Automating the Finite Element Method

Anders Logg

`logg@tti-c.org`

Toyota Technological Institute at Chicago

Johan Hoffman, Johan Jansson, Claes Johnson, Robert C. Kirby, Matthew Knepley, Ridgway Scott

Automating the finite element method

The **FEniCS** project automates the following 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)

$$\texttt{for all elements } e \in \mathcal{T}_{\Omega}$$
$$A \mathrel{+}= A^e$$

Benchmark results: impressive speedups

FFC vs Quadrature at q = 3 (3D)
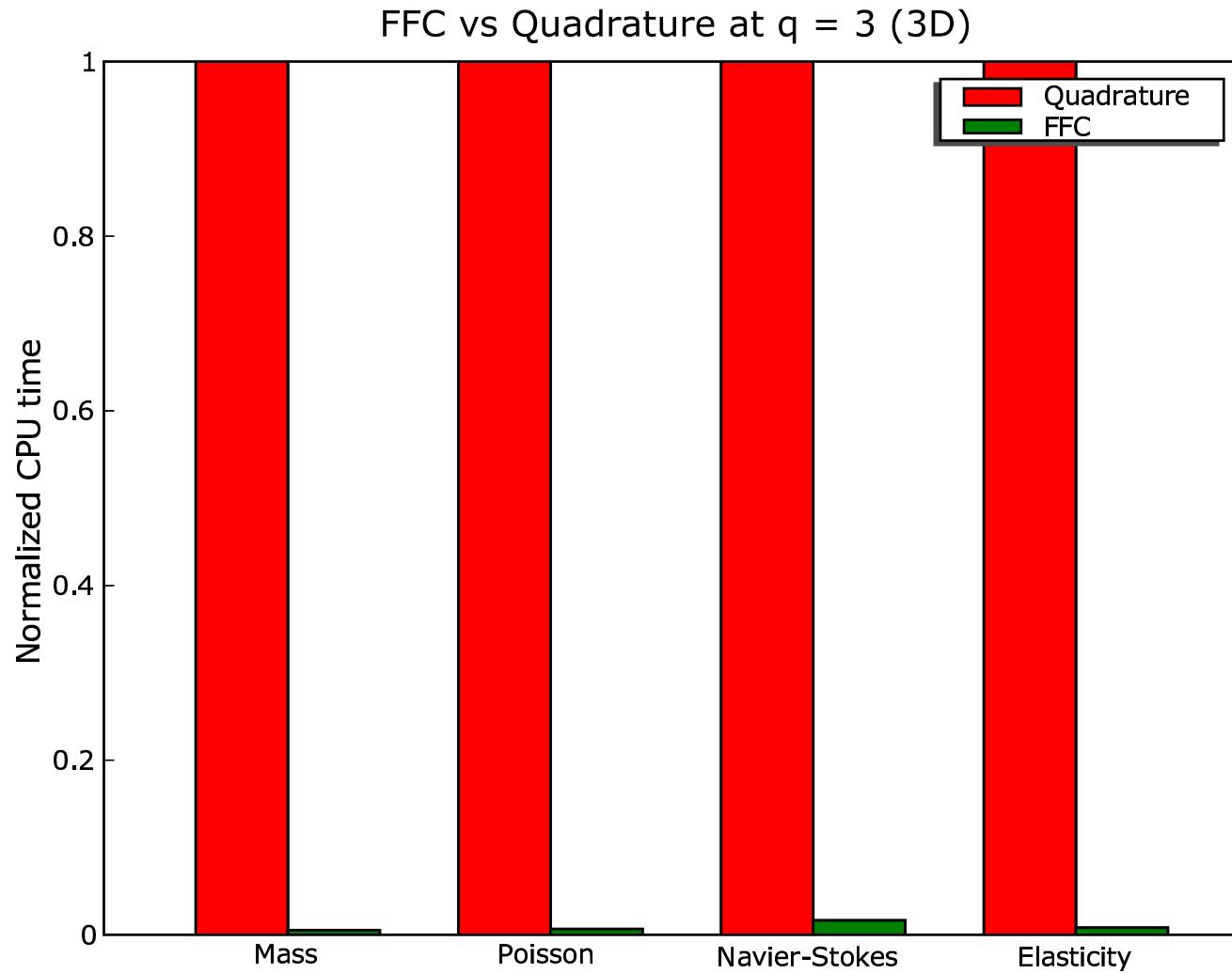
Outline

- **FENICS** and the Automation of CMM

- FIAT, the finite element tabulator

- FFC, the form compiler

- DOLFIN, a PSE for differential equations

- Benchmark results

- Future directions for **FENICS**
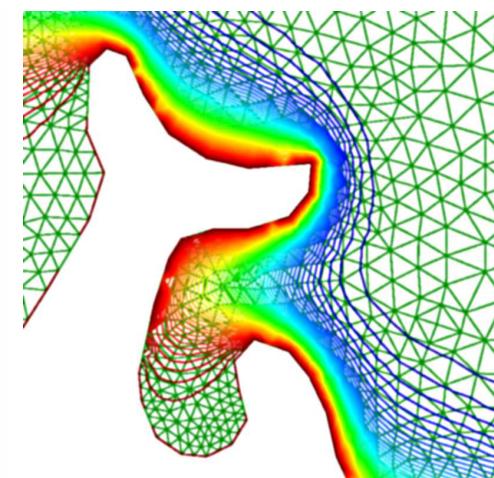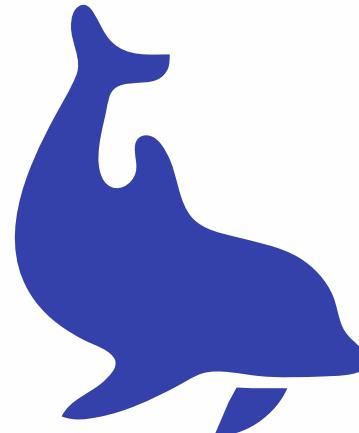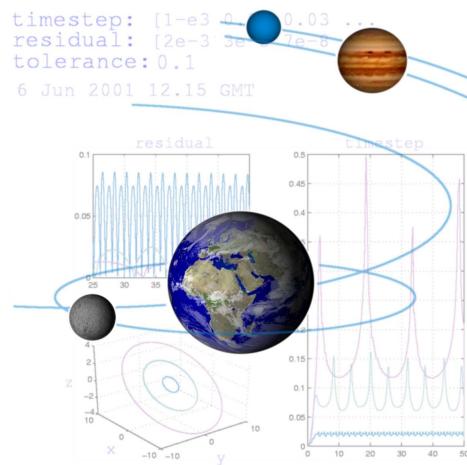
# _FEniCS_ and the Automation of CMM

The **FEniCS** project

A free software project for the Automation of Computational Mathematical Modeling (ACMM), developed at

- The Toyota Technological Institute at Chicago

- The University of Chicago

- Chalmers University of Technology (Göteborg, Sweden)

- NADA, KTH (Stockholm, Sweden)

- Argonne National Laboratory (Chicago)
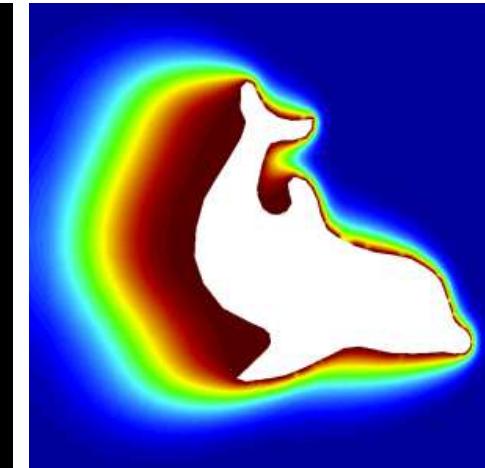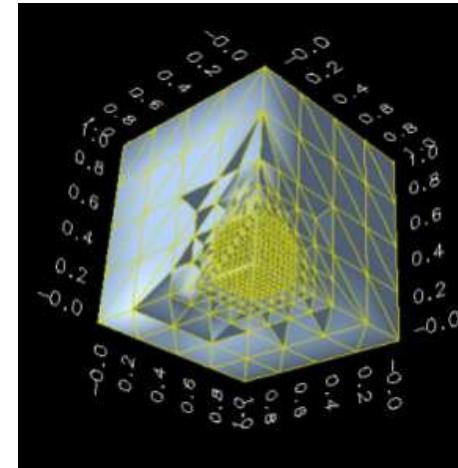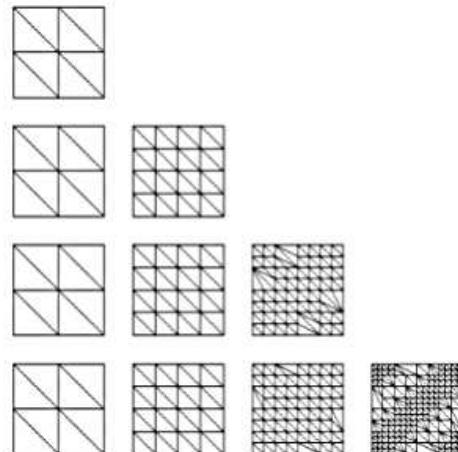
Main goal: the Automation of CMM

$$Au = f \longrightarrow$$

$$\text{TOL} > 0 \longrightarrow$$

$$\longrightarrow U \approx u$$
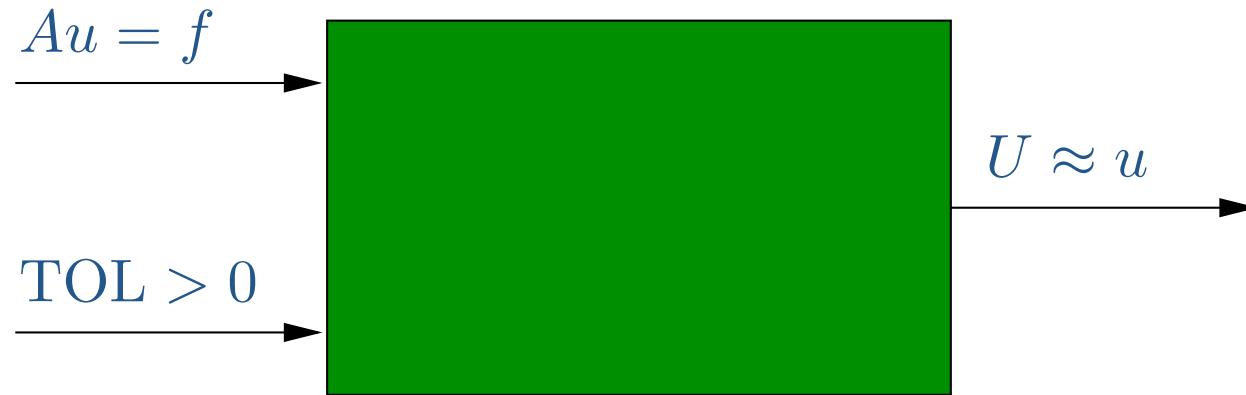
- Input: Model $Au = f$ and tolerance $\text{TOL} > 0$
- Output: Solution $U \approx u$ satisfying $\|U - u\| \leq \text{TOL}$

- Produce a solution U satisfying a given accuracy requirement, using a minimal amount of work

