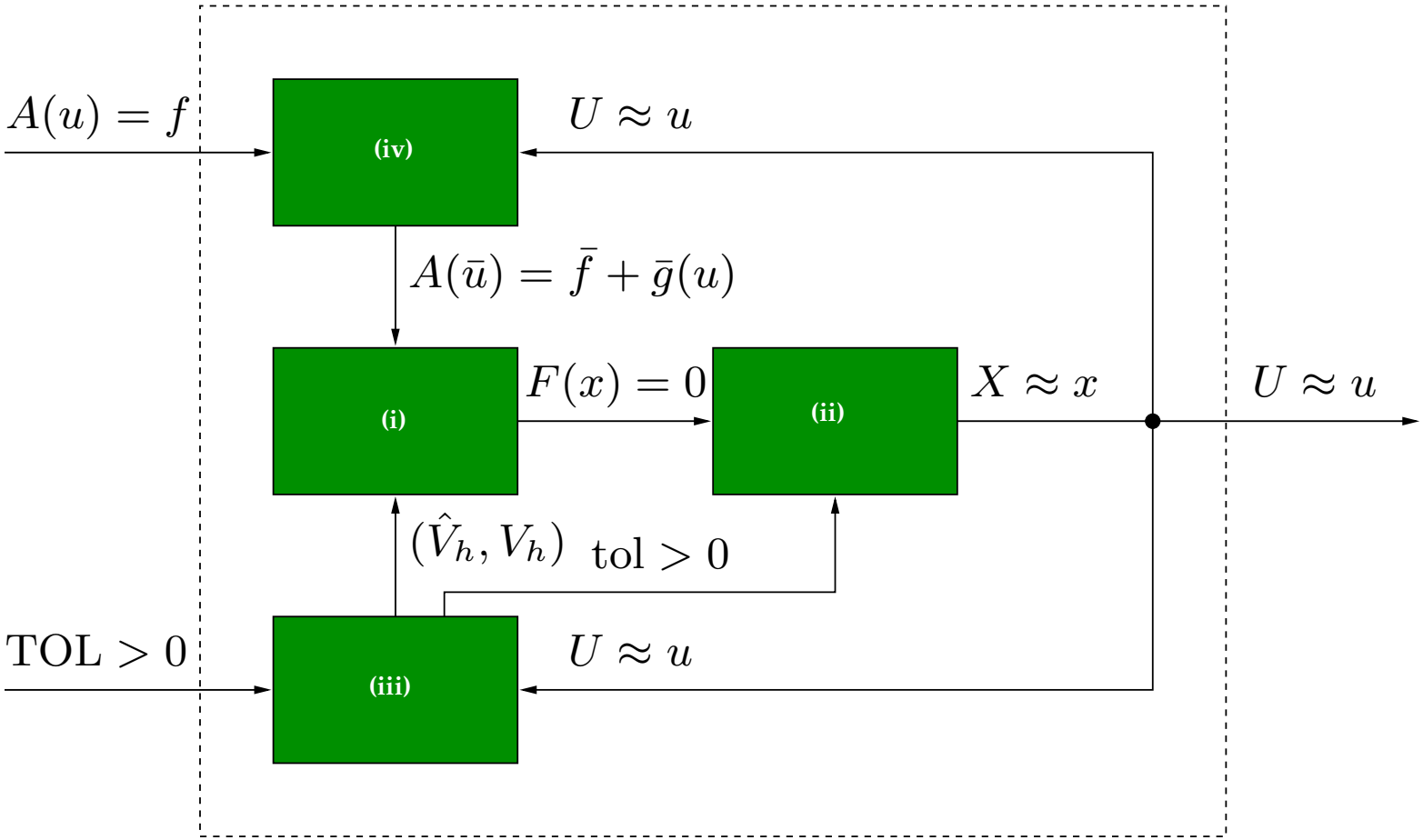
The Automation of CMM



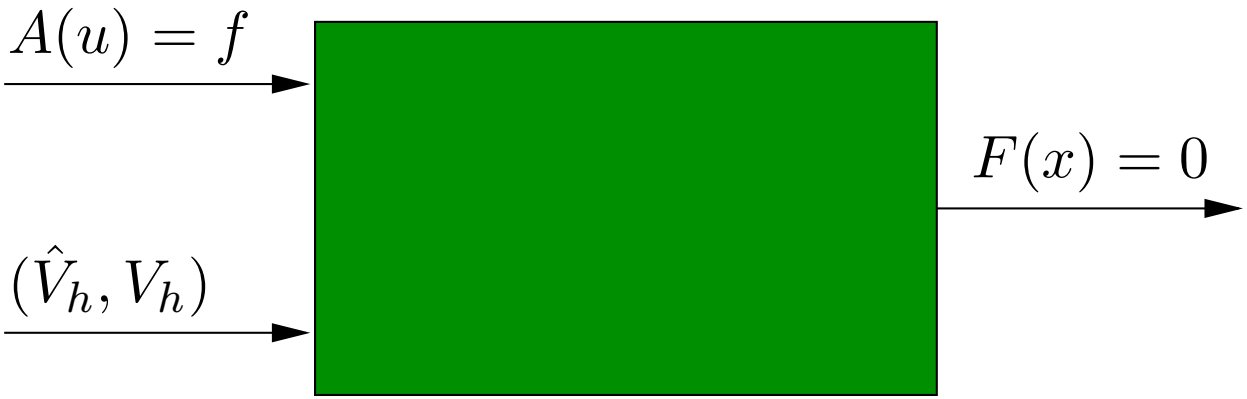
An agenda for the Automation of CMM

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

A modularized view of the Automation of CMM



(i) The automation of discretization



- Input: Model $A(u) = f$ and discrete representation $(\hat{V}_h, V_h)$
- Output: Discrete system $F(x) = 0$

