1 Code generation example: 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, called FErari

Finite Element ReArRangement of Integrals

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.

1.1 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.

Critical step: to determine 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.

1.2 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.

2 Operators related to multilinear forms

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

$$a(v,u) = F(v) \quad \forall v \in \mathcal{V} \tag{2.1}$$

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

$$AU = F \quad \forall v \in \mathcal{V} \tag{2.2}$$

where

$$A_{ij} := a(\phi_i, \phi_j) \quad F_j := F(\phi_j) \quad u := \sum_{i \in \mathcal{I}} U_i \phi_i \tag{2.3}$$

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

$$v := \sum_{i \in \mathcal{I}} V_i \phi_i \tag{2.4}$$

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

$$\begin{aligned}(AV)_i =& \sum_{j \in \mathcal{I}} A_{ij} V_j = \sum_{j \in \mathcal{I}} a(\phi_i, \phi_j) V_j \\ =& a(\phi_i, \sum_{j \in \mathcal{I}} V_j \phi_j) = a(\phi_i, v)\end{aligned} \tag{2.5}$$

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

$$a(\mathcal{V}, v) := AV\,. \qquad (2.6)$$

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 (2.6), 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

$$a(v, u) = 0 \quad \forall v \in \mathcal{V}. \qquad (2.7)$$

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

$$a(v, u^0) = -a(v, g) \quad \forall v \in \mathcal{V}. \tag{2.8}$$

In matrix form, we would write this as

$$AU^0 = -a(\mathcal{V}, g). \tag{2.9}$$

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 (2.7) as

$$a(\mathcal{V}, u^0) = -a(\mathcal{V}, g). \tag{2.10}$$

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

2.1 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

$$a(\mathbf{u}, \mathbf{w}) = c(\mathbf{v}, \tilde{\mathbf{v}}, \mathbf{w}) \quad \forall \mathbf{w} \in \mathcal{V} \tag{2.11}$$

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 (2.3), we set

$$A_{ij} := a(\phi_i, \phi_j) \quad \forall i, j \in \mathcal{I} \tag{2.12}$$

which, by a simple extension of our convention (2.6), can be written as

$$A = a(\mathcal{V}, \mathcal{V}). \tag{2.13}$$

Then (2.11) can be written as

$$
\begin{aligned}
\left(A^t U\right)_i =& \sum_{j\in\mathcal{I}} A_{ji} U_j \\
=& \sum_{j\in\mathcal{I}} a(\phi_j, \phi_i) U_j \\
=& a(\sum_{j\in\mathcal{I}} U_j \phi_j, \phi_i) \\
=& a(\mathbf{u}, \phi_i) \\
=& c(\mathbf{v}, \tilde{\mathbf{v}}, \phi_i) \quad \forall i \in \mathcal{I}.
\end{aligned} \tag{2.14}
$$

In notation analogous to that of (2.6), we can write (2.14) as

$$
a(\mathbf{u}, \mathcal{V}) = A^t U = c(\mathbf{v}, \tilde{\mathbf{v}}, \mathcal{V}), \tag{2.15}
$$

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

2.2 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

$$C_{ij} := c(\mathbf{v}, \phi_i, \phi_j) \quad \forall i, j \in \mathcal{I} \tag{2.16}$$

which we write in our shorthand as

$$C = c(\mathbf{v}, \mathcal{V}, \mathcal{V}) \tag{2.17}$$

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

$$\mathbf{u} + \mathbf{v} \cdot \nabla \mathbf{u} = \mathbf{f} \tag{2.18}$$

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

$$(\mathbf{u}, \mathbf{w})_{L^2} + c(\mathbf{v}, \mathbf{u}, \mathbf{w}) = (\mathbf{f}, \mathbf{w})_{L^2} \quad \forall \mathbf{w} \in \mathcal{V} \tag{2.19}$$

Now write the variational equation

$$(\mathbf{u},\mathbf{w})_{L^2} + c(\mathbf{v},\mathbf{u},\mathbf{w}) = (\mathbf{f},\mathbf{w})_{L^2} \quad \forall \mathbf{w} \in \mathcal{V} \tag{2.20}$$

in component form:

$$\sum_{i\in\mathcal{I}} U_i\left((\phi_i,\phi_j)_{L^2} + c(\mathbf{v},\phi_i,\phi_j)\right) = (\mathbf{f},\phi_j)_{L^2} \quad \forall j \in \mathcal{I} \tag{2.21}$$

In operator notation, this becomes

$$U^t\left((\mathcal{V},\mathcal{V})_{L^2} + c(\mathbf{v},\mathcal{V},\mathcal{V})\right) = F \tag{2.22}$$

2.3 General tensors from Forms

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

$$b(v,p) := \int \nabla\cdot \mathbf{v}(x)p(x)\,dx \tag{2.23}$$

The form $b(\cdot,\cdot)$ in (2.23) 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 (2.12) and (2.13):

$$\left(b(\mathcal{V},\Pi)\right)_{ij} := b(\phi_i, q_i) \tag{2.24}$$

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

$$a\left(\ldots, \mathcal{V}^1, \ldots, \mathcal{V}^k, \ldots\right) \tag{2.25}$$

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.

