High-performance evaluation of finite element variational forms via commuting diagrams and duality

Robert C. Kirby, Baylor University

We revisit the question of optimizing the construction and application of finite element matrices. By using commuting properties of the reference mappings and duality, we reorganize stiffness matrix construction and matrix-free application so that the bulk of the work can be done by optimized matrix multiplication libraries. We provide examples, including numerical experiments, with the Laplace and curl-curl operators as well as develop a general framework. Our techniques are applicable in general geometry and are not restricted to constant coefficient operators.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Finite element methods, high-performance computing, commuting diagram

ACM Reference Format:
Robert C. Kirby, 2012. High performance evaluation of finite element variational forms via commuting diagrams and duality *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2013), 24 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The literature and lore of finite elements contain many methods for evaluating and applying variational forms. Beyond straightforward loops over quadrature points, we can point to precomputation of certain reference element integrals in affine geometry (utilized both by the FEniCS project [Logg et al. 2012] and Sundance [Long et al. 2010], although the technique is surely much older), sum-factorization methods of Orszag [Orszag 1979], the nodal spectral elements of Hesthaven and Warburton [Hesthaven and Warburton 2008], and the low-complexity Bernstein basis methods found in [Ainsworth et al. 2011; Kirby 2011]. This wide variety of methodology is largely driven by diverse properties of the bases, geometry, and coefficients. Affine geometry allows certain optimizations that are not possible in more general geometry, $H(\mathrm{div})$ and $H(\mathrm{curl})$ spaces require more complex transformations than H^1, and particular bases have tensor product or other structure that allows low-complexity algorithms for matrix-vector products.

At risk of introducing Yet Another Method, we propose in this paper an approach based on *duality*, in which the test function transformations are transferred onto the trial functions, which depends on certain commuting diagram properties for pullback operations. Elementwise stiffness matrix formation and matrix-free application (by which we mean the action of the element stiffness matrix being calculated without its

This work is supported by the National Science Foundation, under grant CCF-1117794.
Author's address: R. C. Kirby, Department of Mathematics, Baylor University; One Bear Place #97328; Waco, TX 76798-7328.

DOI:http://dx.doi.org/10.1145/0000000.0000000

entries being explicitly formed) are then cast predominantly in terms of dense matrix-multiplication, which can be outsourced to optimized BLAS routines. This approach works for generic element shapes and in the presence of variable coefficients. It is important to note that this approach does not appreciably alter the operation count over existing quadrature-based methods, although processing many cells together requires some amount of intermediate storage. Finally, our approach provides a natural abstraction barrier beneath which matrix multiplication can be replaced with equivalent algorithms utilizing setting-specific structure. We point this out, but do not dwell on this here.

We require some degree of "homogeneity" to apply our method. While we support variable coefficients, we require batches of cells where the same operator, quadrature rule, and bases are used throughout. So, it would be straightforward to adapt our method to Darcy-Stokes flow or fluid-structure interaction (apply the method on each subdomain), but p-adaptive methods where the finite element basis varies from cell to cell would present a much more significant challenge.

In the rest of the paper, we begin with a lengthy example based on the Poisson operator in Section 2. The essential features of our approach are present in this problem, and we easily adapt the technique to the curl-curl operator on Nédélec spaces in Section 3 and present an abstract setting in Section 4. In Section 5 we present some timing results indicating that our methods yield very high performance. Finally, we offer some concluding remarks and indicate future research directions in Section 6.

2. AN EXAMPLE: THE LAPLACE OPERATOR

Our approach is best presented with an example; we begin with the weak form of the Poisson operator:

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, dx. \tag{1}$$

Suppose we have a reference cell $\hat{K}$, which could be a simplex, box, or some other shape, with some reference basis $\{\Psi_i\}_{i=1}^{N_f}$, and some collection of cells $\{K_c\}_{i=1}^{N_c}$ in $\mathbb{R}^d$. These cells may be the entire collection of subdomains in a tesselation of Ω, or they may be a subset chosen for whatever reason (e.g. a subdomain in domain decomposition or a batch of cells with size chose to optimize some performance). $N_c = 1$ is permissible.

Following the standard methodology, we assume each K_c is diffeomorphic to $\hat{K}$ via a mapping $F_{K_c} : \hat{K} \rightarrow K_c$. Although we require differentiability and invertibility, we do not require affinity or other properties of the mappings.

The reference basis is transferred to each cell by changing variables

$$\psi_i^{K_c} = \Psi_i \circ F_{K_c}^{-1}.$$

Following the notation in [Bochev and Gunzburger 2009], we define this pullback as Φ_G^* so that $\psi_i^{K_c} = \Phi_G^*(\Psi_i) \equiv \Psi_i \circ F_{K_c}^{-1}$.

We denote by $DF_{K_c} : \hat{K} \rightarrow \mathbb{R}^{d\times d}$ the Jacobian matrix and $J_{K_c} = \det DF_{K_c}$. The cellwise basis functions then are differentiated by

$$\nabla \psi_i^{K_c} = \left(DF_{K_c}^{-T} \hat{\nabla} \Psi_i \right) \circ F_{K_c}^{-1}, \tag{2}$$

where $\hat{\nabla}$ indicates differentiation in the reference coordinates. We also give this a transformation a name, Φ_C^*, so that

$$\Phi_C^*(\hat{u}) = \left(DF_{K_c}^{-T} \hat{u} \right) \circ F_{K_c}^{-1}. \tag{3}$$

The mapping Φ^*_G is the pullback operator associated with the local H^1 space, and Φ^*_C is the pullback for $H(\mathrm{curl})$. The relationship between differentiation and these pullbacks is naturally expressed by the commuting relationship

$$\nabla \Phi^*_G\left(\hat{u}\right) = \Phi^*_C\left(\hat{\nabla}\hat{u}\right). \tag{4}$$

Our first task is to construct the element stiffness matrices for each cell in the batch. Define

$$A_{cij} = \int_{K_c} \nabla \psi_i^{K_c} \cdot \nabla \psi_j^{K_c} \, dx. \tag{5}$$

Using the pullbacks to transform the integrals, we have that

$$\begin{aligned}\int_{K_c} \nabla \psi_i^{K_c} \cdot \nabla \psi_j^{K_c} dx &= \int_{K_c} \nabla\left(\Phi^*_G\left(\Psi_i\right)\right) \cdot \nabla\left(\Phi^*_G\left(\Psi_j\right)\right) dx \\ &= \int_{\hat{K}} \left(DF_{K_c}^{-T}\hat{\nabla}\Psi_i\right) \cdot \left(DF_{K_c}^{-T}\hat{\nabla}\Psi_j\right) \left|J_{K_c}\right| \, d\hat{x}\end{aligned} \tag{6}$$

Now, for any vectors $x, y \in \mathbb{R}^n$ and matrix $A \in \mathbb{R}^{n\times n}$, it is true that $x^T\left(Ay\right) = \left(A^T x\right)^T y$. This is just a particular case of the adjoint of a linear operator within an inner product. After using this fact, we collect the geometric terms together to obtain

$$\begin{aligned}\int_{K_c} \nabla \psi_i^{K_c} \cdot \nabla \psi_j^{K_c} \, dx &= \int_{\hat{K}} \left(DF_{K_c}^{-1} DF_{K_c}^{-T}\hat{\nabla}\Psi_i\right) \cdot \hat{\nabla}\Psi_j \left|J_{K_c}\right| \, d\hat{x} \\ &= \int_{\hat{K}} \left(\left|J_{K_c}\right| DF_{K_c}^{-1} DF_{K_c}^{-T}\hat{\nabla}\Psi_i\right) \cdot \hat{\nabla}\Psi_j \, d\hat{x} \\ &= \int_{\hat{K}} \left(G_{K_c}\hat{\nabla}\Psi_i\right) \cdot \hat{\nabla}\Psi_j \, d\hat{x},\end{aligned} \tag{7}$$

where the $G_{K_c} : \hat{K} \rightarrow \mathbb{R}^{d\times d}$ is defined by $G_{K_c} \equiv \left|J_{K_c}\right| DF_{K_c}^{-1} DF_{K_c}^{-T}$.

If we further define $\Gamma_i^{K_c} = G_{K_c}\hat{\nabla}\Psi_i$, then

$$A_{cij} = \int_{K_c} \nabla \psi_i^{K_c} \cdot \nabla \psi_j^{K_c} \, dx = \int_{\hat{K}} \Gamma_i^{K_c} \cdot \hat{\nabla}\Psi_j \, d\hat{x}. \tag{8}$$

In this last expression all cell-specific information is contained in the Γ^{K_c} array.

Now, we consider the (possibly approximate) evaluation of these integrals via numerical quadrature. Let $\{\xi_q\}_{q=1}^{N_q} \subset \hat{K}$ be a collection of quadrature points with associated quadrature weights $\{w_q\}_{q=1}^{N_q}$. Application of this quadrature in (7) gives

$$\int_{K_c} \nabla \psi_i^{K_c} \cdot \nabla \psi_j^{K_c} \, dx \approx \sum_{q=1}^{N_q} w_q \Gamma_i^{K_c}\left(\xi_q\right) \cdot \hat{\nabla}\Psi_j\left(\xi_q\right). \tag{9}$$

To expedite our algorithmic discussion, we will introduce several multidimensional arrays. This is already suggested by labeling the i, j entry of the stiffness matrix on K_c as A_{cij} – it is a rank three $N_c \times N_f \times N_f$ array. Let the $N_c \times N_q \times d \times d$ array

$$DF_{cqk\ell} = \left(DF_{K_c}\left(\xi_q\right)\right)_{k\ell} \tag{10}$$

be the Jacobian matrices tabulated at quadrature points, and $DF^{-1}_{cqk\ell}$ the pointwise inverses. We also let the $N_c \times N_q$ array J_{cq} be the tabulated Jacobian determinants

$J_{cq} = |J_{K_c}(\xi_q)|$, we store the reference gradients as

$$D\Psi_{iqk} = \frac{\partial \Psi_i}{\partial \hat{x}_k}(\xi_q). \tag{11}$$

It will also be useful to pre-scale the tabulated gradients by the quadrature weights, so we introduce

$$D\Psi^w_{iqk} = w_q D\Psi_{iqk}. \tag{12}$$

We let the $N_c \times N_q \times d \times d$ array $G_{cqk\ell}$ contain the values G_{K_c} at the quadrature points on each cell. This is computed by

$$G_{cqk\ell} = |J_{cq}| \sum_{m=1}^{d} DF^{-1}_{cqkm} DF^{-1}_{cq\ell m}. \tag{13}$$

The reader familiar with the the FEniCS Form Compiler [Kirby and Logg 2006] and FErari project [Kirby et al. 2005; Kirby et al. 2006] may recognize this as the geometric tensor generalized to vary over quadrature points and with an additional dimension added to store tensors for several cells at once.

