The FEniCS Project

L. Ridgway Scott

The Institute for Biophysical Dynamics, the Computation Institute, and the Departments of Computer Science and Mathematics, The University of Chicago, Chicago IL 60637, U.S.A.

and the FEniCS Team, especially:

U Chicago/CS: Rob Kirby

ANL/MCS: Matt Knepley

TTI-Chicago: Anders Logg

Chalmers U: Claes Johnson

KTH/NADA: Johan Hoffman

Presented at the Midwest Numerical Analysis Conference, 21 May 2005, University of Iowa.

1 Overview of CMM

Objective of computational mathematical modeling (CMM) is to make **quantitative** predictions of natural phenomena based on some type of **mathematical** description.

Models using partial differential equations can be found today in economics, financial modeling, protein design, materials engineering, as well as more traditional areas.

Automation of generation of software both minimizes errors in approximation and allows more accurate models to be used.

The FEniCS project is devoted to

- studying the fundamental challenges of automating computational mathematical modeling
- developing middleware to automate the generation of software.

1.1 More reliable simulations

The greatest source of error is the model itself.

Modelers need to experiment with different models.

Current software development paradigms make this

costly, time consuming, and unreliable.

FEniCS automates generation of simulation codes based on description of a model (FFC) and the finite element (FIAT).

This allows rapid and reliable simulations with novel models.

Mathematical structure of finite elements

facilitates automation of code generation.

Mathematical modeling can be done using tools of different levels of complexity. We focus on issues relating to partial differential equation models to simplify the discussion.

1.2 Variational form compiler

Variational forms provide a convenient language to describe partial differential equations.

Several systems have dealt with arbitrary problems expressed in variational form. Key step is compilation of variational forms and code generation in a suitable language (e.g., C++ or Fortran).

A form compiler can generate not only the code related directly to the variational forms, but also derived forms: error estimators, derivatives required for Newton's method, or extended systems.

Scientific simulation requires the highest possible level of performance from software and hardware.

This means that code optimization techniques are essential.

The FErari system optimizes the generation of finite element matrices and their actions.

2 FFC examples

```
# Copyright (c) 2005 Johan Jansson.
# Licensed under the GNU GPL Version 2
#
# The bilinear form e(u) : e(u) for linear
# elasticity with e(u) = grad(u) + grad(u)^T
#
# Compile this form with FFC: ffc Elasticity.form

element = FiniteElement("Vector Lagrange", "tetrahedron", 1)

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

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

```
# Copyright (c) 2004 Anders Logg (logg@tti-c.org)
# Licensed under the GNU GPL Version 2
#
# The bilinear form for the nonlinear term in the
# Navier-Stokes equations with fixed convective velocity.
#
# Compile this form with FFC: ffc NavierStokes.form

element = FiniteElement("Vector Lagrange", "tetrahedron", 1)

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

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

This compiles to 388 lines of C++ code (38665 characters)

3 Introduction to matrix formation

Formation of matrices takes a substantial amount of time in finite element computations.

Disadvantage of finite elements over finite differences.

But standard algorithm can be far from optimal.

We give a general formalism which can be automated and linked with FIAT and FFC.

Narrows the efficiency gap between finite elements and finite differences.

Algorithms we present here can be used in "matrix free" representations of finite element operations: substantial reductions in memory requirements and memory traffic.

3.1 Long term goal

Provide guidance regarding the development of a form compiler for finite element variational approximation.

The FEniCS Form Compiler (FFC) is a first step in this direction and is already in production use.

Our examples provide an indication of some of the challenges of designing such a compiler if it is intended to be reasonably efficient.

A critical step is to understand what code needs to be generated.

This is less obvious for higher level languages which have complex operations as elementary units.

There are **opportunities for optimization** which would be difficult to uncover automatically from a low-level representation.

They must be captured at high level.

3.2 Automation in computational mathematical modeling

The idea of automating such tasks not new in scientific computing.

Automatic differentiation tools produce efficient gradient, adjoint, and Hessian for existing code, enabling optimal control calculations, extended system solvers, and Newton-based nonlinear solvers.

Other tools that automate finite element computation:

- FreeFEM and Sundance provide type of variational form compiler and automatic generation of matrices
- Similar tools were provided in the Analysa and Dolfin projects
- Also work in numerical linear algebra, etc.

4 Operators related to multilinear forms

Consider a variational problem to find $u \in \mathcal{V}$ such that

$$a(v,u) = F(v) \quad \forall v \in \mathcal{V} \tag{4.1}$$

for a given (continuous, coercive) bilinear form $a(\cdot,\cdot)$. Corresponds to a linear system of equations

$$AU = F \quad \forall v \in \mathcal{V} \tag{4.2}$$

where

$$A_{ij} := a(\phi_i, \phi_j) \quad F_j := F(\phi_j) \quad u := \sum_{i \in \mathcal{I}} U_i \phi_i \tag{4.3}$$

where, e.g., $\{\phi_i \ : \ i \in \mathcal{I}\}$ is the standard Lagrange nodal basis and where $\mathcal{I}$ denotes the index set for the nodes.

In many iterative methods, the actual matrix A is not needed explicity, rather all that is required is some way to compute the **action** of A, that is, the mapping that sends a vector V to the vector AV. This operation can be defined purely in terms of the bilinear form as follows. Suppose we write

$$v := \sum_{i \in \mathcal{I}} V_i \phi_i \tag{4.4}$$

Then for all $i \in \mathcal{I}$

$$\begin{aligned} (AV)_i =& \sum_{j \in \mathcal{I}} A_{ij} V_j = \sum_{j \in \mathcal{I}} a(\phi_i, \phi_j) V_j \\ =& a(\phi_i, \sum_{j \in \mathcal{I}} V_j \phi_j) = a(\phi_i, v) \end{aligned} \tag{4.5}$$

The vector AV can be computed by evaluating $a(\phi_i, v)$ for all $i \in \mathcal{I}$.

The standard matrix assembly algorithm can be used to compute the action efficiently.

With (4.5) as motivation, we can introduce the notation $a(\mathcal{V}, v)$ where

$$a(\mathcal{V}, v) := AV\,. \tag{4.6}$$

Note that the notation "$\mathcal{V}$" inserted in a slot in the variational form indicates implicitly the range of the index variable i. Note that evaluating $Y_i := a(v, \phi_i)$ for all $i \in \mathcal{I}$ computes the vector $Y = A^t V$. In the notation of (4.6), we have $A^t V = a(v, \mathcal{V})$. Correspondingly, it is natural to define $a(\mathcal{V}, \mathcal{V}) = A$.