A key step: the automation of discretization

$$Au = f$$

$$(V, \hat{V})$$

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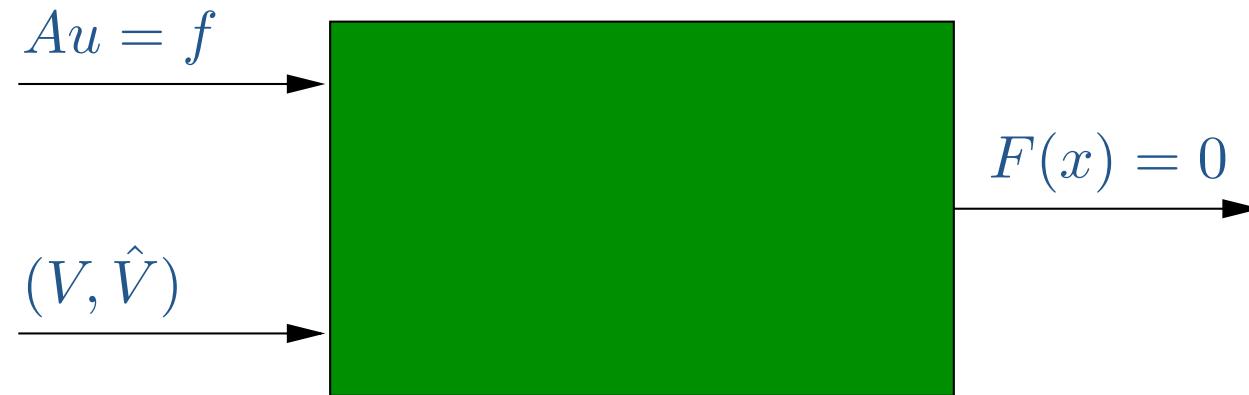
- Input: Model $Au = f$ and discrete representation $(V, \hat{V})$
- Output: Discrete system $F(x) = 0$

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

Basic algorithm: the (Galerkin) finite element method

$$a(v, u) = L(v)$$

$$(V, \hat{V})$$

$$F(x) = 0$$
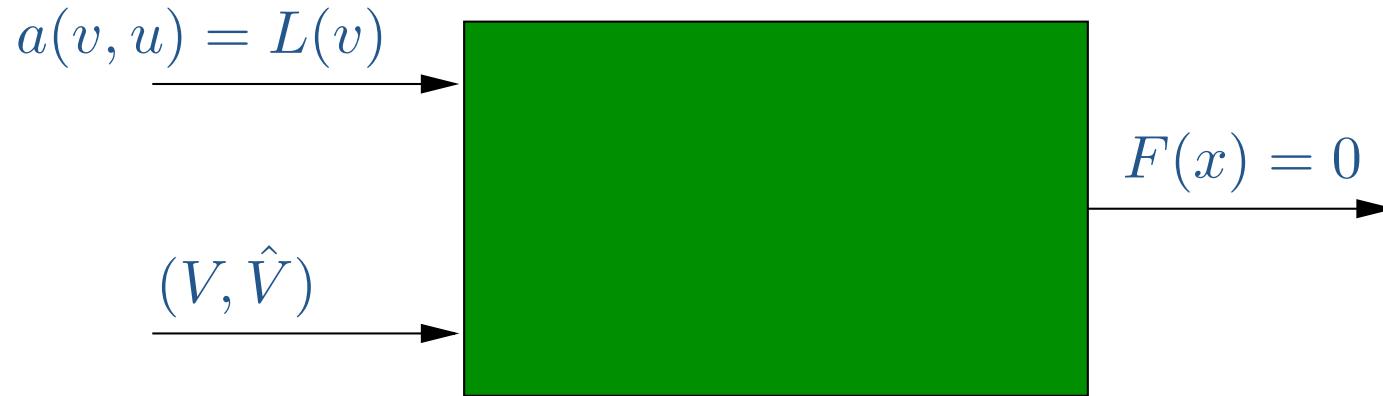
- Input: Variational problem $a(v, u) = L(v)$ for all v and discrete representation $(V, \hat{V})$
- Output: Discrete system $F(x) = 0$

FEniCS development

Basic principles:

- Focus on automation
- Focus on efficiency
- Focus on consistent user-interfaces
- Focus on applications

Implementation:

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

FEniCS components

- **FIAT**, the finite element tabulator
  - Robert C. Kirby
- **FFC**, the form compiler
  - Anders Logg
- **DOLFIN**, a PSE for differential equations
  - Hoffman, Jansson, Logg, et al.
- **FErari**, optimized form evaluation
  - Kirby, Knepley, Logg, Scott, Terrel
- **Ko**, simulation of mechanical systems
  - Johan Jansson
- **Puffin**, educational version for Octave/MATLAB
  - Hoffman, Logg
- (PETSc)

FIAT, *the finite element tabulator*

FIAT: FInite element Automatic Tabulator

- Automates the generation of finite element basis functions
- Simplifies the specification of new elements
- Continuous and discontinous Lagrange elements of arbitrary order
- Crouzeix–Raviart (CR) elements
- Nedelec elements
- Raviart–Thomas (RT) elements
- Brezzi–Douglas–Marini (BDM) elements
- Brezzi–Douglas–Marini–Fortin (BDM) elements
- In preparation: Taylor-Hood, Arnold-Winther, . . .

Implementation

- Use orthonormal basis on triangles and tets (Dubiner)
- Express basis functions in terms of the orthonormal basis
- Translate conditions on function space into linear algebraic relations

- Implemented in Python
- Efficient linear algebra
- Efficient tabulation mode
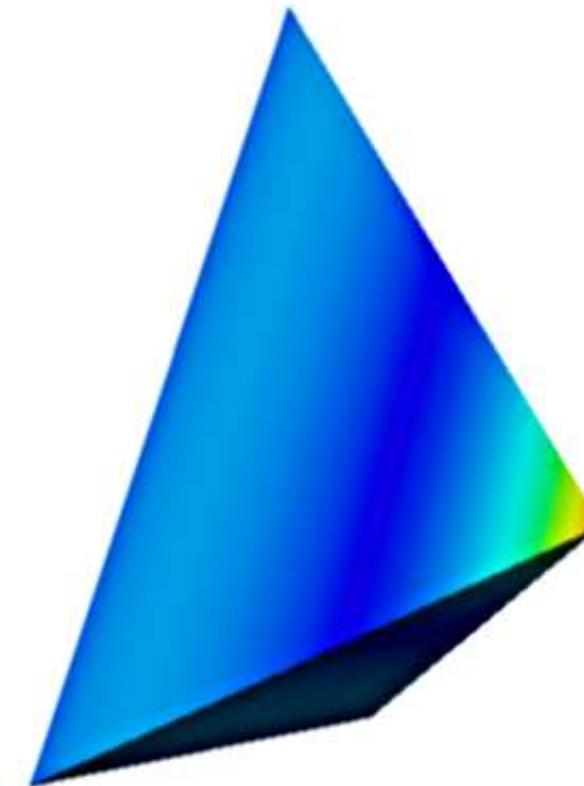
# Basic usage

```
>>> from FIAT.Lagrange import *
>>> from FIAT.shapes import *
>>> element = Lagrange(TETRAHEDRON, 3)
>>> basis = element.function_space()
>>> v = basis[0]
>>> v((-1.0, -1.0, -1.0))
1.0

>>> from FIAT.quadrature import *
>>> quadrature = make_quadrature(TETRAHEDRON, 2)
>>> points = quadrature.get_points()
>>> table = basis.tabulate_jet(2, points)
```

FFC, *the form compiler*

FFC: the FEniCS Form Compiler

- Automates a key step in the implementation of finite element methods for partial differential equations

- Input: a variational form and a finite element

- Output: optimal C/C++

$$a(v, u) = \int_{\Omega} \nabla u(x) \cdot \nabla v(x)\, dx \longrightarrow \boxed{\textbf{FFC}} \longrightarrow \texttt{Poisson.h}$$
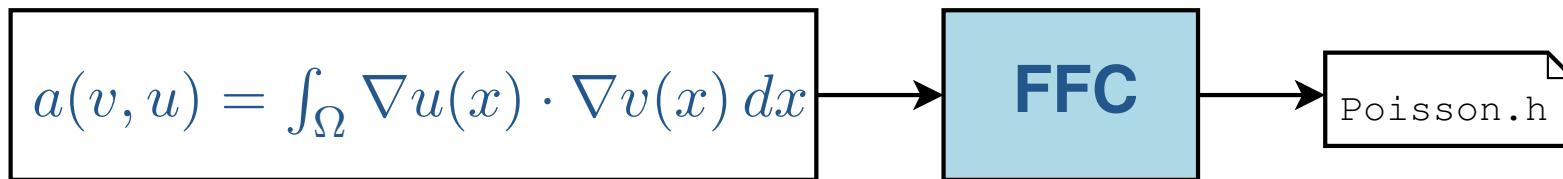
```
>> ffc [-l language] poisson.form
```

Design goals

- Any form
- Any element
- Maximum efficiency

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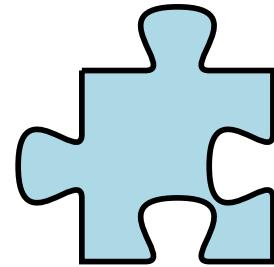
Generality   Efficiency
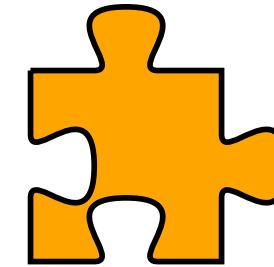
Compiler

Features

- Command-line or Python interface

- Support for a wide range of elements (through **FIAT**):
  - Continuous scalar or vector Lagrange elements of arbitrary order ($q \geq 1$) on triangles and tetrahedra
  - Discontinuous scalar or vector Lagrange elements of arbitrary order ($q \geq 0$) on triangles and tetrahedra
  - Crouzeix–Raviart on triangles and tetrahedra
  - Others in preparation

- Efficient, close to optimal, evaluation of forms

- Support for user-defined formats

- Primary target: **DOLFIN**/PETSc

Basic usage

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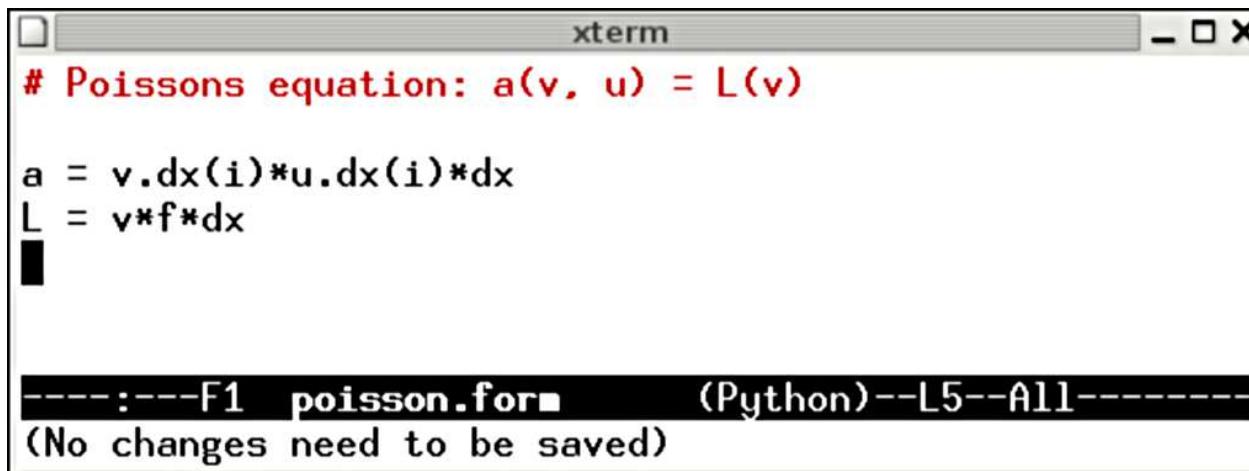

```
# Poissons equation: a(v, u) = L(v)

a = v.dx(i)*u.dx(i)*dx
L = v*f*dx

----:---F1   poisson.form       (Python)--L5--All--------
(No changes need to be saved)
```