Now, our algorithm proceeds in three separate steps. First, calculating the G array from the pre-tabulated Jacobian information, assuming a fused multiply-add, requires

$$N_c N_q d^2 (d+1) \tag{14}$$

operations. Next, we calculate $\Gamma_i^{K_c}$, storing it in a $N_c \times N_f \times N_q \times d$ array, which we call Γ_{ciqd}. This array stores the action of pointwise G matrices on the reference element gradients. It is computed by

$$\Gamma_{ciqk} = \sum_{\ell=1}^{d} G_{cqk\ell} D\Psi_{iq\ell}, \tag{15}$$

which requires $N_c N_f N_q d^2$ flops.

In the third step, we calculate array A using Γ and the weighted basis function gradients. We rewrite (9) as

$$\begin{aligned} A_{cij} &= \sum_{q=1}^{N_q} w_q \sum_{k=1}^{d} \Gamma_{ciqk} D\Psi_{jqk} \\ &= \sum_{q=1}^{N_q} \sum_{k=1}^{d} \Gamma_{ciqk} D\Psi^w_{jqk}. \end{aligned} \tag{16}$$

The second version of this sum requires

$$N_c N_f^2 N_q d \tag{17}$$

flops.

Summing the operation counts from each of these three phases gives a total flop count of

$$N_c N_q d \left[d(d+1) + N_f d + N_f^2 \right]. \tag{18}$$

To summarize this discussion, we present pseudocode in Algorithm 1.

We require temporary storage of $N_c N_q \left(d^2 + N_f d\right)$ floating point numbers for G and Γ, although this can be reduced to $N_c N_q N_f d$ if the computation of G and Γ are fused (for matrix construction, G is no longer required after Γ is known, but this will not be

ALGORITHM 1: Algorithm for computing Laplace element stiffness matrices using duality assuming basis functions and Jacobians are tabulated.

```
Require: Tabulated inverse Jacobians $DF^{-1}_{cqk\ell}$ and determinants $J_{cq}$.
Require: Tabulated basis functions $D\Psi_{iqk}$ and weighted basis functions $D\Psi^w_{iqk}$.
Require: Allocated space for $G_{cqk\ell}$, $\Gamma_{ciqk}$, and $A_{cij}$
  {Compute $G$}
  for $c = 1$ to $N_c$ do
    for $q = 1$ to $N_q$ do
      for $k, \ell = 1$ to $d$ do
        $G_{cqk\ell} \leftarrow |J_{cq}| \sum_{m=1}^{d} DF^{-1}_{cqkm} DF^{-1}_{cq\ell m}$
      end for
    end for
  end for
  {Compute $\Gamma$}
  for $c = 1$ to $N_c$ do
    for $i = 1$ to $N_f$ do
      for $q = 1$ to $N_q$ do
        for $k = 1$ to $d$ do
          $\Gamma_{ciqk} \leftarrow \sum_{\ell=1}^{d} G_{cqk\ell} D\Psi_{iq\ell}$
        end for
      end for
    end for
  end for
  {Compute $A$}
  for $c = 1$ to $N_c$ do
    for $i, j = 1$ to $N_f$ do
      $A_{cij} \leftarrow \sum_{q=1}^{N_q} \sum_{k=1}^{d} \Gamma_{ciqk} D\Psi^w_{jqk}$
    end for
  end for
```

the case when we consider the matrix action). At any rate, assuming that $N_q \approx N_f$, the storage requirement is a little larger than d times the $N_c N_f^2$ required for A itself.

To fix ideas about the sizes of these objects, suppose (as we will in our numerical tests later) that we use the relatively large value of $N_c = 1000$. For tricubic hexahedra, we have $4^3 = 64$ basis functions per cell. In double precision, then, storing A_{cij} takes $1000 \times 64^2 \times 8$ bytes, or about 33 MB. The intermediate storage, a factor of $d = 3$ larger, gives about 100MB. This comfortably fits on typical GPU cards, so this does not preclude the use of accelarators even if the entire problem cannot fit on board.

The introduction of variable coefficients does not materially affect this technique. To briefly demonstrate this, consider the weak form

$$\int_\Omega \kappa(x) \nabla u \cdot \nabla v \, dx, \tag{19}$$

where $\kappa : \Omega \rightarrow \mathbb{R}$ is some nonnegative function. As before, we can change variables from a cell K_c to the reference cell and apply duality in the inner product to obtain

$$\int_{K_c} \kappa(x) \nabla \psi_i^{K_c} \cdot \nabla \psi_j^{K_c} \, dx = \int_{\hat{K}} \left(\kappa(F_{K_c}(\hat{x})) \, |J_{K_c}| \, DF_{K_c}^{-1} DF_{K_c}^{-T} \hat{\nabla} \Psi_i \right) \cdot \hat{\nabla} \Psi_j \, d\hat{x}. \tag{20}$$

So, if we redefine $G_{K_c}(\hat{x}) \equiv \kappa\left(F_{K_c}(x)\right) |J_{K_c}| \, DF_{K_c}^{-1} DF_{K_c}^{-T}$, the rest of the discussion proceeds unchanged. Once κ is tabulated at the quadrature points for each cell, we simply include an extra pointwise multiplication by this data in the definition of the array $G_{cqk\ell}$ given in (13). If κ were matrix-valued, the same computational pattern would hold, but at an additional cost of d^2 operations per entry of G.

2.1. Some more standard loop variants

We shall show soon the main advantage of our technique, that it can be recast as matrix multiplication, but first we discuss the operation count relative to some other standard variants.

The most straightforward approach may be to loop over the cells and pairs of basis functions on each cell. Thus, we calculate each A_{cij} all at once by looping over quadrature points, transforming basis gradients at that quadrature point, and summing. This is used, for example, in the Deal.II tutorial (step-3) [Bangerth et al. 2007; Bangerth and Kanschat 2013]. This is displayed in Algorithm 2. No temporary storage is required, and although the algorithm requires many redundant basis transformations, it is also highly parallelizable as each entry of A_{cij} can in principle be assigned to a separate thread without any write contention. Since the transformation of a single basis function at a single point requires d^2 operations and the loop nest updating A_{cij} requires d operations for the dot product followed by a multiplication by J_{cq} and a multiply-add pair to scale by w_q and accumulate, the operation count for this algorithm is

$$N_c N_f^2 N_q \left(2d^2 + d + 2\right).$$

A comparison of the leading terms (i.e. those with N_f^2) with (18) shows that this requires about $2d$ times the work as the duality-approach.

ALGORITHM 2: A simple strategy for calculating A_{cij} entry-by-entry. This approach has a high flop count because of redundant basis function transformations, but yields maximal concurrency.

```
Require: Tabulated inverse Jacobians $DF^{-1}_{cqk\ell}$ and determinants $J_{cq}$.
Require: Tabulated basis functions $D\Psi_{iqk}$.
Require: Allocated space for $A_{cij}$, initialized to zero.
Require: Allocates space for $\mu_k$ and $\nu_k$ in $\mathbb{R}^d$
  for $c = 1$ to $N_c$ do
    for $i, j = 1$ to $N_f$ do
      for $q = 1$ to $N_q$ do
        {Transform basis functions at current quadrature point}
        for $\ell = 1$ to $d$ do
          $\mu_\ell \leftarrow \sum_{k=1}^{d} DF^{-1}_{cqk\ell} D\Psi_{iqk}$
        end for
        for $\ell = 1$ to $d$ do
          $\nu_\ell \leftarrow \sum_{k=1}^{d} DF^{-1}_{cqk\ell} D\Psi_{jqk}$
        end for
        {Sum contribution at current quadrature point}
        $A_{cij} \leftarrow A_{cij} + w_q J_{cq} \sum_{k=1}^{d} \mu_k \nu_k$
      end for
    end for
  end for
```

Alternately, we can lower the operation count with a small amount of additional storage by putting the quadrature loop outermost on each cell. Before accumulating any contributions to A_{cij}, we transform all of the basis gradients at ξ_q. We present this approach in Algorithm 3.

Removing redundant transformations reduces the operation count relative to Algorithm 2. The cost of the basis transformations is

$$N_c N_q N_f d^2,$$

while accumulating the contributions to the element matrix costs

$$N_c N_q N_f^2 \left(d+2\right)$$

flops, for a total of

$$N_c N_q N_f \left(d^2 + N_f \left(d+2\right)\right). \tag{21}$$

Such an approach, however, makes it difficult for multiple threads to cooperate on a single element matrix. While this would limit applicability on a GPU, this variant should perform well on a multicore CPU.

Even so, the leading term of (21) is slightly larger than in (18), accounting for the fact that we included the multiplication by Jacobian determinant in G and the quadrature weights are rolled into $D\Psi^w$. So, we actually have a slightly lower operation count than some standard loop nests, whether or not handle many elements at once.