The action of a bilinear form can be used in several contexts. Perhaps the simplest is when non-homogeneous boundary conditions are posed. Suppose g represents a function defined on the whole domain which satisfies the correct boundary conditions. A typical variational problem is to find u such that $u - g \in \mathcal{V}$ and

$$a(v, u) = 0 \quad \forall v \in \mathcal{V}. \tag{4.7}$$

This can be re-written using the difference $u^0 := u - g \in \mathcal{V}$. The variational problem becomes: Find $u^0 \in \mathcal{V}$ such that

$$a(v, u^0) = -a(v, g) \quad \forall v \in \mathcal{V}. \tag{4.8}$$

In matrix form, we would write this as

$$AU^0 = -a(\mathcal{V}, g). \tag{4.9}$$

This could be solved by a direct method (e.g., Gaussian elimination) with $-a(\mathcal{V}, g)$ as the right-hand-side vector. However, we could equally well think of (4.7) as

$$a(\mathcal{V}, u^0) = -a(\mathcal{V}, g). \tag{4.10}$$

which does not require the explicit evaluation of a matrix and could be solved by an interative method.

4.1 The Action of Trilinear Forms

The nonlinear term in the Navier–Stokes provides an example of the action of a general multi-linear form. Certain algorithms might involve a variational problem to find $\mathbf{u} \in \mathcal{V}$ such that

$$a(\mathbf{u}, \mathbf{w}) = c(\mathbf{v}, \tilde{\mathbf{v}}, \mathbf{w}) \quad \forall \mathbf{w} \in \mathcal{V} \tag{4.11}$$

for two different $\mathbf{v} \in \mathcal{V}$ and $\tilde{\mathbf{v}} \in \mathcal{V}$. Choose $\mathbf{w} = \phi_i$ for a generic basis function ϕ_i. Write as usual $\mathbf{u} := \sum_{i \in \mathcal{I}} U_i \phi_i$. By analogy with the definition (4.3), we set

$$A_{ij} := a(\phi_i, \phi_j) \quad \forall i, j \in \mathcal{I} \tag{4.12}$$

which, by a simple extension of our convention (4.6), can be written as

$$A = a(\mathcal{V}, \mathcal{V}). \tag{4.13}$$

Then (4.11) can be written as

$$
\begin{aligned}
\left(A^t U\right)_i =& \sum_{j \in \mathcal{I}} A_{ji} U_j \\
=& \sum_{j \in \mathcal{I}} a(\phi_j, \phi_i) U_j \\
=& a(\sum_{j \in \mathcal{I}} U_j \phi_j, \phi_i) \\
=& a(\mathbf{u}, \phi_i) \\
=& c(\mathbf{v}, \tilde{\mathbf{v}}, \phi_i) \quad \forall i \in \mathcal{I}.
\end{aligned}
\tag{4.14}
$$

In notation analogous to that of (4.6), we can write (4.14) as

$$
a(\mathbf{u}, \mathcal{V}) = A^t U = c(\mathbf{v}, \tilde{\mathbf{v}}, \mathcal{V}), \tag{4.15}
$$

where the latter term introduces notation for the action of a trilinear form.

4.2 Generating matrices from multilinear forms

With forms of two or more variables, there are other objects that can be generated automatically in a way that is similar to what we can do to generate the action of a form. For trivarariate forms, it is of interest to work with the matrix

$$C_{ij} := c(\mathbf{v}, \phi_i, \phi_j) \quad \forall i, j \in \mathcal{I} \tag{4.16}$$

which we write in our shorthand as

$$C = c(\mathbf{v}, \mathcal{V}, \mathcal{V}) \tag{4.17}$$

For example, one might want to solve (for $\mathbf{u}$, given $\mathbf{f}$) the equation

$$\mathbf{u} + \mathbf{v} \cdot \nabla \mathbf{u} = \mathbf{f} \tag{4.18}$$

for a fixed, specified $\mathbf{v} \in \mathcal{V}$, using the variational form

$$(\mathbf{u}, \mathbf{w})_{L^2} + c(\mathbf{v}, \mathbf{u}, \mathbf{w}) = (\mathbf{f}, \mathbf{w})_{L^2} \quad \forall \mathbf{w} \in \mathcal{V} \tag{4.19}$$

Now write the variational equation

$$(\mathbf{u}, \mathbf{w})_{L^2} + c(\mathbf{v}, \mathbf{u}, \mathbf{w}) = (\mathbf{f}, \mathbf{w})_{L^2} \quad \forall \mathbf{w} \in \mathcal{V} \tag{4.20}$$

in component form:

$$\sum_{i \in \mathcal{I}} U_i \left((\phi_i, \phi_j)_{L^2} + c(\mathbf{v}, \phi_i, \phi_j) \right) = (\mathbf{f}, \phi_j)_{L^2} \quad \forall j \in \mathcal{I} \tag{4.21}$$

In operator notation, this becomes

$$U^t \left((\mathcal{V}, \mathcal{V})_{L^2} + c(\mathbf{v}, \mathcal{V}, \mathcal{V}) \right) = F \tag{4.22}$$

4.3 General tensors from Forms

Frequently the spaces in a form are not all the same, e.g.,

$$b(v,p) := \int \nabla\cdot \mathbf{v}(x) p(x)\, dx \tag{4.23}$$

The form $b(\cdot,\cdot)$ in (4.23) involves spaces of scalar functions (say, Π) as well as vector functions (say, $\mathcal{V}$). The matrix $b(\mathcal{V},\Pi)$ is defined analogously to (4.12) and (4.13):

$$(b(\mathcal{V},\Pi))_{ij} := b(\phi_i, q_i) \tag{4.24}$$

where $\{\phi_i \ : \ i\in\mathcal{I}\}$ is a basis of $\mathcal{V}$ as before, and $\{q_i \ : \ i\in\mathcal{J}\}$ is a basis of Π. Note that $b(\mathcal{V},\Pi)$ will not, in general, be a square matrix.

In general, if we have a form $a(v^1, \ldots, v^n)$ of n entries, then the expression

$$a\left(\ldots, \mathcal{V}^1, \ldots, \mathcal{V}^k, \ldots\right) \tag{4.25}$$

defines a tensor of rank k. More precisely, each of the n arguments in the form $a(v^1, \ldots, v^n)$ may be a function space or a member of a function space.

