BIT Numerical Mathematics
2003, Vol. 43, No. 1, pp. 001–018

0006-3835/03/4301-0001 $16.00

Evaluation of the Action of Finite Element Operators *

Robert C. Kirby[1] and Matthew G. Knepley[2] and L. Ridgway Scott[3] †

[1] *Department of Computer Science, The University of Chicago; Chicago, Illinois 60637-1581, USA. email: kirby@cs.uchicago.edu*

[2] *Division of Mathematics and Computer Science, Argonne National Laboratory, Argonne, Illinois 60439-4844, USA. email: knepley@mcs.anl.gov*

[3] *The Computation Institute and Departments of Computer Science and Mathematics, University of Chicago, Chicago, Illinois 60637-1581, USA. email: ridg@cs.uchicago.edu*

Abstract.

The Krylov methods frequently used to solve linear systems associated with finite element discretizations of partial differential equations (PDE) rely only on the matrix-vector product. This work considers the relative costs, in terms both of floating point operations and memory traffic, of several approaches to computing the matrix action. These include forming and using a global sparse matrix, building local element matrices and using them with a local-to-global indexing, and computing the action of the local matrices directly by numerical quadrature. Which approach is most efficient depends on several factors, including the relative cost of computation to memory access, how quickly local element matrices may be formed, and how quickly a function expressed in a finite element basis may be differentiated at the quadrature points.

AMS subject classification (2000): 65D05, 65N15, 65N30.

Key words: finite element, compiler, variational form

1 Introduction

The systematic formulation of finite element methods provides the opportunity to automate a substantial amount of finite element computation. In [8], we have considered the problem of automating the efficient computation of finite element matrices. Other efforts related to automation of finite element codes are also discussed in [8].

Here we focus on the efficiency for implementing the computation of the action of operators associated with variational forms on finite element spaces. Such techniques provide what are known as "matrix free" methods, since they allow problems to be solved without ever expliciting computing (or storing) the matrix associated with the linear operators. In particular, we study the use of quadrature to evaluate the integrals that define the action. While this provides a general approach for code generation, it is not the optimal algorithm in all cases.

*
†

One objective of this work is to establish a baseline for designing a system for automated finite element variational approximation. Our examples illustrate trade-offs that should be considered in developing a compiler for variational forms.

2 Examples of variational forms

Simple linear p.d.e. problems use linear and bilinear integro-differential forms. Nonlinear problems and problems with variable coefficients introduce extra functions into such variational forms. The number of functions is the "arity" of the form. In theory, multi-linear forms of any arity can arise, and in practice, trilinear forms are common, and ones of arity four arise frequently. Our objective here is not to describe broad classes of forms that are used frequently, but rather we just introduce a small number to provide some guidance as examples.

Let us list some typical forms that arise in applications. For scalar valued functions

$$a_{\text{Laplace}}(u,v) = \int_\Omega (\nabla u(x)) \cdot (\nabla v(x))\, dx \tag{2.1}$$

is the basic form related to Laplace's equation [3, 5]. The Navier-Stokes equations for vector-valued functions involve several forms, including

$$a_{\text{Stokesgrad}}(\mathbf{u},\mathbf{v}) = \int_\Omega (\nabla \mathbf{u}(x)) : (\nabla \mathbf{v}(x))\, dx, \tag{2.2}$$

$$b_{\text{pressure}}(\mathbf{u},p) = \int_\Omega (\nabla\cdot \mathbf{u}(x)) p(x)\, dx, \tag{2.3}$$

$$c_{\text{momentum}}(\mathbf{u},\mathbf{v},\mathbf{w}) = \int_\Omega (\mathbf{u}(x)\cdot\nabla \mathbf{v}(x))\cdot \mathbf{w}(x)\, dx. \tag{2.4}$$

We use a dot "$\cdot$" to denote the vector dot product and a colon ":" to denote the Frobenius product of matrices.

3 Operators related to multilinear forms

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

$$a(u,v) = F(v) \quad \forall v \in \mathcal{V} \tag{3.1}$$

for a given (continuous, coercive) bilinear form $a(\cdot,\cdot)$. We will want to consider more general problems (defined on pairs of different spaces, etc.), but for now we will keep it simple. This corresponds to a linear system of equations, and one can write the corresponding matrix equation in terms of a basis. In practice, an interpolation basis is typically used, e.g., the standard Lagrange nodal basis $\{\phi_i \,:\, i \in \mathcal{I}\}$ where $\mathcal{I}$ denotes the index set for the nodes. The matrix equation is

$$AU = F \quad \forall v \in \mathcal{V} \tag{3.2}$$

where

$$\begin{aligned} A_{ij} &:= a(\phi_i,\phi_j) \\ F_j &:= F(\phi_j) \\ u &:= \sum_{i\in\mathcal{I}} U_i \phi_i. \end{aligned} \tag{3.3}$$

3.1 Action of bilinear forms

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

$$v := \sum_{i \in \mathcal{I}} V_i \phi_i \tag{3.4}$$

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

$$\begin{aligned} (AV)_i =& \sum_{j \in \mathcal{I}} A_{ij} V_j \\ =& \sum_{j \in \mathcal{I}} a(\phi_i, \phi_j) V_j \\ =& a(\phi_i, \sum_{j \in \mathcal{I}} V_j \phi_j) \\ =& a(\phi_i, v) \end{aligned} \tag{3.5}$$