ALGORITHM 3: Lower-cost algorithm for computing Laplace element matrices based on a quadrature-first approach. This algorithm avoids redundant basis function transformations at the cost of a small amount of local storage and reduced available concurrency.

```
Require: Tabulated inverse Jacobians $DF^{-1}_{cqk\ell}$ and determinants $J_{cq}$.
Require: Tabulated basis functions $D\Psi_{iqk}$.
Require: Allocated space for $A_{cij}$, initialized to zero.
Require: Allocated space for $\mu_{ik}$.
  for $c = 1$ to $N_c$ do
    for $q = 1$ to $N_q$ do
      {Transform every basis function at current quadrature point}
      for $i = 1$ to $N_f$ do
        for $\ell = 1$ to $d$ do
          $\mu_{i\ell} \leftarrow \sum_{k=1}^{d} DF^{-1}_{cqk\ell} D\Psi_{iqk}$
        end for
      end for
      {Compute contributation to each local matrix entry at current quadrature point}
      for $i = 1$ to $N_f$ do
        for $j = 1$ to $N_f$ do
          $A_{cij} \leftarrow A_{cij} + w_q J_{cq} \sum_{k=1}^{d} \mu_{ik}\mu_{jk}$
        end for
      end for
    end for
  end for
```

2.2. Multicell matrix-matrix multiplication

So, our duality-based approach does not degrade the operation count. Now, we show how our algorithm can be recast as matrix multiplication. The use of optimized level 3 BLAS has been seen to accelerate the computation or application of element stiffness matrices in other settings, as well. Hesthaven and Warburton [Hesthaven and Warburton 2008] cast the application of nodal spectral element operators in terms of dense derivative matrices acting on sets of elemental degrees of freedom. For affine geometry and constant coefficients, the Sundance project [Long et al. 2010] uses BLAS-based multiplication to construct batches of element stiffness matrices assuming certain reference element quantities are pre-integrated and the geometry is affine. For individual cells, the Intrepid project [Bochev et al. 2012] is able construct stiffness matrices with matrix multiplication without assuming affinity or constant coefficients. However, previously, no known technique generates stiffness matrices for general batches of possibly non-affine cells with variable coefficients with the work predominantly done by a single matrix multiplication.

In our three-stage formulation presented above, the third stage (16) may be recast as a single matrix multiplication for the entire batch of cells. Moreover, as the number of basis functions increases, the level 3 fraction (the proportion of the calculation done in matrix multiplication) approaches one.

First, we reshape $A \in \mathbb{R}^{N_c \times N_f \times N_f}$ as a two-dimensional array $\bar{A} \in \mathbb{R}^{(N_c N_f) \times N_f}$. For $1 \leq c \leq N_c$ and $1 \leq i \leq N_f$, let $I = (c-1)N_f + i$. Given I, the associated c and i are obtained by integer division with remainder. For any $1 \leq I \leq N_c N_f$ and $1 \leq j \leq N_f$, we define

$$\bar{A}_{Ij} = A_{cij}.$$

Assuming that arrays are stored with generalized row-major indexing into flat storage, the mapping $A \mapsto \bar{A}$ requires no data motion on a computer.

We similarly reshape the array Γ given in (15). With I still equal to $(c-1)N_f + i$ and now $Q = (q-1)d + k$ for $1 \leq q \leq N_q$ and $1 \leq k \leq d$, we define $\bar{\Gamma}_{IQ} = \Gamma_{ciqk}$ by

$$\bar{\Gamma}_{IQ} = \Gamma_{ciqk}$$

and $\bar{E} \in \mathbb{R}^{N_f \times (N_q d)}$ by

$$\bar{E}_{iQ} = D\Psi^{w}_{iqk},$$

and as with $\bar{A}$, no data motion is required. With these definitions, (16) becomes

$$A_{cij} = \bar{A}_{Ij} = \sum_{Q=1}^{N_q d} \bar{\Gamma}_{IQ} \bar{E}_{jQ}, \tag{22}$$

or, more compactly, $\bar{A} = \bar{\Gamma}\bar{E}^T$.

The level 3 fraction of our approach is given by

$$\frac{N_c N_f^2 N_q d}{N_c N_q d \left[d\,(d+1) + N_f d + N_f^2 \right]},$$

which after some basic manipulations, is

$$\frac{1}{\frac{d(d+1)}{N_f^2} + \frac{d}{N_f} + 1}. \tag{23}$$

Notice that the level 3 fraction is independent of the number of quadrature points N_q.

For large numbers of basis functions, the level 3 fraction approaches 1, which is to be expected. While explicit matrix construction is typically more practical for lower-order methods where N_f may not be large enough for an optimized BLAS library to deliver peak performance, some reasonable gain should be expected due to internal optimizations such as loop unrolling, tiling, and so on. We explore this later. The level 3 fraction is tabulated for orders one through three for the simplex in Table I and cubical domains in Table II. These are for the constant coefficient case; for variable coefficients, there is an additional cost in forming G leading to a slight reduction in the level 3 fraction.

$d=2$			$d=3$		
Degree	N_f	Level 3 Fraction	Degree	N_f	Level 3 Fraction
1	3	0.43	1	4	0.4
2	6	0.67	2	10	0.70
3	10	0.79	3	20	0.85

Table I: Level 3 fraction for Laplace stiffness matrix construction for linear, quadratic, and cubic bases on triangular and tetrahedral elements.

$d=2$			$d=3$		
Degree	N_f	Level 3 Fraction	Degree	N_f	Level 3 Fraction
1	4	0.53	1	8	0.64
2	9	0.77	2	27	0.89
3	16	0.87	3	64	0.95

Table II: Level 3 fraction for Laplace stiffness matrix construction for linear, quadratic, and cubic bases on quadrilateral and hexahedral elements.

2.3. Relation to tensor contraction in `ffc`/FErari

We return to our previous comment about the tensor contraction formulation of form evaluation used in [Kirby et al. 2005; Kirby and Logg 2006], showing that we may recover it by assuming affine geometry and rearranging the order of summation.

If we assume affine geometry, the Jacobian matrix will take the same value at each quadrature point on a given cell. We denote the rank-4 array $G_{cqk\ell}$ that is constant in q for each cell as $G_{cqk\ell} = \mathcal{G}_{ck\ell}$ and revisit our stiffness matrix computation. We modify the computation of Γ_{cqik} in (15) by

$$\Gamma_{ciqk} = \sum_{\ell=1}^{d} \mathcal{G}_{ck\ell} D\Psi_{iq\ell}. \tag{24}$$

Now, we substitute this into (16), interchange summation so that the quadrature loop is innermost, and hoist $\mathcal{G}$ appropriately to obtain

$$\begin{aligned} A_{cij} &= \sum_{q=1}^{N_q} \sum_{k=1}^{d} \Gamma_{ciqk} D\Psi^w_{jqk} \\ &= \sum_{q=1}^{N_q} \sum_{k=1}^{d} \left(\sum_{\ell=1}^{d} \mathcal{G}_{ck\ell} D\Psi_{iq\ell} \right) D\Psi^w_{jqk} \\ &= \sum_{k=1}^{d} \sum_{\ell=1}^{d} \mathcal{G}_{ck\ell} \sum_{q=1}^{N_q} w_q D\Psi_{iq\ell} D\Psi_{jq\ell}. \end{aligned} \tag{25}$$

Defining

$$A^0_{ijk\ell} = \sum_{q=1}^{N_q} w_q D\Psi_{iq\ell} D\Psi_{jq\ell}, \tag{26}$$

we have

$$A_{cij} = \sum_{k=1}^{d} \sum_{\ell=1}^{d} \mathcal{G}_{ck\ell} A^0_{ijk\ell}, \tag{27}$$

which is, up to the exactness of the quadrature rule, the formulation used in [Kirby and Logg 2006] and optimized by discrete structure in [Kirby et al. 2005].

Assuming that A^0_{ijkl} is precomputed, evaluating A_{cij} by (27) this formulation has a much lower operation count. Computing $\mathcal{G}$ requires

$$N_c d^2 \left(d+1\right)$$

flops, a factor of N_q lower than G. Then, forming A^0 by (26) requires

$$2N_f^2 d^2 N_q$$

flops. This is done once on the reference element – the operation count is independent of N_c. Finally, computing A from A^0 and $\mathcal{G}$ via (27) requires

$$N_c N_f^2 d^2$$

operations. The total operation count is

$$2N_f^2 d^2 N_q + N_c d^2 \left(d + 1 + N_f^2\right). \tag{28}$$

The final stage can be cast as matrix multiplication. We can reshape $\mathcal{G}$ to a $N_c \times d^2$ array and multiply by the transpose of reshaping A^0 to $N_f^2 \times d^2$. While the relatively small (d^2) shared shared dimension of this matrix multiplication may limit performance gains obtained from an optimized BLAS routine, the vastly reduced operation count provides a major advantage for constant coefficient operators in affine geometry.

2.4. Matrix-free action

Krylov methods for the solution of linear systems only require the application of the system matrix to a vector at each iteration. Although many effective preconditioners, such as incomplete factorizations and algebraic multigrid, require matrix entries, higher-order discretizations frequently make use of preconditioners for low-order methods on finer meshes [Heys et al. 2005; Orszag 1979]. This allows one to bypass the high memory footprint and large cost associated with forming the element stiffness matrices and assembling them into a gobal sparse matrix, provided that the operator can be applied relatively efficiently. Our present techniques are readily adapted to computing the matrix action, and the use of an optimized BLAS library leads to efficiency that approaches that of Trilinos sparse matrix multiplication.

