Extending and Optimizing the FEniCS Form Compiler

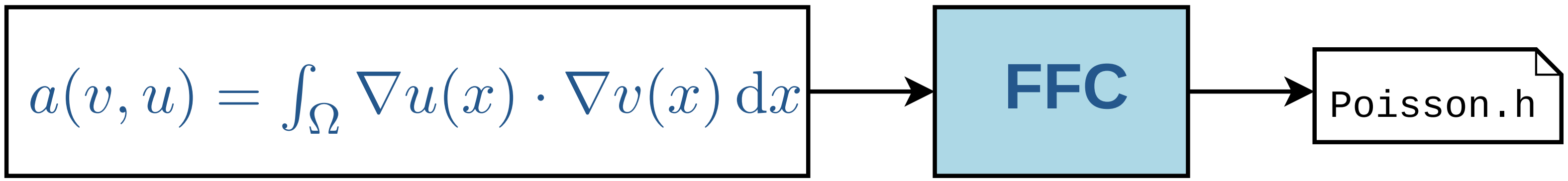
Anders Logg

`logg@tti-c.org`

Toyota Technological Institute at Chicag

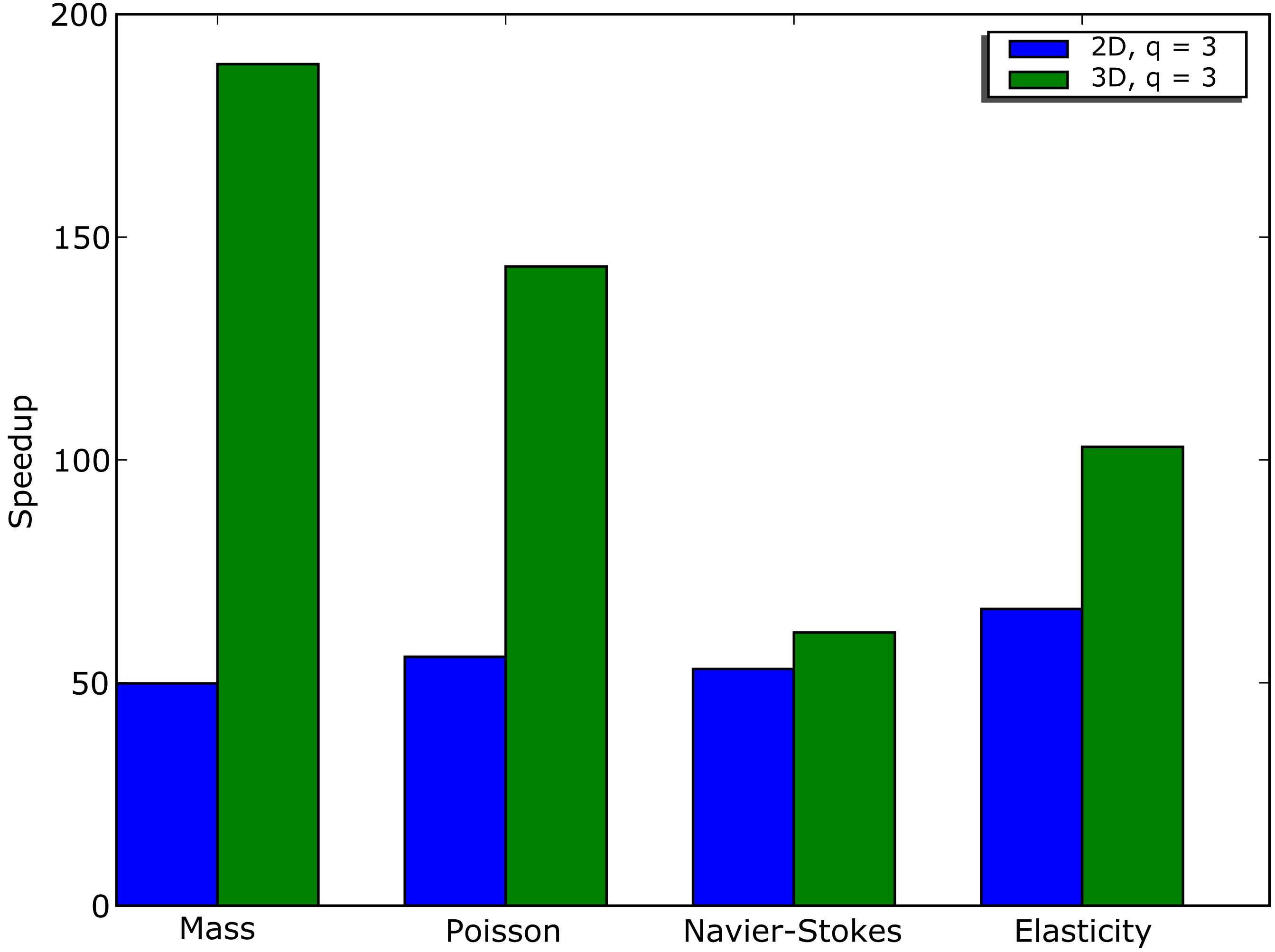
FFC: the FEniCS Form Compiler

- 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++




```
# ffc [-l language] [-f option] poisson.form
```

Benchmark results: impressive speedups



Extensions and optimizations (new since 0.2.0)

- Improved Python interface
- Extensions of the form language:
  - `D`, `grad`, `div`, `rot` (`curl`)
  - `rank`, `trace`, `dot`, `cross`
  - (Factorization of terms with equal signatures)
- Support for arbitrary mixed elements
- New (level 2) BLAS mode (option `-f blas`)

Other improvements:

- A first version of a manual
- New XML output format

Outline

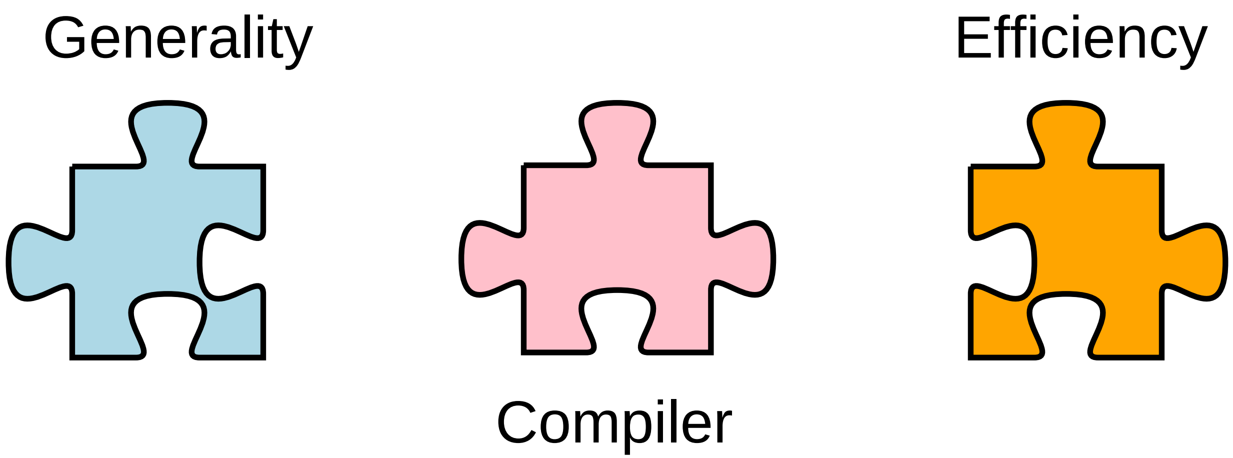
- Introduction to **FFC**
  - Basic usage
  - Tensor-representation of forms
  - Benchmarks
- Extensions and optimizations
  - Language extensions
  - Mixed elements
  - The new BLAS mode
- Future plans for **FFC**

Introduction to ***FFC***

Design goals

- 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 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
  - 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):

```
galerkin
element = FiniteElement("Lagrange", "triangle", 1)

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

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

-uu-:**-F1  Poisson.form       (Python CVS:1.21)--L8--C10--All----
```

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

```
>> ffc Poisson.form
```

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

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 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 $u \in H^1_0(\Omega)$ such that

$$\int_\Omega \nabla v(x) \cdot \nabla u(x) \,\mathrm{d}x = \int_\Omega v(x) f(x) \,\mathrm{d}x \quad \text{for all } v \in H^1_0(\Omega)$$

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

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

with $a : \hat{V} \times V \rightarrow \mathbb{R}$ a *bilinear form* and $L : \hat{V} \rightarrow \mathbb{R}$ 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 $V = \mathrm{span}\{\phi_i\}_{i=1}^M$ and $\hat{V} = \mathrm{span}\{\hat{\phi}_i\}_{i=1}^M$, we obtain the linear system

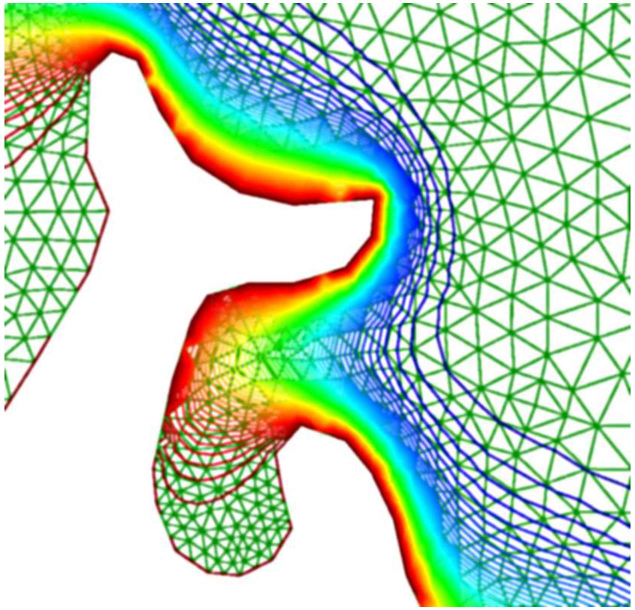
$$Ax = b$$

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

$A = 0$
`for all elements` $e \in \mathcal{T}$
    A += 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

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

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:

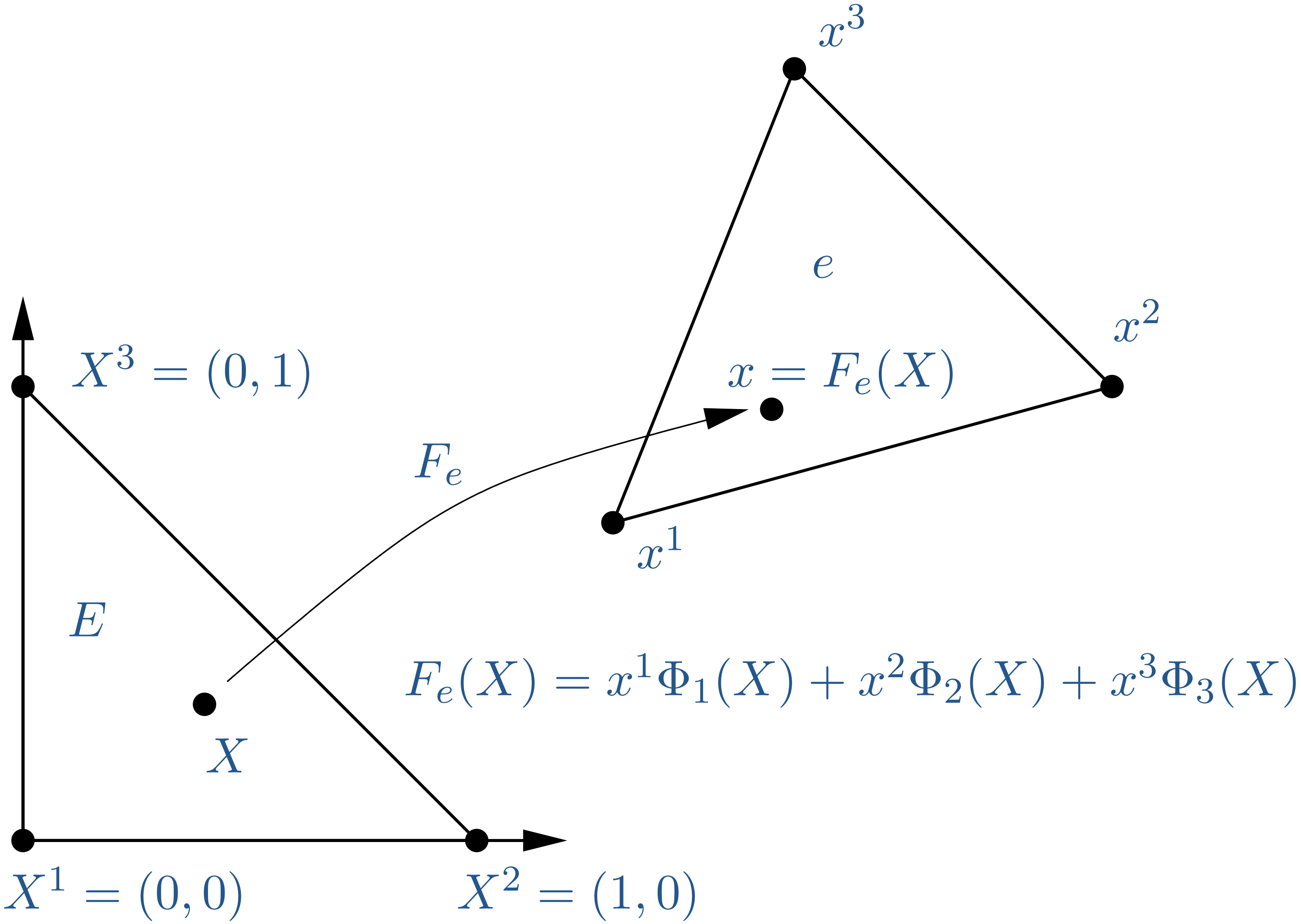
$$A_i^e = A_{i\alpha}^0 G_e^\alpha$$

- 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 $F_e : E \rightarrow e$



Example 1: the mass matrix

- Form:

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

- Evaluation:

$$\begin{aligned} A_i^e &= \int_e \phi_{i_1}\phi_{i_2}\,\mathrm{d}x \\ &= \det F_e' \int_E \Phi_{i_1}(X)\Phi_{i_2}(X)\,\mathrm{d}X = A_i^0 G_e \end{aligned}$$

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

Example 2: Poisson

- Form:

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

- Evaluation:

$$A_i^e = \int_e \nabla \phi_{i_1}(x) \cdot \nabla \phi_{i_2}(x) \, \mathrm{d}x$$
$$= \det F_e' \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_E \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, \mathrm{d}X = A_{i\alpha}^0 G_e^\alpha$$

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

Test cases

- Mass matrix:

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

- Poisson:

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

- Navier–Stokes (nonlinear term):

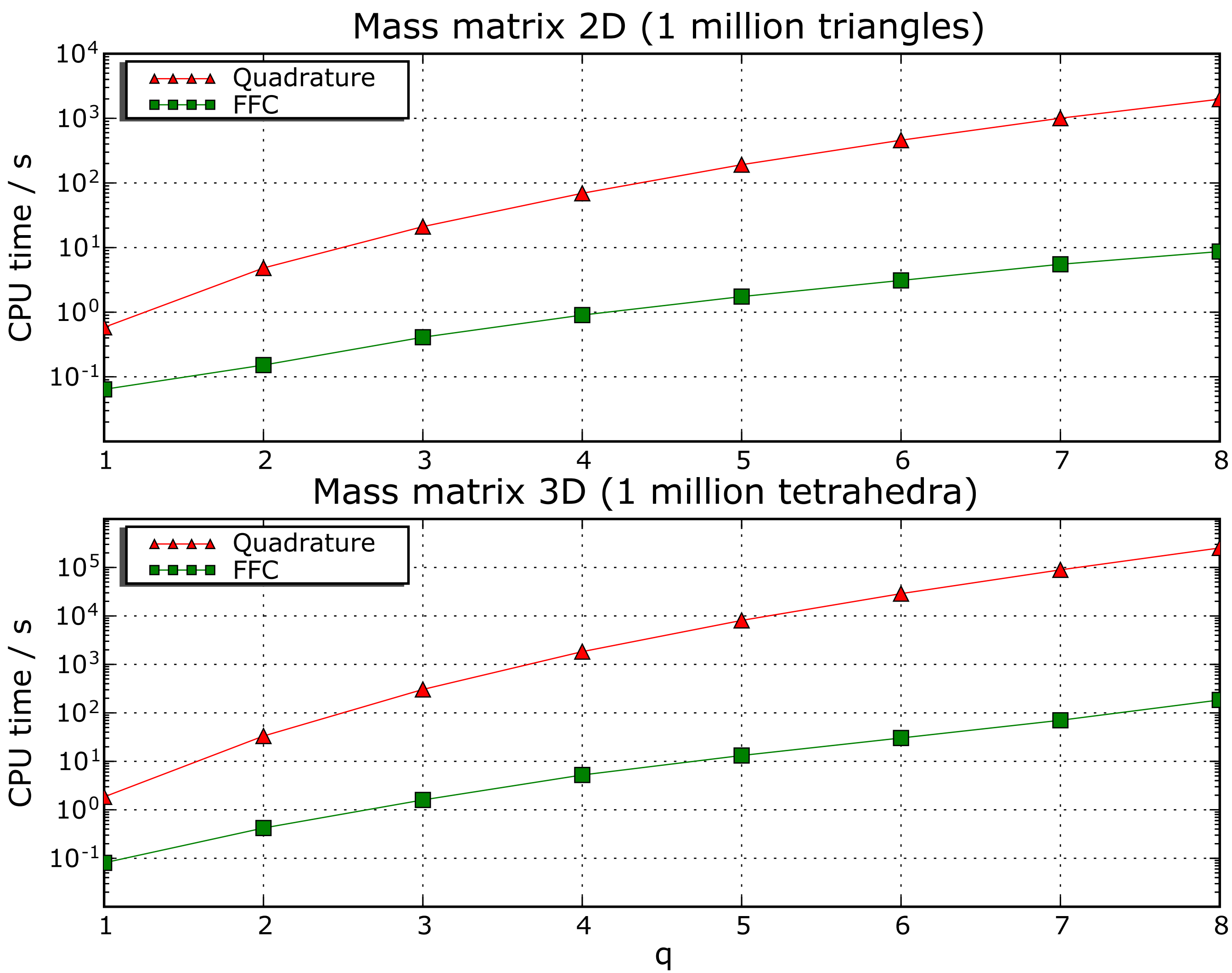
$$a(v, u) = \int_\Omega v \cdot (u \cdot \nabla) u \, \mathrm{d}x$$

- Linear elasticity (the strain-strain term):