Thus 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 (3.5) as motivation, we can introduce the notation $a(\mathcal{V}, v)$ where

$$a(\mathcal{V}, v) := AV\,. \tag{3.6}$$

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 (3.6), 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

$$a(v, u) = 0 \quad \forall v \in \mathcal{V}. \tag{3.7}$$

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

$$a(v, u^0) = -a(v, g) \quad \forall v \in \mathcal{V}. \tag{3.8}$$

In matrix form, we would write this as

$$AU^0 = -a(\mathcal{V}, g). \tag{3.9}$$

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 (3.7) as

$$a(\mathcal{V}, u^0) = -a(\mathcal{V}, g). \tag{3.10}$$

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

3.2 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

$$a_{\text{Stokesgrad}}(\mathbf{u}, \mathbf{w}) = c_{\text{momentum}}(\mathbf{v}, \tilde{\mathbf{v}}, \mathbf{w}) \quad \forall \mathbf{w} \in \mathcal{V} \tag{3.11}$$

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 (3.3), we set

$$A_{ij} := a_{\text{Stokesgrad}}(\phi_i, \phi_j) \quad \forall i, j \in \mathcal{I} \tag{3.12}$$

which, by a simple extension of our convention (3.6), can be written as

$$A = a_{\text{Stokesgrad}}(\mathcal{V}, \mathcal{V}). \tag{3.13}$$

Then (3.11) can be written as

$$\begin{aligned} \left(A^t U\right)_i =& \sum_{j \in \mathcal{I}} A_{ji} U_j \\ =& \sum_{j \in \mathcal{I}} a_{\text{Stokesgrad}}(\phi_j, \phi_i) U_j \\ =& a_{\text{Stokesgrad}}(\sum_{j \in \mathcal{I}} U_j \phi_j, \phi_i) \\ =& a_{\text{Stokesgrad}}(\mathbf{u}, \phi_i) \\ =& c_{\text{momentum}}(\mathbf{v}, \tilde{\mathbf{v}}, \phi_i) \quad \forall i \in \mathcal{I}. \end{aligned} \tag{3.14}$$

In notation analogous to that of (3.6), we can write (3.14) as

$$a_{\text{Stokesgrad}}(\mathbf{u}, \mathcal{V}) = A^t U = c_{\text{momentum}}(\mathbf{v}, \tilde{\mathbf{v}}, \mathcal{V}), \tag{3.15}$$

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

3.3 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 such as (2.4), it might be of interest to work with the matrix

$$C_{ij} := c_{\text{momentum}}(\mathbf{v}, \phi_i, \phi_j) \quad \forall i, j \in \mathcal{I} \tag{3.16}$$

which we write in our shorthand as

$$C = c_{\text{momentum}}(\mathbf{v}, \mathcal{V}, \mathcal{V}) \tag{3.17}$$

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

$$\mathbf{u} + \mathbf{v} \cdot \nabla \mathbf{u} = \mathbf{f} \tag{3.18}$$

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

$$(\mathbf{u}, \mathbf{w})_{L^2} + c_{\text{momentum}}(\mathbf{v}, \mathbf{u}, \mathbf{w}) = (\mathbf{f}, \mathbf{w})_{L^2} \quad \forall \mathbf{w} \in \mathcal{V} \tag{3.19}$$

to define $\mathbf{u} \in \mathcal{V}$, where $(\mathbf{u}, \mathbf{w})_{L^2}$ is the standard L^2 inner-product. In component form, this becomes

$$\sum_{i \in \mathcal{I}} U_i \left((\phi_i, \phi_j)_{L^2} + c_{\text{momentum}}(\mathbf{v}, \phi_i, \phi_j) \right) = (\mathbf{f}, \phi_j)_{L^2} \quad \forall j \in \mathcal{I}. \tag{3.20}$$

In operator notation, this becomes

$$U^t \left((\mathcal{V}, \mathcal{V})_{L^2} + c_{\text{momentum}}(\mathbf{v}, \mathcal{V}, \mathcal{V}) \right) = F \tag{3.21}$$

3.4 General tensors from Forms

It may frequently happen that the spaces in a form are not all the same. The form $b_{\text{pressure}}(\cdot,\cdot)$ in (2.3) 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 (3.12) and (3.13):

$$(b(\mathcal{V},\Pi))_{ij} := b(\phi_i, q_i) \tag{3.22}$$

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 $a(v^1,\dots,v^n)$ of n entries, then the expression

$$a\left(\dots,\mathcal{V}^1,\dots,\mathcal{V}^k,\dots\right) \tag{3.23}$$

defines a tensor of rank k. More precisely, each of the n arguments in the form $a(v^1,\dots,v^n)$ may be a function space or a member of a function space. Suppose that we have some k of the n (not necessarily consecutive) arguments that are function spaces, $\mathcal{V}^1,\dots,\mathcal{V}^k$, and $n-k$ members of a function space $v^1,\dots,v^{n-k}$. Then, suppose we have a partition of $\{1,\dots,n\} = \{i_\ell \,:\, \ell = 1,\dots k\} \bigcup \{j_\ell \,:\, \ell = 1,\dots n-k\}$ suppose that $w_{i_\ell} \to \mathcal{V}^{i_\ell}$ for $\ell = 1,\dots k$ and $w_{j_\ell} \to v^{j_\ell}$ for $\ell = 1,\dots n-k$. Here $a \to b$ should be read as "a is a pointer to b." Then $a(w_1,\dots,w_n)$ is a rank k tensor. 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 $a(v^1,\dots,v^n)$. 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.