Again, we consider a batch of cells $\{K_c\}_{c=1}^{N_c}$ with mappings to a reference element and basis functions as before. However, we suppose we have some function $u(x)$ defined over the cells $\cup_{c=1}^{N_c} K_c$ that is a member of the global H^1 finite element space. After scattering the global degrees of freedom to the individual cells, we obtain the $N_c \times N_f$ array U such that

$$u(x)|_{K_c} = \sum_{i=1}^{N_f} U_{ci} \psi_i^{K_c}(x).$$

Our goal is to construct the array $A \in \mathbb{R}^{N_c \times N_f}$ such that

$$A_{ci} = \int_{K_c} \nabla u \cdot \nabla \psi_i^{K_c} \, dx. \tag{29}$$

This array is then gathered into a global storage vector, summing elementwise contributions.

We map the relevant integrals to the reference element by

$$A_{ci} = \int_{\hat{K}} DF_{K_c}^{-T} \hat{\nabla} \left(\sum_{j=1}^{N_f} U_{cj} \Psi_j \right) \cdot DF_{K_c}^{-T} \hat{\nabla} \Psi_i \left| J_{K_c} \right| \, dx.$$

Next, we interchange summation with differentiation and move the transformation from the test function gradient to the trial to obtain

$$\begin{aligned} A_{ci} &= \int_{\hat{K}} DF_{K_c}^{-T} \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \Psi_j \right) \cdot DF_{K_c}^{-T} \hat{\nabla} \Psi_i \left| J_{K_c} \right| \, dx \\ &= \int_{\hat{K}} \left(\left| J_{K_c} \right| DF_{K_c}^{-1} DF_{K_c}^{-T} \right) \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \Psi_j \right) \cdot \hat{\nabla} \Psi_i \, d\hat{x} \\ &= \int_{\hat{K}} \left(G_{K_c} \right) \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \Psi_j \right) \cdot \hat{\nabla} \Psi_i \, d\hat{x} \\ &= \int_{\hat{K}} \Gamma_{K_c} \cdot \hat{\nabla} \Psi_i \, d\hat{x}. \end{aligned} \tag{30}$$

where the same definition of G_{K_c} holds as above and we have now defined

$$\Gamma_{K_c} = G_{K_c} \left(\sum_{j=1}^{N_f} U_{c,j} \hat{\nabla} \Psi_j \right).$$

Note that Γ now has one less index than for matrix construction – we have one field per cell (tabulated at quadrature points) rather than one per each basis function.

Evaluating the integrals in (30) via numerical quadrature gives

$$A_{ci} \approx \sum_{q=1}^{N_q} w_q \sum_{k=1}^{d} \Gamma_{cqk} D\Psi_{iqk} = \sum_{q=1}^{N_q} \sum_{k=1}^{d} \Gamma_{cqk} D\Psi^{w}_{iqk}, \tag{31}$$

where Γ_{cqk} contains the components of Γ_{K_c} tabulated at each quadrature point.

For the stiffness matrix construction, Γ was a rank 4 array with indices over the cell, basis function, quadrature point, and spatial dimension. For the matrix action, we do not have a basis function index, only a rank 3 array with indices over cell, quadrature point, and spatial dimension.

The computation of $G_{cqk\ell}$ by (13) using $N_c N_q d^2 \left(d+1\right)$ flops proceeds as before. However, since it is computed once and reused for each matrix-vector product, we omit this small cost from our subsequent operation counts. To construct Γ, we require the quantity $\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \Psi_i \left(\xi_q\right)$ at each quadrature point q. We let Δ_{cqk} be the $N_c \times N_q \times d$ array such that

$$\Delta_{cqk} = \sum_{j=1}^{N_f} U_{cj} D\Psi_{jqk}. \tag{32}$$

Constructing this array costs $N_c N_q N_f d$ operations, after which we form Γ by

$$\Gamma_{cqk} = \sum_{\ell=1}^{d} G_{cqk\ell} \Delta_{cq\ell}, \tag{33}$$

$d = 2$			$d = 3$		
Degree	N_f	Level 3 Fraction	Degree	N_f	Level 3 Fraction
1	4	0.75	1	8	0.73
2	9	0.86	2	27	0.87
3	16	0.91	3	64	0.93

Table III: Level 3 fraction for matrix-free stiffness matrix action for linear, quadratic, and cubic bases on triangular and tetrahedral elements.

requiring another $N_c N_q d^2$ operations. Computing (31) requires a final $N_c N_f N_q d$ operations. Summing up the flop counts of these three stages gives a total of

$$N_c N_q d \left[d + 2N_f \right]. \tag{34}$$

For temporary storage beyond the elementwise storage of the function coefficients, we require space for $N_c N_q d^2$ numbers to store G and $N_c N_q d$ numbers to store Δ. The computation of Γ can overwrite Δ.

The formation of both Δ_{cqk} and A_{ci} can be cast as matrix multiplication. With $Q = (q-1)d + k$ as before, let $\bar{\Delta}_{cQ} = \Delta_{cqk}$ and $\bar{E}_{iQ} = D\Psi_{iqk}$. Then,

$$\Delta_{cqk} = \bar{\Delta}_{cQ} = \sum_{i=1}^{N_f} U_{ci} \bar{E}_{iQ}, \tag{35}$$

so that $\bar{\Delta} = U\bar{E}$. Similarly, (31) can be rewritten as

$$A_{ci} = \sum_{Q=1}^{N_q d} \bar{\Gamma}_{cQ} \bar{E}_{iQ}, \tag{36}$$

with $\bar{\Gamma}_{cQ} = \Gamma_{cqk}$ and $\bar{E}_{iQ} = D\Psi^w_{iqk}$ so that $\bar{A} = \bar{\Gamma}\bar{E}^T$.

So then, these two stages admit $2N_c N_q N_f d$ flops being done in matrix multiplication, giving a level 3 fraction of

$$\frac{2N_c N_q N_f d}{N_c N_q d \left[d + 2N_f \right]} = \frac{1}{\frac{d}{2N_f} + 1}. \tag{37}$$

This quantity approaches 1 with the number of basis functions, and as seen in Tables III and IV, is quite close to 1 for relatively low-order elements.

Finally, consider the temporary storage requirements associated with this approach. Again taking $N_c = 1000$ and using tricubic elements with $N_f = 64$, we require storage for G consisting of $N_c \times N_q \times 3 \times 3$ doubles. If $N_q = N_f$ (e.g. Gauss-Lobatto quadrature), this is about 4.6 MB of storage. For Γ, we require $N_c \times N_q \times d$ doubles, about 1.5MB of storage. Storing the element degrees of freedom in U and the result of the action each require $N_c \times N_f$ storage, about half a megabyte. Such sizes are quite modest, even on low-memory accelerators.

3. ANOTHER EXAMPLE: THE CURL-CURL OPERATOR

Our approach is specific neither to the Poisson operator nor to H^1 finite element spaces. For example, we consider the variational form

$$\int_\Omega (\nabla \times u) \cdot (\nabla \times v)\, dx, \tag{38}$$

$d=2$			$d=3$		
Degree	N_f	Level 3 Fraction	Degree	N_f	Level 3 Fraction
1	3	0.8	1	4	0.84
2	6	0.9	2	10	0.95
3	10	0.94	3	20	0.98

Table IV: Level 3 fraction for matrix-free stiffness matrix action for linear, quadratic, and cubic bases on quadrilateral and hexahedral elements.

where $u, v \in H(\mathrm{curl})$ and again consider a patch of cells $\{K_c\}_{c=1}^{N_c}$ with associated mappings $F_{K_c} : \hat{K} \to K_c$ and a reference basis for $\{\Psi_i\}_{i=1}^{N_f}$ a finite-dimensional subspace of $H(\mathrm{curl})$ such as that of Nédélec [Nédélec 1980]. Following [Bochev and Gunzburger 2009], $H(\mathrm{curl})$ bases transform by the covariant Piola transform Φ_C^* defined earlier, so that

$$\psi_i^{K_c} = \Phi_C^*\left(\Psi_i\right) = \left(DF_{K_c}^{-T}\Psi_i\right) \circ F_{K_c}^{-1}. \tag{39}$$

When the curl is taken of this transformed field on K_c, it becomes a different transformation applied to the curl of the reference element field [Bochev and Gunzburger 2009]:

$$\nabla \times \psi_i^{K_c} = \left(J_{K_c}^{-1} DF_{K_c} \hat{\nabla} \times \Psi_i\right) \circ F_{K_c}^{-1} \equiv \Phi_D^*\left(\Psi_i\right), \tag{40}$$

which is the contravariant Piola transformation used for $H(\mathrm{div})$. This commmuting relationship allows us to proceed as with the Laplace operator.

Let $A \in \mathbb{R}^{N_c \times N_f \times N_f}$ be the element stiffness matrices

$$A_{cij} = \int_{K_c} \left(\nabla \times \psi_i^{K_c}\right) \cdot \left(\nabla \times \psi_j^{K_c}\right) \, dx. \tag{41}$$

Using (40), we map these integrals back to $\hat{K}$ and use duality so that

$$\begin{aligned} A_{cij} &= \int_{\hat{K}} \left(J_{K_c}^{-1} DF_{K_c} \hat{\nabla} \times \Psi_i\right) \cdot \left(J_{K_c}^{-1} DF_{K_c} \hat{\nabla} \times \Psi_j\right) |J_{K_c}| \, d\hat{x} \\ &= \int_{\hat{K}} \left(|J_{K_c}|^{-1} DF_{K_c}^T DF_{K_c} \hat{\nabla} \times \Psi_i\right) \cdot \left(\hat{\nabla} \times \Psi_j\right) d\hat{x} \\ &= \int_{\hat{K}} \left(G_{K_c} \hat{\nabla} \times \Psi_i\right) \cdot \left(\hat{\nabla} \times \Psi_j\right) d\hat{x}, \\ &= \int_{\hat{K}} \Gamma_i^{K_c} \cdot \left(\hat{\nabla} \times \Psi_j\right) d\hat{x}, \end{aligned} \tag{42}$$

where

$$G_{K_c} = J_{K_c}^{-1} DF_{K_c}^{-T} DF_{K_c}^{-1} \tag{43}$$

$$\Gamma_i^{K_c} = G_{K_c}\left(\hat{\nabla} \times \Psi_i\right). \tag{44}$$

We are using same symbols as for the Laplacian with slightly different definitions to stress the near-complete analogy between the two cases.