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

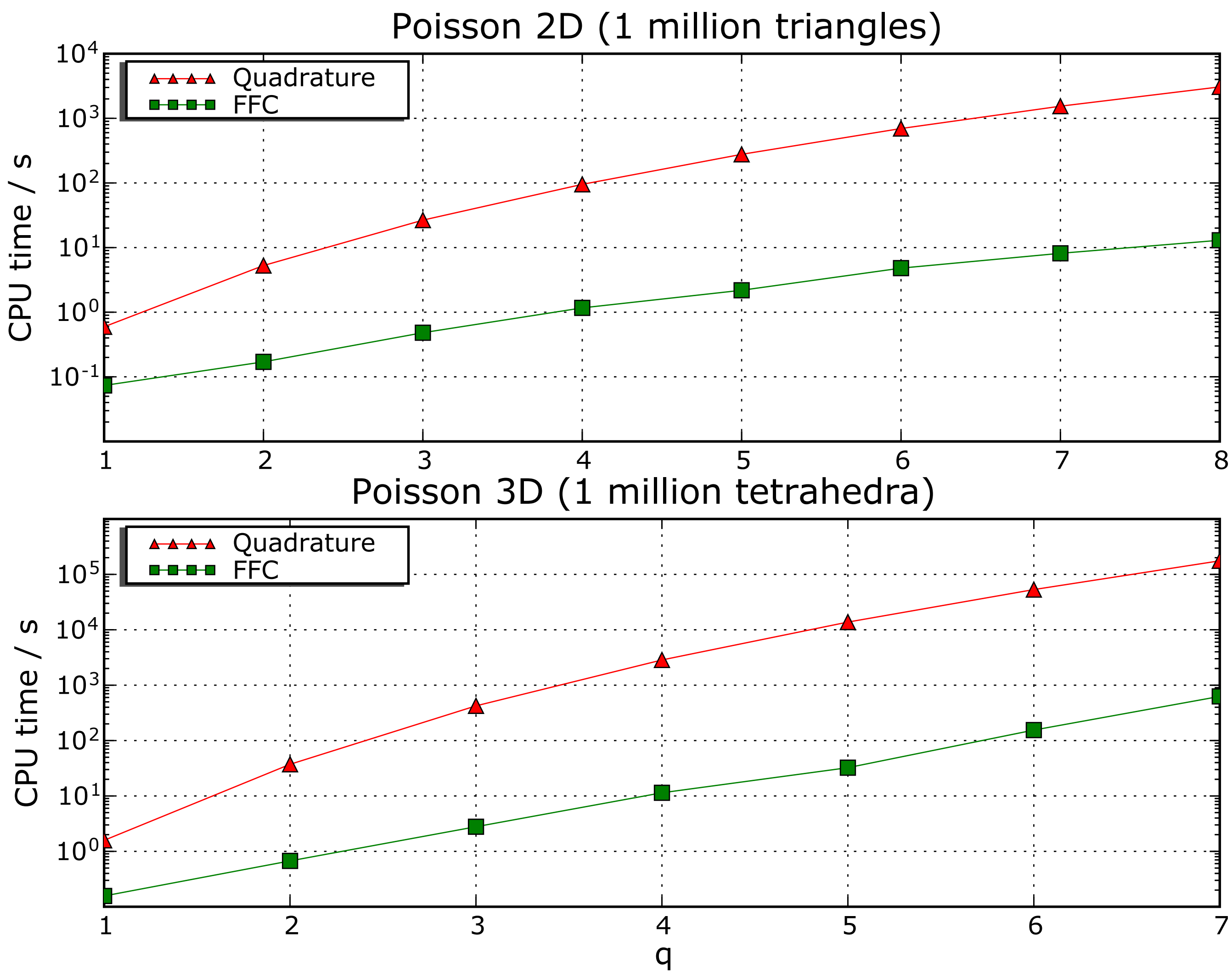
where $\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top)$