3 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

$$\sum_{e} \sum_{\lambda \in \mathcal{L}} f_{\iota(e,\lambda)} \phi_{\lambda}^{e} \tag{3.26}$$

where f_i denotes the "nodal value" of the finite element function at the i-th node in the global numbering scheme and $\{\phi_{\lambda}^{e} \ : \ \lambda \in \mathcal{L}\}$ 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, $\xi \rightarrow J\xi + x_e$, of $\mathcal{T}$ to T_e:

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

The inverse mapping, $x \rightarrow \xi = J^{-1}(x - x_e)$ 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, $\det J = 1/\det J^{-1}$.

3.1 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

$$\begin{aligned} a_e(v,w) &:= \int_{T_e} \nabla v(x) \cdot \nabla w(x)\, dx \\ &= \int_{\mathcal{T}} \sum_{j=1}^{d} \frac{\partial}{\partial x_j} v(J\xi + x_e) \frac{\partial}{\partial x_j} w(J\xi + x_e) \det(J)\, d\xi \end{aligned} \tag{3.27}$$

by transofrming to the reference element.

Finite element matrices computed via assembly in a similar way.

The local element form is computed as follows.

3.2 Evaluation of bilinear forms—continued

$$
\begin{aligned}
a_e(v,w) &= \int_{\mathcal{T}} \sum_{j=1}^{d} \frac{\partial}{\partial x_j} v(J\xi + x_e) \frac{\partial}{\partial x_j} w(J\xi + x_e) \det(J)\, d\xi \\
&= \int_{\mathcal{T}} \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial}{\partial \xi_m} \left(\sum_{\lambda \in \mathcal{L}} v_{\iota(e,\lambda)} \phi_\lambda(\xi) \right) \times \\
&\qquad\qquad \frac{\partial \xi_{m'}}{\partial x_j} \frac{\partial}{\partial \xi_{m'}} \left(\sum_{\mu \in \mathcal{L}} w_{\iota(e,\mu)} \phi_\mu(\xi) \right) \det(J)\, d\xi \qquad (3.28) \\
&= \begin{pmatrix} v_{\iota(e,1)} \\ \cdot \\ \cdot \\ v_{\iota(e,|\mathcal{L}|)} \end{pmatrix}^t \mathbf{K}^e \begin{pmatrix} w_{\iota(e,1)} \\ \cdot \\ \cdot \\ w_{\iota(e,|\mathcal{L}|)} \end{pmatrix} .
\end{aligned}
$$

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

$$K^e_{\lambda,\mu} := \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \det(J) \int_{\mathcal{T}} \frac{\partial}{\partial \xi_m} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{m'}} \phi_\mu(\xi)\, d\xi$$
$$= \sum_{m,m'=1}^{d} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{3.29}$$

where

$$K_{\lambda,\mu,m,m'} = \int_{\mathcal{T}} \frac{\partial}{\partial \xi_m} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{m'}} \phi_\mu(\xi)\, d\xi \tag{3.30}$$

and

$$G^e_{m,m'} := \det(J) \sum_{j=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \tag{3.31}$$

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

3.3 Computation of Bilinear Form Matrices

The matrix associated with a bilinear form,

$$A_{ij} := a(\phi_i, \phi_j) = \sum_e a_e(\phi_i, \phi_j) \tag{3.32}$$

for all i, j, can be computed 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

$$A_{\iota(e,\lambda),\iota(e,\mu)} += K^e_{\lambda,\mu} = \sum_{m,m'} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{3.33}$$

where $G^e_{m,m'}$ and $K_{\lambda,\mu,m,m'}$ are defined via

$$G^e_{m,m'} = \det(J) \sum_{j=1}^{d} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \tag{3.34}$$

$$K_{\lambda,\mu,m,m'} = \int_{\mathcal{T}} \frac{\partial}{\partial \xi_m} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{m'}} \phi_\mu(\xi)\, d\xi \tag{3.35}$$

3.4 Matrix computation strategy

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

We optimize the computation of each

$$K^e_{\lambda,\mu} = \sum_{m,m'} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{3.36}$$

This is a collection of dot products of fixed vectors (the tensors K) with a varying set of vectors (the "geometry" information encoded in the G's).

Pre-computations can be done, based on relations among the K's, that reduce computational effort substantially.

3.5 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

The detailed algorithm for computing K for quadratics: $6 * K^e =$

$$\begin{pmatrix} 3G_{11} & -G_{12} & \gamma_{11} & \gamma_0 & 4G_{12} & 0 \\ -G_{21} & 3G_{22} & \gamma_{22} & 0 & 4G_{21} & \gamma_1 \\ \gamma_{11} & \gamma_{22} & 3(\gamma_{11}+\gamma_{22}) & \gamma_0 & 0 & \gamma_1 \\ \gamma_0 & 0 & \gamma_0 & \gamma_2 & -\gamma_3 - 8G_{22} & \gamma_3 \\ 4G_{21} & 4G_{12} & 0 & -\gamma_3 - 8G_{22} & \gamma_2 & -\gamma_3 - 8G_{11} \\ 0 & \gamma_1 & \gamma_1 & \gamma_3 & -\gamma_3 - 8G_{11} & \gamma_2 \end{pmatrix}$$