2. Compile the form using **FFC**:

```
>> ffc poisson.form
```

This will generate C++ code (`Poisson.h`) for **DOLFIN**

Components of **FFC**
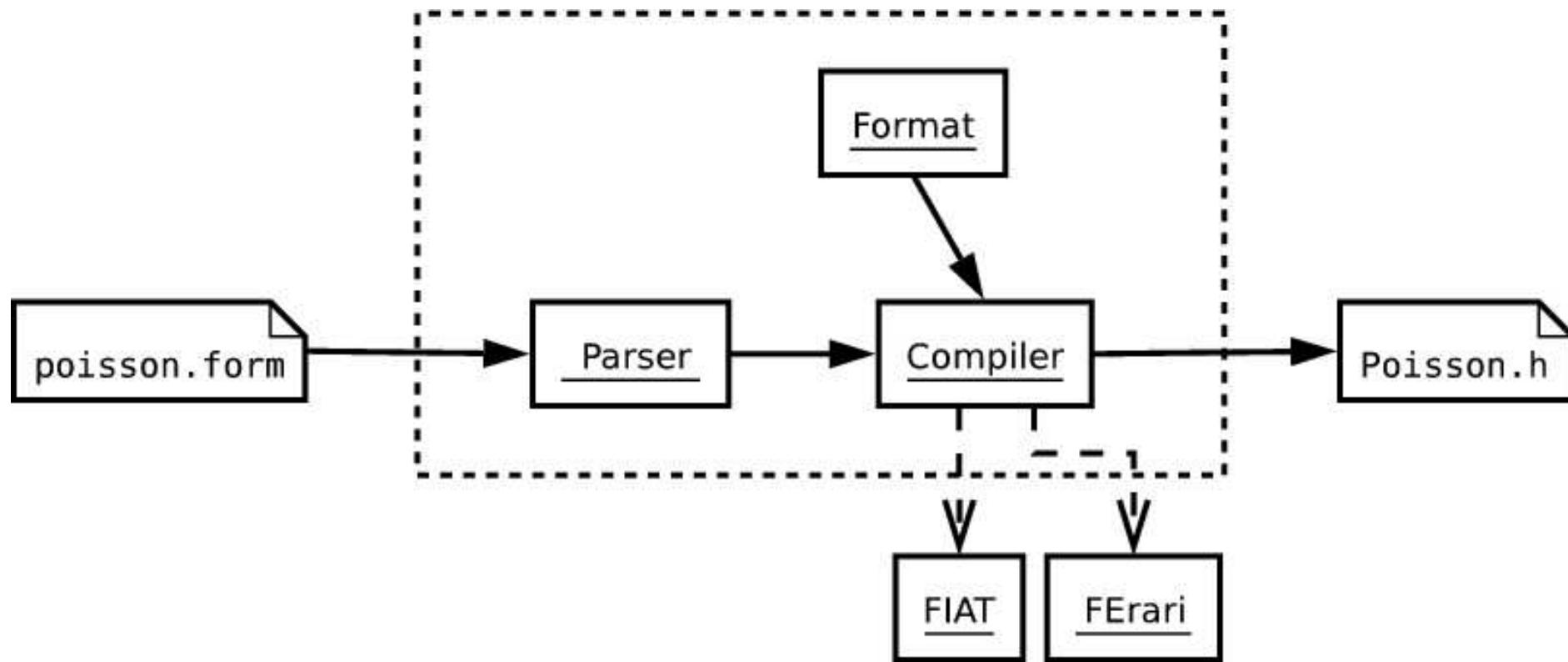
Basic example: Poisson's equation

- Strong form: Find $u \in \mathcal{C}^2(\overline{\Omega})$ with $u = 0$ on $\partial\Omega$ such that

$$-\Delta u = f \quad \text{in } \Omega$$

- Weak form: Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla v(x) \cdot \nabla u(x) \, dx = \int_\Omega v(x) f(x) \, dx \quad \text{for all } v \in H_0^1(\Omega)$$

- Standard notation: Find $u \in V$ such that

$$a(v, u) = L(v) \quad \text{for all } v \in \hat{V}$$

with $a : \hat{V} \times V \to \mathbb{R}$ a *bilinear form* and $L : \hat{V} \to \mathbb{R}$ a *linear form* (functional)

Obtaining the discrete system

Let V and $\hat{V}$ be discrete function spaces. Then

$$a(v, U) = L(v) \quad \text{for all } v \in \hat{V}$$

is a discrete linear system for the approximate solution $U \approx u$. With $V = \operatorname{span}\{\phi_i\}_{i=1}^M$ and $\hat{V} = \operatorname{span}\{\hat{\phi}_i\}_{i=1}^M$, we obtain the linear system

$$Ax = b$$

for the degrees of freedom $x = (x_i)$ of $U = \sum_{i=1}^M x_i \phi_i$, where

$$A_{ij} = a(\hat{\phi}_i, \phi_j)$$

$$b_i = L(\hat{\phi}_i)$$

Computing the linear system: assembly
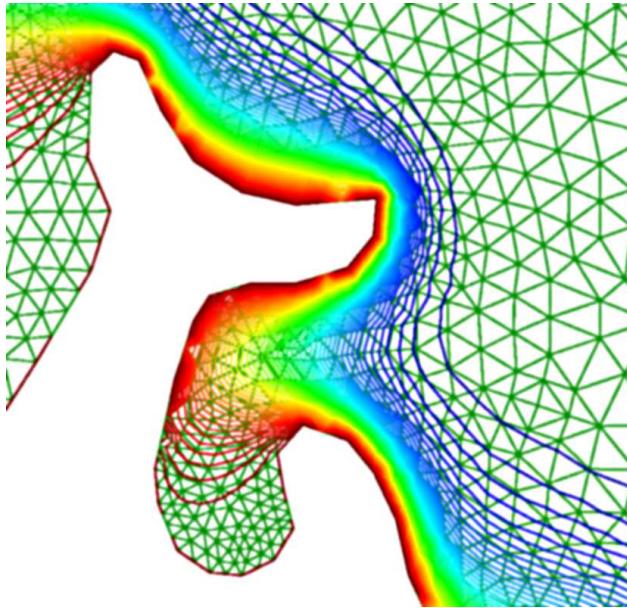
Noting that $a(v, u) = \sum_{e \in \mathcal{T}} a_e(v, u)$, the matrix A can be assembled by

$$A = 0$$
$$\texttt{for all elements}\, e \in \mathcal{T}$$
$$A\, \mathtt{+=}\, A^e$$

The *element matrix* A^e is defined by

$$A^e_{ij} = a_e(\hat{\phi}_i, \phi_j)$$

for all local basis functions $\hat{\phi}_i$ and ϕ_j on e

Multilinear forms

Consider a multilinear form

$$a : V_1 \times V_2 \times \cdots \times V_r \to \mathbb{R}$$

with $V_1, V_2, \ldots, V_r$ function spaces on the domain Ω

- Typically, $r = 1$ (linear form) or $r = 2$ (bilinear form)
- Assume $V_1 = V_2 = \cdots = V_r = V$ for ease of notation

Want to compute the rank r *element tensor* A^e defined by

$$A_i^e = a_e(\phi_{i_1}, \phi_{i_2}, \ldots, \phi_{i_r})$$

with $\{\phi_i\}_{i=1}^n$ the local basis on e and multiindex $i = (i_1, i_2, \ldots, i_r)$

Tensor representation of forms

In general, the element tensor A^e can be represented as the product of a *reference tensor* A^0 and a *geometric tensor* G_e:

$$A_i^e = A_{i\alpha}^0 G_e^\alpha$$

- A^0: a tensor of rank $|i| + |\alpha| = r + |\alpha|$
- G_e: a tensor of rank $|\alpha|$

Basic idea:

- Precompute A^0 at compile-time
- Generate optimal code for run-time evaluation of G_e and the product $A_{i\alpha}^0 G_e^\alpha$

The (affine) map $F_e : E \rightarrow e$

$$F_e(X) = x^1 \Phi_1(X) + x^2 \Phi_2(X) + x^3 \Phi_3(X)$$

Where the diagram shows:
- x^3
- e
- $x = F_e(X)$
- x^2
- x^1
- $X^3 = (0,1)$
- F_e
- E
- X
- $X^1 = (0,0)$
- $X^2 = (1,0)$
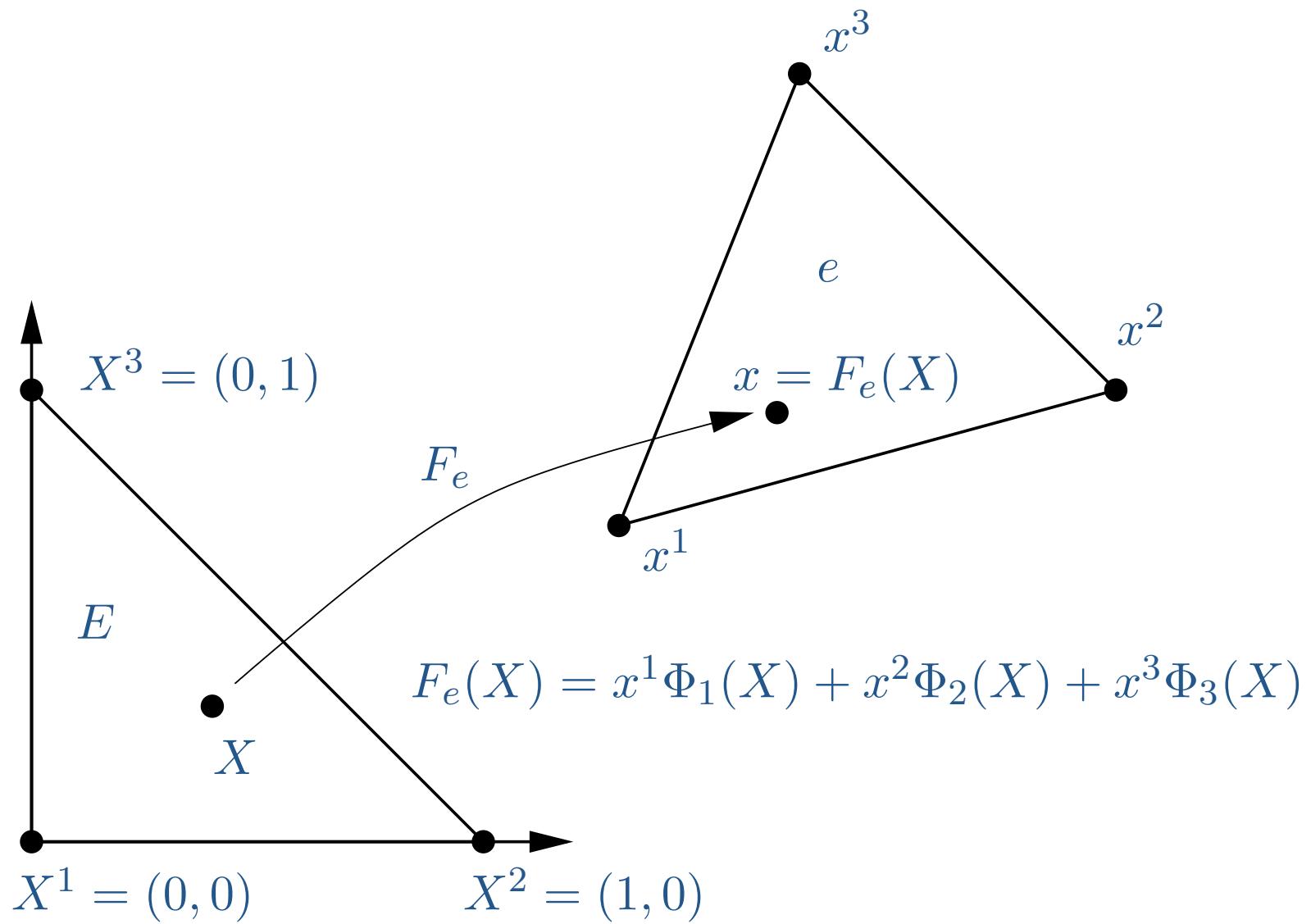
Example 1: the mass matrix