Rather than the gradients tabulated on the reference element, we require the curls, so we define $D\Psi_{iqk}$ to be the k^{th} vector component of $\hat{\nabla} \times \Psi_i$ tabulated at ξ_q, and we also define $D\Psi_{iqk}^w = w_q D\Psi_{iqk}$. We let $G \in \mathbb{R}^{N_c \times N_q \times d \times d}$ to be the G_{K_c} matrices tabulated at quadrature points. This is calculated from the precomputed Jacobians in $N_c N_q d^2$ flops by

$$G_{cqk\ell} = \left(\sum_{m=1}^{d} DF^T_{cqmk} DF_{cqm\ell} \right) / |J_{cq}|. \tag{45}$$

Second, analagous to (15), G must act on the reference element curls to give

$$\Gamma_{ciqk} = \sum_{\ell=1}^{d} G_{cqk\ell} D\Psi_{iq\ell}, \tag{46}$$

and finally

$$A_{cij} = \sum_{q=1}^{N_q} \sum_{k=1}^{3} \Gamma_{ciqk} D\Psi^w_{jqk}. \tag{47}$$

The operation counts associated with (46) and (47) are exactly the same as for (15) and (16), respectively, considered as functions of N_c, N_f, N_q, and d. The curl-curl stiffness matrix costs the same to compute, per entry, as the Poisson matrix, but the dimension of the Nédélec space is $3k\left(k+1\right)^2$ rather than $\left(k+1\right)^3$.

We may also adapt this approach to apply the curl-curl operator elementwise without forming either the global or element stiffness matrices. If we have $u|_{K_c} = \sum_{i=1}^{N_f} U_{ci} \psi_i^{K_c}$, then we wish to compute

$$A_{ci} = \int_{K_c} \left(\nabla \times u\right) \cdot \left(\nabla \times \psi_i^{K_c}\right) dx \tag{48}$$

and then collect the elementwise results into a global vector. By expanding u in the basis, using linearity and the pullbacks, we have

$$\begin{aligned} A_{ci} &= \int_{K_c} \left(\nabla \times \left(\sum_{j=1}^{N_f} U_{cj} \psi_j^{K_c} \right) \right) \cdot \left(\nabla \times \psi_i^{K_c} \right) \\ &= \int_{K_c} \left(\sum_{j=1}^{N_f} U_{cj} \left(\nabla \times \psi_j^{K_c} \right) \right) \cdot \left(\nabla \times \psi_i^{K_c} \right) dx \\ &= \int_{K_c} \left(\sum_{j=1}^{N_f} U_{cj} \nabla \times \Phi_C^* \left(\Psi_j\right) \right) \cdot \left(\nabla \times \Phi_C^* \left(\Psi_i\right) \right) dx. \end{aligned} \tag{49}$$

Now, we use the commuting relationship for the curl operator, linearity, and change variables to write

$$\begin{aligned} A_{ci} &= \int_{K_c} \left(\sum_{j=1}^{N_f} U_{cj} \Phi_D^* \left(\hat{\nabla} \times \Psi_j \right) \right) \cdot \left(\Phi_D^* \left(\hat{\nabla} \times \Psi_i \right) \right) dx \\ &= \int_{K_c} \left(\Phi_D^* \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \times \Psi_j \right) \right) \cdot \left(\Phi_D^* \left(\hat{\nabla} \times \Psi_i \right) \right) dx \\ &= \int_{\hat{K}} \left(J_{K_c}^{-1} DF_{K_c} \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \times \Psi_j \right) \right) \cdot \left(J_{K_c}^{-1} DF_{K_c} \left(\hat{\nabla} \times \Psi_i \right) \right) |J_{K_c}| d\hat{x}. \end{aligned} \tag{50}$$

Using the adjoint relation, we have

$$\begin{aligned} A_{ci} &= \int_{K_c} \left(|J_{K_c}|^{-1} DF_{K_c}^T DF_{K_c} \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \times \Psi_j \right) \right) \cdot \left(\hat{\nabla} \times \Psi_i \right) d\hat{x} \\ &= \int_{K_c} \left(G_{K_c} \sum_{j=1}^{N_f} U_{cj} \nabla \times \Psi_j \right) \cdot (\nabla \times \Psi_i)\, d\hat{x} \\ &= \int_{K_c} \Gamma^{K_c} \cdot (\nabla \times \Psi_i)\, d\hat{x}, \end{aligned} \tag{51}$$

where $\Gamma^{K_c} = G_{K_c} \left(\sum_{j=1}^{N_f} U_{cj} \hat{\nabla} \times \Psi_j \right)$. To evaluate this numerically, we first compute the quantity

$$\Delta_{cqk} = \sum_{j=1}^{N_f} U_{cj} D\Psi_{jqk},$$

After this, the transformation G is applied so that

$$\Gamma_{cqk} = \sum_{\ell=1}^{d} G_{cqk\ell} \Delta_{cq\ell},$$

and finally

$$A_{ci} = \sum_{q=1}^{N_q} \sum_{k=1}^{d} \Gamma_{cqk} D\Psi^w_{iqk}.$$

The computation proceeds exactly the same as for the Laplacian, and the operation count and level three fraction are exactly parallel to those for the Laplacian, as well.

4. A GENERAL FRAMEWORK

Having seen some examples, we now present an abstract setting. While this requires some amount of additional notation, it will illustrate the general utility of optimized matrix multiplication for bilinear form evaluation. We still let $\{K_c\}_{c=1}^{N_c}$ denote a collection of cells diffeomorphic to the reference cell $\hat{K}$ via mappings F_{K_c}.

On each cell K_c, let $\mathcal{W}_i$, for $i = 1, 2$ be finite-dimensional vector spaces and $\mathcal{U}_i$ for $i = 1, 2$ be finite-dimensional spaces of functions mapping K_c into $\mathcal{W}_i$, respectively. We also require $(\mathcal{W}, \langle \cdot, \cdot \rangle)$ to be a finite-dimensional inner product space. The members of $\mathcal{W}$ are indexed by some set $\mathcal{A}$, and we assume that $\mathcal{W}$ is equipped with an inner product $\langle \cdot, \cdot \rangle$ of the form

$$\langle v, w \rangle = \sum_{\alpha \in \mathcal{A}} v_\alpha w_\alpha \tag{52}$$

for all $v, w \in \mathcal{W}$. Let $\mathcal{V}_i$, $i = 1, 2$ be a second pair of finite-dimensional spaces of functions mapping K_c into the inner-product space $(\mathcal{W}, \langle \cdot, \cdot \rangle)$. Also, let $D_i : \mathcal{U}_i \to \mathcal{V}_i$ for $i = 1, 2$ be a pair of linear mappings.

We consider elementwise bilinear forms of the form $a_c : \mathcal{U}_1 \times \mathcal{U}_2 \to \mathbb{R}$ of the form

$$a_c(u_1, u_2) = \int_{K_c} \langle D_1 u_1, D_2 u_2 \rangle dx. \tag{53}$$

While it may not be obvious, this abstract formulation does admit variable coefficients; they will be absorbed into the definitions of D_1 and $\mathcal{V}_1$.

We assume that $\mathcal{U}_i$ and $\mathcal{V}_i$ are the images of some spaces $\hat{\mathcal{U}}_i$ and $\hat{\mathcal{V}}_i$ under pullbacks $\Phi^*_{\mathcal{U}_i}$ and $\Phi^*_{\mathcal{V}_i}$, where the hatted spaces contain functions mapping $\hat{K}$ into the finite-dimensional spaces $\mathcal{W}_i$ and $\mathcal{W}$. We also assume reference-element analogs of the linear maps D_i, which we denote by $\hat{D}_i : \hat{\mathcal{U}}_i \rightarrow \hat{\mathcal{V}}_i$. We require that the commuting properties

$$\begin{aligned} D_1 \Phi^*_{\mathcal{U}_1}(\hat{u}) &= \omega \Phi^*_{\mathcal{V}_1}\left(\hat{D}\hat{u}\right) \\ D_2 \Phi^*_{\mathcal{U}_2}(\hat{u}) &= \Phi^*_{\mathcal{V}_2}\left(\hat{D}\hat{u}\right) \end{aligned} \tag{54}$$

hold for all $\hat{u} \in \hat{\mathcal{U}}_i$. ω is some linear operator necessary to encode a coefficient.

We require that at least the pullbacks for the $\mathcal{V}_i$ spaces to take the form of a linear transformation plus a coordinate change. That is, we require the existence of $R_{\mathcal{V}_i}$ such that

$$\Phi^*_{\mathcal{V}_i}(\hat{u}) = R_i\left(\hat{u} \circ F^{-1}_{K_c}\right) \tag{55}$$

for each $\hat{u}$. We have

$$\begin{aligned} D_1 \Phi^*_{\mathcal{U}_1}(\hat{u}) &= \omega\left(R_1 \hat{u}\right) \circ F^{-1}_{K_c} \\ D_2 \Phi^*_{\mathcal{U}_2}(\hat{u}) &= \left(R_2 \hat{u}\right) \circ F^{-1}_{K_c} \end{aligned} \tag{56}$$

Now, we let $N_{\mathcal{U}_i} = \dim \mathcal{U}_i = \dim \hat{\mathcal{U}}_i$. We let $\left\{\Psi^i_j\right\}_{j=1}^{N_{\mathcal{U}_i}}$ be a basis for $\hat{\mathcal{U}}_i$ and let $\psi^i_j = \Phi^*_{\mathcal{U}_i}\left(\Psi^i_j\right)$ be the basis functions for $\mathcal{U}_i$. Before formulating the matrix construction and application in this abstract setting, we turn to some examples.