4 Evaluation by Assembly

Let us consider how this all might work in the case of finite elements, especially Lagrange piecewise polynomial spaces. There is considerable simplification in this case, as the spaces can be generated from the Ciarlet definition of a finite element, at least for affine-equivalent elements [5, 4, 7]. Moreover, the quadrature can be generated from a single quadrature rule on the reference element.

It is interesting to think about the roles of the three ingredients in the Ciarlet definition of "element." The function space is clear; this is what we use to represent the approximation. The nodal variables are similarly used computationally to represent the functions and enforce continuity across adjacent elements. The primary computational role of the domain K is in specifying the integral

$$u \to \int_K u(x)\,dx\,, \tag{4.1}$$

where u is some algebraic combination of members of the function space and their derivatives. That is, K is a proxy for this integral. This role can equally be played by a quadrature rule

$$u \to \int_K u(x)\,dx \approx \sum_\xi u(\xi)\omega_\xi\,, \tag{4.2}$$

where the quadrature might or might not be exact on the forms we are interested in. Moreover, this quadrature rule may involve mapping the integrand to a reference element and manipulating any derivatives appropriately.

Thus we can think of the Ciarlet definition of a finite element as a triple consisting of:

- a function space,
- a set of nodal variables (a basis for the dual space of the function space), and
- quadrature rules for integrating members of the function space (including derivatives and algebraic combinations) that may include transforming to a reference domain.

Here we are using the term "quadrature rule" loosely in the sense of anything that produces the correct integrals. It may be necessary to use different rules in different contexts. For example, the product of gradients of quadratics can be integrated exactly using the edge mid-point rule on simplices. However, the product of quadratics themselves cannot. Similarly, a tri-linear term such as the momentum term in Navier-Stokes (2.4) would not be exact with this rule. One can show that the corresponding methods using inexact quadrature would be optimal-order convergent, so it might be useful to be able to use the simple rules for efficiency.

The *assembly* of integrated differential forms by summing its constituent parts over each *element*, which are computed separately is facilitated 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 the interpolant of a continuous function for the space of all piecewise polynomial (of some degree) functions (no boundary conditions imposed) via

$$f_I := \sum_e \sum_{\lambda \in \mathcal{L}} f(x_{\iota(e,\lambda)}) \phi^e_\lambda \tag{4.3}$$

where $\{\phi^e_\lambda \,:\, \lambda \in \mathcal{L}\}$ denotes the set of basis functions for polynomials of the given degree on T_e. We can relate all of the "element" basis functions ϕ^e_λ to a fixed set of basis functions on a "reference" element, $\mathcal{T}$, via an affine mapping, $\xi \to J\xi + x_e$, of $\mathcal{T}$ to T_e:

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

(By definition, the element basis functions, ϕ^e_λ, are extended by zero outside T_e.) 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, $\det J = 1/\det J^{-1}$.

More complex elements involve more complex mappings to the reference element, such as the Piola transform [1].

4.1 Evaluation of bilinear forms

We consider Laplace's equation to get started. For example,

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

where the "element" bilinear form is defined (and evaluated) via

$$
\begin{aligned}
a_e(v,w) &:= \int_{T_e} \nabla v(x)\cdot\nabla w(x)\,dx \\
&= \int_{T_e} \nabla v(x)\cdot\nabla w(x)\,dx \\
&= \int_{\mathcal{T}} \sum_{j=1}^{d} \frac{\partial}{\partial x_j} v(J\xi + x_e) \frac{\partial}{\partial x_j} w(J\xi + x_e)\det(J)\,d\xi \\
&= \int_{\mathcal{T}} \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j}\frac{\partial}{\partial \xi_m}\left(\sum_{\lambda\in\mathcal{L}} v_{\iota(e,\lambda)}\phi_\lambda(\xi)\right)\times \\
&\quad \frac{\partial \xi_{m'}}{\partial x_j}\frac{\partial}{\partial \xi_{m'}}\left(\sum_{\mu\in\mathcal{L}} w_{\iota(e,\mu)}\phi_\mu(\xi)\right)\det(J)\,d\xi \\
&= \begin{pmatrix} v_{\iota(e,1)} \\ \cdot \\ \cdot \\ v_{\iota(e,|\mathcal{L}|)} \end{pmatrix}^t \mathbf{K}^e \begin{pmatrix} w_{\iota(e,1)} \\ \cdot \\ \cdot \\ w_{\iota(e,|\mathcal{L}|)} \end{pmatrix}.
\end{aligned} \tag{4.4}
$$

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

$$
\begin{aligned}
K^e_{\lambda,\mu} &:= \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j}\frac{\partial \xi_{m'}}{\partial x_j}\det(J)\int_{\mathcal{T}} \frac{\partial}{\partial \xi_m}\phi_\lambda(\xi)\frac{\partial}{\partial \xi_{m'}}\phi_\mu(\xi)\,d\xi \\
&= \sum_{j,m,m'=1}^{d} \frac{\partial \xi_m}{\partial x_j}\frac{\partial \xi_{m'}}{\partial x_j}\det(J) K_{\lambda,\mu,m,m'} \\
&= \sum_{m,m'=1}^{d} G^e_{m,m'} K_{\lambda,\mu,m,m'}
\end{aligned} \tag{4.5}
$$

where

$$
G^e_{m,m'} := \det(J)\sum_{j=1}^{d} \frac{\partial \xi_m}{\partial x_j}\frac{\partial \xi_{m'}}{\partial x_j} \tag{4.6}
$$

for $\lambda, \mu \in \mathcal{L}$.

