Optimizing FIAT with Level 3 BLAS

ROBERT C. KIRBY[1]
The University of Chicago

The code FIAT (FInite element Automatic Tabulator) provides necessary abstractions to define and tabulate a wide range of different finite elements on a reference element. We show here how the performance of the critical operations in its algorithms may be greatly improved by representing polynomials and linear functionals internally as vectors and hence setting up dense matrix operations. The performance gains are up to three orders of magnitude in cases studied. We also discuss how dimensional independence can be obtained through a use of graded incidence relations and give some applications to Lagrange and Brezzi-Douglas-Marini elements.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: —*Algorithm Design, Efficiency*; G.1.8 [**Partial Differential Equations**]: Finite Element Methods—

General Terms: Algorithms,Performance

Additional Key Words and Phrases: finite element, numerical linear algebra, Hilbert space

1. INTRODUCTION

Previously, we presented a computational representation theory for finite element spaces based on the mathematical definitions of Ciarlet [Kirby 2004; Ciarlet 1978]. The abstractions used allowed us to implement a Python code capable of evaluating the nodal basis for arbitrary order instances of many different triangular finite elements, including those of Lagrange [Brenner and Scott 2002], Raviart-Thomas [Raviart and Thomas 1977], and Arnold-Winther [Arnold and Winther 2002]. The capability to employ a wide range of elements will become useful both for theoreticians studying new elements and methods and practitioners seeking to leverage the best theoretical work, especially as FIAT becomes fully merged with language/compiler projects for variational forms such as Sundance [Long 2003], PETSc [Laboratory], and FFC [Logg 2005].

In order to improve both the performance and the generality of FIAT, we have reinterpreted many aspects to pay more attention to granularity and mathematical structure. For one, the previous memoizing system encouraged programming at too low granularity. We have implemented a batch evaluation mode for the orthornormal polynomials and sought out ways to rephrase the calculation at this higher level. Also, we have exploited the Hilbert space structure of polynomials to implement integration and functional application in terms of Euclidean dot products. This has led to a much higher fraction of work being performed in the context of dense

matrix operations. We have observed up to a thousand-fold improvement in performance while using no additional C or Fortran code beyond Python's numerical and linear algebra extension modules, which rely upon the level 3 BLAS [Dongarra et al. 1990]

Since [Kirby 2004], the work of Bientinesi *et al* [Bientinesi et al. 2005] for dense linear algebra has come to our attention. They develop a formal structure for deriving linear algebra algorithms that allows generation of provably correct, high-performance code. One interpretation of this work is that it moves beyond an "enumerative" phase of scientific computating in which a set of algorithms is specified and manually implemented in a library to "grammatical" phase in which structures capable of generating the entire suite of such algorithms are exploited. Through FIAT and its integration into codes for variational forms, we hope to make an analagous contribution to numerical PDE. Rather than enumerating a list of finite elements and variational forms and laboriously implementing each, we are developing abstract structures covering an entire class of finite element methods.

In this paper, we first review the major mathematical results in [Kirby 2004], then present our techniques for representing polynomials, sets of polynomials, and linear functionals over polynomial spaces. We discuss both how our new code achieves dimensional abstraction and how we may formulate the construction of the Vandermonde and constraint matrices needed in [Kirby 2004] with level 3 BLAS. Then, we present two examples, the Lagrange and the Brezzi-Douglas-Marini [Brezzi et al. 1985] elements in light of our new framework and study the performance of the old and new codes. We believe this to be the first implementation of higher order BDM elements on tetrahedra. Finally, we conclude by discussing the ongoing and future work for FIAT and its implications for the automation of finite element computation.

2. FINITE ELEMENT DEFINITIONS

Before discussing our techniques for optimizing FIAT, we review the basic definitions of Ciarlet [Ciarlet 1978] and the linear algebraic framework presented in [Kirby 2004].

We recall that Ciarlet defines a finite element as a triple (K, P, N) where K is a bounded domain in $\mathbb{R}^d$ with a piecewise smooth boundary, P is a finite-dimensional function space over K, and $N = \{n_i\}_{i=1}^{|P|}$ is a basis for the dual space P'. Finite element computations rely on a *nodal* basis for P. That is, a basis $\{\psi_i\}_{i=1}^{|P|}$ such that $n_i(\psi_j) = \delta_{i,j}$ for each $1 \leq i, j \leq |P|$. For example, for the Lagrange elements, this nodal basis would consist of polynomials that are either one or zero at a collection of lattice points over some lattice on a triangle or tetrahedron. The nodes are chosen, among other things, to enforce appropriate continuity across adjacent elements.