4.1. Examples

For the weak Laplacian

$$a_c(u, v) = \int_{K_c} \nabla u \cdot \nabla v dx, \tag{57}$$

$\mathcal{U}_1 = \mathcal{U}_2$ is some finite element subspace of $H^1(K_c)$ and $D_1 = D_2 = \nabla$ is the gradient. $\mathcal{W}_1 = \mathcal{W}_2 = \mathcal{W} = \mathbb{R}^d$ equipped with the Euclidean inner product. The spaces $\mathcal{V}_i$ are simply the codomains of ∇ and $\hat{\mathcal{U}}_i$ are the reference element spaces. In the commuting relations (54) and (55) we have $\kappa = I$ and $R_1 = R_2 = DF^{-T}_{K_c}$.

For the variable coefficient operator

$$a_c(u, v) = \int_{K_c} \kappa \nabla u \cdot \nabla v dx, \tag{58}$$

we take $D_1 = \kappa\nabla$. Then, while $\mathcal{U}_i$ and $\hat{\mathcal{U}}_i$ are unchanged, $\mathcal{V}_1 = \{\kappa \nabla u : u \in \mathcal{U}_1\}$ and $\hat{\mathcal{V}}_1$ is similarly modified. In (54), we have that $\omega = \kappa$, and the commuting relations (54) and (55) for D_1 just become

$$\kappa \nabla \Phi^*_{\mathcal{U}_1}(\hat{u}) = \kappa DF^{-T}_{K_c} \hat{\nabla} \hat{u} \tag{59}$$

To cast the curl-curl example from Section 3 in this framework, we let $\hat{\mathcal{U}}_i$ be the the reference element Nedelec spaces with $\hat{\mathcal{V}}_i$ the codomain of the reference element curl. The physical spaces $\mathcal{U}_i$ are obtained from $\hat{\mathcal{U}}_i$ by the pullback Φ^*_C. The operators $D_1 = D_2 = \nabla\times$ are the curl operator, with $\hat{D}_i$ the curl in the reference element coordinates. From (40), we know that $R_1 = R_2 = J^{-1}_{K_c} DF_{K_c}$ and that ω is the identity.

4.2. Matrix construction

Returning to the abstract setting, our element stiffness matrices take the form

$$A_{cij} = a_c(\psi_i^1, \psi_j^2) = \int_{K_c} \langle D_1 \psi_i^1, D_2 \psi_j^2 \rangle dx. \tag{60}$$

Using the pullbacks of the basis functions and linearity, we can change these integrals to the reference elements by

$$\begin{aligned} A_{cij} &= \int_{K_c} \langle D_1 \psi_i^1, D_2 \psi_j^2 \rangle dx \\ &= \int_{K_c} \langle D_1 \Phi^*_{\mathcal{U}_1}\left(\Psi_i^1\right), D_2 \Phi^*_{\mathcal{U}_2}\left(\Psi_j^2\right) \rangle dx \\ &= \int_{K_c} \langle \omega \Phi^*_{\mathcal{V}_1}\left(\hat{D}_1 \Psi_i^1\right), \Phi^*_{\mathcal{V}_2}\left(\hat{D}_2 \Psi_j^2\right) \rangle dx \\ &= \int_{\hat{K}} \langle \omega R_1 \hat{D}_1 \Psi_i^1, R_2 \hat{D}_2 \Psi_j^2 \rangle |J_{K_c}| d\hat{x} \\ &= \int_{\hat{K}} \langle |J_{K_c}| R_2' \omega R_1 \hat{D}_1 \Psi_j^1, \hat{D}_2 \Psi_j^2 \rangle d\hat{x}, \end{aligned} \tag{61}$$

where R_2' denotes the adjoint operator to R_2. We define $G_{K_c} = |J_{K_c}| R_2' \omega R_1$ and then $\Gamma_i^{K_c} = G_{K_c} \hat{D}_1 \Psi_i^1$. We then have

$$A_{cij} = \int_{\hat{K}} \langle \Gamma_i^{K_c}, \hat{D}_2 \Psi_j^2 \rangle d\hat{x}. \tag{62}$$

We evaluate (possibly approximately) the integrals in (62) by numerical quadrature. Analagous to the $D\Psi$ arrays used before, we let $D\Psi^j_{iq\alpha}$ be the α entry of $\hat{D}_j$ applied to Ψ_i^j evaluated at quadrature point ξ_q on the reference element. We also let $D\Psi^{2,w}_{iq\alpha} = w_q D\Psi^2_{iq\alpha}$

Next, we define the pointwise matrices that store the composite transformations. In the worst case, this will require storing $N_c N_q |\mathcal{A}|^2$ numbers in an array $G_{cq\alpha\beta}$. This corresponds to an array storing a $|\mathcal{A}| \times |\mathcal{A}|$ matrix at each quadrature point of each cell that is For many elements, this may be typical, but there are particular cases where there are alternatives. For example, the Arnold-Winther elements [Arnold and Winther 2002] provide a polynomial-based symmetric tensor finite element for use in two-dimensional elasticity. After defining a reference element, they use the matrix Piola transform so that a reference tensor $\hat{\tau}$ is transformed to $DF_{K_c}\left(\hat{\tau} \circ F_{K_c}^{-1}\right) DF_{K_c}^T$. This linear transformation is more compactly stored and more efficiently applied than reshaping the symmetric tensor input as a 3-vector and forming a 3×3 matrix that encodes the action of this transformation.

At any rate, we require this array to act on the array $D\Psi^1$ so that we form an array $\Gamma_{ciq\alpha}$ by

$$\Gamma_{ciq\alpha} = \sum_{\beta \in \mathcal{A}} G_{cq\alpha\beta} D\Psi^1_{iq\beta}, \tag{63}$$

realizing that in certain cases like the Arnold-Winther element this may be done without explicit values for G.

After this, we can write A_{cij} as a sum over quadrature points and components as

$$A_{cij} \approx \sum_{q=1}^{N_q} w_q \sum_{\alpha \in \mathcal{A}} \Gamma_{ciq\alpha} D\Psi^2_{jq\alpha} = \sum_{q=1}^{N_q} \sum_{\alpha \in \mathcal{A}} \Gamma_{ciq\alpha} D\Psi^{2,w}_{jq\alpha}. \tag{64}$$

We can impose a linear ordering on the elements of $\mathcal{A}$ and reshape Γ as a $N_c N_{\mathcal{U}_1} \times N_q|\mathcal{A}|$ array $\bar{\Gamma}$ and $D\Psi^{2,w}$ as a $N_{\mathcal{U}_2} \times N_q|\mathcal{A}|$ array $\bar{D}$. $\bar{A}$, the $N_c N_{\mathcal{U}_1} \times N_{\mathcal{V}_1}$ reshaping of A, is computed by the matrix-matrix product $\bar{\Gamma}^T \bar{D}$.

Assuming that all calculations are carried out explicitly (i.e. ω, R_1 and R_2 are stored as $|\mathcal{A}| \times |\mathcal{A}|$ matrices), we can show this approach yields a high level 3 fraction. Assuming that R_1 and R_2 are given, the formation of G requires $N_c N_q |\mathcal{A}|^2 \left(|\mathcal{A}| + 1\right)$ operations. Forming $\Gamma_{ciq\alpha}$ requires a further $N_c N_q N_{\mathcal{U}_1} |\mathcal{A}|^2$ operations. The last step, forming A requires $N_c N_q N_{\mathcal{U}_1} N_{\mathcal{U}_2} |\mathcal{A}|$ operations. The total operation count is

$$N_c N_q |\mathcal{A}| \left(|\mathcal{A}| \left(|\mathcal{A}| + 1 \right) + N_{\mathcal{U}_1} |\mathcal{A}| + N_{\mathcal{U}_1} N_{\mathcal{U}_2} \right), \tag{65}$$

of which

$$N_c N_q |\mathcal{A}| N_{\mathcal{U}_1} N_{\mathcal{U}_2} \tag{66}$$

are performed in matrix multiplication, giving (after some algebra) a level 3 fraction of

$$\frac{1}{\frac{|\mathcal{A}|(|\mathcal{A}|+1)}{N_{\mathcal{U}_1} N_{\mathcal{U}_2}} + \frac{|\mathcal{A}|}{N_{\mathcal{U}_2}} + 1}. \tag{67}$$

When $|\mathcal{A}| = d$ and $N_{\mathcal{U}_1} = N_{\mathcal{U}_2} \equiv N_f$, this exactly recovers the count found in (23).

4.3. Matrix action

We also consider the construction of the elementwise action of variational forms. Letting $u \in \mathcal{U}_1$ on each K_c with $u|_{K_c} = \sum_{j=1}^{N_\mathcal{U}} U_{cj} \psi^1_j$, we need to compute

$$A_{ci} = \int_{K_c} \langle D_1 u, D_2 \psi^2_i \rangle dx. \tag{68}$$

for all $1 \leq c \leq N_c$ and $1 \leq i \leq N_{\mathcal{U}_2}$.