There are two different ways to utilize the expressions above. One is to compute and store all element stiffness matrices. Then to compute a given form $a(u,v)$ we loop over all elements e and increment the accumulator (set initially to zero) by adding

$$
\begin{pmatrix} v_{\iota(e,1)} \\ \cdot \\ \cdot \\ v_{\iota(e,|\mathcal{L}|)} \end{pmatrix}^t \mathbf{K}^e \begin{pmatrix} w_{\iota(e,1)} \\ \cdot \\ \cdot \\ w_{\iota(e,|\mathcal{L}|)} \end{pmatrix}. \tag{4.7}
$$

For each evaluation of (4.7), the $\mathbf{K}^e$'s are brought from memory, together with v and w.

Another approach is to avoid such a large amount of storage and instead compute (4.5) for each element each time a form is to be computed, and then apply (4.7) to increment the accumulator. We compare the trade-offs subsequently.

Note that in the first case, we must load vectors u and v plus $\mathbf{K}^e$ for each element, as well as store all of the $\mathbf{K}^e$ matrices. In the second case, we just need

to load u, v, and the Jacobian on each element. The only additional storage is for the tensor $K_{\lambda,\mu,m,m'}$ regardless of the number of elements in the mesh. Thus we need to consider the complexity of evaluating $\mathbf{K}^e$.

In Section 7, we study a third approach which involves quadrature to compute the integrals in (4.4).

4.2 Evaluation of Bilinear Form Actions

Following (3.5), we need to evaluate

$$v_i := a(\phi_i, w) = \sum_e a_e(\phi_i, w)$$

for all i, where ϕ_i denotes a typical basis function. We can do this as follows.

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

$$\begin{aligned} v_{\iota(e,\lambda)} += & \sum_{m,m',\mu} G^e_{m,m'} K_{\lambda,\mu,m,m'} w_{\iota(e,\mu)} \\ = & \sum_{\mu} K^e_{\lambda,\mu} w_{\iota(e,\mu)} \end{aligned} \tag{4.8}$$

where $G^e_{m,m'}$ are defined in (4.6). We have indicated that there are two ways of computing this: pre-compute and store the element-stiffness matrices $\mathbf{K}^e$ and essentially compute them on the fly.

Regardless of how we evaluate the expressions, there is a natural way to think about the action algorithm. We can write the set of increments indicated in (4.8) as a vector operation

$$\begin{pmatrix} v_{\iota(e,1)} \\ \cdot \\ \cdot \\ v_{\iota(e,|\mathcal{L}|)} \end{pmatrix}^t += [K^e_{\lambda,\mu}] * [w_{\iota(e,\mu)}] = \left[\sum_{\mu,m,n} G^e_{mn} K_{\lambda,\mu,m,n} w_{\iota(e,\mu)} \right] \tag{4.9}$$

where $[K^e_{\lambda,\mu}]$ denotes the element stiffness matrix, $[w_{\iota(e,\mu)}]$ the vector of local nodal values, the matrix-vector product is indicated by $*$, and in the last vector expression the vector index is λ. Using (4.9) to up-date the action vector requires $d^2|\mathcal{L}|^2$ multiply-add pairs per element, and the memory traffic is $3|\mathcal{L}|+d(d+1)/2$ (not counting storing K) if we use the fact that G^e is symmetric. Note that v must be both read from and written to memory.

The performance of different algorithms will depend on the relative cost of memory references and floating point operations. For some machines, memory references are much more costly than floating point operations, whereas on others it is nearer parity. To determine the optimal algorithm may require tuning based on the particular architecture being used.

There is a natural duality between the algorithm for computing forms and the algorithm for computing actions. In the form evaluation, we bring in two vectors v and w from memory, and we compute a scalar (that can be accumulated in a register). This can be thought of as taking the dot-product of the vectors v and $K * w$. In the action algorithm, we bring in two vectors v and w from memory, and we compute an increment to v that is given by $K * w$. The action algorithm incurs additional memory references since it must write the resulting v back to memory. The computation of a form (i.e., a scalar value) occurs relatively rarely, whereas the action is quite common. However, it is easy to define the action from the form, and the duality in their assembly algorithms makes it useful to think about the form evaluation problem as a primary problem.

4.3 Computation of Bilinear Form Matrices

The matrix associated with a bilinear form is

$$A_{ij} := a(\phi_i, \phi_j) = \sum_e a_e(\phi_i, \phi_j) \tag{4.10}$$

for all i, j, where ϕ_i denotes a typical basis function. 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

$$A_{\iota(e,\lambda),\iota(e,\mu)} += K^e_{\lambda,\mu} = \sum_{m,m'} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{4.11}$$

where $G^e_{m,m'}$ are defined in (4.6). Here it is interesting that there appear to be very few options. One can imagine trying to optimize the computation of each

$$K^e_{\lambda,\mu} = \sum_{m,m'} G^e_{m,m'} K_{\lambda,\mu,m,m'} \tag{4.12}$$

but each such term must be computed separately. We consider this optimization in Section 6.

5 Evaluation Using a Global Matrix

The evaluation of a global bilinear form such as $a(v, w)$ can be done via a global stiffness matrix, A, as

$$a(v, w) = \sum_{i,j} A_{ij} v_i w_j$$

where A_{ij} is defined in the obvious way. Here i and j range over the set, $\mathcal{I}$, of global indices $\iota(e, \lambda)$. Thus it is a $|\mathcal{I}| \times |\mathcal{I}|$ matrix. The action of the variational form is just the action of this matrix.

The approaches considered in Section 4 involved an amount of work proportional to $|\mathcal{E}|$ where $\mathcal{E}$ is the set of all elements, independent of whether element stiffness matrices (ESMs) are precomputed or not.