- Produce a discrete system $F(x) = 0$ corresponding to the model $A(u) = f$ for the given discrete representation $(\hat{V}_h, V_h)$

(ii) The automation of discrete solution

- Input: Discrete system $F(x) = 0$ and discrete tolerance $\mathrm{tol} > 0$
- Output: Discrete solution $X \approx x$ (corresponding to U) satisfying $|m(X) - m(x)| \leq \mathrm{tol}$
- Produce a discrete solution X satisfying a given accuracy requirement using a minimal amount of work

(iii) The automation of error control

- Input: Current solution $U \approx u$ and tolerance $\mathrm{TOL} > 0$
- Output: New discrete representation $(\hat{V}_h, V_h)$ and tolerance for the solution of the discrete system
- Produce an optimal discrete representation $(\hat{V}_h, V_h)$ based on feed-back from the computed solution

The dual problem

The dual problem of $A(u) = f$ for the given output functional M is given by

$$\overline{A'}^* \varphi = \psi \quad \text{on } \Omega \times [0, T)$$

where $\overline{A'}^*$ denotes the adjoint of a mean value of the Fréchet derivative A' of A:

$$\overline{A'} = \int_0^1 A'\left(sU + (1-s)u\right) \, \mathrm{d}s$$

and where ψ is the Riesz representer of a similar mean value of the Fréchet derivative M' of M:

$$(v, \psi) = \overline{M'}v \quad \forall v \in V$$

Estimating the error

Error representation:

$$\begin{aligned} M(U) - M(u) &= \overline{M'}(U - u) = (U - u, \psi) = (U - u, \overline{A'}^* \varphi) \\ &= (\overline{A'}(U - u), \varphi) = (A(U) - A(u), \varphi) \\ &= (A(U) - f, \varphi) = (R(U), \varphi) \end{aligned}$$

If the solution U is computed by a Galerkin method so that $(R(U), v) = 0$ for any $v \in \hat{V}_h$ we obtain

$$M(U) - M(u) = (R(U), \varphi - \pi_h \varphi)$$

where $\pi_h \varphi$ is a suitable approximation of φ in $\hat{V}_h$

The weak dual problem

Nonlinear variational problem: Find $u \in V$ such that

$$a(u; v) = L(v) \quad \forall v \in \hat{V}$$

Dual (linear) variational problem: Find $\varphi \in \hat{V}$ such that

$$\overline{a'}^*(U, u;\, v, \varphi) = \overline{M'}(U, u; v) \quad \forall v \in V$$

where $\overline{a'}^*$ denotes the adjoint of the bilinear form $\overline{a'}$

Note:

$$\overline{a'}(U, u;\, v, w) = \overline{a'}^*(U, u;\, w, v)$$
$$\overline{a'}(U, u;\, v, U - u) = a(U; v) - a(u; v)$$

Estimating the error

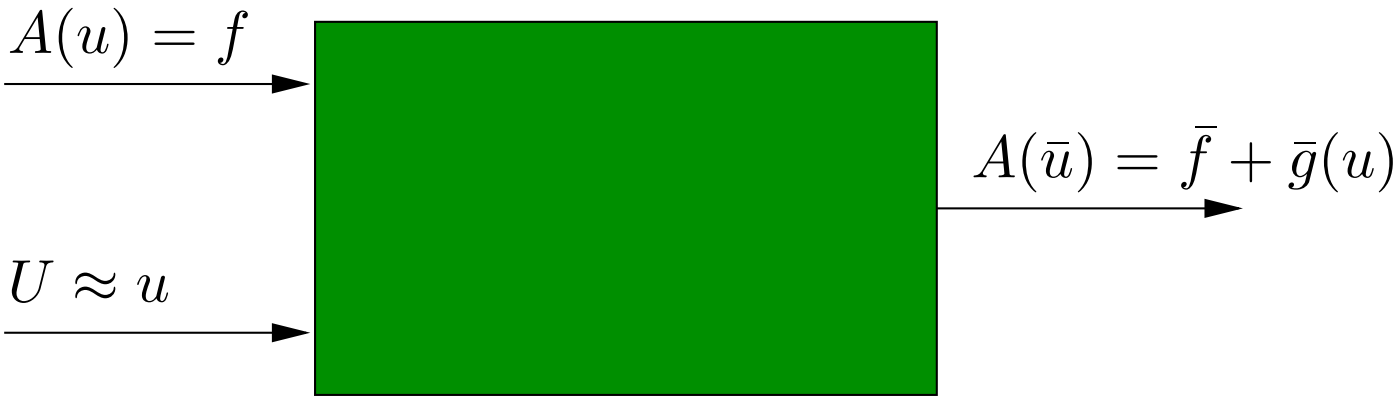
Error representation:

$$\begin{aligned} M(U) - M(u) &= \overline{M'}(U, u;\, U - u) = \overline{a'}^{*}(U, u;\, U - u, \varphi) \\ &= \overline{a'}(U, u;\, \varphi, U - u) = a(U;\, \varphi) - a(u;\, \varphi) \\ &= a(U;\, \varphi) - L(\varphi) \end{aligned}$$

As before we use the Galerkin orthogonality to subtract $a(U, \pi\varphi) - L(\pi\varphi) = 0$ for some $\pi_h\varphi \in \hat{V}_h \subset \hat{V}$:

$$M(U) - M(u) = a(U; \varphi - \pi\varphi) - L(\varphi - \pi\varphi)$$

(iv) The automation of modeling



- Input: Model $A(u) = f$ and current solution $U \approx u$
- Output: Reduced model $A(\bar{u}) = \bar{f} + \bar{g}(u)$