a = v*u*dx

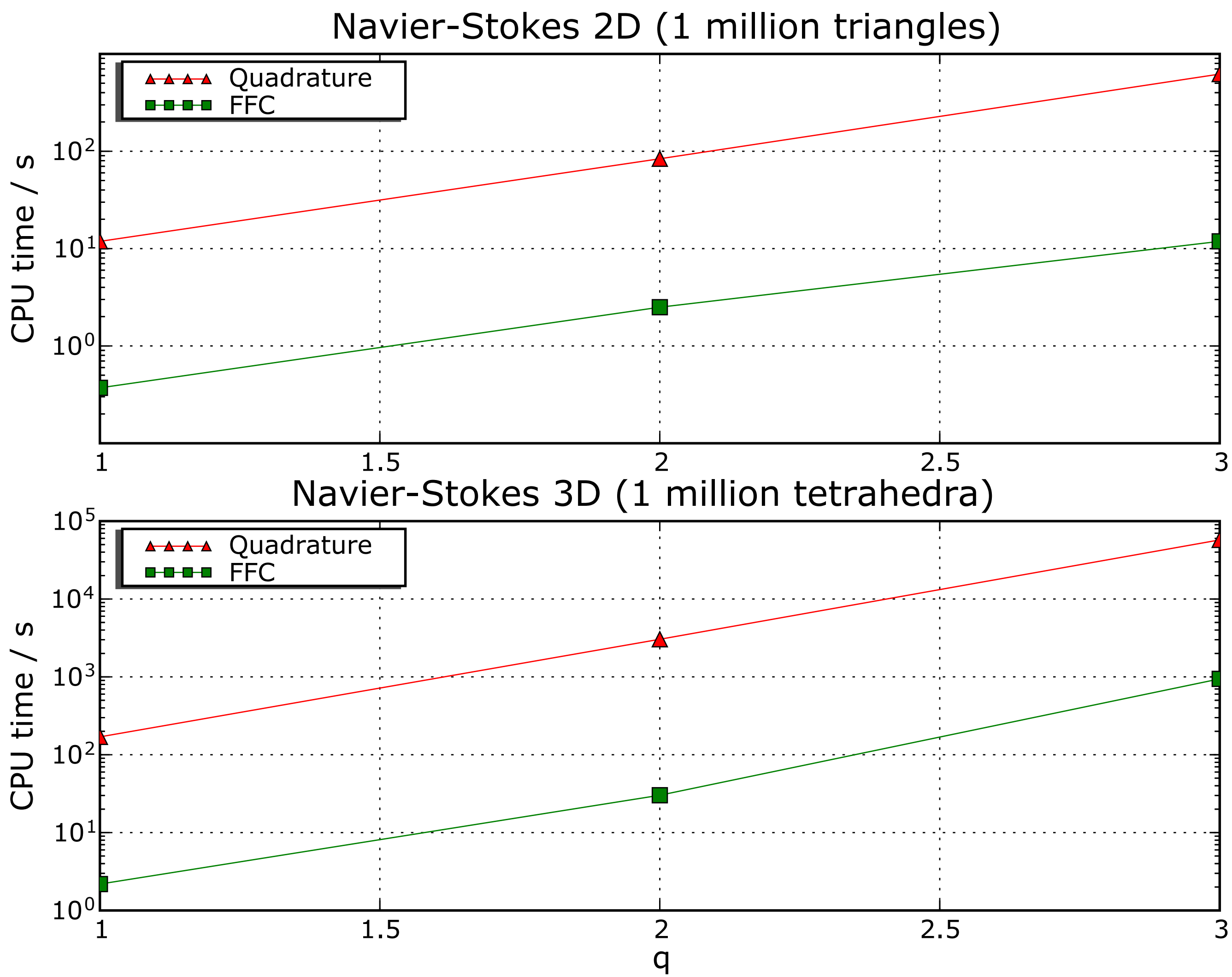


Mass matrix 2D (1 million triangles)
Quadrature
FFC
CPU time / s
Mass matrix 3D (1 million tetrahedra)
Quadrature
FFC
CPU time / s
q

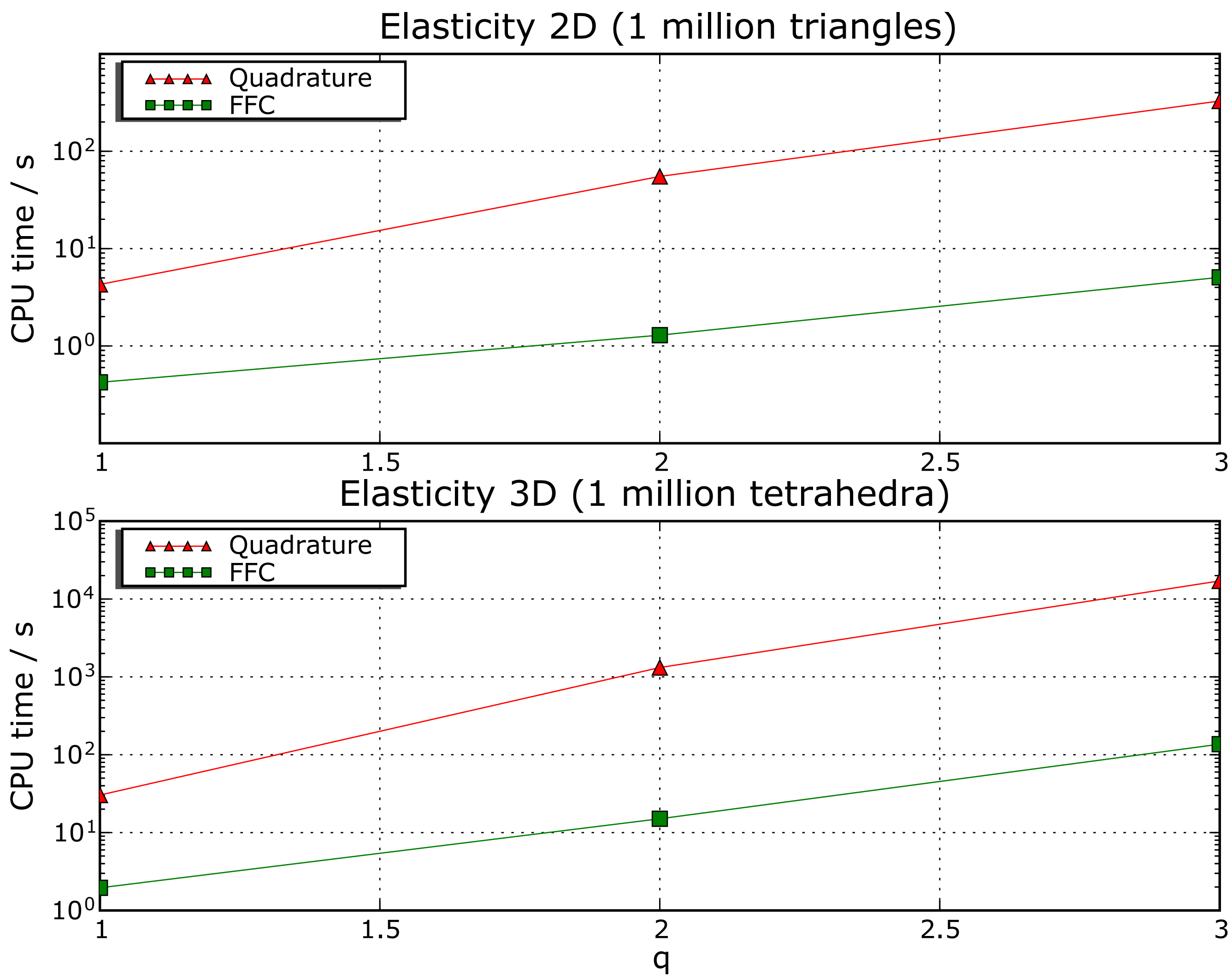
a = dot(grad(v), grad(u))



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



a = dot(eps(v), eps(u))*dx



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)

Language extensions

Basic form language

- Basic scalar operators:
  - Scalar addition:`v + w`, subtraction: `v - w`
  - Scalar multiplication: `v*w`, `c*v`
  - Scalar division: `v/c`
- Component access: `v[i]`
- Index summation: `v[i]*w[i]`
- Differentiation: `v.dx(i)`
- Integration: `*dx`, `*ds`

New operators

- Vector operators:
  - Vectorization: `vec(v)`, vector length (`len(v)`)
  - Products: `dot(v, w)`, `cross(v, w)`
- Tensor operators:
  - Transpose: `transp(A)`, trace: `trace(A)`
  - Matrix-vector product: `mult(A, v)`
  - Matrix-matrix product: `mult(A, B)`
- Differential operators:
  - Scalar partial derivative: `D`(v, i)
  - Vector differential operators: `grad`(v), `div`(v), `rot`(v)