For example, $a\left(v^1, v^2, \mathcal{V}^1, v^3, \mathcal{V}^2, \mathcal{V}^3, v^4\right)$ denotes a tensor of rank 3, whereas $a\left(v^1, \mathcal{V}^1, v^2, v^3, \mathcal{V}^2, v^4, v^5\right)$ denotes a tensor of rank 2.

Note that a tensor of rank zero is just a scalar, consistent with the usual interpretation of $a(v^1, \ldots, v^n)$.

A tensor of rank one is a vector, and a tensor of rank two is a matrix.

Tensors of rank three or higher are less common in computational linear algebra.

5 Matrix Evaluation by Assembly

The *assembly* of integrated differential forms is done by summing its constituent parts over each *element*, which are computed separately through the use of a numbering scheme called the *local-to-global* index. This index, $\iota(e, \lambda)$, relates the local (or element) node number, $\lambda \in \mathcal{L}$, on a particular element, indexed by e, to its position in the global data structure.

We may write a finite element function f in the form

$$\sum_{e} \sum_{\lambda \in \mathcal{L}} f_{\iota(e,\lambda)} \phi^e_\lambda \tag{5.26}$$

where f_i denotes the "nodal value" of the finite element function at the i-th node in the global numbering scheme and $\{\phi^e_\lambda \ : \ \lambda \in \mathcal{L}\}$ denotes the set of basis functions on the element domain T_e.

The element basis functions, ϕ^e_λ, are extended by zero outside T_e.

Can relate "element" basis functions ϕ^e_λ to fixed set of basis functions on "reference" element, $\mathcal{T}$, via mapping of $\mathcal{T}$ to T_e.

Could involve changing both the "x" values and the "ϕ" values in a coordinated way, as with the Piola transform , or it could be one whose Jacobian is non-constant, as with tensor-product elements or isoparametric elements.

For an affine mapping, $\xi \to J\xi + x_e$, of $\mathcal{T}$ to T_e:

$$\phi^e_\lambda(x) = \phi_\lambda\left(J^{-1}(x - x_e)\right).$$

The inverse mapping, $x \to \xi = J^{-1}(x - x_e)$ has as its Jacobian

$$J^{-1}_{mj} = \frac{\partial \xi_m}{\partial x_j},$$

and this is the quantity which appears in the evaluation of the bilinear forms. Of course, $\det J = 1/\det J^{-1}$.

5.1 Evaluation of bilinear forms

The assembly algorithm utiizes the decomposition of a variational form as a sum over "element" forms

$$a(v,w) = \sum_e a_e(v,w)$$

where "element" bilinear form for Laplace's equation defined via

$$\begin{aligned} a_e(v,w) &:= \int_{T_e} \nabla v(x) \cdot \nabla w(x)\, dx \\ &= \int_{\mathcal{T}} \sum_{j=1}^{d} \frac{\partial}{\partial x_j} v(J\xi + x_e) \frac{\partial}{\partial x_j} w(J\xi + x_e) \det(J)\, d\xi \end{aligned} \tag{5.27}$$

by transofrming to the reference element.

Finite element matrices computed via assembly in a similar way.

The local element form is computed as follows.

5.2 Evaluation of bilinear forms—continued

$$
\begin{aligned}
a_e(v,w) &= \int_{\mathcal{T}} \sum_{j=1}^{d} \frac{\partial}{\partial x_j} v(J\xi + x_e) \frac{\partial}{\partial x_j} w(J\xi + x_e) \det(J)\, d\xi \\
&= \int_{\mathcal{T}} \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial}{\partial \xi_m} \left(\sum_{\lambda \in \mathcal{L}} v_{\iota(e,\lambda)} \phi_\lambda(\xi) \right) \times \\
&\qquad \frac{\partial \xi_{m'}}{\partial x_j} \frac{\partial}{\partial \xi_{m'}} \left(\sum_{\mu \in \mathcal{L}} w_{\iota(e,\mu)} \phi_\mu(\xi) \right) \det(J)\, d\xi \\
&= \begin{pmatrix} v_{\iota(e,1)} \\ \cdot \\ \cdot \\ v_{\iota(e,|\mathcal{L}|)} \end{pmatrix}^t \mathbf{K}^e \begin{pmatrix} w_{\iota(e,1)} \\ \cdot \\ \cdot \\ w_{\iota(e,|\mathcal{L}|)} \end{pmatrix}.
\end{aligned}
\tag{5.28}
$$

Here, the *element stiffness matrix*, $\mathbf{K}^e$, is given by

$$K^e_{\lambda,\mu} := \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \det(J) \int_{\mathcal{T}} \frac{\partial}{\partial \xi_m} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{m'}} \phi_\mu(\xi)\, d\xi$$

$$= \sum_{m,m'=1}^{d} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{5.29}$$

where

$$K_{\lambda,\mu,m,m'} = \int_{\mathcal{T}} \frac{\partial}{\partial \xi_m} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{m'}} \phi_\mu(\xi)\, d\xi \tag{5.30}$$

and

$$G^e_{m,m'} := \det(J) \sum_{j=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \tag{5.31}$$

for $\lambda, \mu \in \mathcal{L}$ and $m, m' = 1, \ldots, d$.

5.3 Computation of Bilinear Form Matrices

The matrix associated with a bilinear form is

$$A_{ij} := a(\phi_i, \phi_j) = \sum_e a_e(\phi_i, \phi_j) \tag{5.32}$$

for all i, j.We can compute this again by assembly.

First, set all the entries of A to zero. Then loop over all elements e and local element numbers λ and μ and compute

$$A_{\iota(e,\lambda),\iota(e,\mu)} += K^e_{\lambda,\mu} = \sum_{m,m'} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{5.33}$$

where $G^e_{m,m'}$ are defined in (5.31).

We optimize the computation of each

$$K^e_{\lambda,\mu} = \sum_{m,m'} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{5.34}$$

6 Computing K for general elements

Tensor K for quadratics represented as a matrix of 2×2 matrices.

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

The tensor $K_{i,j,m,n}$ can be presented as an $|\mathcal{L}| \times |\mathcal{L}|$ matrix of $d \times d$ matrices, as presented in the table for the case of quadratics in two dimension.

The entries of resulting matrix K^e can be viewed as the dot (or Frobenius) product of the entries of K and G^e:

$$K^e_{i,j} = \mathbf{K}_{i,j} : G^e \tag{6.35}$$

The key point to consider is how many independent entries there are in $\mathbf{K}$, and the complexity of them.