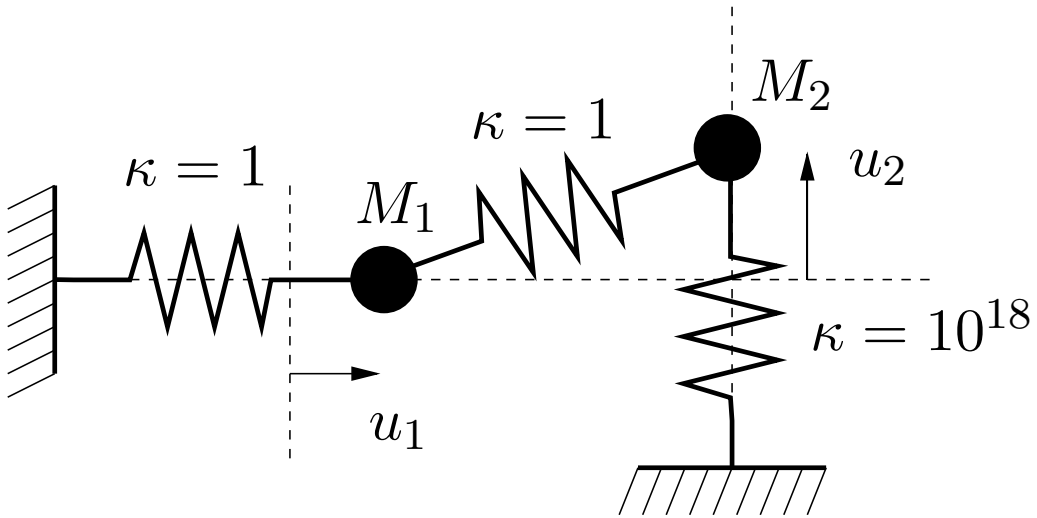
- Produce a reduced model $A(\bar{u}) = \bar{f} + \bar{g}(u)$ for the variation of the solution u on resolvable scales

(iv) The automation of modeling

The model for the average $\bar{u}$ is obtained by taking the average of the full model $A(u) = f$:

$$A(\bar{u}) = \bar{A}(u) + (A(\bar{u}) - \bar{A}(u)) = \bar{f} + \bar{g}(u)$$

Resolve the variation of the solution u of the full model in a short time simulation to determine a subgrid model for the *variance* $\bar{g}(u) = A(\bar{u}) - \bar{A}(u)$



(v) The automation of optimization

- Input: Model $A(u,p) = f$ and cost functional $\mathcal{J}(u,p)$
- Output: Solution $U \approx u$ and control parameter $P \approx p$

- Produce the solution $U \approx u$ and control parameter $P \approx p$ minimizing the cost functional using a minimal amount of work

(v) The automation of optimization

Find the stationary points of the associated Lagrangian:

$$L(u, p, \varphi) = \mathcal{J}(u, p) + (A(u, p) - f(p), \varphi)$$

Stationarity with respect to (u, p) gives a system of PDEs (the primal and the dual) and a direction for the update of p:

$$\begin{aligned} A(u, p) &= f(p) \\ (A')^*(u, p)\varphi &= -\partial\mathcal{J}/\partial u \\ \partial\mathcal{J}/\partial p &= (\partial f/\partial p)^*\varphi - (\partial A/\partial p)^*\varphi \end{aligned}$$

The FEniCS project

- Initiated in 2003
- Develop free software for the Automation of CMM

Partners (in order of appearance):

- Toyota Technological Institute at Chicago
- The University of Chicago
- Chalmers University of Technology
- The Royal Institute of Technology
- Argonne National Laboratory
- Simula Research Laboratory
- Delft University of Technology

Components/projects:

- DOLFIN, FFC, FIAT, Ko, Puffin, Sieve, FErari

FEniCS and the Automation of FEM

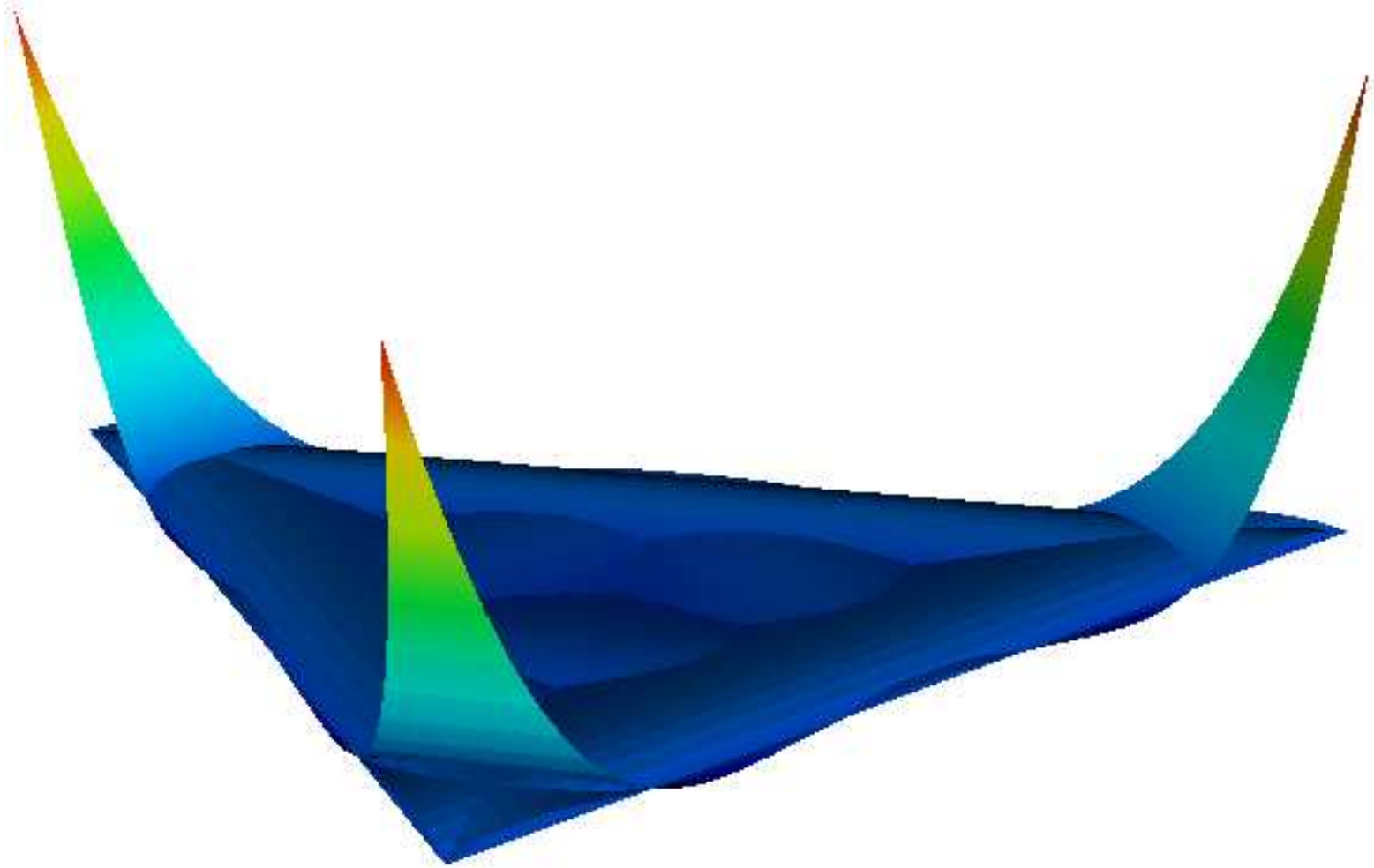
The *automation of the finite element method* realizes the *automation of discretization* and is thus a key step towards the Automation of CMM:

FIAT	automates the tabulation of basis functions
FFC	automates the computation of the element tensor
DOLFIN	automates the assembly of the discrete system

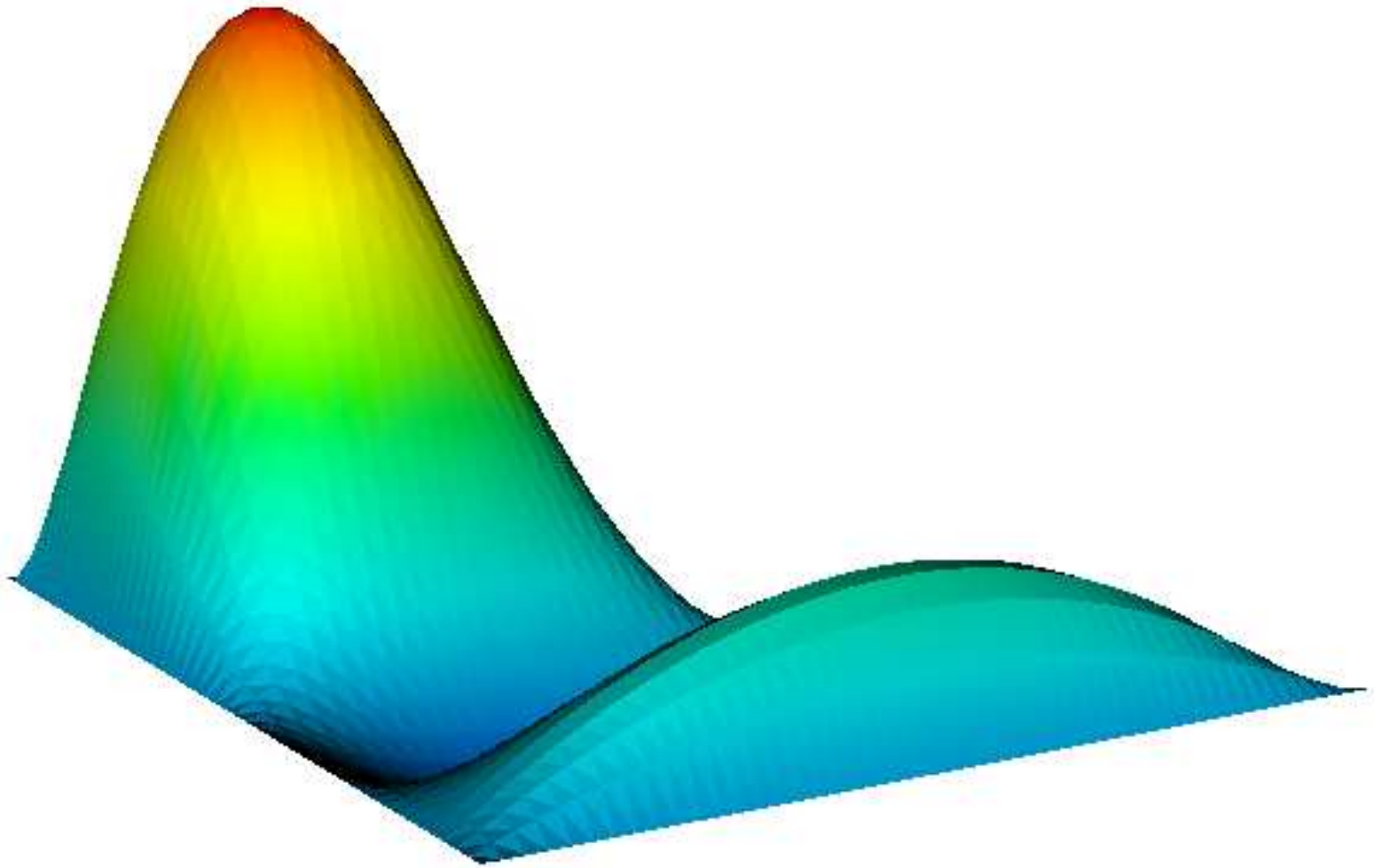
FIAT

- Developed by Robert C. Kirby (University of Chicago)
- First released in 2004
- Automates the tabulation of basis functions
- Implemented in Python (C++ version: FIAT++)
- Currently supported finite elements:
  - Lagrange
  - Hermite
  - Raviart–Thomas
  - Brezzi–Douglas–Marini
  - Nedelec
  - Crouzeix–Raviart
- In progress:
  - Brezzi–Douglas–Fortin–Marini
  - Arnold–Winther