Applying the pullbacks, linearity, the commuting relations, and adjoints leads to

$$\begin{aligned} A_{ci} &= \int_{K_c} \langle D_1 u, D_2 \psi^2_i \rangle dx \\ &= \int_{\hat{K}} \langle G_{K_c} \left(\sum_{j=1}^{N_{\mathcal{U}_1}} U_{cj} \hat{D}_1 \Psi^1_j \right), \hat{D}_2 \Psi^2_i \rangle d\hat{x} \\ &= \int_{\hat{K}} \langle \Gamma^{K_c}, \hat{D}_2 \Psi^2_i \rangle d\hat{x} \end{aligned} \tag{69}$$

where the operator G_{K_c} and its tabulation $G_{cq\alpha\beta}$ are as for matrix construction and

$$\Gamma_{K_c} = G_{K_c} \left(\sum_{i=1}^{N_\mathcal{U}} U_{ci} \hat{D} \Psi^\mathcal{U}_i \right). \tag{70}$$

To calculate the action by quadrature, we tabulate Γ_{K_c} at the quadrature points by first finding

$$\Delta_{cq\alpha} = \sum_{i=1}^{N_\mathcal{U}} U_{ci} D\Psi^\mathcal{U}_{iq\alpha}, \tag{71}$$

to which we apply the G array to obtain Γ by

$$\Gamma_{cq\alpha} = \sum_{\beta \in \mathcal{A}} G_{cq\alpha\beta} \Delta_{cq\beta}. \tag{72}$$

Finally, A_{ci} is computed by

$$A_{ci} = \sum_{q=1}^{N_q} w_q \sum_{\alpha \in \mathcal{A}} \Gamma_{cq\alpha} D\Psi^2_{iq\alpha} = \sum_{q=1}^{N_q} \sum_{\alpha \in \mathcal{A}} \Gamma_{cq\alpha} D\Psi^{2,w}_{iq\alpha}, \tag{73}$$

which, as the first stage, can be reshaped into matrix multiplication.

5. SOME NUMERICAL RESULTS

We have implemented several of these these algorithms in C++ using the Trilinos project [Heroux et al. 2005]. In particular, we use Intrepid [Bochev et al. 2012] to provide basis functions, integration rules, and elementwise Jacobians as well as its `FieldContainer` as a general multi-dimensional array. We also use `Epetra_FECrsMatrix` and `Epetra_FEVector` to store sparse matrices and global vectors. We also provide some comparisons against FEniCS version 1.2.0. We have run all our experiments on a single 2.70GHz Intel Xeon core of a Dell Precision workstation with 128GB of RAM running Linux Mint. The peak performance of a single thread on this machine is approximately 10.8 GFlops (four times the clock rate). All code was compiled with the Ubuntu-packaged gcc 4.6.3 suite. We compiled and linked against a single-threaded build of OpenBLAS [OpenBlas 2013], which proved notably faster than the Ubuntu-packaged ATLAS [Whaley and Dongarra 1998] routines for the sizes and shapes of matrix multiplication we require.

In our C++ code, we used as a baseline a straightforward implementation of Algorithm 2 for assembling the Laplacian and an analgous one for the curl-curl operator. We also used a similar approach to implement the actions of both of these operators. We also wrote similar implementations of the duality-based method Algorithm 3 using basic C++ loops. Because of the slightly lower operation count, we observed slight decreases in the run-time. Then, we also replaced the relevant loops with calls to the general matrix-matrix multiplication routine `DGEMM` from OpenBLAS to measure the impact of an optimized matrix multiplication routine on the run-time. Also relevant to the discussion is the time spent interacting with global storage. For assembling the stiffness matrices, we also report the time to sum the element matrices into a global sparse matrix. For the matrix actions, we report the time element computation, the total time for the matrix-vector product, and a comparison to the sparse matrix-vector product provided by an assembled Epetra matrix. In the next two subsections, we summarize our findings for these situations, and then provide a brief comparison to FEniCS in the final subsection.

5.1. The Laplace operator

In Table V, we present timings for the Laplace operator on a set of 1000 hexahedra, discretized with tensor-product polynomials of orders 1, 2, and 3. The second through fourth columns give the time required to build A_{cij} using the standard variant in Algorithm 2, our straight C++ implementation of Algorithm 1, and the build time using OpenBLAS. All of these columns include the time to compute $G_{cqk\ell}$ and Γ_{ciqk} as well as A_{cij}, but not to sum into global storage. The final column gives the time required to perform that phase, which is independent of the technique used for building A_{cij}.

We observe a modest speedup going from the standard variant to duality, simply because of the lower operation count. However, replacing the computation of A_{cij} from

k	basic loop	dual	BLAS (GFLOPs)	Assembly time
1	6.64E-003	4.74E-003	1.91E-003 (1.26)	2.65E-003
2	2.51E-001	1.16E-001	2.60E-002 (2.56)	5.18E-002
3	3.29E+000	1.39E+000	1.77E-001 (4.66)	3.96E-001

Table V: Performance numbers for assembling the Laplace operator on a mesh of 1000 hexahedra. The first column (k) indicates the polynomial degree. The second through fourth columns present timings (in seconds) for a basic looping strategy, a loop-based implementation of the duality algorithm, and the duality algorithm utilizing `DGEMM`. The parenthetical number in the "BLAS" column indicates the number of gigaflops this timing represents. The final column indicates the time required to assemble the elementwise matrices into a sparse Epetra matrix.

k	basic	dual	BLAS (GFLOPs)	Total matvec	Epetra
1	1.79E-003	8.85E-004	2.61E-004 (1.75)	2.78E-004	3.22E-005
2	1.88E-002	8.58E-003	9.61E-004 (4.80)	1.02E-003	2.81E-004
3	1.04E-001	4.68E-002	3.21E-003 (7.84)	3.34E-003	1.11E-003

Table VI: Performance numbers for matrix-free application of the Laplace operator on a mesh of 1000 hexahedra. The first column (k) indicates the polynomial degree. The second through fourth columns give timings (in seconds) for a basic looping strategy, a C++ implementation of the duality-based algorithm, and the same optimized utilizing `DGEMM`. The parenthetical number in the "BLAS" column indicates the number of gigaflops this timing represents. The "Total matvec" column gives the time for elementwise BLAS-based calculations plus the time to scatter and gather into a global vector. The final column gives the time to apply the already-assembled global Epetra matrix onto a vector.

Γ_{ciqk} by optimized matrix multiplication gives a marked improvement in the overall run-time. As the polynomial degree increases, so does the FLOP rate. For tricubic elements, our 4.66 GFLOPs corresponds to about 43% of peak performance.

We also tested the three approaches to the action of the Laplacian. We showed earlier that the level 3 fraction for matrix actions is larger, and can expect higher flop rates as a result. In Table VI, the 7.84 GFLOPs for tri-cubic elements corresponds to just over 75% of peak performance. It is also interesting to note from this table that the time to scatter and gather the elementwise degrees of freedom is a small amount of the total computation and that by third degree, our matrix-free method is only three times slower than the Epetra matrix-vector product. When one observes that the cost of assembling the global sparse matrix costs about one hundred matrix-vector products, this is a notable result.

5.2. The curl-curl operator

We repeated the same experiments as above for the curl-curl operator. Since, for the same degree, the Nédélec space is vector-valued and larger than the H^1 space ($3k\,(k+1)^2$ versus $(k+1)^3$ degrees of freedom per element), this will correspond to larger matrix dimensions. Larger matrix dimensions typically lead to better performance for `DGEMM`, and we find this to be the case. In Table VII, we see flop rates of 1.35 GFLOPs for the lowest order case up to 6.62 GFLOPs (61% of peak) for the third-order case. In each case, we have achieved a substantial speedup over the loop-based implementation and reduced the elementwise construction time below that of global assembly.

k	basic loop	dual	BLAS (GFLOPs)	Assembly time
1	1.98E-002	8.77E-003	2.76E-003 (1.35)	5.60E-003
2	1.29E+000	4.20E-001	6.20E-002 (3.83)	1.79E-001
3	2.18E+001	6.76E+000	6.02E-001 (6.62)	1.30E+000

Table VII: Performance numbers for assembling the curl-curl operator with Nédélec elements on a mesh of 1000 hexahedra. The first column (k) indicates the order of the space. The second through fourth columns present timings (in seconds) for an analog of Algorithm 2, an pure C++ analog of Algorithm 1, and Algorithm 1 adapted to utilize `DGEMM`. The parenthetical number in the "BLAS" column indicates the number of gigaflops this timing represents. The final column indicates the time required to assemble the elementwise matrices into a sparse Epetra matrix.

k	basic	dual	BLAS (GFLOPs)	Total matvec	Epetra
1	2.99E-003	1.22E-003	2.54E-004 (2.55)	2.81E-004	9.70E-005
2	4.25E-002	1.66E-002	1.34E-003 (6.69)	1.47E-003	3.15E-003
3	2.71E-001	1.07E-001	5.60E-003 (9.97)	5.94E-003	2.22E-002

Table VIII: Performance numbers for matrix-free application of the curl-curl operator using Nedelec elements on a mesh of 1000 hexahedra. The first column (k) indicates the order of the space degree. The second through fourth columns give timings (in seconds) for a basic looping strategy, a loop-based implementation of the duality algorithm, and the duality algorithm utilizing `DGEMM`. The parenthetical number in the "BLAS" column indicates the number of gigaflops this timing represents. The "Total matvec" column gives the time for elementwise BLAS-based calculations plus the time to scatter and gather into a global vector. The final column gives the time to apply the already-assembled Epetra matrix onto a vector.

Our approach is also quite successful in optimizing the action of the curl-curl operator, as presented in Table VIII. Our optimized matrix-free implementation is only marginally slower than the Epetra matvec in the lowest-order case and is actually faster for second and third order. Also note that our third-order optimized matrix-vector product, running at 9.97 GFLOPs, amounts to 92% of single-thread peak performance.