Need to factorize forms in component-form

Poisson in tensor-notation:

$$A_i^K = \int_K \frac{\partial \phi_{i_1}}{\partial x_\beta} \frac{\partial \phi_{i_2}}{\partial x_\beta} \, \mathrm{d}x$$

$$= \det F_K' \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_E \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, \mathrm{d}X = A^0_{i\alpha} G_e^\alpha$$

Poisson in component form:

$$A_i^e = \int_e \frac{\partial \phi_{i_1}}{\partial x_1} \frac{\partial \phi_{i_2}}{\partial x_1} + \frac{\partial \phi_{i_1}}{\partial x_2} \frac{\partial \phi_{i_2}}{\partial x_2} + \frac{\partial \phi_{i_1}}{\partial x_3} \frac{\partial \phi_{i_2}}{\partial x_2} \, \mathrm{d}x$$

$$= (A^0_{i\alpha} \tilde{G}_e^\alpha)_1 + (A^0_{i\alpha} \tilde{G}_e^\alpha)_2 + (A^0_{i\alpha} \tilde{G}_e^\alpha)_3 = A^0_{i\alpha}((\tilde{G}_e^\alpha)_1 + (\tilde{G}_e^\alpha)_2 + (\tilde{G}_e^\alpha)_3)$$

$$= A^0_{i\alpha} G_e^\alpha$$

Factorization

- Expensive and difficult to compare numerical tensors
- Compute a unique string *signature* for each reference tensor and compare signatures
- *Hard signatures* match iff reference tensors are equal
- *Soft signatures* match iff hard signatures match after reordering of indices
- Factorize terms with equal soft signatures

Hard and soft signatures for Poisson:

```
{Lagrange finite element of degree 1 on a Triangle;i0;[];[(d/dXa)]}* \
{Lagrange finite element of degree 1 on a Triangle;i1;[];[(d/dXa)]}

{Lagrange finite element of degree 1 on a Triangle;i0;[];[(d/dXa0)]}* \
{Lagrange finite element of degree 1 on a Triangle;i1;[];[(d/dXa1)]}
```