Due to the sparseness of A, the cost of using a global matrix is more difficult to estimate. It can be done in $|A|$ operations, where $|A|$ denotes the number of nonzeros. In the two dimensional case, work estimates for the global-matrix approach can be compared with the element-matrix approach [2]. The global matrix approach is typically more efficient for low order methods.

For higher-order elements, using a local-element approach becomes more attractive. Table 5.1 collects data from [2]. For quadratics, the number of entries in A can be grouped in two classes. For the vertex nodes, we have thrice the number of non-zero elements as for piecewise linears ($9|\mathcal{E}|$) since there are two nodes for each edge (the vertex and the edge mid-point) and a node on at the edge-midpoint on each edge in the edges going around the boundary of the star of the vertex. For edge-midpoint nodes, there are nine nodes in the union of the triangles sharing the edge, and this corresponds to $13\frac{1}{2}|\mathcal{E}|$ more non-zero elements in A.

For quadratics, we show in Section 7.4 that the action can be computed using quadrature for computing the element stiffness actions using only 62 operations and only 21 memory references per element. This requires $31/23 \approx 1.34$ more work, but $7/9 \approx 0.78$ of the memory references. On many architectures, the quadrature approach will thus be more efficient. But as well, the total memory

Table 5.1: Amounts of work (number of multiply-add pairs) **per element** for computing the action using the global matrix as a function of element degree in two dimensions. The number of memory references is the number of multiply-add pairs plus twice the number of nodal values.

degree	multiply-add pairs	nodal values	memory references
1	$3\frac{1}{2}$	$\frac{1}{2}$	$4\frac{1}{2}$
2	23	2	27
3	$76\frac{1}{2}$	$4\frac{1}{2}$	85
4	188	8	204
$t-1$	$\frac{1}{4}t^4+\frac{1}{2}t^3-\frac{5}{4}t^2+\frac{1}{2}$	$\frac{1}{2}t^2-t+\frac{1}{2}$	$\frac{1}{4}t^4+\frac{1}{2}t^3-\frac{1}{4}t^2-2t+\frac{3}{2}$

storage is quite different. The global stiffness matrix storage is $23|\mathcal{E}|$, whereas the element-quadrature approach requires the same $3|\mathcal{E}|$ as required for piecewise linears. Thus one could work with much larger meshes. For reference, a given piecewise quadratic interpolant v requires storage equal to $E+V \approx 2|\mathcal{E}|$. A typical solution algorithm will have a small number of vectors of this size.

6 Computing K for general elements

The tensor $K_{i,j,m,n}$ can be presented as an $|\mathcal{L}| \times |\mathcal{L}|$ matrix of $d \times d$ matrices. The entries of resulting matrix K^e can be viewed as the dot (or Frobenius) product of the entries of K and G^e. That is,

$$K^e_{i,j} = \mathbf{K}_{i,j} : G^e \tag{6.1}$$

In [8], we show that the matrices $\mathbf{K}_{i,j}$ are frequently very simple (e.g., have at most one non-zero entry) or can be expressed as linear combinations of other matrices for different values of i, j. This can lead to a reduction in cost of evaluation of K^e of more than an order of magnitude.

In the case of linears in three dimensions, we find [8] a reduction to 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. 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.

7 Evaluation of Forms via quadrature

There is another way to evaluate objects related to multilinear forms that can be more efficient both in time and storage than the above approach using element matrices. Let us return to the example involving $a(\cdot,\cdot)$. We have

$$\begin{aligned}
&a_e(v,w) = \\
&\int_{\mathcal{T}} \left(\left(J^{-1}\right)^t \sum_{\lambda\in\mathcal{L}} v_{\iota(e,\lambda)} \nabla\phi_\lambda(\xi) \right) \cdot \left(\left(J^{-1}\right)^t \sum_{\lambda\in\mathcal{L}} w_{\iota(e,\lambda)} \nabla\phi_\lambda(\xi) \right) \det(J)\, d\xi \\
&= \int_{\mathcal{T}} \left(\sum_{\lambda\in\mathcal{L}} v_{\iota(e,\lambda)} \nabla\phi_\lambda(\xi) \right) \cdot \left(J^{-1} \left(J^{-1}\right)^t \sum_{\lambda\in\mathcal{L}} w_{\iota(e,\lambda)} \nabla\phi_\lambda(\xi) \right) \det(J)\, d\xi \\
&= \int_{\mathcal{T}} \left(\sum_{\lambda\in\mathcal{L}} v_{\iota(e,\lambda)} \nabla\phi_\lambda(\xi) \right) \cdot \left(\mathbf{G}^e \sum_{\lambda\in\mathcal{L}} w_{\iota(e,\lambda)} \nabla\phi_\lambda(\xi) \right) d\xi
\end{aligned} \tag{7.1}$$

where $\mathbf{G}^e = \det(J)J^{-1}\left(J^{-1}\right)^t$ as before. Assuming we use a quadrature rule to evaluate the integral:

$$\int_{\mathcal{T}} f(\xi)\,d\xi := \sum_{\xi\in\Xi} \omega_\xi f(\xi)$$

we can convert the above as follows:

$$a_e(v,w) = \sum_{\xi\in\Xi} \omega_\xi \left(\sum_{\lambda\in\mathcal{L}} v_{\iota(e,\lambda)}\nabla\phi_\lambda(\xi)\right)\cdot\left(\mathbf{G}^e \sum_{\lambda\in\mathcal{L}} w_{\iota(e,\lambda)}\nabla\phi_\lambda(\xi)\right) \tag{7.2}$$