Fifth-degree Lagrange basis functions

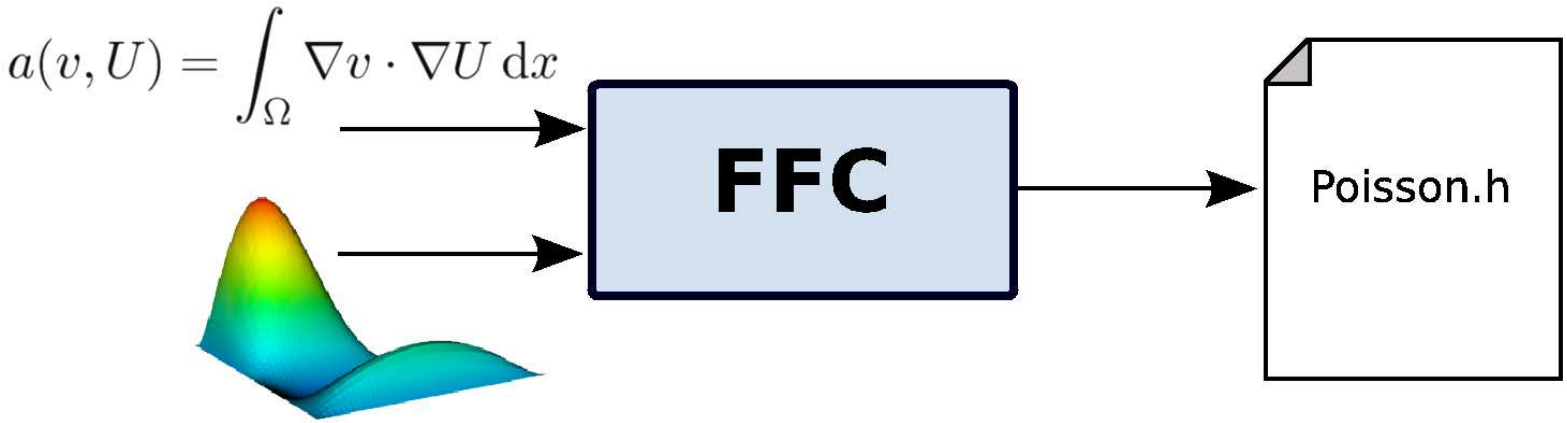


Another fifth-degree Lagrange basis function



FFC

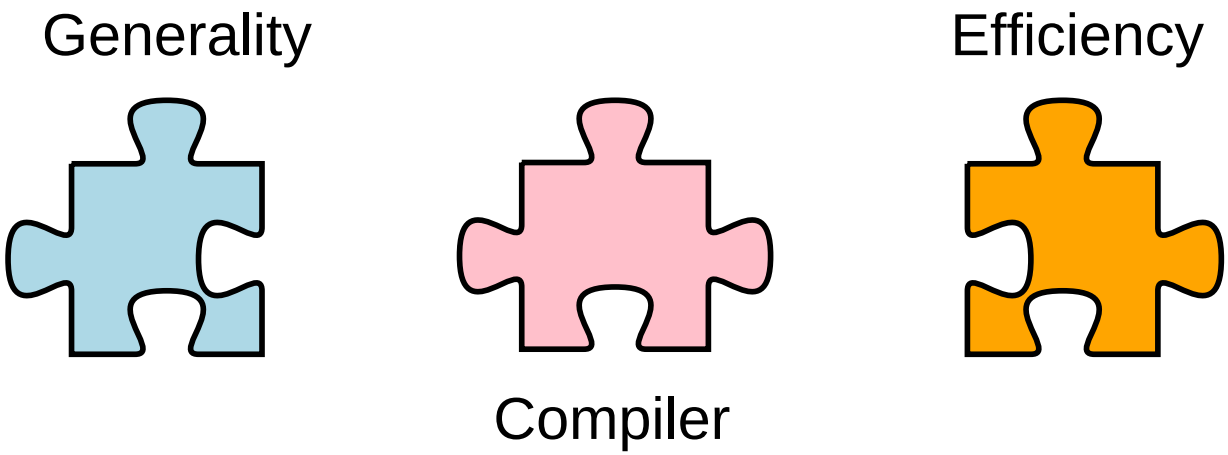
- Developed by Anders Logg (TTI Chicago)
- First released in 2004
- Automates the computation of the element tensor
- A compiler for multilinear forms and finite elements