User-defined operators (example by Johan Jansson)

```
lmbda = Constant()
mu    = Constant()

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

def E(e, lmbda, mu):
    return 2.0*mult(mu, e) + \
           mult(lmbda, mult(trace(e), Identity(d)))

sigma = E(epsilon(u), lmbda, mu)

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

Mixed elements

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**:

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

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

Stokes with mixed elements (in **FFC**)

$$\int_\Omega \nabla v \cdot \nabla u - (\nabla \cdot v)\, p = \int_\Omega v f \,\mathrm{d}x$$

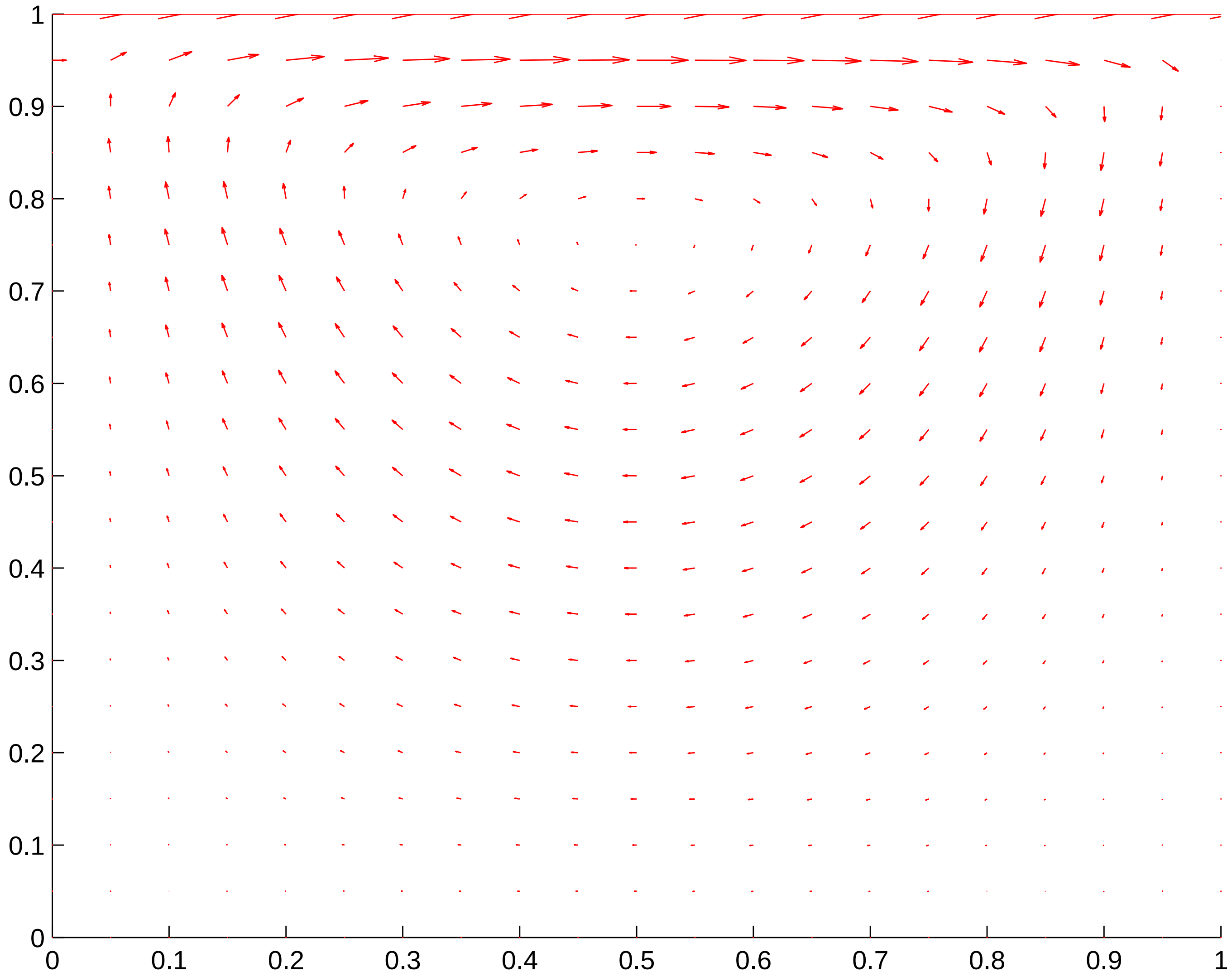
$$\int_\Omega q\, \nabla \cdot u = 0$$

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

(v, q) = BasisFunctions(TH)
(u, p) = BasisFunctions(TH)

f = Function(P2)

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

Stokes with mixed elements (in **DOLFIN**)

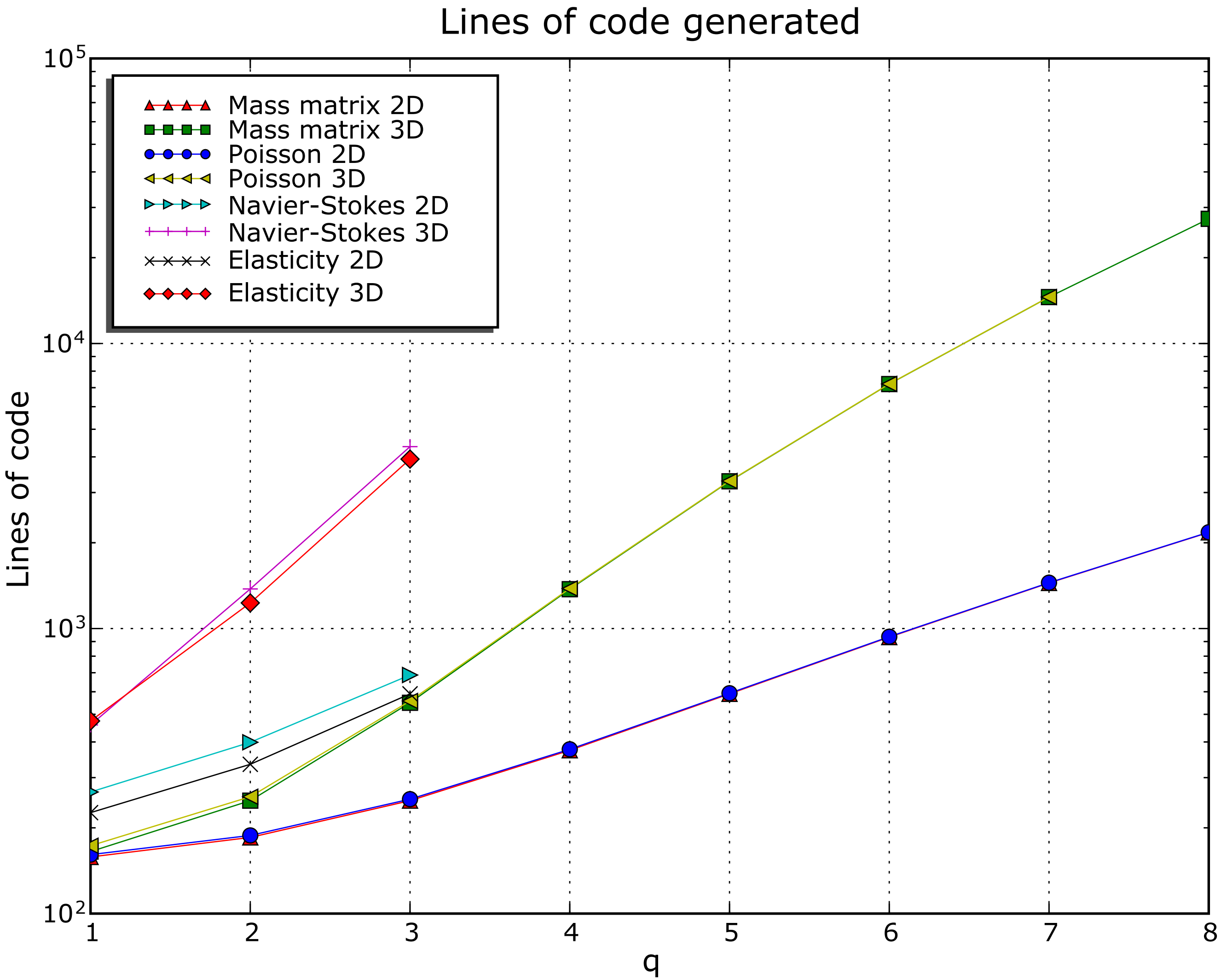
```
Stokes::BilinearForm a;
Stokes::LinearForm L(f);

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

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

Function u(x, mesh, a.trial());
File file("stokes.m");
file << u;
```

Stokes with mixed elements (solution)



The new BLAS mode

Motivation



Lines of code generated
Mass matrix 2D
Mass matrix 3D
Poisson 2D
Poisson 3D
Navier-Stokes 2D
Navier-Stokes 3D
Elasticity 2D
Elasticity 3D
10^5
10^4
10^3
10^2
Lines of code
1
2
3
4
5
6
7
8
q

Motivation

Large code size means

- Generated code expensive to compile (with g++)
- Non-optimal run-time performance

Use BLAS to

- Reduce compile-time (fewer instructions)
- Improve run-time performance (access memory better)

Three stages to optimize:

- Compile-time: **FFC**
- Compile-time: g++
- Run-time

Rewrite as a matrix-vector product

Compute sum of tensor products:

$$A_i^e = \sum_k \left(A^0_{i\alpha} G_e^\alpha\right)_k$$

Enumerate multiindices:

$\{i^j\}_j$ sequence of multiindices of length $|i|$

$\{\alpha^j\}_j$ sequence of multiindices of length $|\alpha|$

Example (2D Poisson for $q = 1$):

$$\{i^j\}_j = \{(1,1),(1,2),(1,3),(2,1),\ldots,(3,3)\}$$

$$\{\alpha^j\}_j = \{(1,1),(1,2),(2,1),(2,2)\}$$

Rewrite as a matrix-vector product

Flatten each element tensor A^K and geometry tensor G_K:

$$a_K = (A^K_{i^j})^\top = (A^K_{i^1}, A^K_{i^2}, \ldots)^\top$$

$$g_K = (G^{\alpha^1}_K)^\top = (G^{\alpha^1}_K, G^{\alpha^2}_K, \ldots)^\top$$

Define the matrix $\bar{A}^0$ by $\bar{A}^0_{jk} = A^0_{i^j \alpha^j}$ to obtain

$$a_K = \bar{A}^0 g_K$$

Write sum of tensor products as one matrix-vector product:

$$A^K_i = \sum_k \left(A^0_{i\alpha} G^\alpha_e\right)_k \leftrightarrow a_K = \sum_k \left(\bar{A}^0 g_K\right)_k = \left[(\bar{A}^0)_1 \, (\bar{A}^0)_2 \ldots\right] \begin{bmatrix} (g_K)_1 \\ (g_K)_2 \\ \vdots \end{bmatrix}$$

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] = 4.999999999999998e-01*G0_0_0 + 4.999999999999997e-01*G0_0_1 +
             4.999999999999997e-01*G0_1_0 + 4.999999999999996e-01*G0_1_1;
  block[1] = -4.999999999999998e-01*G0_0_0 - 4.999999999999997e-01*G0_1_0;
  block[2] = -4.999999999999997e-01*G0_0_1 - 4.999999999999996e-01*G0_1_1;
  block[3] = -4.999999999999998e-01*G0_0_0 - 4.999999999999997e-01*G0_0_1;
  block[4] = 4.999999999999998e-01*G0_0_0;
  ...
  block[8] = 4.999999999999996e-01*G0_1_1;
}
```