For example, six of the entries are all zero, the four 2×2 matrices in the upper-left corner of the table as well as four other entries representing K are trivial, and there are significant redundancies among the rest. For example, $-4\mathbf{K}_{3,1} = \mathbf{K}_{4,1} = \mathbf{K}_{3,4}$.

6.1 Tensor K for quadratics

zero entries, trivial entries and related entries ($-4\mathbf{K}_{3,1} = \mathbf{K}_{3,4} = \mathbf{K}_{4,1}$)

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

6.2 Computing K for quadratics

Taking advantage of these simplifications, each K^e for quadratics in two dimensions can be computed with at most 18 floating point operations instead of 288 floating point operations: an improvement of a factor of sixteen in computational complexity.

On the other hand, there are only 64 nonzero entries in each **K**. So eliminating multiplications by zero gives a four fold improvement.

Sparse matrix accumulation requires at least 76 (=36+36+4) memory references, not including sparse matrix indexing. Even if the matrix is stored in symmetric form, at least 46 (=21+21+4) memory references are needed.

Computational complexity less than cost of memory references.

6.3 Linears in three dimensions

The tensor $K_{i,j,m,n}$ for the case of linears in three dimensions is presented in the following table: $4\mathbf{K} =$

1	0	0	0	1	0	0	0	1	-1	-1	-1
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	-1	-1	-1
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	-1	-1	-1
-1	0	0	0	-1	0	0	0	-1	1	1	1
-1	0	0	0	-1	0	0	0	-1	1	1	1
-1	0	0	0	-1	0	0	0	-1	1	1	1

6.4 Algorithm for linears in three-D

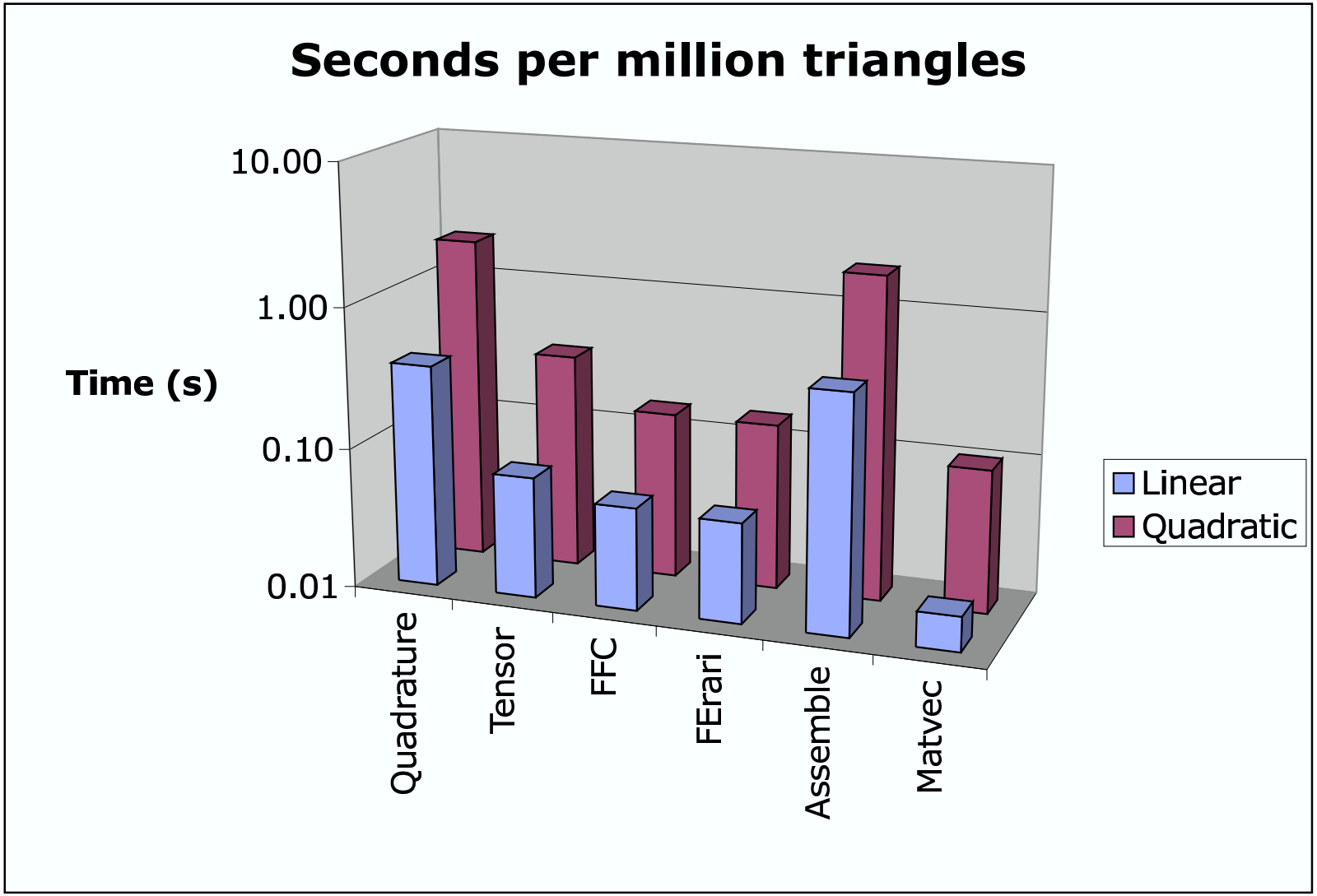
Each K^e can be computed by computing the three row sums of G^e, the three column sums, and the sum of one of these sums.

We also have to negate all of the column and row sums, leading to a total of 20 floating point operations instead of 288 floating point operations using the straightforward definition, an improvement of a factor of nearly fifteen in computational complexity.

On the other hand, there are only 36 non-zero elements in $\mathbf{K}$, and all of these are ± 1.

At most 57 (=16+16+16+9) memory references are needed to do a general sparse matrix update for each element.

Using symmetry of G^e (row sums equal column sums) we can reduce the computation to only 10 floating point operations, leading to a improvement of nearly 29. For a sparse matrix update, at most 39 (=10+10+10+9) memory references are needed.


Seconds per million triangles
Time (s)
10.00
1.00
0.10
0.01
Quadrature
Tensor
FFC
FErari
Assemble
Matvec
Linear
Quadratic

7 Evaluation of general multi-linear forms

Arbitrary multi-linear forms can appear in finite element calculations.