The work estimate for the latter approach depends on how the gradients at the quadrature points are caclulated. The simplest approach is just to do a matrix multiplication: $[\nabla\phi_\lambda(\xi)] * [w_{\iota(e,\lambda)}]$. Assuming that the dimension of the polynomial space and the number of quadrature points is sufficiently large (such that the multiplication by the matrix $\mathbf{G}^e$ is a negligible addition to the work), then the work using quadrature is of the order $|\Xi|\times|\mathcal{L}|$. The work for, say, using a stored element stiffness matrix is of the order $|\mathcal{L}|^2$. Depending on the number of quadrature points versus the number of basis functions, the quadrature approach may be more or less efficient. For quadratics in two dimensions, we have $|\mathcal{L}| = 6$ but we only need three quadrature points ($|\Xi| = 3$) to compute the products of gradients exactly.

7.1 Form actions by quadrature

The action can be computed as follows. First, set all the entries of V to zero. Then loop over all elements e and local element numbers λ and compute

$$v_{\iota(e,\mu)} += \sum_{\xi\in\Xi} \omega_\xi \nabla\phi_\mu(\xi)\cdot\left(\mathbf{G}^e\sum_{\lambda\in\mathcal{L}} w_{\iota(e,\lambda)}\nabla\phi_\lambda(\xi)\right). \tag{7.3}$$

Unlike the algorithm using element stiffness matrices, we cannot write this as a simple vector up-date. The expression

$$\nabla w(\xi) = \mathbf{G}^e \sum_{\lambda\in\mathcal{L}} w_{\iota(e,\lambda)}\nabla\phi_\lambda(\xi) \tag{7.4}$$

So, we need a mapping $[w_{\iota(e,\lambda)}] \to [\nabla w(\xi)]$ to compute the action of the local matrix by quadrature. As we shall see, how fast this mapping can be computed is critical to the efficiency of the quadrature based approach.

Let us formalize this mapping. Let $\widehat{w}$ be defined (locally only) by

$$\widehat{w}_{m,\xi} = \sum_{\lambda\in\mathcal{L},n} G^e_{mn} w_{\iota(e,\lambda)}\phi_{\lambda,n}(\xi) = \sum_n G^e_{mn} w_{,n}(\xi) \tag{7.5}$$

for $m = 1,\ldots,d$ and $\xi\in\Xi$. It may be necessary to account for memory storage of these quantities unless we are sure they can be held in registers. However, it is reasonable to assume that they will be held in cache, with immediate re-use in the typical case, so we only count one memory reference per variable.

Let us define (once and for all) the tensor quantity

$$\Phi_{\mu,m,\xi} = \omega_\xi \phi_{\mu,m}(\xi) \tag{7.6}$$

Then for all $\mu\in\mathcal{L}$, we have

$$\Delta v_{\iota(e,\mu)} = \sum_{m,\xi} \Phi_{\mu,m,\xi}\widehat{w}_{m,\xi} = \sum_{m,n,\xi} \Phi_{\mu,m,\xi} G^e_{m,n} w_{,n}(\xi)\,. \tag{7.7}$$

Note that we can evaluate $w_{,n}(\xi)$ as

$$w_{,n}(\xi) = \sum_{\lambda \in \mathcal{L}} w_{\iota(e,\lambda)} \phi_{\lambda,n}(\xi) \tag{7.8}$$

but there may be other ways to compute these expressions that are less costly. Using (7.8), we can write

$$\Delta v_{\iota(e,\mu)} = \sum_{m,\xi} \Phi_{\mu,m,\xi} \left(\sum_{\lambda \in \mathcal{L}} w_{\iota(e,\lambda)} \left(\sum_{n=1}^{d} G^e_{mn} \phi_{\lambda,n}(\xi) \right) \right) \tag{7.9}$$

Comparing (7.9) with (4.9) shows that (if the quadrature is exact)

$$K_{\lambda,\mu,m,n} = \sum_{\xi} \Phi_{\mu,m,\xi} \phi_{\lambda,n}(\xi) \tag{7.10}$$

as would be expected: this is just a quadrature representation of the integrals defining K.

A straightforward evaluation of (7.9) would require $d(d+1)|\Xi||\mathcal{L}|(|\mathcal{L}|+1)$ multiply-add pairs. However, the computation can be done in only $d(d+2)|\Xi||\mathcal{L}|$ multiply-add pairs. To see this write (7.5) as

$$\widehat{w}_{m,\xi} = \sum_{\lambda \in \mathcal{L}} w_{\iota(e,\lambda)} \left(\sum_{n=1}^{d} G^e_{mn} \phi_{\lambda,n}(\xi) \right) \tag{7.11}$$

which requires $d(d+1)|\Xi||\mathcal{L}|$ multiply-add pairs. Then using (7.7) completes the up-date in only $d|\Xi||\mathcal{L}|$ additional multiply-add pairs. Thus the total computation takes only only $d(d+2)|\Xi||\mathcal{L}|$ multiply-add pairs. However, $d|\Xi|$ temporary storage locations (or additional memory references) are needed for the temporary data generated in (7.11).