Compiling Poisson (new BLAS mode)

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

Preliminary benchmarks (timings in seconds)

- 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)

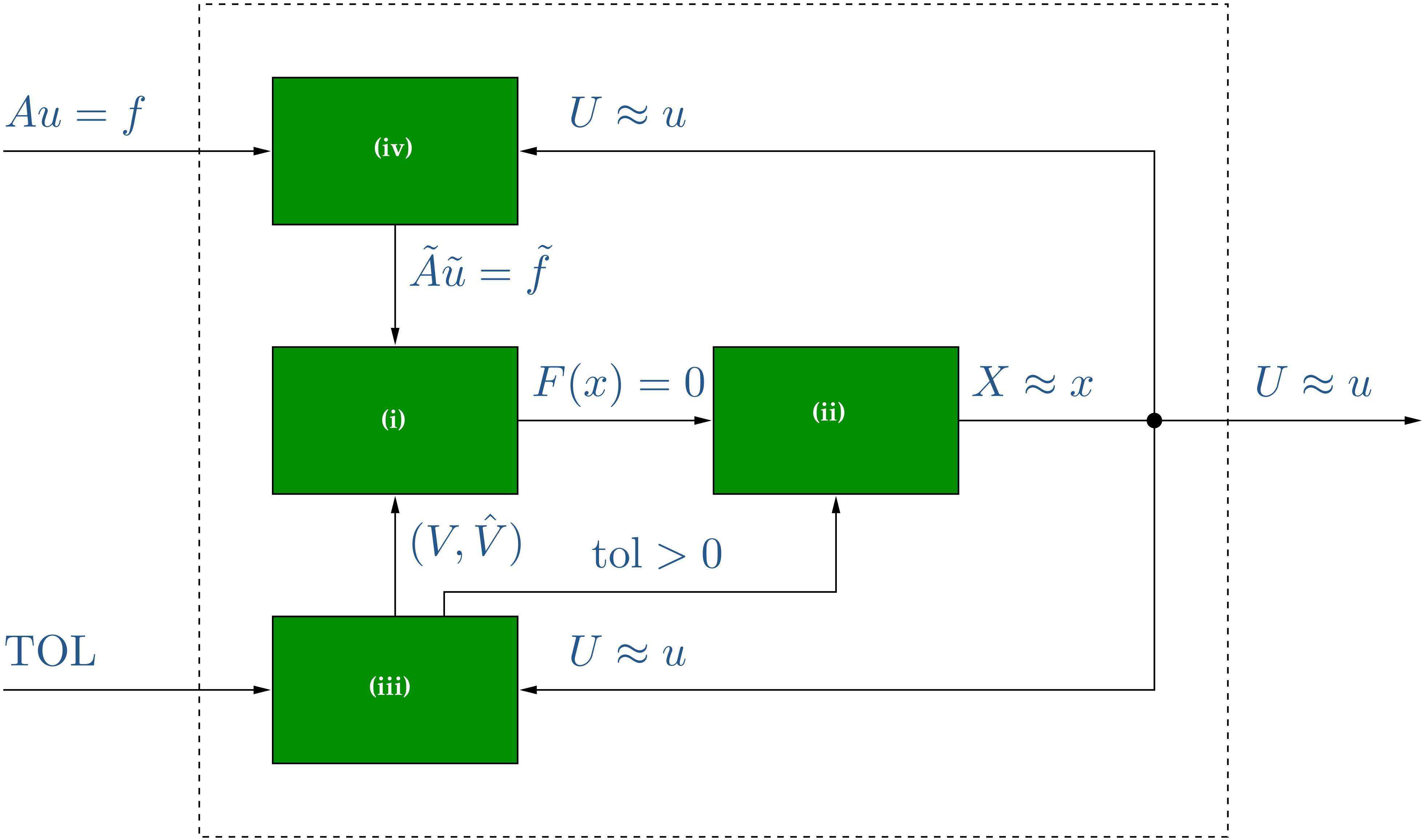
Future plans for ***FFC***

Future plans for **FFC**

- Optimizations:
  - Optimize precomputation
  - FErari optimizations
  - Level 3 BLAS optimizations
  - Optimize compuation of geometry tensor
- Features:
  - Projections, division by functions
  - Improved support for mixed elements
  - Take full advantage of Sieve for connectivity
  - Support for new elements and non-affine mappings
  - Tensor-representation of quadrature-based schemes
  - Representation of nonlinear forms
  - Generation of dual problems and error estimators

Additional slides

The Automation of CMM



Example 3: Navier–Stokes

- Form:

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

- Evaluation:

$$A_i^e = \int_e \phi_{i_1} \cdot (w \cdot \nabla) \phi_{i_2} \, \mathrm{d}x$$
$$= \det F_e' \frac{\partial X_{\alpha_3}}{\partial x_{\alpha_1}} w_{\alpha_2} \int_E \Phi_{i_1}[\beta] \Phi_{\alpha_2}[\alpha_1] \frac{\partial \Phi_{i_2}[\beta]}{\partial X_{\alpha_3}} \, \mathrm{d}X = A_{i\alpha}^0 G_e^\alpha$$

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