We use as the next example the nonlinear form $c(\cdot,\cdot,\cdot)$ in the Navier–Stokes term using "element" trilinear forms:

$$
\begin{aligned}
c_e(\mathbf{u},\mathbf{v},\mathbf{w}) &:= \int_{T_e} \mathbf{u}\cdot\nabla\mathbf{v}(x)\cdot\mathbf{w}(x)\,dx \\
&= \int_{T_e} \sum_{j,k=1}^{d} u_j(x)\frac{\partial}{\partial x_j} v_k(x) w_k(x)\,dx \\
&= \int_{\mathcal{T}} \sum_{j,k=1}^{d} u_j(J\xi+x_e)\frac{\partial}{\partial x_j} v_k(J\xi+x_e) w_k(J\xi+x_e)\det(J)\,d\xi
\end{aligned}
\tag{7.36}
$$

Therefore

$$c_e(\mathbf{u},\mathbf{v},\mathbf{w}) = \int_{\mathcal{T}} \sum_{j,k,m=1}^{d} \left(\sum_{\lambda\in\mathcal{L}} u_j^{\iota(e,\lambda)} \phi_\lambda(\xi) \right) \frac{\partial \xi_m}{\partial x_j} \left(\sum_{\mu\in\mathcal{L}} v_k^{\iota(e,\mu)} \frac{\partial}{\partial \xi_m} \phi_\mu(\xi) \right) \times$$

$$\left(\sum_{\rho\in\mathcal{L}} w_k^{\iota(e,\rho)} \phi_\rho(\xi) \right) \det(J)\, d\xi$$

$$= \sum_{j,k,m=1}^{d} \sum_{\lambda,\mu,\rho\in\mathcal{L}} u_j^{\iota(e,\lambda)} \frac{\partial \xi_m}{\partial x_j} v_k^{\iota(e,\mu)} w_k^{\iota(e,\rho)} \det(J) \times$$

$$\int_{\mathcal{T}} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_m} \phi_\mu(\xi) \phi_\rho(\xi)\, d\xi$$

$$= \sum_{j,k=1}^{d} \sum_{\lambda,\mu,\rho\in\mathcal{L}} u_j^{\iota(e,\lambda)} v_k^{\iota(e,\mu)} w_k^{\iota(e,\rho)} \sum_{m=1}^{d} \frac{\partial \xi_m}{\partial x_j} \det(J) N_{\lambda,\mu,\rho,m} \tag{7.37}$$

where

$$N_{\lambda,\mu,\rho,m} := \int_{\mathcal{T}} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_m} \phi_\mu(\xi) \phi_\rho(\xi)\, d\xi \tag{7.38}$$

To summarize, we have

$$
\begin{aligned}
c_e(\mathbf{u},\mathbf{v},\mathbf{w}) &= \sum_{j,k=1}^{d} \sum_{\lambda,\mu,\rho\in\mathcal{L}} u_j^{\iota(e,\lambda)} v_k^{\iota(e,\mu)} w_k^{\iota(e,\rho)} N^e_{\lambda,\mu,\rho,j} \\
&= \sum_{k=1}^{d} \sum_{\mu,\rho\in\mathcal{L}} v_k^{\iota(e,\mu)} w_k^{\iota(e,\rho)} \sum_{j=1}^{d} \sum_{\lambda\in\mathcal{L}} u_j^{\iota(e,\lambda)} N^e_{\lambda,\mu,\rho,j}
\end{aligned}
\tag{7.39}
$$

where the element coefficients $N^e_{\lambda,\mu,\rho,j}$ are defined by

$$
N^e_{\lambda,\mu,\rho,j} := \sum_{m=1}^{d} \frac{\partial \xi_m}{\partial x_j} \det(J) N_{\lambda,\mu,\rho,m\cdot} =: \sum_{m=1}^{d} \widetilde{G}_{mj} N_{\lambda,\mu,\rho,m\cdot} \tag{7.40}
$$

where $\widetilde{G}_{mj} := \frac{\partial \xi_m}{\partial x_j} \det(J)$.

Recall that J is the Jacobian above, and J^{-1} is its inverse, and

$$\left(J^{-1}\right)_{m,j} = \frac{\partial \xi_m}{\partial x_j}.$$

Note that both $N_{\lambda,\mu,\rho,(\cdot)}$ and $N^e_{\lambda,\mu,\rho,(\cdot)}$ can be thought of as d-vectors. Moreover

$$N^e_{\lambda,\mu,\rho,(\cdot)} = \det(J) N_{\lambda,\mu,\rho,(\cdot)} J^{-1}.$$

Also note that $N_{\lambda,\mu,\rho,(\cdot)} = N_{\rho,\mu,\lambda,(\cdot)}$, so that considerable storage reduction could be made if desired.

The matrix C defined by $C_{ij} = c(\mathbf{u}, \phi_i, \phi_j)$ can be computed using the assembly algorithm as follows. First, note that C can be written as a matrix of dimension $|\mathcal{V}| \times |\mathcal{V}|$ with entries that are $d \times d$ diagonal blocks. In particular, let I_d denote the $d \times d$ identity matrix. Now set C to zero, loop over all elements and up-date the matrix by

$$\begin{aligned} C_{\iota(e,\mu),\iota(e,\rho)} + &= I_d \sum_{j=1}^{d} \sum_{\lambda \in \mathcal{L}} u_j^{\iota(e,\lambda)} N^e_{\lambda,\mu,\rho,j} \\ &= I_d \sum_{m,j=1}^{d} \widetilde{G}_{mj} \left(\sum_{\lambda \in \mathcal{L}} u_j^{\iota(e,\lambda)} N_{\lambda,\mu,\rho,m} \right) \\ &= I_d \sum_{m,\lambda \in \mathcal{L}} \gamma_{m\lambda} N_{\lambda,\mu,\rho,m} \end{aligned} \tag{7.41}$$

for all μ and ρ, where (continued on next slide)

$$\gamma_{m\lambda} = \sum_{j=1}^{d} \widetilde{G}_{mj} u_j^{\iota(e,\lambda)}. \tag{7.42}$$

and $\widetilde{G}_{mj} := \frac{\partial \xi_m}{\partial x_j} \det(J)$.