In [Kirby 2004], we argued that the difficulty in constructing computable representations for nodal bases for general finite elements such as high-order Raviart-Thomas elements is a major factor limiting their adoption by practitioners. Hence, we proceeded to develop a linear algebraic framework for computing a wide class of such elements relying on the existence of nice recurrence relations for orthonormal polynomials on simplices. In particular, supposing that we had some basis $\{\phi_i\}_{i=1}^{|P|}$ for P and the ability to apply members N to that basis for P, we could construct

the nodal basis by forming the matrix $V \in \mathbb{R}^{|P|,|P|}$ such that

$$V_{i,j} = n_i(\phi_j). \tag{1}$$

We showed that the columns of the inverse of this matrix (a generalized Vandermonde matrix) are in fact the expansion coefficients for the nodal basis for (K, P, N). Thus, computation of the nodal basis is nothing more than computing the basis for which we have an implementation followed by some arithmetic.

In some cases, such as p-adaptivity of the Brezzi-Douglas-Fortin-Marini [Brezzi and Fortin 1991] or Arnold-Winther [Arnold and Winther 2002] elements, the polynomial space does not coincide exactly with polynomials of some total degree. Hence, the standard orthonormal expansions do not exactly span the space. In this situation, we set the polynomial space inside of some larger space $\bar{P}$ and described P as the intersection of the null spaces of several functionals acting on $\bar{P}$. For example, we showed how the BDFM elements of degree k on a triangle can specified as the subspace of $(P_k)^2$ that the normal components integrated against the k^{th} Legendre polynomial on each edge vanish. Given a set of constraint functionals $\{\ell_i\}_{i=1}^m$ and a basis $\{\bar{\phi}_i\}_{i=1}^{|P|}$ for $\bar{P}$, we constructed the matrix L with

$$L_{i,j} = \ell_i(\bar{\phi}_j), \tag{2}$$

and hence constructed a basis for P using the null space of the matrix, computed with the singular value decomposition.

For both Vandermonde matrices and constraint matrices, we are required to construct a "functional outer product" that is a collection of functionals applied to a collection of polynomials. In the former implementation, the cost of building such matrices could be considerable. In the following section, we show how a new interpretation of polynomials and functionals leads to a much more streamlined implementation.

3. REPRESENTING AND COMPUTING POLYNOMIALS

3.1 Bulk evaluation of orthonormal polynomials

Even with memoization, repeated evaluation of the orthornormal polynomials created a considerable bottleneck in [Kirby 2004]. We have increased the granularity of evaluation, yielding substantial performance gains.

Consider first one space dimension and the Jacobi polynomials. Using recurrence relations to calculate $P_n^{a,b}$, we must calculate all of the preceding Jacobi polynomials $\{P_i^{a,b}\}_{i=1}^{n-1}$. If the recurrence relations are written as a loop rather than recursively, then memoization will not detect these redundancies if we call $P_i^{a,b}$ with several different values of i.

Now, consider the two-dimensional orthogonal polynomials over the triangle due to Dubiner [Dubiner 1991]. These polynomials are L^2-orthogonal and can be ordered hierarchically. They are defined by a mapping from the reference triangle $(-1,-1),(1,-1),(-1,1)$ to the reference square with vertices at ± 1. Away from the top vertex, this change of coordinates from the triangle to the rectangle is given

by

$$\begin{pmatrix} \xi \\ \eta \end{pmatrix} = \begin{pmatrix} 2\left(\frac{1+x}{1-y}\right) - 1 \\ y \end{pmatrix} \tag{3}$$

For a degree n, the polynomials are defined by

$$\phi_{i,j}(x,y) = P_i^{0,0}(\xi)\left(\frac{1-\eta}{2}\right)^i P_j^{2i+1,0}(\eta), \tag{4}$$

where $0 \leq i,j \leq n$ and $0 \leq i+j \leq n$. In order to tabulate all of the polynomials of degree n, we must run through the Jacobi recurrence relations for $P_i^{0,0}$ for $0 \leq i \leq n$, and also $P_j^{2i+1,0}$ for all $0 \leq j \leq n$ and $0 \leq i \leq j$. In our new implementation, we have been careful to make calls to tabulate all the Jacobi polynomials with particular weights that we need, then incorporate them into the final result. Similar (but more detailed) considerations apply in three dimensions.