where the G_{ij}'s are the inputs and the quantities γ_i are defined by

$$\begin{aligned} \gamma_0 =& -4\gamma_{11}, \\ \gamma_1 =& -4\gamma_{22}, \\ \gamma_2 =& 4G_{1221} + 8G_{1122} = \gamma_3 + 8\gamma_{12} = 8(G_{12} + \gamma_{12}), \\ \gamma_3 =& 4G_{1221} = 4\gamma_{21} = 8G_{12} \end{aligned} \tag{3.37}$$

where we use the notation $G_{ijk\ell} := G_{ij} + G_{k\ell}$; finally the γ_{ij}'s are

$$\begin{pmatrix} \gamma_{11} = G_{11} + G_{12} = G_{1112} & \gamma_{12} = G_{11} + G_{22} = G_{1122} \\ \gamma_{21} = G_{12} + G_{21} = G_{1221} & \gamma_{22} = G_{12} + G_{22} = G_{1222} \end{pmatrix} \tag{3.38}$$

3.6 Symmetry properties

Let us distinguish different types of operations.

The above formulas involve (a) negation, (b) multiplication of integers and floating-point numbers, and (c) additions of floating-point numbers.

Since the order of addition is arbitrary, we may assume that the operations (c) are commutative (although changing the order of evaluation may change the result).

Thus we have $G_{1222} = G_{2212}$ and so forth. The symmetry of G implies that $G_{1112} = G_{1121}$ and $G_{2122} = G_{1222}$.

The symmetry of G implies that K^e is also symmetric, by inspection, as it must be from the definition.

3.7 Algorithm details

The computation of the entries of K^e procedes as follows.

The computations in (3.38) are done first and require only four (c) operations, or three (c) operations and one (b) operation ($\gamma_{21} = 2G_{12}$).

Next, the γ_i's are computed via (3.37), requiring four (b) operations and one (c) operation.

Finally, the matrix K^e is completed, via three (a) operations, seven (b) operations, and three (c) operations.

This makes a total of three (a) operations, twelve (b) operations, and three (c) operations.

Thus only eighteen operations are required to evaluate K^e, compared with 288 operations via the formula (3.36).

3.8 Optimality?

There may be other algorithms with the same amount of work (or less) since there are many ways to decompose some of the sub-matrices in terms of others.

Finding (or proving) the absolute minimum may be difficult.

The metric for minimization should be run time, not some arbitrary way of counting operations.

May need to identify sets of ways to evaluate finite element matrices. These could then be tested on different systems (architectures plus compilers) to see which is the best.

It takes fewer operations to compute K^e than it does to write it down, so memory traffic must be considered.

Just elimating multiplication by zero often reduces floating point operation cost below memory cost.

3.9 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 can be less than cost of memory references.

Table 1: The tensor K for cubics two dimensions on triangles, represented as a matrix of two by two matrices. Common denominator is 160.

68	68	-14	0	0	-14	-6	-6	-6	-6	6	54	6	-108	-108	6	54	6	0	0
68	68	-14	0	0	-14	-6	-6	-6	-6	6	54	6	-108	-108	6	54	6	0	0
-14	-14	68	0	0	14	6	114	6	-48	-6	0	-6	0	54	48	-108	-114	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	0	0	0	0	0	0
-14	-14	14	0	0	68	-48	6	114	6	-114	-108	48	54	0	-6	0	-6	0	0
-6	-6	6	0	0	-48	270	135	-54	135	54	27	54	27	0	27	0	-135	-324	-162
-6	-6	114	0	0	6	135	270	-27	-54	27	0	27	0	27	54	-135	-270	-162	0
-6	-6	6	0	0	114	-54	-27	270	135	-270	-135	54	27	0	27	0	27	0	-162
-6	-6	-48	0	0	6	135	-54	135	270	-135	0	27	0	27	54	27	54	-162	-324
6	6	-6	0	0	-114	54	27	-270	-135	270	135	-54	-27	0	-27	0	-27	0	162
54	54	0	0	0	-108	27	0	-135	0	135	270	-189	-216	-27	0	-27	0	162	0
6	6	-6	0	0	48	54	27	54	27	-54	-189	270	135	0	135	0	-27	-324	-162
-108	-108	0	0	0	54	27	0	27	0	-27	-216	135	270	135	0	-27	0	-162	0
-108	-108	54	0	0	0	0	27	0	27	0	-27	0	135	270	135	-216	-27	0	-162
6	6	48	0	0	-6	27	54	27	54	-27	0	135	0	135	270	-189	-54	-162	-324
54	54	-108	0	0	0	0	-135	0	27	0	-27	0	-27	-216	-189	270	135	0	162
6	6	-114	0	0	-6	-135	-270	27	54	-27	0	-27	0	-27	-54	135	270	162	0
0	0	0	0	0	0	-324	-162	0	-162	0	162	-324	-162	0	-162	0	162	648	324
0	0	0	0	0	0	-162	0	-162	-324	162	0	-162	0	-162	-324	162	0	324	648

3.10 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:

$$
4\mathbf{K} = \begin{array}{|ccc|ccc|ccc|ccc|}
\hline
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & -1 \\
\hline
-1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 1 & 1 & 1 \\
-1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 1 & 1 & 1 \\
-1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 1 & 1 & 1 \\
\hline
\end{array}
$$