Design goals for FFC

- Any form
- Any element
- Maximum efficiency

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

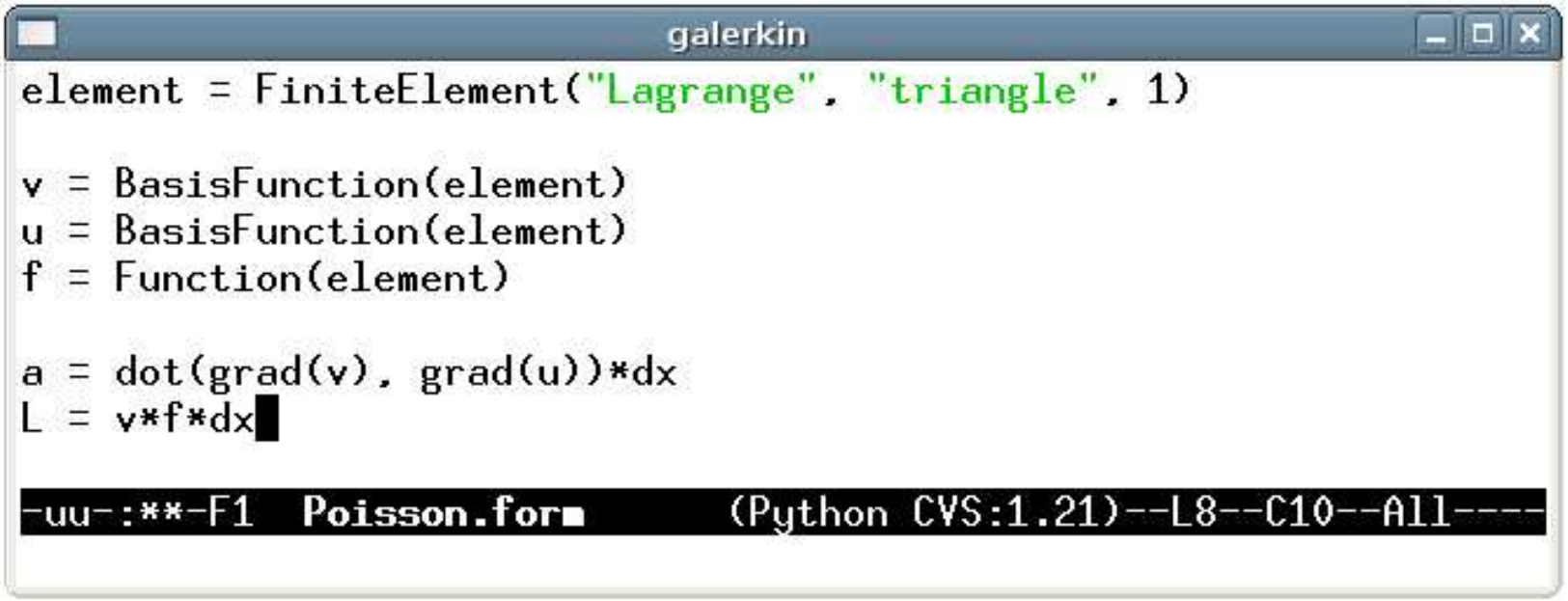


Features

- Command-line or Python interface
- Support for a wide range of elements (through FIAT):
  - Continuous scalar or vector Lagrange elements of arbitrary degree ($q \geq 1$) on triangles and tetrahedra
  - Discontinuous scalar or vector Lagrange elements of arbitrary degree ($q \geq 0$) on triangles and tetrahedra
  - Crouzeix–Raviart on triangles and tetrahedra
  - Arbitrary mixed elements
  - Others in preparation
- Efficient, close to optimal, evaluation of forms
- Support for user-defined formats
- Primary target: DOLFIN/PETSc

Command-line interface

1. Implement the form using your favorite text editor (emacs):



2. Compile the form using FFC:

```
# ffc Poisson.form
```

Python interface

```
from ffc import *

element = FiniteElement("Lagrange", "triangle", 1)
dx = Integral("interior")

v = BasisFunction(element)
U = BasisFunction(element)
f = Function(element)

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

compile([a, L])

#forms = build([a, L])
#write(forms)
```

Basic form language

Basic scalar operators

addition	`v + w`
subtraction	`v - w`
multiplication	`v*w, c*v`
division	`v/c`

Vector/tensor operators

indexing	`v[i]`
summation	`v[i]*w[i]`

Differentiation/integration

differentiation	`v.dx(i)`
integration	`*dx, *ds`

New operators

Vector operators

vectorization	`vec(v)`
vector length	`len(v)`
scalar product	`dot(v, w)`
cross product	`cross(v, w)`

Tensor operators

transpose	`transp(A)`
trace	`trace(A)`
products	`mult(A, v)`, `mult(A, B)`

Differential operators

partial derivative	`D(v, i)`
vector operators	`grad(v)`, `div(v)`, `rot(v)`

User-defined operators

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

def sigma(v):
    return 2*mu*epsilon(v) + \
           lmbda*mult(trace(epsilon(v)), I)

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

Mixed elements

Define new function space V as direct sum of two (or more) given function spaces:

$$V = (V_1, 0) \oplus (0, V_2)$$

Constructing mixed elements in FFC:

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

E = TH + TH + P1 + P2 + ...
E = MixedElement([TH, TH, P1, P2, ...])
```

Compiling Poisson (default mode)

```
void eval(real block[], const AffineMap& map) const
{
  // Compute geometry tensors
  real G0_0_0 = map.det*map.g00*map.g00 + map.det*map.g01*map.g01;
  real G0_0_1 = map.det*map.g00*map.g10 + map.det*map.g01*map.g11;
  real G0_1_0 = map.det*map.g10*map.g00 + map.det*map.g11*map.g01;
  real G0_1_1 = map.det*map.g10*map.g10 + map.det*map.g11*map.g11;

  // Compute element tensor
  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[8] = 0.5*G0_1_1;
}
```

Compiling Poisson (BLAS mode)

```
void eval(real block[], const AffineMap& map) const
{
  // Reset geometry tensors
  for (unsigned int i = 0; i < blas.ni; i++)
    blas.Gi[i] = 0.0;

   // Compute entries of G multiplied by nonzero entries of A
   blas.Gi[0] = map.det*map.g00*map.g00 + map.det*map.g01*map.g01;
   blas.Gi[1] = map.det*map.g00*map.g10 + map.det*map.g01*map.g11;
   blas.Gi[2] = map.det*map.g10*map.g00 + map.det*map.g11*map.g01;
   blas.Gi[3] = map.det*map.g10*map.g10 + map.det*map.g11*map.g11;