It thus appears that the computation of C can be viewed as similar in form to (6.35), and similar optimization techniques applied. In fact, we can introduce the notation $K^{e,u}$ where

$$K^{e,u}_{\mu,\rho} = \sum_{m,\lambda \in \mathcal{L}} \gamma_{m\lambda} N_{\lambda,\mu,\rho,m} \tag{7.43}$$

Then the update of C is done in the obvious way with $K^{e,u}$.

The memory traffic required to compute γ is at most $|\mathcal{V}| \times d + d^2$, and the update of C could require as few as $3|\mathcal{V}|^2$ memory references.

7.1 Trilinear Forms with Piecewise Linears

In the piecewise linear case, (7.38) simplifies to

$$N_{\lambda,\mu,\rho,m} := \frac{\partial \phi_\mu}{\partial \xi_m} \int_{\mathcal{T}} \phi_\lambda(\xi)\phi_\rho(\xi)\, d\xi \tag{7.44}$$

We can think of $N_{\lambda,\mu,\rho,m}$ defined from two matrices:
$N_{\lambda,\mu,\rho,m} = D_{\mu,m}F_{\lambda,\rho}$ where

$$D_{\mu,m} := \frac{\partial \phi_\mu}{\partial \xi_m} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} (d=2) \quad \text{and} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} (d=3) \tag{7.45}$$

and

$$F_{\lambda,\rho} := \int_{\mathcal{T}} \phi_\lambda(\xi)\phi_\rho(\xi)\, d\xi \tag{7.46}$$

The latter matrix is easy to determine.

In the piecewise linear case, we can compute integrals of products using the quadrature rule that is based on edge mid-points (with equal weights given by the area of the simplex divided by the number of edges).

Thus the weights are $\omega = 1/6$ for $d = 2$ and $\omega = 1/24$ for $d = 3$.

Each of the values $\phi_\lambda(\xi)$ is either $\frac{1}{2}$ or zero, and the products are equal to $\frac{1}{4}$ or zero.

For the diagonal terms $\lambda = \rho$, the product is non-zero on d edges, so $F_{\lambda,\lambda} = 1/12$ for $d = 2$ and $1/32$ for $d = 3$.

If $\lambda \neq \rho$, then the product $\phi_\lambda(\xi)\phi_\rho(\xi)$ is non-zero for exactly one edge (the one connecting the corresponding vertices), so $F_{\lambda,\rho} = 1/24$ for $d = 2$ and $1/96$ for $d = 3$.

Thus we can describe the matrices F in general as having d on the diagonals, 1 on the off-diagonals, and scaled by $1/24$ for $d = 2$ and $1/96$ for $d = 3$. Thus $4(d+1)!F =$

$$\begin{pmatrix} d & 1 & \cdots & 1 \\ 1 & d & \cdots & 1 \\ \cdot & \cdot & \cdots & \cdot \\ 1 & 1 & \cdots & d \end{pmatrix} = (d-1)I_{d+1} + \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \cdot & \cdot & \cdots & \cdot \\ 1 & 1 & \cdots & 1 \end{pmatrix} \qquad (7.47)$$

for $d = 2$ or 3, where I_d denotes the $d \times d$ identity matrix. Note that for a given d, the matrices in (7.47) are $d+1 \times d+1$ in dimension.

The tensor N (multiplied by ninety-six) for piecewise linears in three dimensions represented as a matrix of four by three matrices.

3	1	1	1	0	0	0	0	0	0	0	0	3	1	1	1
0	0	0	0	3	1	1	1	0	0	0	0	3	1	1	1
0	0	0	0	0	0	0	0	3	1	1	1	3	1	1	1
1	3	1	1	0	0	0	0	0	0	0	0	1	3	1	1
0	0	0	0	1	3	1	1	0	0	0	0	1	3	1	1
0	0	0	0	0	0	0	0	1	3	1	1	1	3	1	1
1	1	3	1	0	0	0	0	0	0	0	0	1	1	3	1
0	0	0	0	1	1	3	1	0	0	0	0	1	1	3	1
0	0	0	0	0	0	0	0	1	1	3	1	1	1	3	1
1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3
0	0	0	0	1	1	1	3	0	0	0	0	1	1	1	3
0	0	0	0	0	0	0	0	1	1	1	3	1	1	1	3

We see now a new ingredient for computing the entries of $K^{e,u}$ from the matrix $\gamma_{m,\lambda}$. Define $\gamma_m = \sum_{\lambda=1}^{4} \gamma_{m,\lambda}$ for $m = 1, 2, 3$, and then $\tilde{\gamma}_{m,\lambda} = 2\gamma_{m,\lambda} + \gamma_m$ for $m = 1, 2, 3$ and $\lambda = 1, 2, 3, 4$. Then

$$K^{e,u} = \begin{pmatrix} \tilde{\gamma}_{11} & \tilde{\gamma}_{21} & \tilde{\gamma}_{31} & \tilde{\gamma}_{11} + \tilde{\gamma}_{21} + \tilde{\gamma}_{31} \\ \tilde{\gamma}_{12} & \tilde{\gamma}_{22} & \tilde{\gamma}_{32} & \tilde{\gamma}_{12} + \tilde{\gamma}_{22} + \tilde{\gamma}_{32} \\ \tilde{\gamma}_{13} & \tilde{\gamma}_{23} & \tilde{\gamma}_{33} & \tilde{\gamma}_{13} + \tilde{\gamma}_{23} + \tilde{\gamma}_{33} \\ \tilde{\gamma}_{14} & \tilde{\gamma}_{24} & \tilde{\gamma}_{34} & \tilde{\gamma}_{14} + \tilde{\gamma}_{24} + \tilde{\gamma}_{34} \end{pmatrix} \tag{7.48}$$

However, note that the γ_m's are not computations that would have appeared directly in the formulation of $K^{e,u}$ but are intermediary terms that we have defined for convenience and efficiency. This requires 39 operations, instead of 384 operations using (7.43).

Only 21 memory references are required to compute γ, and at most 48 memory references are required to update C.

7.2 Algorithmic implications

The examples provide guidance for the general case.

The "vector" space of the evaluation problem (7.43) can be arbitrary in size.

In the case of the trilinear form in Navier-Stokes considered there, the dimension is the spatial dimension times the dimension of the approximation (finite element) space.

High-order finite elements would lead to very high-dimensional problems.

We need to look for relationship among the "computational vectors" in high-dimensional spaces, e.g., up to several hundred in extreme cases. The lowest order case in three space dimensions requires a twelve-dimensional space for the complexity analysis.