3.11 Algorithm for linears in three-D

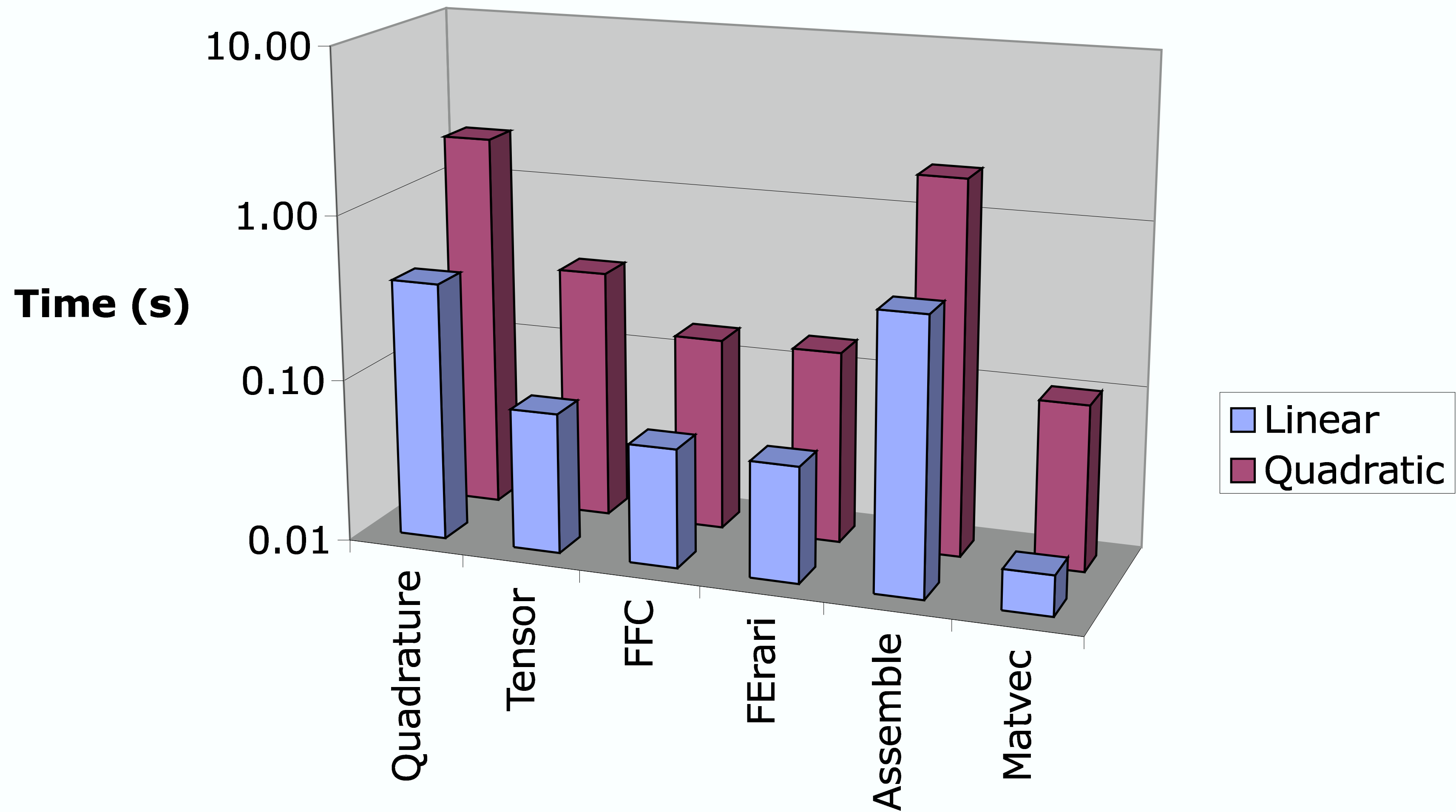
Each K^e can be computed by computing the three row sums of G^e, the three column sums, and the sum of one of these sums.

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.



Using FErari and tensor reduction (FFC): low order elements in 2D.

3.12 Collinearity tests

Two vectors in $I\!R^m$ 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 $\mathcal{O}(m)$ 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 $\mathcal{O}(mn \log n)$ arithmetic operations.

A hash table can be used to reduce this to an expected $\mathcal{O}(mn)$ arithmetic operations.

3.13 A (random) dimensional reduction algorithm

A randomized approach could be faster for large m.

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 m) 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.

4 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.

Algorithm is simple and attractive but

cost is cubic in the number of vectors.

4.1 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.

4.2 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.

4.3 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.

4.4 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

$$x + y + z = M, \quad x \geq 0, y \geq 0, z \geq 0 \tag{4.39}$$

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

$$(i, j), \quad i = 1, \ldots, 2k, \quad j = 1, 2, 3 \tag{4.40}$$

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 $i = 1, 2, 3, 4$.