- Form:

$$a(v, u) = \int_\Omega v(x)u(x) \, dx$$

- Evaluation:

$$A_i^e = \int_e \phi_{i_1} \phi_{i_2} \, dx$$

$$= \det F_e' \int_E \Phi_{i_1}(X)\Phi_{i_2}(X) \, dX = A_i^0 G_e$$

with $A_i^0 = \int_E \Phi_{i_1}(X)\Phi_{i_2}(X) \, dX$ and $G_e = \det F_e'$

Example 2: Poisson

- Form:

$$a(v, u) = \int_\Omega \nabla v(x) \cdot \nabla u(x) \, dx$$

- Evaluation:

$$A_i^e = \int_e \nabla \phi_{i_1}(x) \cdot \nabla \phi_{i_2}(x) \, dx$$

$$= \det F_e' \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_E \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, dX = A_{i\alpha}^0 G_e^\alpha$$

with $A_{i\alpha}^0 = \int_E \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, dX$ and $G_e^\alpha = \det F_e' \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta}$

An algebra for forms

- Basic elements: basis functions and their derivatives

- Define addition, subtraction and multiplication with scalars to generate a vector space

- Define multiplication between elements of $\mathcal{A}$ to generate an algebra:

$$\mathcal{A} = \left\{ v : v = \sum c \prod \partial^{|\cdot|} \phi_{(\cdot)} / \partial x_{(\cdot)} \right\}$$

- $\mathcal{A}$ is the algebra of linear combinations of products of basis functions and their derivatives

- $\mathcal{A}$ is closed under addition, subtraction, multiplication and differentiation

- Consider only multilinear forms which can be expressed as integrals over the domain Ω of elements of $\mathcal{A}$

Evaluation of forms

For any $v = \sum c \prod \partial^{|\cdot|} \phi_{(\cdot)}/\partial x_{(\cdot)}$, we have

$$A_i^e = a_e(\phi_{i_1}, \phi_{i_2}, \ldots, \phi_{i_n}) = \int_e v_i \, dx$$

$$= \sum \left(\int_e c \prod \partial^{|\cdot|} \phi_{(\cdot)}/\partial x_{(\cdot)} \, dx \right)_i$$

$$= \sum c'_\alpha \left(\int_E \prod \partial^{|\cdot|} \Phi_{(\cdot)}/\partial X_{(\cdot)} \, dX \right)_{i\alpha}$$

$$= \sum A_{i\alpha}^0 G_e^\alpha,$$

where $A_{i\alpha}^0 = \left(\int_E \prod \partial^{|\cdot|} \Phi_{(\cdot)}/\partial X_{(\cdot)} \, dX \right)_{i\alpha}$ and $G_e^\alpha = c'_\alpha$

Optimization

- Each entry of the element tensor is given by a scalar product

- Find dependencies between entries:
  - Colinearity
  - Coplanarity
  - Edit distance

- Optimization handled by **FErari** (in preparation)

- Operation count can sometimes be reduced to less than one multiply-add pair per entry

- Alternative approach: BLAS level 2 or 3

Reference tensor ($\mathcal{P}^2$ Poisson in 2D)

|					
3 3 3 3	1 0 1 0	0 1 0 1	0 0 0 0	0 -4 0 -4	-4 0 -4 0
1 1 0 0	3 0 0 0	0 -1 0 0	0 4 0 0	0 0 0 0	-4 -4 0 0
0 0 1 1	0 0 -1 0	0 0 0 3	0 0 4 0	0 0 -4 -4	0 0 0 0
0 0 0 0	0 0 4 0	0 4 0 0	8 4 4 8	-8 -4 -4 0	0 -4 -4 -8
0 0 -4 -4	0 0 0 0	0 -4 0 -4	-8 -4 -4 0	8 4 4 8	0 4 4 0
-4 -4 0 0	-4 0 -4 0	0 0 0 0	0 -4 -4 -8	0 4 4 0	8 4 4 8

Minimum spanning tree ($\mathcal{P}^2$ Poisson in 2D)
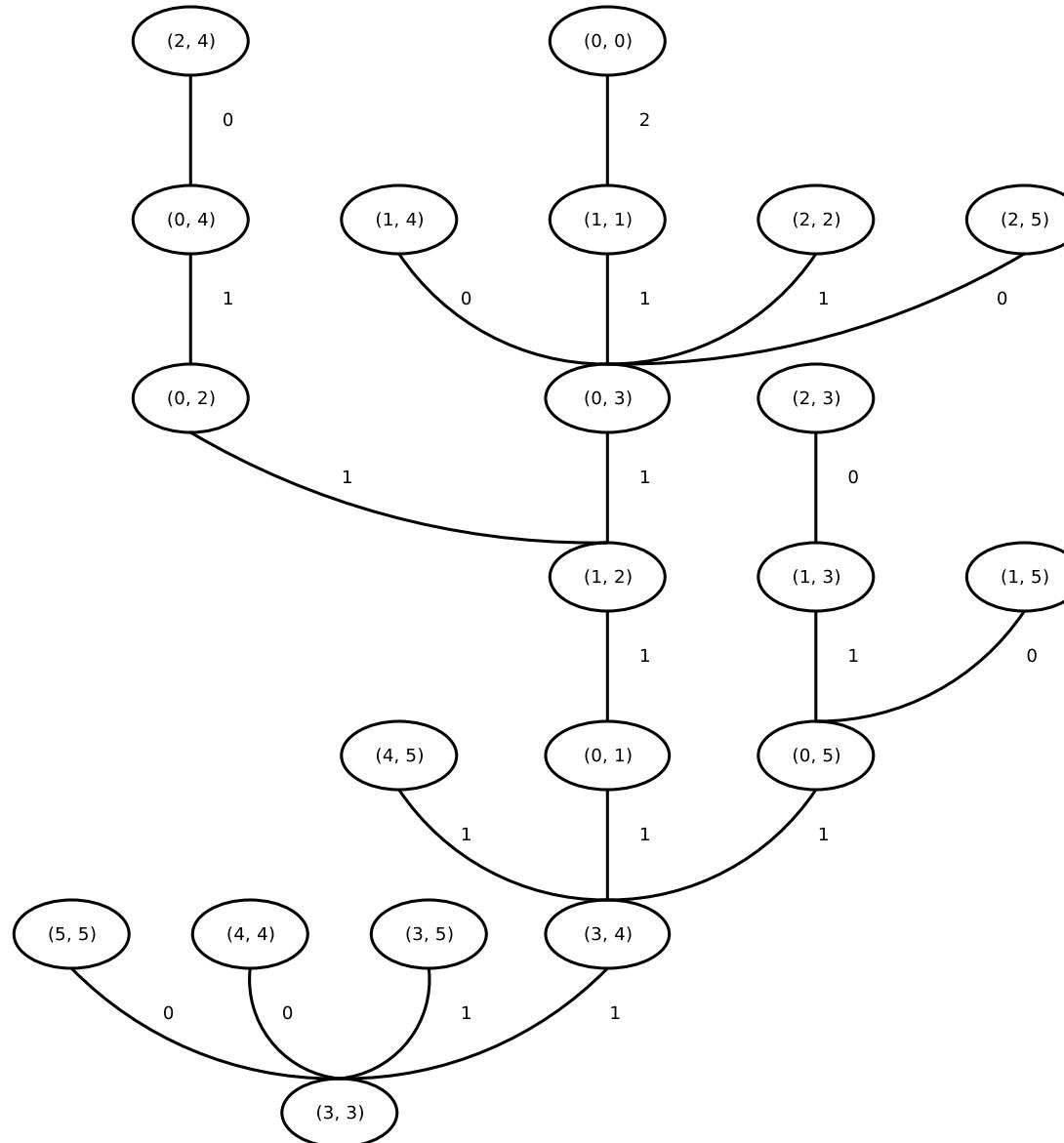
DOLFIN, *a PSE for differential equations*

DOLFIN

- Provides a simple, consistent and intuitive API to **FEniCS**

- Provides solvers for a collection of standard PDEs

- Implemented as a C++ library
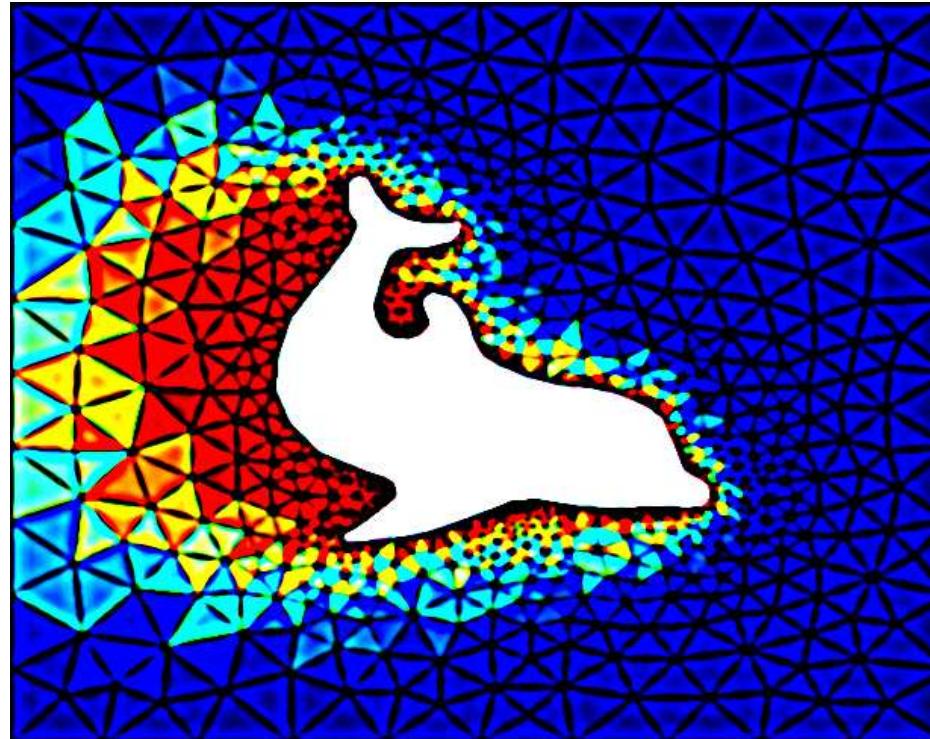
Basic usage

```cpp
#include <dolfin.h>
#include "Poisson.h"
using namespace dolfin;
int main()
{
  ...
  Mesh mesh("mesh.xml.gz");
  Poisson::BilinearForm a;
  Poisson::LinearForm L(f);

  Matrix A;
  Vector x, b;
  FEM::assemble(a, L, A, b, mesh, bc);

  GMRES solver;
  solver.solve(A, x, b);

  return 0;
}
```

Basic concepts (classes)

- Linear algebra
  - `Vector`, `Matrix`, `VirtualMatrix`
  - `LinearSolver`, `Preconditioner`
- Geometry
  - `Mesh`, `Boundary`, `MeshHierarchy`
  - `Vertex`, `Cell`, `Edge`, `Face`
- Finite elements
  - `FiniteElement`, `Function`
  - `BilinearForm`, `LinearForm`
- Ordinary differential equations
  - `ODE`, `ComplexODE`, `ParticleSystem`, `Homotopy`
- Other: `File`, `Progress`, `Event`, `Parameter`

Components of **DOLFIN**

User level — main.cpp (user program)

Modules/solvers:
- Poisson solver/module
- Navier-Stokes solver/module
- Elasticity solver/module
- Convection-diffusion solver/module
- Wave equation solver/module
- Multi-adaptive solver/module