It will not be sufficient just to look for simple combinations to determine optimal algorithms. We need to think of this as an approximation problem.

Must look for vectors (matrices) which closely approximate a set of vectors that we need to compute. The vectors $\mathbf{v}_1 = (1, 1, 1, 1, 0, \ldots, 0), \mathbf{v}_2 = (0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0), \mathbf{v}_3 = (0, \ldots, 0, 1, 1, 1, 1)$ are each edit-distance one from four vectors we need to compute. The quantities γ_m represent the computations (dot-product) with $\mathbf{v}_m$. The quantities $\tilde{\gamma}_{m\lambda}$ are simple perturbations of γ which require only two operations to evaluate. A simple rescaling can reduce this to one operation.

Edit-distance is a useful measure to approximate the computational complexity distance, since it provides an upper-bound on the number of computations it takes to get from one vector to another. Thus we need to add this type of optimization.

8 The FErari system

We have implemented a prototype system called FErari, for Finite Element Re-arrangement Algorithm to Reduce Instructions.

We used FErari to verify that simple algorithms can find substantial reduction in operations that we exposed in our examples.

We give FErari results for conforming and nonconforming Lagrange elements in two dimensions in the following tables.

We have grouped the vectors according to whether they are zero (0), equal (=), colinear ($\|$), have only one nonzero entry (1e), differ by edit distance one (ED1), have only two nonzero entries (2e), are a linear combination of two other vectors (LC2).

Table 1: FErari at work on Conforming Lagrange elements in two dimensions. All of the vectors are accounted for by the algorithm. Key: $\mathcal{O}$ is the order of polynomials; Tot is the total number of vectors. The remaining entries are the number of vectors that are zero (0), equal (=), colinear ($\|$), have only one nonzero entry (1e), differ by edit distance one (ED1), have only two nonzero entries (2e), are a linear combination of two other vectors (LC2). MAPs is an upper bound on floating point operations required.

$\mathcal{O}$	Tot	0	=	$\|$	1e	ED1	2e	LC2	MAPs
1	9	0	0	0	4	4	0	1	10
2	36	6	11	6	4	8	0	1	20
3	100	6	41	10	4	16	8	15	76
4	225	0	98	6	4	35	16	66	209
5	441	0	183	15	4	51	28	160	446
6	784	0	342	21	4	75	32	310	784

The picture for non-conforming elements has fewer simple relations, but coplanarity relations can still be used to reduce computation substantially.

Table 2: FErari at work on Nonconforming Lagrange elements in two dimensions. All of the vectors (Total) are accounted for by the algorithm. See Table 1 for explanation of terms.

$\mathcal{O}$	Tot	0	=	$\|\|$	1e	ED1	2e	LC2	MAPs
1	9	0	0	0	4	4	0	1	10
3	100	0	11	1	0	0	0	88	177
5	441	0	105	0	0	0	0	336	672

FErari searches through the vectors as follows. (The operation counts for FErari to find the dependences or properties are given in parentheses.) The operation counts that result from using the discovered property are listed at the end, and are counted as multiply-add pairs (MAPs). FErari starts with the entire list of *(Total)* vectors and marks vectors in the list at the i-th state that have the i-th property:

1. *zero* vectors ($O(n)$) – these entries of K are free
2. vectors that are *equal* ($O(n \log(n))$ – these entries of K are free
3. vectors that are *colinear* ($O(n \log(n))$) – costs one MAP each
4. vectors that have only *one* nonzero *entry* ($O(n)$) – one MAP each
5. vectors that are edit distance one (ED1) from another vector or its negation ($O(n^2)$) – one MAP each, plus maybe a (cheap) sign flip

6. vectors that have only *two* nonzero *entries* ($O(n)$) – two MAPs each

7. vectors that are linear combinations (LC2) of two other vectors ($O(n^2)$) – two MAPs each

Note that the cheaper operations to perform and the ones that have the biggest payoff are done first.

FErari did not search here among alternate evaluation graphs, but rather it assigned evaluation strategies to each vector iteratively following the above scheme.

The examples are limited to two-dimensional cases for the Poisson operator, for simplicity. The data were generated with the Fiat system. However, FErari can be applied to data supplied by any method.

8.1 Collinearity tests

Two vectors in $\mathbb{R}^D$ are collinear if and only if the absolute value of the cosine of the angle between them is one. If the vectors are normalized to have Euclidean length one, then just check whether their dot-product has absolute value one or not. Test can be performed in $\mathcal{O}(D)$ arithmetic operations.

Further normalization: make the first non-zero coordinate of the unit vectors positive, by multiplying the vector by -1 if necessary. This provides a unique representation of the vectors in projective space, and we can check for collinearity by simply checking equality of individual components.

In a sense, we use a lexicographic ordering to check for equality. Using a sorting algorithm with this ordering determines collinearity in $\mathcal{O}(D \log n)$ arithmetic operations.

8.2 A (random) dimensional reduction algorithm

A randomized approach could be faster for large D.

If two vectors are collinear, so will be any projection of the vectors onto a subset of coordinates.

Pick at random k different coordinates (numbers from 1 to D) and apply an appropriate algorithm in k dimensions. (If $k = 2$ or 3, special techniques apply.)

When two vectors are collinear in these k dimensions, apply to the algorithm again in two other randomly selected coordinates.

It is only necessary to apply the algorithm to subsets of vectors linked by potential collineartiy.

When such equivalence classes are sufficiently small, test all remaining coordinates.

8.3 Efficient Computation of co-planarity

One vector can be written as a linear combination of two others if and only if the three vectors (and the origin) are co-planar.

A simple approach to finding co-planar trios of vectors would require an amount of computation cubic in the number of vectors. For example, we could randomly select three coordinates and consider the projection of all trios of vectors in these coordinates. Form the matrix from the three projected vectors and compute the determinant. If it is non-zero, then the vectors are linearly independent, so this trio of vectors need not be considered further. Apply the algorithm recursively to the subset of vectors that appear to be linearly dependent in the coordinates currently chosen. This is a simple and attractive algorithm, but its cost is still cubic in the number of vectors.

8.4 A (nearly) quadratic algorithm

The basic idea is to determine the set of planes generated by all pairs of vectors.

We assume that collinear pairs have been removed.

Then three vectors lie in a plane if and only if the planes of each of the pairs are the same (co-planar). Thus we have reduced the problem to a form similar to the efficient collinearity algorithm.