4.5 Yes quadratic is optimal

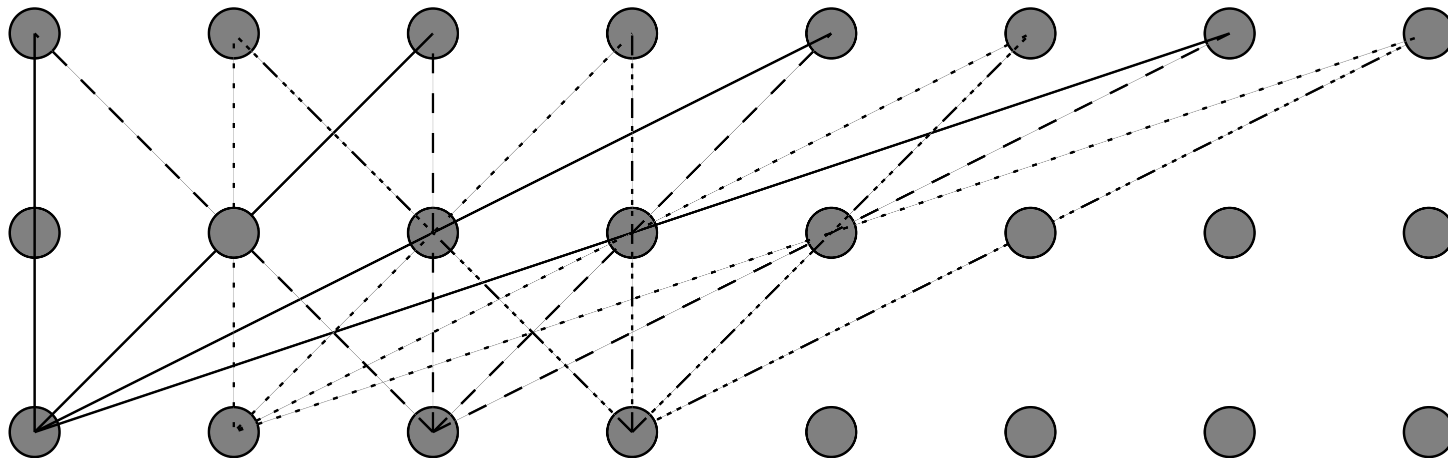


Figure 1: Example of lattice with $k = 4$. For each point on the lower line, there are exactly four planes. Only the planes for $i = 1, 2, 3, 4$ are shown.

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.

5 Evaluation of general multi-linear forms

Arbitrary multi-linear forms can appear in finite element calculations, e.g., the nonlinear form $c(\cdot,\cdot,\cdot)$ in Navier–Stokes:

$$\begin{aligned}
c_e(\mathbf{u},\mathbf{v},\mathbf{w}) &:= \int_{T_e} \mathbf{u}\cdot\nabla\mathbf{v}(x)\cdot\mathbf{w}(x)\,dx \\
&= \int_{T_e} \sum_{j,k=1}^{d} u_j(x)\frac{\partial}{\partial x_j} v_k(x) w_k(x)\,dx \\
&= \int_{\mathcal{T}} \sum_{j,k=1}^{d} u_j(J\xi+x_e)\frac{\partial}{\partial x_j} v_k(J\xi+x_e) w_k(J\xi+x_e)\det(J)\,d\xi \\
&= \int_{\mathcal{T}} \sum_{j,k,m=1}^{d} \left(\sum_{\lambda\in\mathcal{L}} u_j^{\iota(e,\lambda)}\phi_\lambda(\xi)\right)\frac{\partial\xi_m}{\partial x_j}\left(\sum_{\mu\in\mathcal{L}} v_k^{\iota(e,\mu)}\frac{\partial}{\partial\xi_m}\phi_\mu(\xi)\right)\times \\
&\qquad \left(\sum_{\rho\in\mathcal{L}} w_k^{\iota(e,\rho)}\phi_\rho(\xi)\right)\det(J)\,d\xi
\end{aligned} \tag{5.41}$$

Therefore

$$c_e(\mathbf{u},\mathbf{v},\mathbf{w}) = \sum_{j,k=1}^{d} \sum_{\lambda,\mu,\rho\in\mathcal{L}} u_j^{\iota(e,\lambda)} v_k^{\iota(e,\mu)} w_k^{\iota(e,\rho)} \sum_{m=1}^{d} \frac{\partial \xi_m}{\partial x_j} \det(J) N_{\lambda,\mu,\rho,m}$$
$$= \sum_{k=1}^{d} \sum_{\mu,\rho\in\mathcal{L}} v_k^{\iota(e,\mu)} w_k^{\iota(e,\rho)} \sum_{j=1}^{d} \sum_{\lambda\in\mathcal{L}} u_j^{\iota(e,\lambda)} N^e_{\lambda,\mu,\rho,j} \tag{5.42}$$

where

$$N_{\lambda,\mu,\rho,m} := \int_{\mathcal{T}} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_m} \phi_\mu(\xi) \phi_\rho(\xi)\, d\xi \tag{5.43}$$

$$N^e_{\lambda,\mu,\rho,j} := \sum_{m=1}^{d} \frac{\partial \xi_m}{\partial x_j} \det(J) N_{\lambda,\mu,\rho,m}. =: \sum_{m=1}^{d} \widetilde{G}_{mj} N_{\lambda,\mu,\rho,m}. \tag{5.44}$$

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

.