Kernel/tools

Components of **DOLFIN**

Kernel/tools	Multi-adaptivity	Automatic assembly	Form evaluation
	Linear algebra	Mesh and geometry	Function representation
	Basic data structures	Element library	Quadrature rules
Log system/ diagnostics	Parameter database	Input/ output	Basic math

Replacing the backend of **DOLFIN**

Kernel/tools

|		
Multi-adaptivity	Automatic assembly	Form evaluation (FFC)
Linear algebra (PETSc)	Mesh and geometry (Matt?)	Function representation
Basic data structures (STL)	Element library (FIAT)	Quadrature rules (FIAT)
Log system/ diagnostics	Parameter database	Input/ output	Basic math
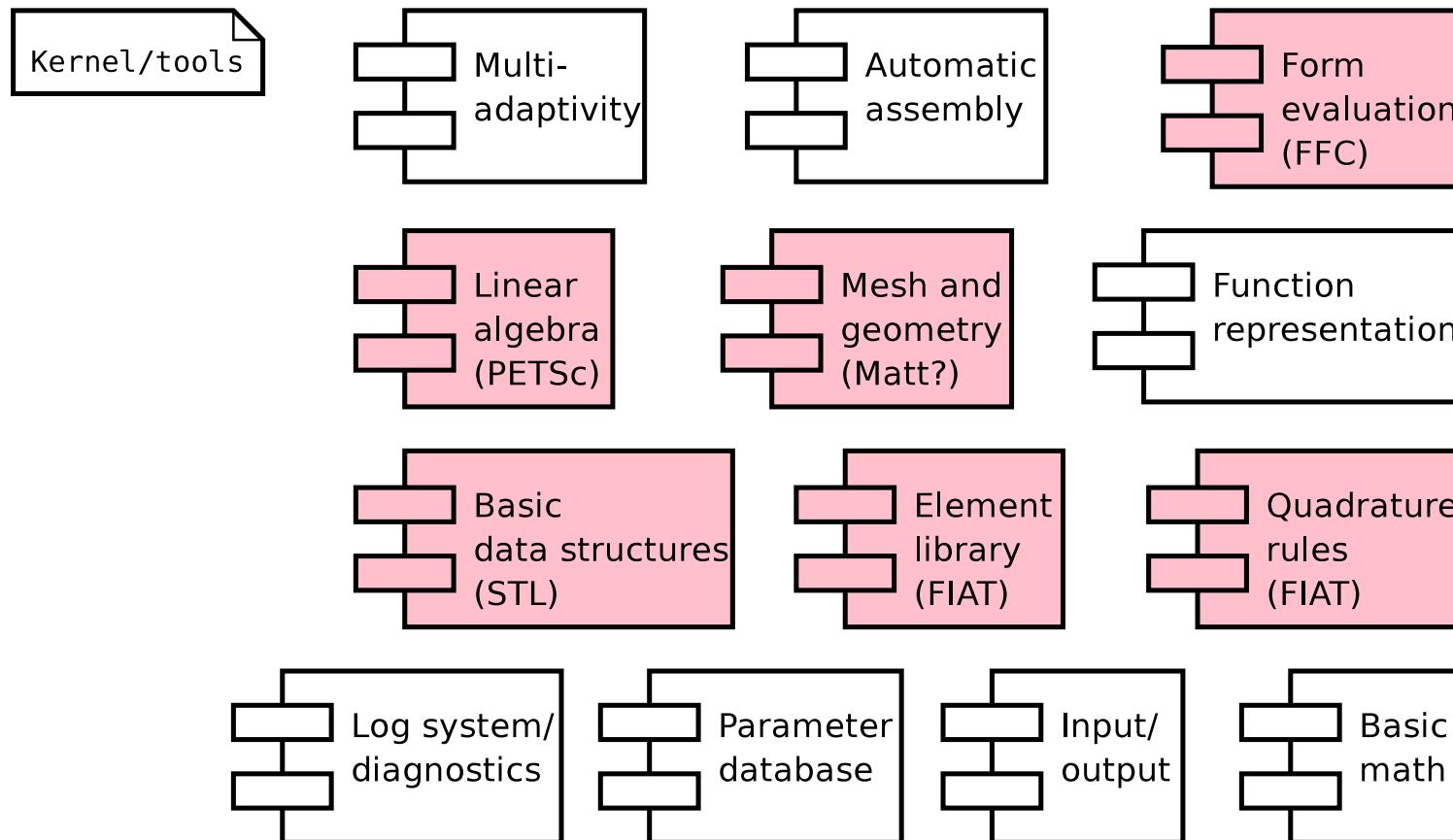
Example: updated elasticity/plasticity
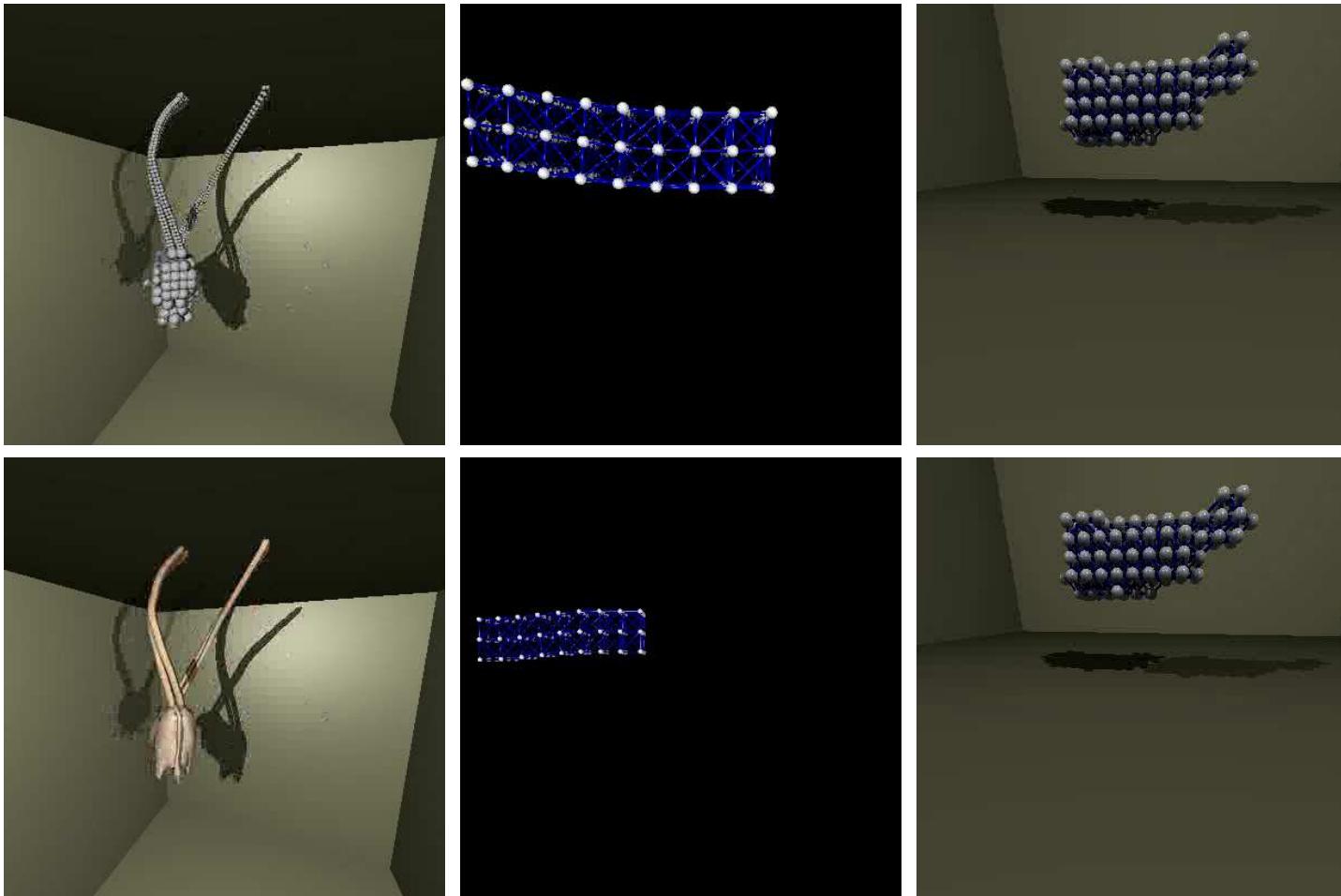

Simulations/animations by Johan Jansson, Computational Technology, Chalmers

Benchmark results

Impressive speedups

FFC vs Quadrature at q = 3 (2D)