Determining whether two planes are the same could be done in a variety of ways.

8.5 Determining co-planarity of three vectors

Pick three random coordinates and project vectors onto them.

For each pair of vectors a and b, represent the plane that they span by the normal vector (which can be computed using the vector cross-product $a \times b$).

Finding equal planes equivalent to finding normals that are collinear.

Thus we have reduced to the collinearity problem for $\frac{1}{2}n(n+1)$ vectors. The cost of the algorithm will be nearly quadratic in the number of vectors.

As in the collinearity algorithm, we find equivalence classes of vectors that are co-planar in the three coordinates chosen. We apply the algorithm recursively to the equivalence classes, but now the equivalence relation is more complicated.

8.6 Co-planarity equivalence relation

Suppose that a, b, c and b, c, d are co-planar. Then all of a, b, c, d are co-planar in the three coordinates chosen. Thus we would apply the algorithm to the subset a, b, c, d. That is, we see that we can define a precise equivalence relation among triples: two triples are equivalent if they have a pair in common.

But now suppose that we find that a, b, c and c, d, e are co-planar in the chosen three coordinates, but there are no other relations involving a, b, d, e. Then we could apply the algorithm separately to a, b, c and c, d, e. However, this may not be a big win computationally, since c is in both sets. That is, it may make sense to apply the algorithm instead to the set a, b, c, d, e. This means that we use a different, weaker notion of equivalence: two triples are equivalent if they have a single entry in common.

There are obvious trade-offs between the two equivalence relations. One is more precise but may generate a larger number of smaller equivalence classes. The other is weaker and may generate a smaller number of larger equivalence classes. The relative performance may depend on implementation details and be strongly data dependent.

8.7 Is quadratic optimal?

It is interesting to know how close to being optimal this algorithm is. To know this requires knowing just how many common planes there can be. Consider a set of vectors in three dimensions, for simplicity, in the positive orthant ($x \geq 0$, $y \geq 0$, $z \geq 0$). Now consider the projection of the vectors on the triangle T defined by

$$x + y + z = M, \quad x \geq 0, y \geq 0, z \geq 0 \tag{8.49}$$

where $M > 0$ could be arbitrary, but we will take it to be sufficiently large to simplify our notation. Three such vectors lie in a plane through the origin if and only if the projections onto T are collinear. We now construct a set of n points with $\mathcal{O}(n^2)$ common planes.

Let k be a positive integer, and consider the points in the rectangular lattice

$$(i, j), \quad i = 1, \ldots, 2k, \quad j = 1, 2, 3 \tag{8.50}$$

We see that for each point with $j = 0$ we can associate k lines going through three points, and thus there are at least $2k^2$ common planes. Figure 1 shows an example with $k = 4$ showing only four of the eight sets of four planes for $i = 1, 2, 3, 4$.

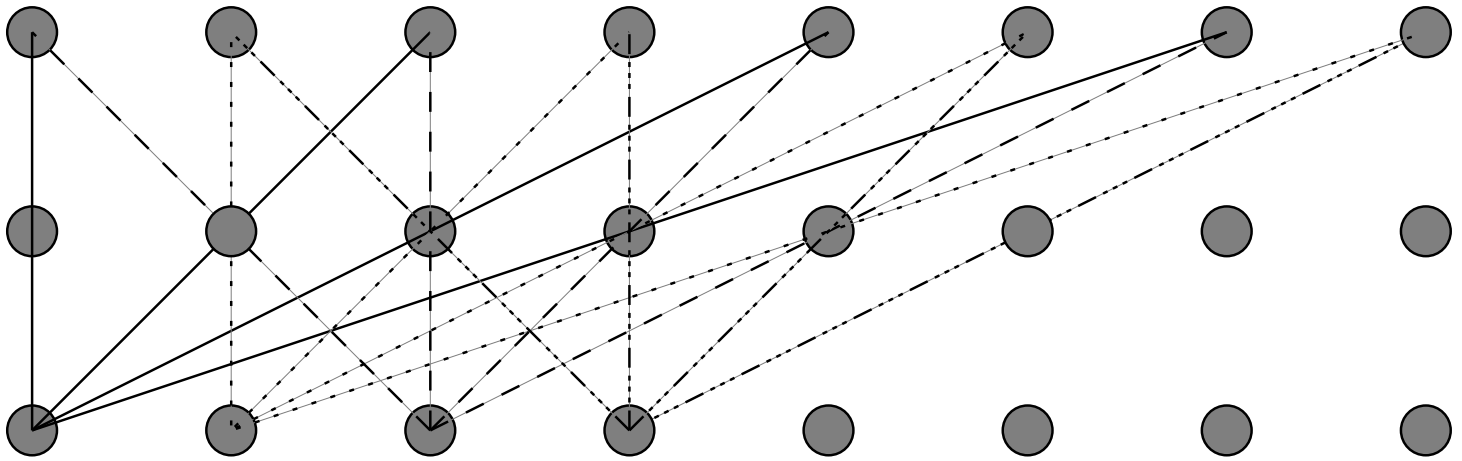


Figure 1: Example of lattice with $k = 4$. For each point on the lower line, there are exactly four planes. Only the planes for $i = 1, 2, 3, 4$ are shown.

Since the number of planes to be determined is quadratic in the number of initial vectors, a quadratic algorithm for determining them is the best we would expect in the worst case.

9 FErari for matrix action

Quadratic Lagrange elements for scalar gradient form in two-D.

Indicated are amounts **per element** (for matrix representation only). A typical vector requires two words per element.

Method used to compute form action	sparse mem refs	local mem refs	floating point ops	total memory
Store Elem. Stiff. Mat.	54	0	72	36
FErari Elem. Stiff. Mat.	21	8	78	3
quadrature/special	21	6	62	3
Global Stiff. Mat.	27	0	46	23

Conclusion: FErari is not compelling, but very competitive.

FErari masks cost of computing local stiffness matrix.

10 Conclusions

The determination of local element matrices involves a novel problem in computational complexity.

There is a mapping from (small) geometry matrices to "difference stencils" that must be computed.

We have demonstrated the potential speed-up available with simple low-order methods.

We have suggested by examples that it may be possible to automate this to some degree by solving abstract graph optimization problems.

Algorithm for determining co-planarity can find dependences in the computation of finite element matrices automatically.

The code `FErari` (for Finite Element ReARrangemnts of Integrals) was developed to carry out this type of optimization automatically.