Matrix $C_{ij} = c(\mathbf{u}, \phi_i, \phi_j)$ can be computed using assembly (C is block diagonal; let I_d denote the $d \times d$ identity matrix)

$$\begin{aligned}
C_{\iota(e,\mu),\iota(e,\rho)} + = & I_d \sum_{j=1}^{d} \sum_{\lambda \in \mathcal{L}} u_j^{\iota(e,\lambda)} N^e_{\lambda,\mu,\rho,j} \\
= & I_d \sum_{m,j=1}^{d} \widetilde{G}_{mj} \left(\sum_{\lambda \in \mathcal{L}} u_j^{\iota(e,\lambda)} N_{\lambda,\mu,\rho,m} \right) \\
= & I_d \sum_{m,\lambda \in \mathcal{L}} \gamma^{e,u}_{m\lambda} N_{\lambda,\mu,\rho,m}
\end{aligned} \tag{5.45}$$

for all μ and ρ, where

$$\gamma^{e,u}_{m\lambda} = \sum_{j=1}^{d} \widetilde{G}_{mj} u_j^{\iota(e,\lambda)}. \tag{5.46}$$

Computation of C similar in form to (**??**), and similar optimization techniques apply. Then the update of C is done in the obvious way with $K^{e,u}$ where

$$K^{e,u}_{\mu,\rho} = \sum_{m,\lambda \in \mathcal{L}} \gamma^{e,u}_{m\lambda} N_{\lambda,\mu,\rho,m} \tag{5.47}$$

5.1 Linear elements in three dimensions

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

Intermediate vectors

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

$$K^{e,u} = \begin{pmatrix} \tilde{\gamma}_{11} & \tilde{\gamma}_{21} & \tilde{\gamma}_{31} & \tilde{\gamma}_{11} + \tilde{\gamma}_{21} + \tilde{\gamma}_{31} \\ \tilde{\gamma}_{12} & \tilde{\gamma}_{22} & \tilde{\gamma}_{32} & \tilde{\gamma}_{12} + \tilde{\gamma}_{22} + \tilde{\gamma}_{32} \\ \tilde{\gamma}_{13} & \tilde{\gamma}_{23} & \tilde{\gamma}_{33} & \tilde{\gamma}_{13} + \tilde{\gamma}_{23} + \tilde{\gamma}_{33} \\ \tilde{\gamma}_{14} & \tilde{\gamma}_{24} & \tilde{\gamma}_{34} & \tilde{\gamma}_{14} + \tilde{\gamma}_{24} + \tilde{\gamma}_{34} \end{pmatrix} \qquad (5.48)$$

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 (5.47).

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

5.2 Algorithmic implications

The examples provide guidance for the general case.

The “vector” space of the evaluation problem (5.47) 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 relationships 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

$$\begin{aligned}\mathbf{V}_1 =&(1,1,1,1,0,0,0,0,0,0,0,0),\\ \mathbf{V}_2 =&(0,0,0,0,1,1,1,1,0,0,0,0),\\ \mathbf{V}_3 =&(0,0,0,0,0,0,0,0,1,1,1,1)\end{aligned} \tag{5.49}$$

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.

6 The FErari system

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

Vectors grouped 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 2: 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

Non-conforming elements have fewer simple relations, but coplanarity relations can still reduce computation substantially.

Table 3: FErari at work on Nonconforming Lagrange elements in two dimensions. All of the vectors (Total) 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
3	100	0	11	1	0	0	0	88	177
5	441	0	105	0	0	0	0	336	672

6.1 FErari search strategy

FErari searched 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

6.1.1 FErari search strategy — continued

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.

The search for high-order geometric relations can be expensive.
FErari needs efficient search strategies.

6.2 Review of where we are

Finite element structure allows automation of software generation (e.g., variational form language).

Need to generate efficient code leads to re-examination of finite element computation. Example: finite element matrix computation.

Finite element matrix computation introduces new problem in computational complexity.

- FErari automates the generation of code to compute finite element matrices.
  - Now we need to optimize FErari to carry out these optimizations efficiently.

7 Higher-dimensional dependences

The algorithm described in section **??** can be generalized to higher-dimensional dependences in an obvious way. The linearly dependence of four vectors can be viewed as the case when two three-dimensional subspaces, generated by two trios of the vectors, coincide. If we take the normals to the three-dimensional subspaces generated by each trio of vectors in four dimensions, then the coincidence of the three-dimensional subspaces is reduced to checking collinearity of the normals. This can be done by the algorithms described in section **??**. Thus finding linear dependence of four vectors among a total of n vectors can be done in nearly $\mathcal{O}(n^3)$ work.

In a similar way, we can determine linearly dependence of $d+1$ vectors among a total of n vectors in nearly $\mathcal{O}(n^d)$ work.

8 Variable coefficients

We begin with a simple example, the weighted Laplacian:

$$a(v, w) := \int_\Omega \omega(x) \nabla v(x) \cdot \nabla w(x)\, dx \tag{8.50}$$

Let V be the space for the approximation ($v, w \in V$). What we really compute with is the projection of ω onto the space S of products of things in ∇V (for the Laplacian).

S provides equivalent "exact integration." Smaller spaces $\widetilde{S} \subset S$ may also yield a desired (maybe optimal) order of approximation.

For linears, grad V is just piecewise constants, so we can take S to be piecewise constants. If V consists of piecewise polynomials of degree k, then we can take S to be piecewise polynomials of degree $2k - 2$.

There is still the problem of determining the projection of ω onto S. For now, we will just assume that $\omega \in S$.