Impressive speedups

FFC vs Quadrature at q = 3 (3D)
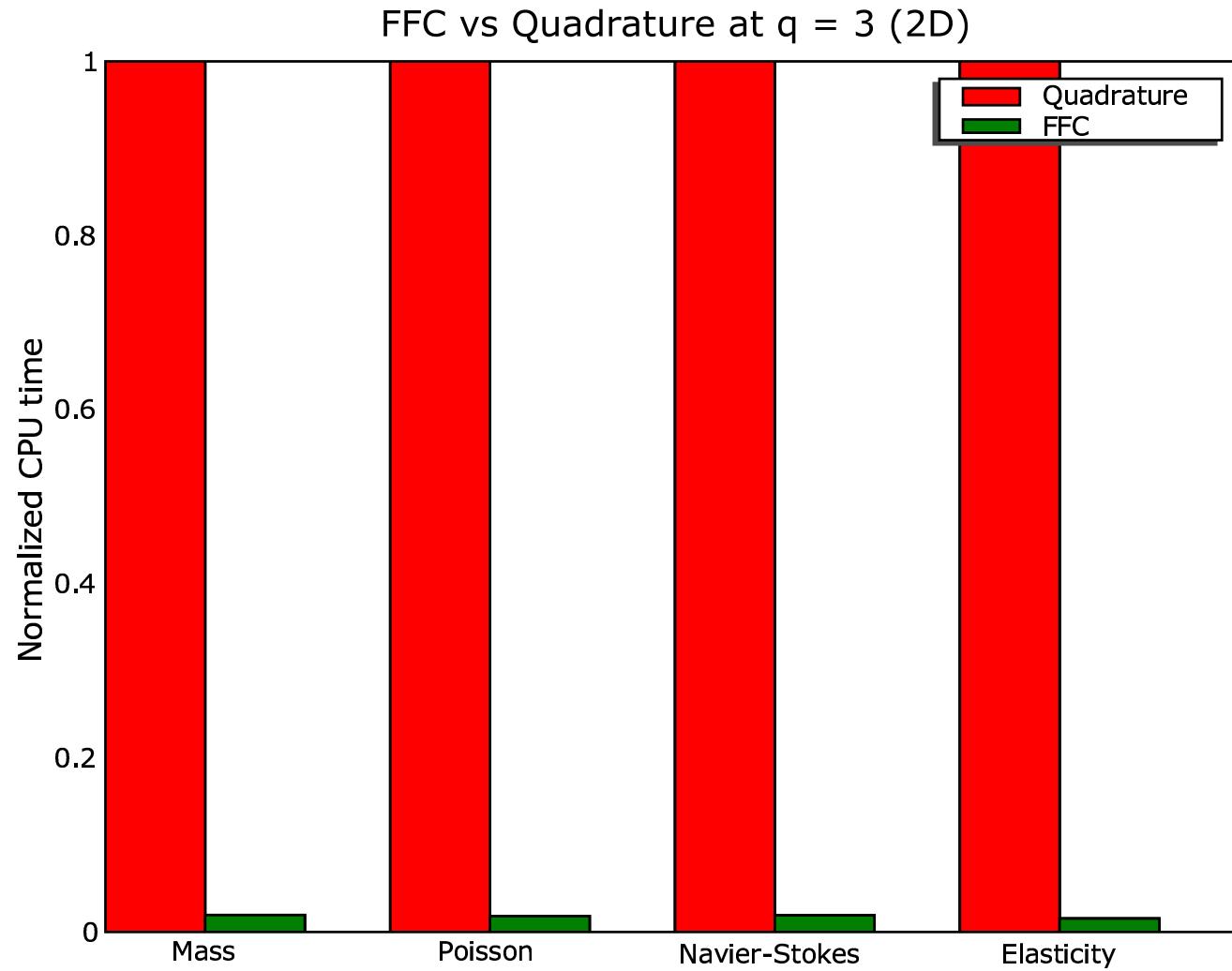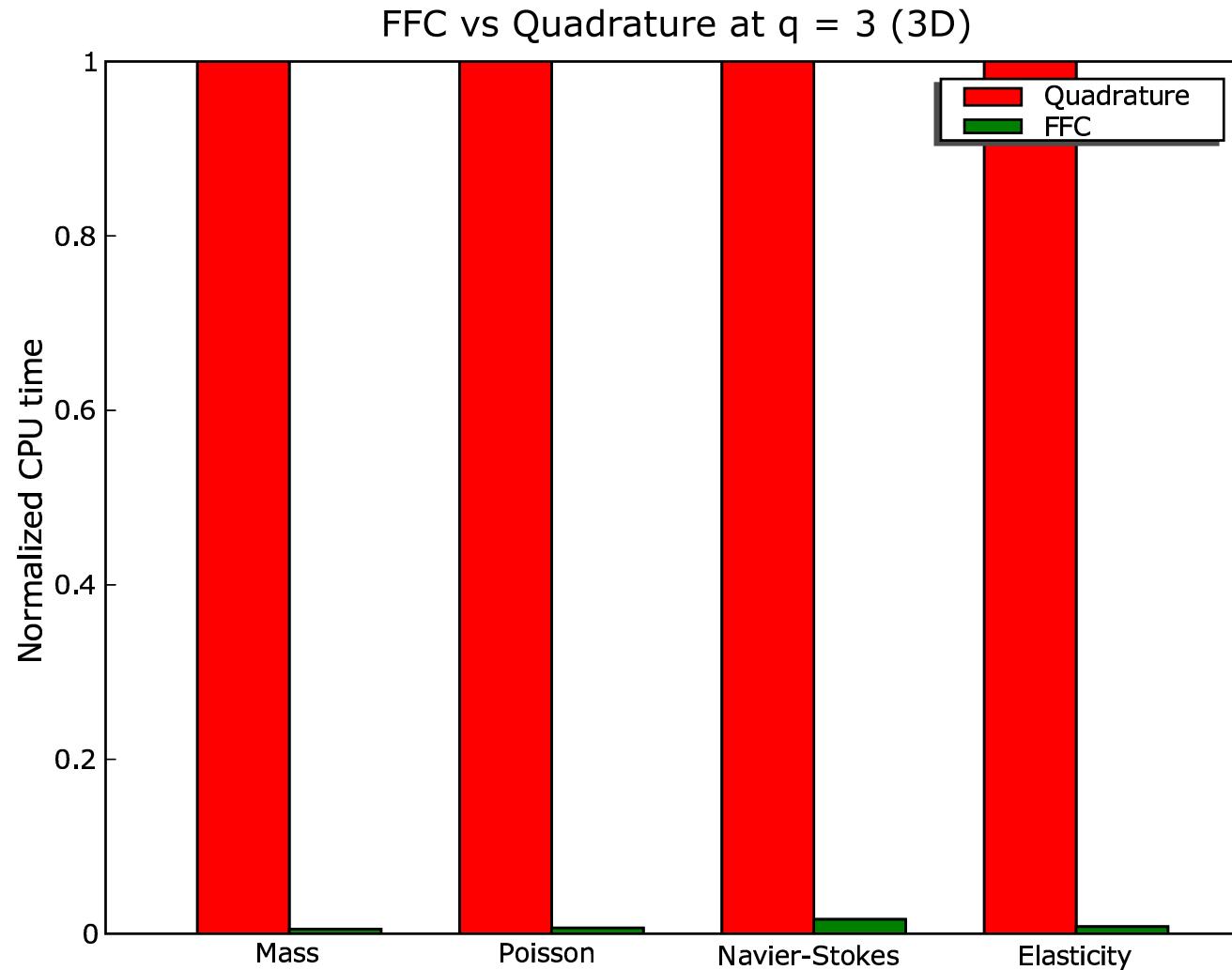
Test case 1: the mass matrix

- Mathematical notation:

$$a(v, u) = \int_\Omega vu \, dx$$

- **FFC** implementation:

```
v = BasisFunction(element)
u = BasisFunction(element)

a = v*u*dx
```

Results

Mass matrix 2D (1 million triangles)

Mass matrix 3D (1 million tetrahedra)
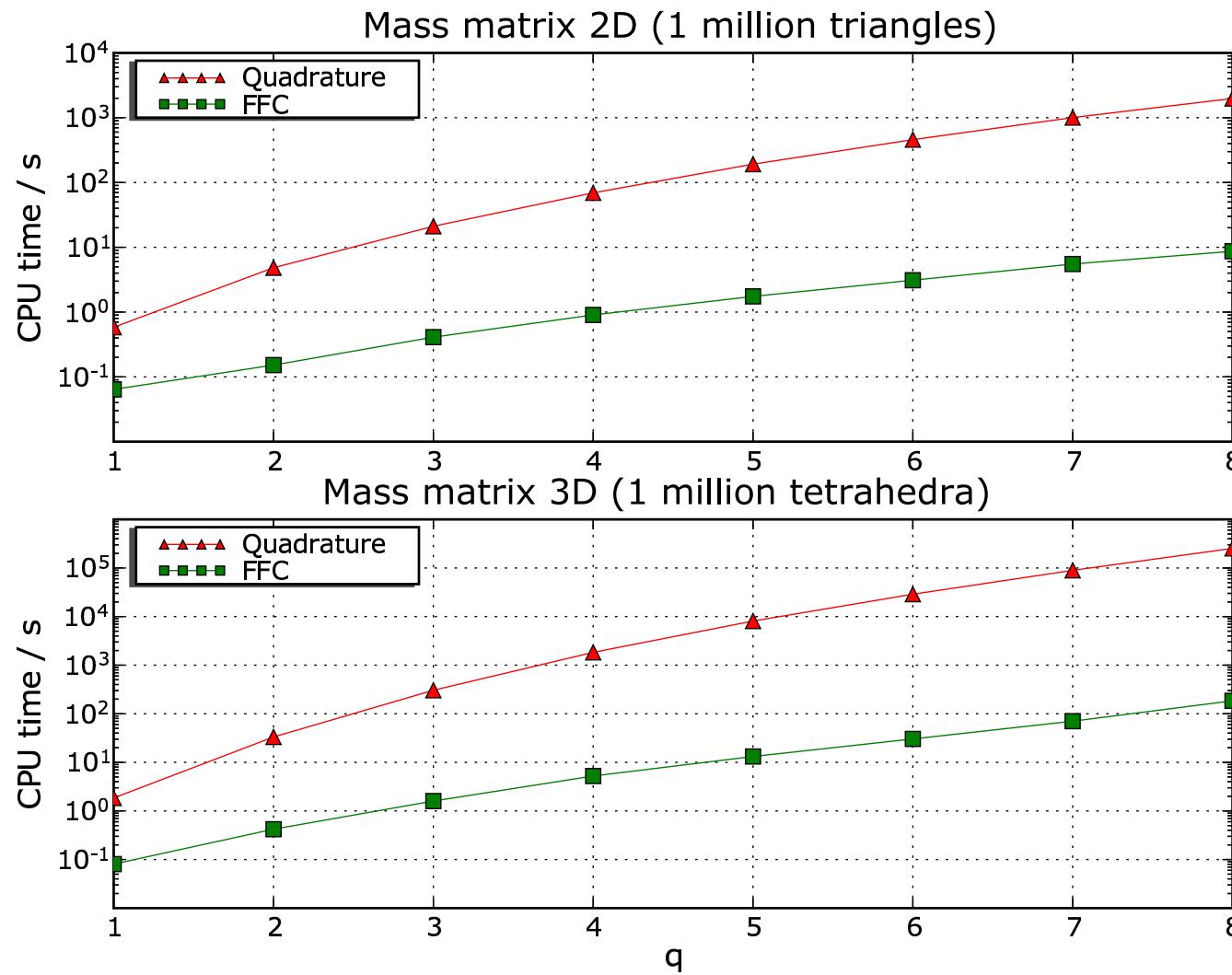
Test case 2: Poisson

- Mathematical notation:

$$a(v, u) = \int_{\Omega} \nabla v \cdot \nabla u \, dx = \int_{\Omega} \sum_{i=1}^{d} \frac{\partial v}{\partial x_i} \frac{\partial u}{\partial x_i} \, dx$$

- **FFC** implementation:

```
v = BasisFunction(element)
u = BasisFunction(element)

a = v.dx(i)*u.dx(i)*dx
```

Results

Poisson 2D (1 million triangles)

Poisson 3D (1 million tetrahedra)
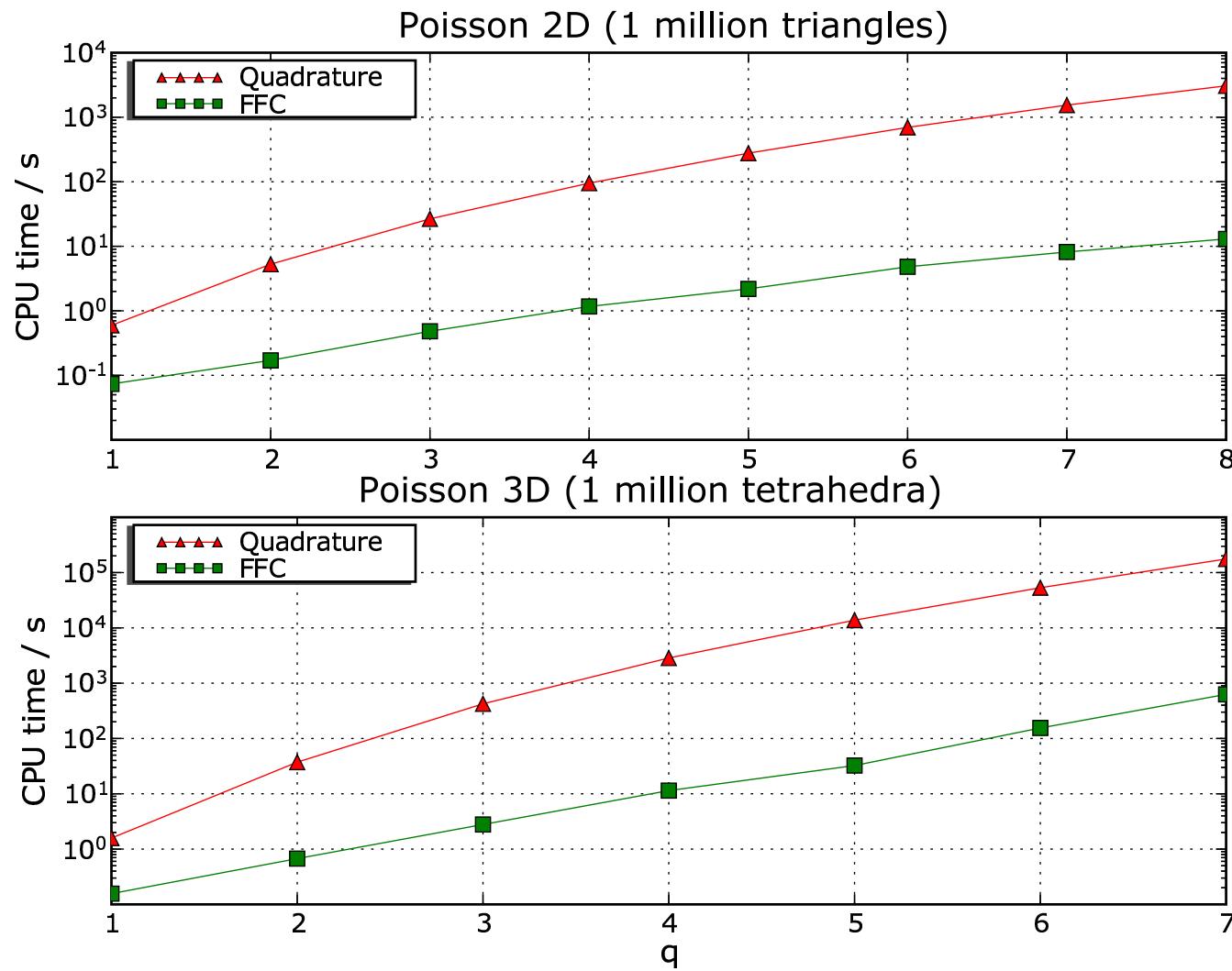
Test case 3: Navier–Stokes