5.3. A comparision to FEniCS

As another point of reference, we also compare our approach to FEniCS [Logg et al. 2012]. While the comparison is imperfect (e.g. FEniCS only supports simplicial meshes), seeing timings verses a known performant code can be helpful. While we use $10^3 = 1000$ hexahedra, we use the `UnitCubeMesh(10,10,10)` consisting of 6000 tetrahedra. While this gives more mesh cells, we have fewer degrees of freedom per cell on tetrahedra for both the Lagrange and Nédélec spaces. For example, trilinear hex elements have 8 basis functions per cell versus 4 linears per tetrahedron. Table IX summarizes the number of degrees freedom for each space of each degree on the two meshes under consideration. Also note that FEniCS' simplicial meshes only require one Jacobian per cell, while we store a Jacobian at each quadrature point on each cell.

FEniCS also provides two distinct modes of form evaluation, the tensor contraction mode described in [Kirby and Logg 2006] and extended to $H(\mathrm{curl})$ elements in [Rognes et al. 2009] and the quadrature mode developed in [Ølgaard and Wells 2010]. The tensor contraction mode precomputes integrals on the reference cell, thereby reducing the run-time cost per-element of form evaluation. The quadrature mode generates performant loops over quadrature points per-cell. The results in [Ølgaard and Wells 2010]

	Lagrange		Nédélec	
k	tet	hex	tet	hex
1	24000	8000	36000	12000
2	36000	27000	120000	54000
3	60000	64000	270000	144000

Table IX: Number of elementwise degrees of freedom for Lagrange and Nédélec spaces on meshes of 1000 hexes and 6000 tetrahedra.

Degree	Tensor mode	Quadrature mode	BLAS
1	4.67e-03	6.43e-03	4.56e-03
2	3.10e-02	9.12e-02	7.78e-02
3	1.62e-01	9.95e-01	5.73e-01

Table X: Comparison of assembling the Laplacian on the unit cube using FEniCS with 6000 tetrahedra versus our BLAS-based approach on 1000 hexahedra. The second and third columns indicate the time to assemble the matrix using code generated by the tensor and quadrature representations in FEniCS. The BLAS column adds together the times from the BLAS and Assembly columns of Table V.

Degree	Tensor mode	Quadrature mode	BLAS
1	2.47e-03	2.64e-03	2.78e-04
2	4.71e-03	8.01e-03	1.02e-03
3	2.39e-02	4.13e-02	3.34e-03

Table XI: Comparison of the action of the Laplacian on the unit cube using FEniCS with 6000 tetrahedra versus our BLAS-based approach on 1000 hexahedra. The second and third columns indicate the time to assemble the matrix using code generated by the tensor and quadrature representations in FEniCS. The BLAS column repeats the "Total matvec" timing in VI.

indicate that it loses to tensor mode for simple, constant-coefficient forms but scales much better with form complexity (including variable coefficients). We will compare our techniques to both the tensor- and quadrature-based representations.

FEniCS also supports the action of a variational form as well as assembly. For example, if `u` is a `Function` and `v` a `TestFunction`, then assembling `inner( grad(u) , grad(v) )*dx` computes the action (without boundary conditions) of the Laplace matrix on the vector of coefficients of `u`.

In Tables X and XII, we compare the assembly time for FEniCS to our approach. We report the total time required by `assemble` in FEniCS compared to the time for us to build A_{cij} and sum entries into global storage. Given that the numbers are comparable for a relatively comparable amount of work and given all the differences we have highlighted, it is probably wise not to draw subtle conclusions from these results.

On the other hand, Tables XI and XIII show a different story for the matrix-free actions of these two operators. Even at $k = 1$, our BLAS formulation wins by about an order of magnitude for the Laplacian. The gap grows with polynomial degree and is more significant for the curl-curl operator.

Degree	Tensor mode	Quadrature mode	BLAS
1	5.13e-02	3.26e-02	8.36e-03
2	2.12e+00	1.42e+00	2.41e-01
3	3.91e+01	2.77e+01	1.90e+00

Table XII: Comparison of assembling the curl-curl operator on the unit cube using FEniCS with 6000 tetrahedra versus our BLAS-based approach on 1000 hexahedra. The second and third columns indicate the time to assemble the matrix using code generated by the tensor and quadrature representations in FEniCS. The BLAS column adds together the times from the BLAS and Assembly columns of Table VII.

Degree	Tensor mode	Quadrature mode	BLAS
1	8.13e-02	5.87e-03	2.81e-04
2	2.56e+00	6.24e-02	1.47e-03
3	3.93e+01	5.12e-01	5.93e-03

Table XIII: Comparison of the action of the curl-curl operator on the unit cube using FEniCS with 6000 tetrahedra versus our BLAS-based approach on 1000 hexahedra. The second and third columns indicate the time to assemble the matrix using code generated by the tensor and quadrature representations in FEniCS. The BLAS column repeats the "Total matvec" timing in VIII.

6. CONCLUSIONS AND FUTURE WORK

Using commuting properties of pullbacks and duality, we have provided a generic mechanism by which finite element operators may be computed with very high performance and minimal assumptions on the particular basis functions, geometry, or coefficients of the problem at hand. By its separation of discrete tasks, our duality-based approach suggests certain useful, though not entirely standard, abstraction barriers. As we have seen, these barriers permit the replacement of FEM-specific for loops with appropriately optimized calls to `DGEMM`. Alternatively, these matrix multiplications could be replaced with alternative, problem-specific algorithms such as tensor-product decomposition of bases.

In this work, we have only considered elementwise variational forms, implicitly assuming natural boundary conditions. While boundary conditions represent a lower-order term with regard to the work estimate for typical conforming methods, a more significant fraction of the work in discontinuous Galerkin methods lies on handling internal boundary terms. Additional research will be required to understand whether the present techniques may be applied to such discretizations.

Finally, In the near future, we hope to extend this approach to utilize multiple CPU cores and also GPU boards. Also, the genericity of our approach suggests that it could be incorporated as an internal target for high-level general-purpose codes such as Sundance.

ACKNOWLEDGMENTS

REFERENCES

Mark Ainsworth, Gaelle Andriamaro, and Oleg Davydov. 2011. Bernstein-Bézier finite elements of arbitrary order and optimal assembly procedures. *SIAM Journal on Scientific Computing* 33, 6 (2011), 3087–3109.

Douglas N. Arnold and Ragnar Winther. 2002. Mixed finite elements for elasticity. *Numer. Math.* 92, 3 (2002), 401–419.

W. Bangerth, R. Hartmann, and G. Kanschat. 2007. deal.II — a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.* 33, 4 (2007), 1–27.

Wolfgang Bangerth and Guido Kanschat. 2013. Deal.II tutorial, "The step-3 tutorial program". http://www.dealii.org/8.0.0/doxygen/tutorial/index.html. (2013).

Pavel Bochev, H. Carter Edwards, Robert C. Kirby, Kara Peterson, and Denis Ridzal. 2012. Solving PDEs with Intrepid. *Scientific Programming* 20, 2 (2012), 151–180.

Pavel B. Bochev and Max D. Gunzburger. 2009. *Least-squares finite element methods*. Applied Mathematical Sciences, Vol. 166. Springer, New York. xxii+660 pages. DOI:http://dx.doi.org/10.1007/b13382

Michael A. Heroux, Roscoe A. Bartlett, Victoria E. Howle, Robert J. Hoekstra, Jonathon J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An overview of the Trilinos project. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 397–423.

Jan S. Hesthaven and Tim Warburton. 2008. *Nodal discontinuous Galerkin methods*. Texts in Applied Mathematics, Vol. 54. Springer, New York. xiv+500 pages. DOI:http://dx.doi.org/10.1007/978-0-387-72067-8 Algorithms, analysis, and applications.

J. J. Heys, T. A. Manteuffel, S. F. McCormick, and L. N. Olson. 2005. Algebraic multigrid for higher-order finite elements. *J. Comput. Phys.* 204, 2 (2005), 520–532. DOI:http://dx.doi.org/10.1016/j.jcp.2004.10.021

Robert C. Kirby. 2011. Fast simplicial finite element algorithms using Bernstein polynomials. *Numer. Math.* 117, 4 (2011), 631–652.

Robert C. Kirby, Matthew G. Knepley, Anders Logg, and L. Ridgway Scott. 2005. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.* 27, 3 (2005), 741–758 (electronic).

Robert C. Kirby and Anders Logg. 2006. A Compiler for Variational Forms. *ACM Trans. Math. Software* 32, 3 (2006), 417–444.

Robert C. Kirby, Anders Logg, L. Ridgway Scott, and Andy R. Terrel. 2006. Topological optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput.* 28, 1 (2006), 224–240 (electronic).

Anders Logg, Kent-Andre Mardal, and Garth N. Wells. 2012. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Vol. 84. Springer.

Kevin Long, Robert C. Kirby, and Bart van Bloemen Waanders. 2010. Unified embedded parallel finite element computations via software-based Fréchet differentiation. *SIAM J. Sci. Comput.* 32, 6 (2010), 3323–3351. DOI:http://dx.doi.org/10.1137/09076920X

J.-C. Nédélec. 1980. Mixed finite elements in $\mathbf{R}^3$. *Numer. Math.* 35, 3 (1980), 315–341.

K. B. Ølgaard and G. N. Wells. 2010. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Software* 37, 1 (2010). http://dx.doi.org/10.1145/1644001.1644009

OpenBlas 2013. OpenBLAS homepage. (2013). http://xianyi.github.com/OpenBLAS.

Steven A. Orszag. 1979. Spectral methods for problems in complex geometries. In *Numerical methods for partial differential equations (Proc. Adv. Sem., Math. Res. Center, Univ. Wisconsin, Madison, Wis., 1978)*. Publ. Math. Res. Center Univ. Wisconsin, Vol. 42. Academic Press, New York, 273–305.

Marie E. Rognes, Robert C. Kirby, and Anders Logg. 2009. Efficient assembly of H(div) and H(curl) conforming finite elements. *SIAM Journal on Scientific Computing* 31, 6 (2009), 4130–4151.

R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1–27.

Received ; revised ; accepted