8.1 Evaluation of weighted bilinear forms

$$
\begin{aligned}
a_e(v,w) &:= \int_{T_e} \omega(x)\nabla v(x)\cdot\nabla w(x)\,dx \\
&= \int_{\mathcal{T}} \omega(J\xi + x_e)\sum_{j=1}^{d}\frac{\partial}{\partial x_j}v(J\xi+x_e)\frac{\partial}{\partial x_j}w(J\xi+x_e)\det(J)\,d\xi \\
&= \int_{\mathcal{T}} \omega(J\xi + x_e)\sum_{j,\alpha_1,\alpha_2=1}^{d}\frac{\partial\xi_{\alpha_1}}{\partial x_j}\frac{\partial}{\partial\xi_{\alpha_1}}\left(\sum_{\lambda\in\mathcal{L}} v_{\iota(e,\lambda)}\phi_\lambda(\xi)\right)\times \\
&\qquad\qquad \frac{\partial\xi_{\alpha_2}}{\partial x_j}\frac{\partial}{\partial\xi_{\alpha_2}}\left(\sum_{\mu\in\mathcal{L}} w_{\iota(e,\mu)}\phi_\mu(\xi)\right)\det(J)\,d\xi
\end{aligned}
\tag{8.51}
$$

Let us assume that $\omega\in S$ can be expanded as

$$
\omega|_{T_e}(J\xi+x_e) = \sum_{\kappa\in\mathcal{S}}\omega^e_\kappa\sigma_\kappa(\xi). \tag{8.52}
$$

Then

$$a_e(v,w) = \sum_{\kappa\in\mathcal{S}} \sum_{\lambda,\mu\in\mathcal{L}} \omega_\kappa^e v_{\iota(e,\lambda)} w_{\iota(e,\mu)} A^e_{\kappa,\lambda,\mu} \tag{8.53}$$

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

$$\begin{aligned} A^e_{\kappa,\lambda,\mu} &:= \sum_{j,\alpha_1,\alpha_2=1}^{d} \frac{\partial \xi_{\alpha_1}}{\partial x_j} \frac{\partial \xi_{\alpha_2}}{\partial x_j} \det(J) \int_{\mathcal{T}} \sigma_\kappa \frac{\partial}{\partial \xi_{\alpha_1}} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{\alpha_2}} \phi_\mu(\xi)\, d\xi \\ &= \sum_{\alpha_1,\alpha_2=1}^{d} G^e_{\alpha_1,\alpha_2} A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2} \end{aligned} \tag{8.54}$$

where

$$A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2} = \int_{\mathcal{T}} \sigma_\kappa \frac{\partial}{\partial \xi_{\alpha_1}} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{\alpha_2}} \phi_\mu(\xi)\, d\xi \tag{8.55}$$

and

$$G^e_{\alpha_1,\alpha_2} := \det(J) \sum_{j=1}^{d} \frac{\partial \xi_{\alpha_1}}{\partial x_j} \frac{\partial \xi_{\alpha_2}}{\partial x_j} \tag{8.56}$$

for $\lambda, \mu \in \mathcal{L}$ and $\alpha_1, \alpha_2 = 1, \ldots, d$.

8.2 Computation of Bilinear Form Matrices

The matrix associated with the bilinear form $a(\cdot,\cdot)$ can be computed by the standard assembly algorithm, with the matrix update being of the form

$$\mathcal{A}_{\iota(e,\lambda),\iota(e,\mu)} += \sum_{\alpha_1,\alpha_2=1}^{d} \sum_{\kappa\in\mathcal{S}} \omega_\kappa^e G^e_{\alpha_1,\alpha_2} A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2} \tag{8.57}$$

where $G^e_{\alpha_1,\alpha_2} = \det(J)\sum_{j=1}^{d} \frac{\partial\xi_{\alpha_1}}{\partial x_j}\frac{\partial\xi_{\alpha_2}}{\partial x_j}$.

There are several approaches to take at this point.

8.2.1 Full tensor approach

Using (8.57), the product tensors $\omega_\kappa^e G^e_{\alpha_1,\alpha_2}$ could be formed and processed via FErari applied to the full tensor $A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2}$. This requires $d^2|\mathcal{L}|$ multiplications initially, and then FErari is used with $|\mathcal{L}|^2$ tensors in a space of dimension $d^2|\mathcal{L}|$.

8.2.2 Product structure approach: I

The expression (8.57) can be written

$$\mathcal{A}_{\iota(e,\lambda),\iota(e,\mu)} += \sum_{\kappa\in\mathcal{S}} \omega_\kappa^e A_{\kappa,\lambda,\mu}^e \tag{8.58}$$

where $A_{\kappa,\lambda,\mu}^e$ was defined in (8.54):

$$A_{\kappa,\lambda,\mu}^e = \sum_{\alpha_1,\alpha_2=1}^{d} G_{\alpha_1,\alpha_2}^e A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2}$$

FErari can optimize the computation of each $A_{\kappa,\lambda,\mu}^e$, but each such term must be computed separately.

This requires FErari to be used with $|\mathcal{L}|^3$ tensors in a space of dimension d^2, followed by $|\mathcal{L}|$ multiplications in (8.58).