- Mathematical notation:

$$a(v, u) = \int_\Omega v(w \cdot \nabla u)\, dx = \int_\Omega \sum_{i=1}^{d} \sum_{j=1}^{d} v_i w_j \frac{\partial u_i}{\partial x_j}\, dx$$

- **FFC** implementation:

```
v = BasisFunction(element)
u = BasisFunction(element)
w = Function(element)

a = v[i]*w[j]*u[i].dx(j)*dx
```

Results

Navier-Stokes 2D (1 million triangles)

Navier-Stokes 3D (1 million tetrahedra)
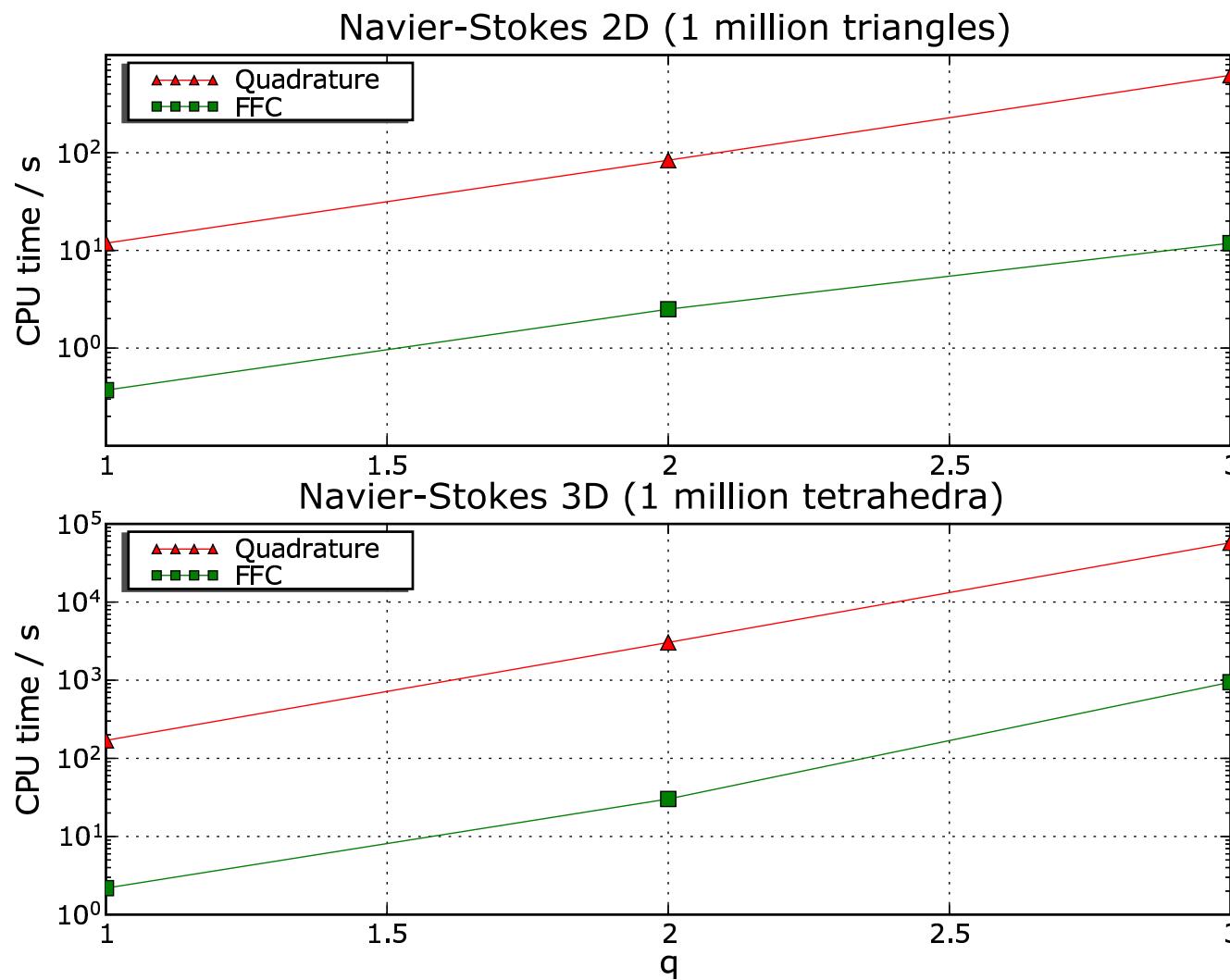
Test case 4: Linear elasticity

- Mathematical notation:

$$
\begin{aligned}
a(v, u) &= \int_\Omega \tfrac{1}{4}(\nabla v + (\nabla v)^\top) : (\nabla u + (\nabla u)^\top)\, dx \\
&= \int_\Omega \sum_{i=1}^{d} \sum_{j=1}^{d} \tfrac{1}{4}\left(\tfrac{\partial v_i}{\partial x_j} + \tfrac{\partial v_j}{\partial x_i}\right)\left(\tfrac{\partial u_i}{\partial x_j} + \tfrac{\partial u_j}{\partial x_i}\right) dx
\end{aligned}
$$

- **FFC** implementation:

```
v = BasisFunction(element)
u = BasisFunction(element)

a = 0.25*(v[i].dx(j) + v[j].dx(i)) * \
         (u[i].dx(j) + u[j].dx(i)) * dx
```

Results

Elasticity 2D (1 million triangles)

Elasticity 3D (1 million tetrahedra)
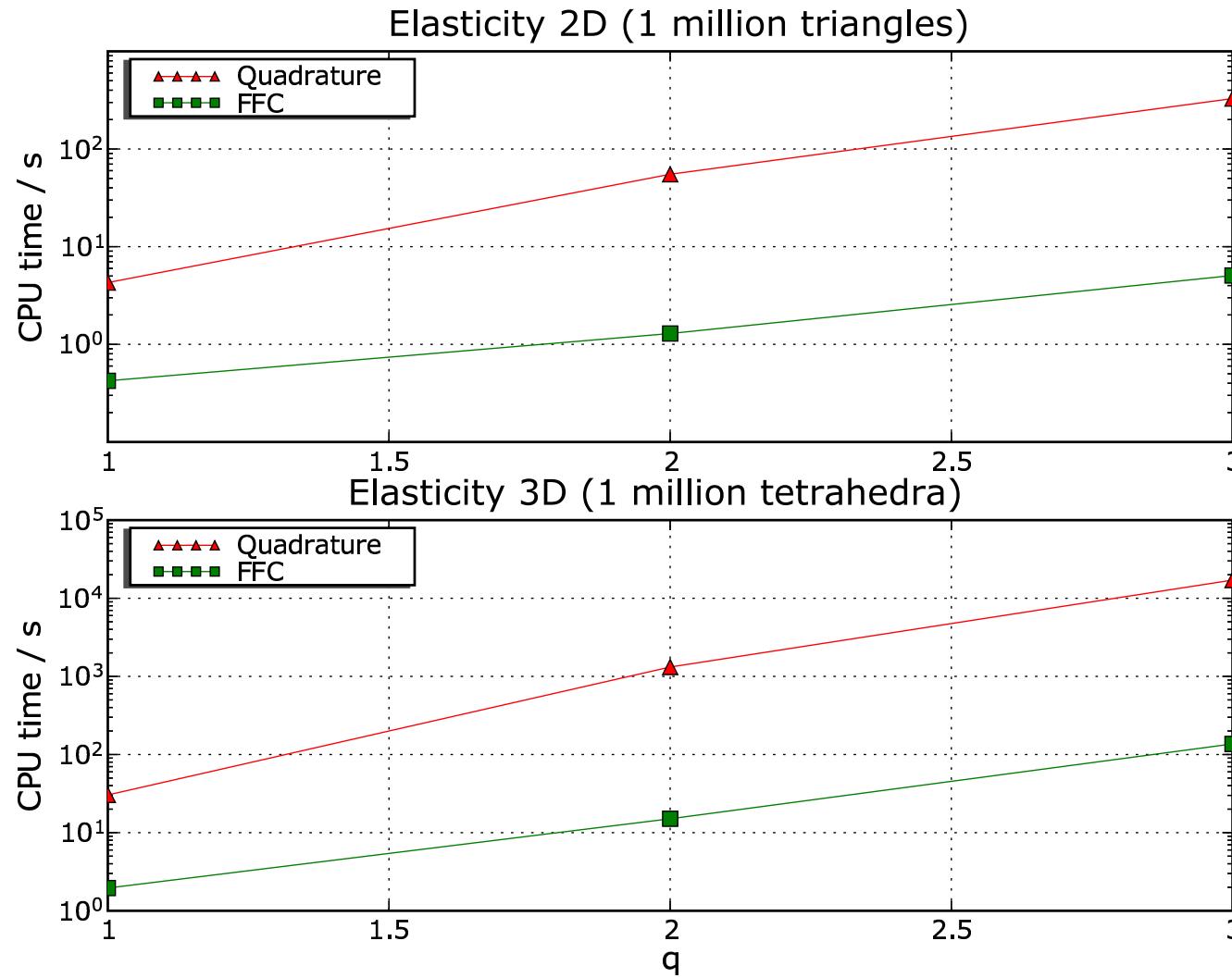
Speedup

Form	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$	$q = 8$
Mass 2D	9.1	31.8	51.5	76.7	109.9	147.8	182.2	227.9
Mass 3D	23.0	79.0	190.5	350.6	612.1	951.0	1270.9	1368.5
Poisson 2D	8.1	30.9	55.2	81.6	126.9	144.6	189.0	236.1
Poisson 3D	10.1	55.4	152.1	249.9	425.2	343.8	280.6	—
Navier–Stokes 2D	32.0	33.5	52.3	—	—	—	—	—
Navier–Stokes 3D	77.7	100.7	60.9	—	—	—	—	—
Elasticity 2D	10.1	42.7	64.8	—	—	—	—	—
Elasticity 3D	15.5	87.5	125.0	—	—	—	—	—

- Impressive speedups but far from optimal

- Data access costs more than flops

- Solution: build arrays and call BLAS (Level 2 or 3)