7.2 Quadrature as a general model

Using quadrature provides a general model of how to compile forms. All that we require is values and derivatives of basis functions at quadrature points. The resulting scheme may be approximate if the quadrature is not sufficiently accurate, but in many cases it will be exact. For example, the matrix update is computed via quadrature as

$$\begin{aligned} A_{\iota(e,\lambda),\iota(e,\mu)} += & \sum_{\xi \in \Xi} \omega_\xi \nabla \phi_\lambda(\xi) \cdot (\mathbf{G}^e \nabla \phi_\mu(\xi)) \\ = & \sum_{\xi \in \Xi} \omega_\xi \sum_{m,n=1}^{d} \phi_{\lambda,m}(\xi) G^e_{m,n} \phi_{\mu,n}(\xi) \\ = & \sum_{m,n=1}^{d} G^e_{m,n} \sum_{\xi \in \Xi} \omega_\xi \phi_{\lambda,m}(\xi) \phi_{\mu,n}(\xi) \\ = & \sum_{m,n=1}^{d} G^e_{m,n} K_{\lambda,\mu,m,n} \end{aligned} \tag{7.12}$$

where the coefficients $K_{\lambda,\mu,m,n}$ are analogous to those defined in (4.5), but here they are defined by quadrature:

$$K_{\lambda,\mu,m,n} = \sum_{\xi \in \Xi} \omega_\xi \phi_{\lambda,m}(\xi) \phi_{\mu,n}(\xi) \tag{7.13}$$

(The coefficients are exactly those of (4.5) 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 (7.13), and then proceeding as in Section 4.3. The work and memory references are similar to those in Section 4.3: $\mathbf{G^e}$ must be read from memory, and there are $|\mathcal{L}|^2$ reads and writes required to up-date A.

7.3 Spectral elements

The technology of "spectral elements" exploits the quadrature approach in a systematic way. The spectral-element approximating spaces are simply high-order piecewise polynomials, perhaps with carefully chosen local bases to mitigate conditioning problems arising when the degree of the polynomials gets large. In the simplest case, tensor-product polynomials are used (on quadralateral and hexahedral meshes). In this case, the evaluation of derivatives at quadrature points as indicated in (7.5) can be done by an especially simple algorithm. Using the tensor product structure, derivatives in the coordinate directions can be computed as derivatives of one-dimensional polynomials determined by restricting to lines parallel to the coordinate axes. Naturally, it makes sense to take the quadrature points to be Cartesian products of one-dimensional quadrature rules to get the most benefit from these one-dimensional polynomial representations.

It is well known that the spectral-element technology can be extended to simplicial meshes [6]. For high-degree polynomials, substantial reductions in operations costs are obtained. Since this work is well known, we will not review the known complexity results. However, we will note that the general question of the complexity of mappings from a given polynomial representation (say, in terms of nodal values) to gradients (and function values) at quadrature points is an open problem. More precisely, suppose that we have representation $W \in I\!R^{|\mathcal{L}|}$ of nodal values of some $w \in \mathcal{V}$ and we wish to compute the set of values $W' = \{w_{,n}(\xi) \,:\, n = 1, \ldots, d; \xi \in \Xi\}$ in (7.8). Then $W \to W'$ is a linear mapping, and we can ask what is the computational complexity of it. Just evaluating (7.8) as written would require $|\mathcal{L}|d|\Xi|$ multiply-add pairs, which is essentially quadratic in $|\mathcal{L}|$. Using the techniques desribed above, this can be reduced to $\mathcal{O}(|\mathcal{L}|^{(d+1)/d})$. We will give an example to show that there is the oportunity to achieve a substantial reduction in complexity of evaluation algorithms even for low order methods.

7.4 Quadrature for quadratics

Consider the two-dimensional case for the gradient form. Let the reference element be the triangle with vertices at the origin and $(2,0)$ and $(0,2)$. Number the first three nodes for the basis functions as we did for linears: the nodes at on the two axes are numbered 1 and 2, respectively, at the points $(1,0)$ and $(0,1)$, and the origin is number 3. Let the fourth and sixth nodes be the midpoints of the two edges on the axes, i.e., the points $(1,0)$ and $(0,1)$. Let the fifth node be $(1,1)$. Note that these are numbered in a counter-clockwise fashion, with the vertices first and mid-points last. Let q_i denote the value of a quadratic at the i-th node.

To use the standard quadrature rule for quadratics, we need to evaluate the gradient of q at the edge midpoints (nodes 4,5,6). Let q_t^i denote the t-derivative ($t = x$ or y) at the i-th node. Clearly

$$\begin{aligned} q_x^4 &= \tfrac{1}{2}(q_1 - q_3) \\ q_y^6 &= \tfrac{1}{2}(q_2 - q_3) \end{aligned} \tag{7.14}$$

since we can compute the derivatives using the one-dimensional representation on each edge, respectively.

The second derivatives of q are constant, and can be seen to be

$$\begin{aligned} q_{xy} &= q_3 - q_4 + q_5 - q_6 \\ q_{xx} &= q_1 + q_3 - 2q_4 \\ q_{yy} &= q_2 + q_3 - 2q_6 \end{aligned} \tag{7.15}$$

by simply checking this for $q = 1$, $q = x$, $q = y$, $q = x^2$, $q = y^2$, and $q = xy$.

By Taylor's theorem,

$$\begin{aligned} q^5_x &= q^4_x + q_{xy} \\ q^5_y &= q^6_y + q_{xy} \end{aligned} \tag{7.16}$$

and

$$\begin{aligned} q^6_x &= q^5_x - q_{xx} \\ q^4_y &= q^5_y - q_{yy} \end{aligned} \tag{7.17}$$

Thus we can compute the mapping

$$\left(q_1, q_2, q_3, q_4, q_5, q_6\right) \longrightarrow \left(q^4_x, q^4_y, q^5_x, q^5_y, q^6_x, q^6_y\right) \tag{7.18}$$

in only seventeen floating point operations. Such a linear map on six-dimensional space would take 36 multiply-add pairs in general to compute. However, there are not 36 operations needed in this case. This is a reduction by a factor of $72/17 \approx 4.23$ in work. Note that this mapping has a one-dimensional null space, so its image is five dimensional. It is interesting to speculate on what the minimal number of operations required to compute this mapping.