   // Compute element tensor using level 2 BLAS
   cblas_dgemv(CblasRowMajor, CblasNoTrans,
               blas.mi, blas.ni, 1.0, blas.Ai,
               blas.ni, blas.Gi, 1, 0.0, block, 1);
}
```

FFC form language: test case 1

Bilinear form:

$$a(v, U) = \int_\Omega v\, U \,\mathrm{d}x$$

Implementation:

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

v = BasisFunction(element)
U = BasisFunction(element)

a = v*U*dx
```

FFC form language: test case 2

Bilinear form:

$$a(v, U) = \int_\Omega \nabla v \cdot \nabla U \, \mathrm{d}x$$

Implementation:

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

v = BasisFunction(element)
U = BasisFunction(element)

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

FFC form language: test case 3

Bilinear form:

$$a(v, U) = \int_\Omega v \cdot (w \cdot \nabla) U \, \mathrm{d}x$$

Implementation:

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

v = BasisFunction(element)
U = BasisFunction(element)
w = Function(element)

a = v[i]*w[j]*D(U[i],j)*dx
```

FFC form language: test case 4

Bilinear form:

$$a(v, U) = \int_\Omega \epsilon(v) : \epsilon(U) \, \mathrm{d}x$$

Implementation:

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

v = BasisFunction(element)
U = BasisFunction(element)

def epsilon(v):
    return 0.5*(grad(v) + transp(grad(v)))

a = dot(epsilon(v), epsilon(U))*dx
```

FFC benchmark speedups

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	—	—	—	—

- Impressive speedups but far from optimal
- Data access costs more than flops
- Solution: build arrays and call BLAS (level 2 or 3)

Benchmarks for (level 2) BLAS mode

Poisson degree 1 in 3D:

Stage	default mode	BLAS mode
FFC	3.3e-02	3.0e-02
g++	9.2e-01	9.3e-01
Run-time	1.3e-07	9.2e-07

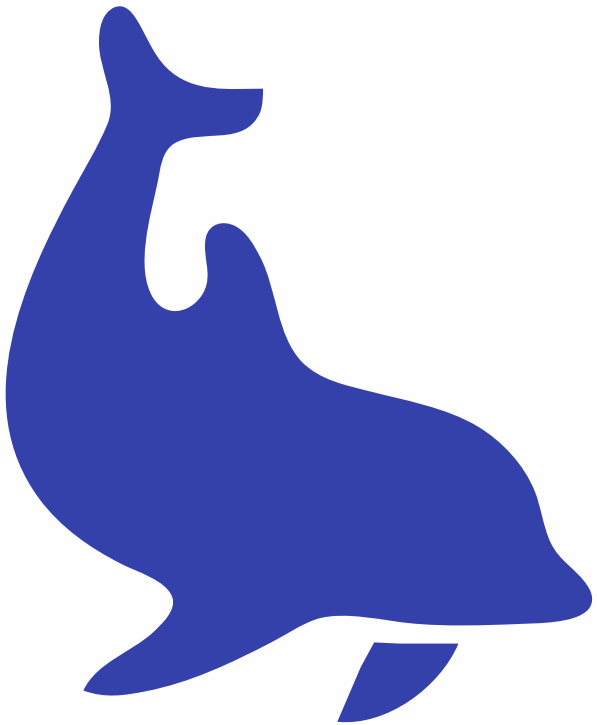
Poisson degree 8 in 3D:

Stage	default mode	BLAS mode
FFC	68	60
g++	91	1.4
Run-time	1.3e-03	5.9e-04

Break-even at $q = 6$ (run-time)

DOLFIN

- Developed by Hoffman/Jansson/Logg/Wells
- First released in 2002
- Automates the assembly of the discrete system
- The C++/Python interface of DOLFIN



Automatic assembly

```
Mesh mesh;

Poisson::BilinearForm a;
Matrix A;
FEM::assemble(a, A, mesh);

Poisson::LinearForm L;
Vector b;
FEM::assemble(L, b, mesh);
```

The mesh

Iterators:

```
for (CellIterator c(m); !c.end(); ++c)
  for (VertexIterator v1(c); !v1.end(); ++v1)
    for (VertexIterator v2(n1); !v2.end(); ++v2)
      cout << *v2 << endl;
```

Mesh refinement:

```
// Mark cells for refinement
for (CellIterator cell(mesh); !cell.end(); ++cell)
  if ( ... )
    cell->mark();

// Refine mesh
mesh.refine();
```

Linear algebra data structures

Matrices and vectors:

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

Matrix-free (action):

```
class MyMatrix : public VirtualMatrix
{
public:
  void mult(const Vector& x, Vector& y);
  ...
};
```

Linear algebra solvers

Simple:

```
Matrix A;
Vector x, b;

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

Advanced:

```
Matrix A;
Vector x, b;

KrylovSolver krylov(KrylovSolver::bicgstab,
                    Preconditioner::hypre_amg);
krylov.solve(A, x, b);
```

Solving ODEs

```
class Lorenz : public ODE
{
public:
  void f(const real u[], real t, real y[])
  {
    y[0] = s*(u[1] - u[0]);
    y[1] = r*u[0] - u[1] - u[0]*u[2];
    y[2] = u[0]*u[1] - b*u[2];
  }
  ...
};

Lorenz ode;
ode.solve();
```

Solving PDEs

```
UnitSquare mesh(16, 16);
Poisson::BilinearForm a;
Poisson::LinearForm L(f);
PDE pde(a, L, mesh, bc);

Function U = pde.solve();

File file(``poisson.pvd'');
file << U;
```

Pre- and post-processing

Input:

```
File file(‘‘mesh.xml’’);
Mesh mesh;
file >> mesh;
```

Output:

```
Function U;
File file(‘‘solution.pvd’’);
//File file(‘‘solution.xml’’);
//File file(‘‘solution.dx’’);
//File file(‘‘solution.m’’);
//File file(‘‘solution.res’’);
//File file(‘‘solution.tec’’);
file << U;
```

Puffin

- Developed by Hoffman/Logg
- First released in 2003
- A minimal educational version of FEniCS for Octave/MATLAB
- Developed as part of the Body and Soul applied mathematics reform project
- Supported by a series of computer sessions

Poisson's equation with Puffin

Variational problem:

$$\int_\Omega \nabla v \cdot \nabla U \,\mathrm{d}x + \int_\Gamma \gamma\, v\, U \,\mathrm{d}s = \int_\Omega v\, f \,\mathrm{d}x + \int_{\partial\Omega} v\,(\gamma g_D - g_N)\,\mathrm{d}s$$

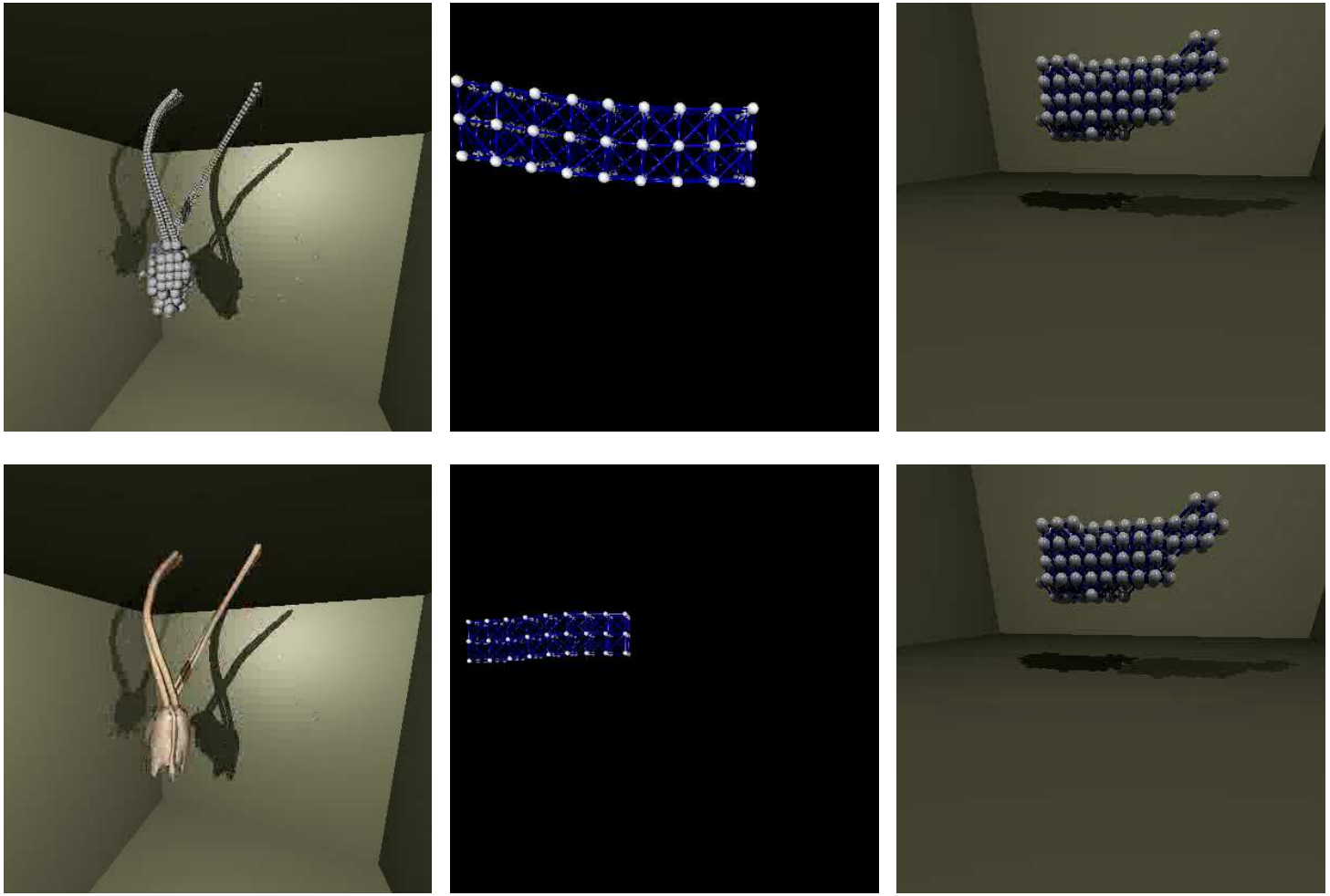
```
function integral = Poisson(v, U, w, dv, dU, dw, ...)

if eq == 1
  integral = dv'*dU*dx + g(x,d,t)*v*U*ds;
else
  integral = v*f(x,d,t)*dx + v*(g(x,d,t)*gd(x,d,t) - ...
                                gn(x,d,t))*ds;
end
```

Ko

- Developed by Johan Jansson (Chalmers)
- First released in 2005
- Mechanical simulator
- Uses DOLFIN as the computational backend
- Simple mass–spring model or full PDE model

Ko demos (courtesy of Johan Jansson)



FEniCS

Visit the FEniCS web page:

`http://www.fenics.org/`

- Software
- Manuals
- Papers
- Presentations
- Discussion

Upcoming lectures

0. Automating the Finite Element Method
1. Survey of Current Finite Element Software
2. The Finite Element Method
3. Automating Basis Functions and Assembly
4. Automating and Optimizing the Computation of the Element Tensor
5. FEniCS and the Automation of CMM
6. FEniCS Demo Session