Using FErari with more vectors in a lower-dimensional space is likely to be more efficient that using FErari with fewer vectors in a higher-dimensional space. And the additional computation is reduced as well in the second approach.

8.2.3 Product structure approach: II

A third way to use FErari is to define the matrix update via

$$\mathcal{A}_{\iota(e,\lambda),\iota(e,\mu)} += \sum_{\alpha_1,\alpha_2=1}^{d} G^e_{\alpha_1,\alpha_2} A^\omega_{\lambda,\mu,\alpha_1,\alpha_2}, \tag{8.59}$$

where $A^\omega_{\lambda,\mu,\alpha_1,\alpha_2}$ are computed via FErari from the definition

$$A^\omega_{\lambda,\mu,\alpha_1,\alpha_2} = \sum_{\kappa\in\mathcal{S}} \omega^e_\kappa A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2}. \tag{8.60}$$

This requires FErari to be used with $d^2|\mathcal{L}|^2$ tensors in a space of dimension $|\mathcal{L}|$, followed by d^2 MAPs in (8.58).

FErari can generate optimized code for all of these approaches together with an estimate of computational complexity.

All three approaches provide different strategies to generate code for finite element matrix definition.

8.2.4 Weighted Laplacian for cubics in 3-D

Comuting the matrix for the weighted Laplacian for cubics in three dimensions: n= number of vector dot products, m is the dimension of the vector space.

Note that $mn = 25200$ is the number of MAPs required for conventional tensor contraction in all cases.

Strategy	n	m	MST MAPs	additional	total MAPs
Full tensor	21	120	14334	120	14454
G first	4200	6	7021	4200	11221
ω first	1260	20	7728	1260	8988

These are based only on certain tensor relations using a Minimum Spanning Tree (MST). Further reductions may acrue from using geometric relations.

Note large size of computation and substantial reductions.

8.3 Evaluation of Bilinear Form Actions

The matrix action associated with the bilinear form $a(\cdot,\cdot)$ can be computed by the standard assembly algorithm as well. For example, the action $v = a(V, w)$ has a vector update update that can be written in the following equivalent forms:

$$
\begin{aligned}
v_{\iota(e,\lambda)} += & \sum_{\alpha_1,\alpha_2,\kappa,\mu} \omega_\kappa^e G^e_{\alpha_1,\alpha_2} A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2} w_{\iota(e,\mu)} \\
= & \sum_{\kappa,\mu} \omega_\kappa^e A^e_{\kappa,\lambda,\mu} w_{\iota(e,\mu)} \\
= & \sum_{\alpha_1,\alpha_2,\mu} G^e_{\alpha_1,\alpha_2} A^\omega_{\lambda,\mu,\alpha_1,\alpha_2} w_{\iota(e,\mu)} \\
= & \sum_{\alpha_1,\alpha_2,\kappa} \omega_\kappa^e G^e_{\alpha_1,\alpha_2} A^w_{\kappa,\lambda,\alpha_1,\alpha_2}
\end{aligned}
\tag{8.61}
$$

where $A^w_{\kappa,\lambda,\alpha_1,\alpha_2} = \sum_\mu A_{\kappa,\lambda,\mu,\alpha_1,\alpha_2} w_{\iota(e,\mu)}$ would be computed via FErari.

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 Computing a matrix via quadrature

The computations in equations (3.32–3.33) can be computed via quadrature as

$$\begin{aligned} A_{\iota(e,\lambda),\iota(e,\mu)} += & \sum_{\xi\in\Xi} \omega_\xi \nabla\phi_\lambda(\xi) \cdot (\mathbf{G}^e \nabla\phi_\mu(\xi)) \\ = & \sum_{\xi\in\Xi} \omega_\xi \sum_{m,n=1}^{d} \phi_{\lambda,m}(\xi) G^e_{m,n} \phi_{\mu,n}(\xi) \\ = & \sum_{m,n=1}^{d} G^e_{m,n} \sum_{\xi\in\Xi} \omega_\xi \phi_{\lambda,m}(\xi)\phi_{\mu,n}(\xi) \\ = & \sum_{m,n=1}^{d} G^e_{m,n} K_{\lambda,\mu,m,n} \end{aligned} \tag{10.62}$$

where the coefficients $K_{\lambda,\mu,m,n}$ are analogous to those defined in (8.52), but here they are defined by quadrature:

$$K_{\lambda,\mu,m,n} = \sum_{\xi \in \Xi} \omega_\xi \phi_{\lambda,m}(\xi) \phi_{\mu,n}(\xi) \tag{10.63}$$

(The coefficients are exactly those of (8.52) if the quadrature is exact.)

The right strategy for computing a matrix via quadrature would thus appear to be to compute the coefficients $K_{\lambda,\mu,m,n}$ first using (9.63), and then proceeding as before.

However, there is a different strategy associated with quadrature when we want only to compute the *action* of the linear operator associated with the matrix and not the matrix itself.

11 Conclusions

Mathematical structure of finite elements supports automation of software generation.

Determining optimal code generation requires re-examination of finite element computations.

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

We have demonstrated the potential speed-up available with simple low-order methods, including their use for matrix action.

The `FErari` system was developed to carry out this type of optimization automatically.

Computational mathematical modeling can become more reliable and efficient by using such tools (see FEniCS.org for more information).