Code bloat

Lines of code generated
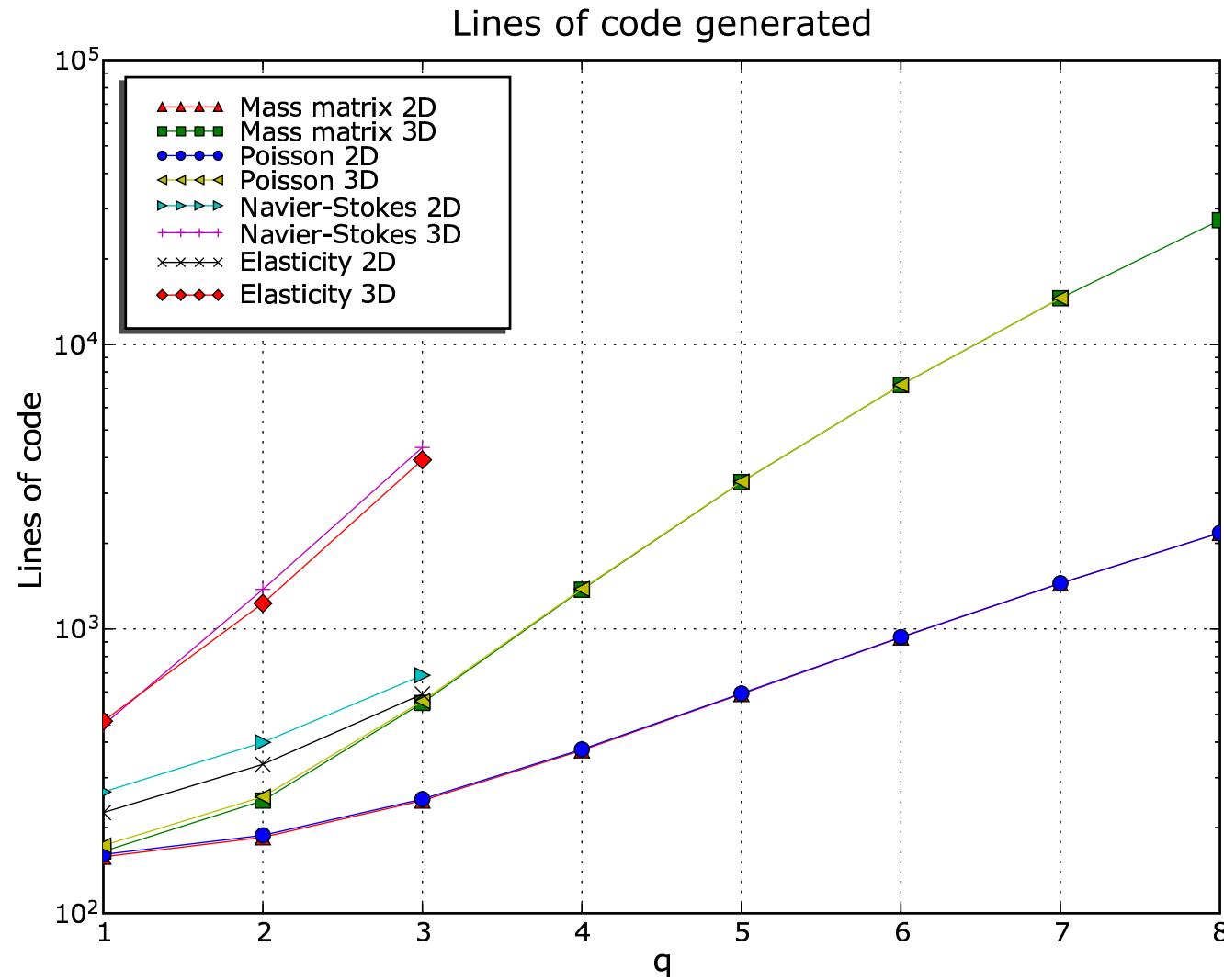
Future directions for **FEniCS**

Future directions for **FEniCS**

- New parallel mesh component (05)

- Completely parallel assembly/solve (05)

- Optimize code generation through FErari (05/06)

- Optimize code generation through BLAS (05/06)

- Optimize compiler to reduce compile time (05/06)

- Complete support for non-standard elements (05/06):
  Crouzeix–Raviart, *Raviart–Thomas*, *Nedelec*,
  Brezzi–Douglas–Marini, *Brezzi–Douglas–Fortin–Marini*,
  Arnold–Winther, *Taylor–Hood*, …

- Automatic generation of dual problems (06)

- Automatic generation of error estimates (06)

- **Manuals**, tutorials, mini-courses (05/06)

References

- *A compiler for variational forms*
  Kirby/Logg, submitted to TOMS (2005)

- *Optimizing the evaluation of finite element matrices*
  Kirby/Knepley/Logg/Scott, SISC (2005)

- *Topological optimization of the evaluation of finite element matrices*
  Kirby/Logg/Scott/Terrel, submitted to SISC (2005)

- *FIAT: A new paradigm for computing finite element basis functions*
  Kirby, TOMS (2004)

- *Optimizing FIAT with the Level 3 BLAS*
  Kirby, submitted to TOMS (2005)

- *Evaluation of the action of finite element operators*
  Kirby/Knepley/Scott, submitted to BIT (2004)

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

Additional slides

The Automation of CMM
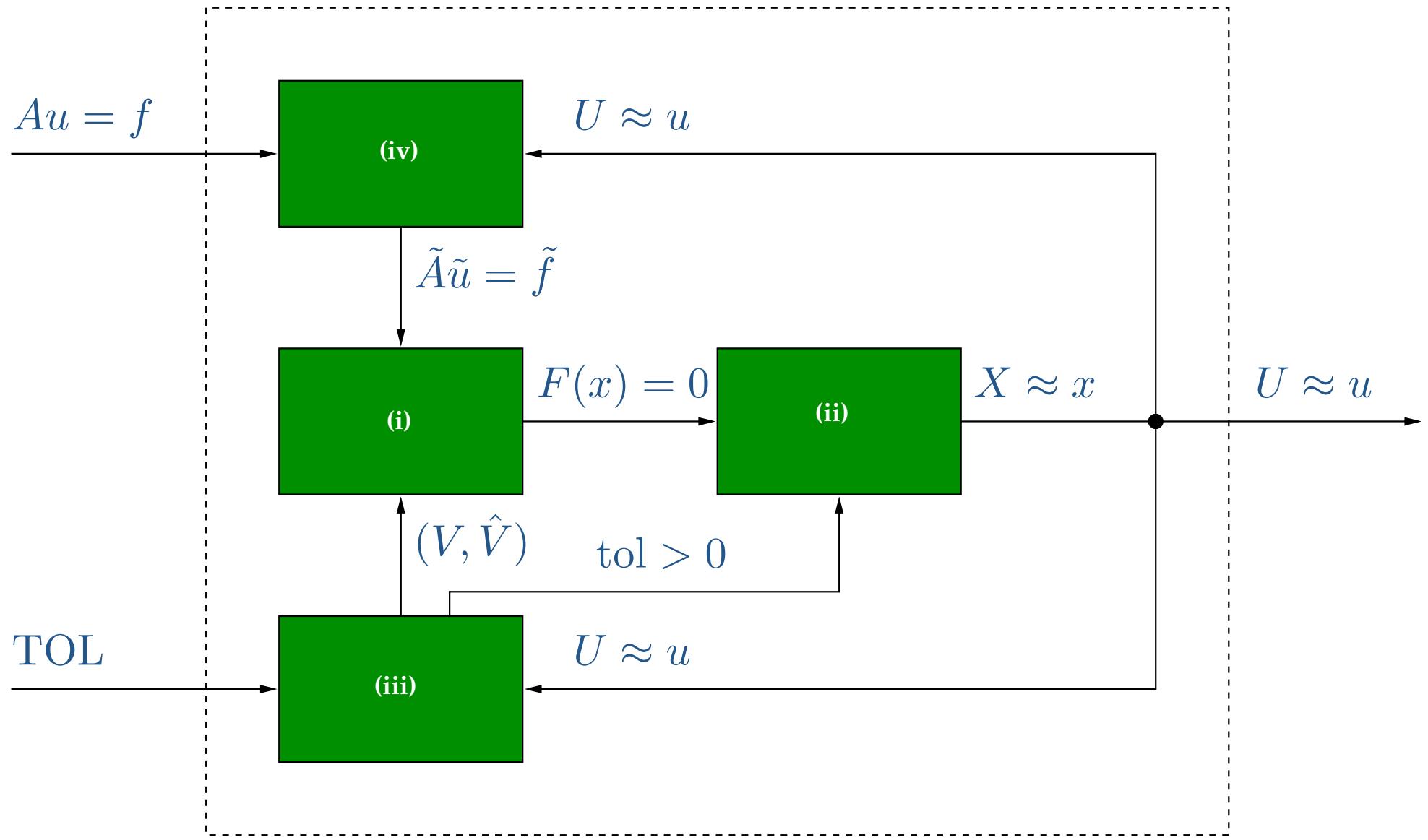
Example 3: Navier–Stokes

- Form:

$$a(v, u) = \int_\Omega v \cdot (w \cdot \nabla) u \, dx$$

- Evaluation:

$$A_i^e = \int_e \phi_{i_1} \cdot (w \cdot \nabla) \phi_{i_2} \, dx$$

$$= \det F_e' \frac{\partial X_{\alpha_3}}{\partial x_{\alpha_1}} w_{\alpha_2} \int_E \Phi_{i_1}[\beta] \Phi_{\alpha_2}[\alpha_1] \frac{\partial \Phi_{i_2}[\beta]}{\partial X_{\alpha_3}} \, dX = A_{i\alpha}^0 G_e^\alpha$$

with $A_{i\alpha}^0 = \int_E \Phi_{i_1}[\beta] \Phi_{\alpha_2}[\alpha_1] \frac{\partial \Phi_{i_2}[\beta]}{\partial X_{\alpha_3}} \, dX$ and

$G_e^\alpha = \det F_e' \frac{\partial X_{\alpha_3}}{\partial x_{\alpha_1}} w_{\alpha_2}$

Complexity of form evaluation

- Basic assumptions:
  - Bilinear form: $|i| = 2$
  - Exact integration of forms

- Notation:
  - q: polynomial order of basis functions
  - p: total polynomial order of form
  - d: dimension of Ω
  - n: dimension of function space $(n \sim q^d)$
  - N: number of quadrature points $(N \sim p^d)$
  - n_C: number of coefficients
  - n_D: number of derivatives

Complexity of tensor contraction

- Need to evaluate $A_i^e = A_{i\alpha}^0 G_e^\alpha$

- Rank of G_e^α is $n_C + n_D$

- Number of elements of A_i^e is n^2

- Number of elements of G_e^α is $n^{n_C} d^{n_D}$

- Total cost:

$$T_C \sim n^2 n^{n_C} d^{n_D} \sim (q^d)^2 (q^d)^{n_C} d^{n_D} \sim \underline{q^{2d+n_C d} d^{n_D}}$$

Complexity of quadrature

- Need to evaluate A_i^e at $N \sim p^d$ quadrature points
- Total order of integrand is $p = 2q + n_C q - n_D$
- Cost of evaluating integrand is $\sim n_C + n_D d + 1$

- Total cost:

$$
\begin{aligned}
T_Q \ &\sim \ n^2 N(n_C + n_D d + 1) \sim (q^d)^2 p^d (n_C + n_D d + 1) \\
&\sim \ \underline{q^{2d}(2q + n_C q - n_D)^d (n_C + n_D d + 1)}
\end{aligned}
$$

Tensor contraction vs quadrature

$$
\begin{aligned}
T_C &\sim q^{2d+n_C d} d^{n_D} \\
T_Q &\sim q^{2d}(2q + n_C q - n_D)^d (n_C + n_D d + 1)
\end{aligned}
$$

Speedup:

$$
T_Q/T_C \sim \frac{(2q + n_C q - n_D)^d (n_C + n_D d + 1)}{q^{n_C d} d^{n_D}}
$$

- Rule of thumb: tensor contraction wins for $n_C = 0, 1$

- Mass matrix ($n_C = n_D = 0$): $T_Q/T_C \sim (2q)^d$

- Poisson ($n_C = 0, n_D = 2$): $T_Q/T_C \sim (2q - 2)^d (2d + 1)/d^2$

- Not clear that tensor contraction wins for the stabilization term of Navier–Stokes: $(w \cdot \nabla)u \, (w \cdot \nabla)v$

- Need an intelligent system that can do both!