After the computation of the mapping (7.18), the multiplication by $\mathbf{G}$ in (7.3) requires a further 12 multiply-add pairs. This produces a 3×2 array $\mathbf{Q}$ of values

$$\mathbf{Q} = \left(Q^4_x, Q^4_y, Q^5_x, Q^5_y, Q^6_x, Q^6_y\right) \tag{7.19}$$

that must be processed to update v accordingly. To do so, we need the array of gradients of the basis functions at the quadrature points. That is, we need

$$\left(q^4_x, q^4_y, q^5_x, q^5_y, q^6_x, q^6_y\right) \tag{7.20}$$

for all six basis functions q. We can summarize the gradient basis function values as a matrix R where

$$\phi_{i,j}(\xi_k) = R_{i,j+2(k-1)},\ i = 1, \ldots, 6;\ j = 1, 2;\ k = 1, 2, 3. \tag{7.21}$$

Here the numbering of the quadrature points is such that ξ_j is at node $j + 3$ for $j = 1, 2, 3$. With this notation, we find

$$\mathbf{R} = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 0 & -1 & -1 & -1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 & -1 & 0 \end{pmatrix} \tag{7.22}$$

Then the vector update just requires adding RQ, which can be done in 21 operations. Thus the total work per element for the matrix action can be done in $17 + 24 + 21 = 62$ operations. Only 21 memory references are required per

Table 7.1: Comparisons of different techniques for evaluating form actions for Lagrange elements of degree 2 for the scalar gradient form in d = two dimensions. Indicated are amounts of work and memory references per element. The amount of memory references for the quadrature approach is listed as "21" assuming that 6 temporary numbers can be held in registers. This would increase to 27 if they had to be written to memory (see text). Note that the matrix G^e is symmetric, so only $d(d+1)/2$ memory references are needed to obtain it.

store ESM (Sect. 4.1) mem refs, FLOPs	evaluate ESM (Sect. 6) mem refs, FLOPs	quadrature/special mem refs, FLOPs	GSM (Section 5) mem refs, FLOPs
54, 72	21, 82	"21" (or 27), 62	27, 46

element: the input of $\mathbf{G}$ (3) and w (6), the update of v (12), except that the temporary storage for Q requires potentially an additional 6 memory references. In principle, this would involve 6 writes followed by 6 reads, but it is quite likely that the reads would be from cache. The temporaries q_{ij} can be assumed to be in registers. The order of computations given above requires only six registers to hold all of the q_{ij} and Q.

Table 7.1 summarizes the comparison between the three ways considered so far for evaluating local form actions, in the special case of quadratics in two dimensions. The quadrature approach has somewhat more operations than the matrix multiplication approach, but with an advantage in memory traffic. This suggests that one approach to compiling variational forms would be to require finite elements to be specified by providing an algorithm (or code) to map nodal values to gradients at quadrature points. This is in the spirit of spectral element methods. Note that the matrix R can be derived from such an algorithm.

One point here is that the winning algorithm may be strongly architecture-dependent. One cannot expect even able programmers to make the correct choice all the time. A system that provides all these options available at run-time would allow users to experimentally determine the best approach for their given machine.

8 Conclusions

We have illustrated several ways that the actions related to variational forms can be evaluated. They provide trade-offs in memory usage and traffic on the one hand and floating point computation on the other. Providing options that can be tested for particular architectures may be the simplest and best way to find optimal performance. For example, using a global matrix is more efficient only when more than one action is being computed. However, this may be hard to determine at "compile" time, so it may be reasonable to leave this to user discretion. That is, these are not compiler issues but language issues: the options need to be available to the user, not determined by a compiler.

The mapping of nodal values to values (and gradients) at quadrature points is essential. If this can be done efficiently, then the quadrature approach to evaluating actions is very competitive even for low-order methods. This suggests a general paradigm for defining the general notion of element computationally: it should include an efficient mapping from the nodal description to values (and gradients) at quadrature points. Unfortunately, this implies a tight linkage with the choice of quadrature rule, and there is not yet a general representation of "optimal" quadrature rules in multiple dimensions.

9 Acknowledgements

We thank the FEniCS team, and Todd Dupont, Johan Hoffman and Anders Logg in particular, for substantial suggestions regarding this paper. Some of the ideas presented here originated in the development of Analysa, a system developed jointly by Scott and Babak Bagheri.

REFERENCES

1. Douglas N. Arnold, Daniele Boffi, and Richard S. Falk. Quadrilateral h(div) finite elements. *SIAM Journal on Numerical Analysis*, to appear(211):1–2, 2004.
2. B. Bagheri, L. R. Scott, and S. Zhang. Implementing and using high–order finite element methods. *Finite Elements in Analysis & Design*, 16:175–189, 1994.
3. S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, 1994.
4. S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods, 2nd Edition*. Springer-Verlag, 2002.
5. P. G. Ciarlet. *The finite element method for elliptic problems*. North-Holland, 1978.
6. George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York, 1999.
7. Robert C. Kirby. FIAT: A new paradigm for computing finite element basis functions. to appear, ACM Trans. Math. Software.
8. Robert C. Kirby, Matthew G. Knepley, and L.R. Scott. Optimal evaluation of finite element matrices. *SISC*, submitted, 2004.