Furthermore, looping through the recurrence relations requires computing the appropriate weights for each stage. We have vectorized the tabulation procedure so as to run through the recurrence relations a single time for all points. Hence, in the following, we shall rely the existence of routines for lines, triangles, and tetrahedra [Karniadakis and Sherwin 1999] that efficiently calculate all of the appropriate orthonormal bases together, making use of vectorized operations and not repeating recurrence relations.

3.2 Computing with polynomials

All polynomials in FIAT are represented as linear combinations of orthonormal polynomials; this representation is particularly amenable to setting up dense linear algebra. In this section, we show how tabulating a set of polynomials is naturally matrix multiplication, even if the polynomials are vector-valued. Then, we argue from basic functional analysis that application of linear functionals and hence construction of functional outer products in FIAT may also be done by matrix multiplication.

Before we begin, we make two preliminary points. First, we throughout will use the standard summation convention for tensors with repeated indices denoting summation. Second, we will always consider that our arrays are stored contiguously in row-major format. This is the C-style layout and is adopted by the Numerical Python module underlying the implementation of FIAT.

3.2.1 *Tabulation.* First, we let P be a set of polynomials of degree n over a reference simplex (line, triangle, tetrahedron) K in $d-$dimensional space and let $\{\phi_i\}_{i=1}^{|P|}$ be an orthonormal basis. Here, $|P|$ is the dimension of the vector space of polynomials. For any $p \in P$, we let $\mathcal{R}$ be the mapping from p to its vector of expansion coefficients in the orthonormal basis. That is, $p = \mathcal{R}(p)_i\phi_i$.

If we want to evaluate a polynomial p at a point x, we then construct the vector $\Phi_i = \phi_i(x)$ and compute the dot product

$$\mathcal{R}(p)_i\Phi_i. \tag{5}$$

Now, suppose that have a set of polynomials $S = \{p_i\}_{i=1}^N$, such as the nodal basis of a finite element set. To this set, we assocate a matrix C such that $C_{i,j} =$

$\mathcal{R}(p_i)_j$. That is, the rows of C store the expansion coefficients of the members of S. Tabulating S at a collection of points $X = \{x_i\}_{i=1}^M$, such as a set of quadrature points, is accomplished by matrix multiplication. We compute the matrix $\Phi_{i,j} = \phi_i(x_j)$ and hence

$$p_i(x_j) = C_{i,k}\Phi_{k,j}. \tag{6}$$

The same methodology may be extended to tabulate sets of vector-valued polynomials by matrix-multiplication with no excess data motion, provided that we store our coefficients properly. Now, we consider a vector-valued polynomial $v \in (P)^m$ with m components, each of which is a polynomial in P. We may represent v as a two-dimensional array $\mathcal{R}(v)_{i,j}$ with $v^i = \mathcal{R}(v)_{i,j}\phi_j$ denoting the i^{th} component of the vector. This corresponds to writing the vector as a linear combination of vectors in $\mathbb{R}^m$ times the expansion coefficients. Rows of the matrix $\mathcal{R}_{i,j}$ are the representations of the scalar-valued polynomials in each component. Perhaps it is more suggestive to write $v = \mathcal{R}(v)_{i,j}\delta_{i,k}e^k\phi_j$, in which we take the i^{th} row times a set of vector-valued functions that only have support in component i. This interpretation will be helpful when we consider linear functionals on vector-valued polynomials.

Now, suppose we have a set $S = \{v_i\}_{i=1}^N$ of vector-valued polynomials. We can associate with this set a rank three tensor of coefficients $C_{i,j,k}$ such that C_i is the matrix of coefficients obtained by applying $\mathcal{R}$ to v_i. That is, $C_{i,j,k} = \mathcal{R}(v_i)_{j,k}$. As before, we build the matrix $\Phi_{i,j} = \phi_i(x_j)$ of the orthornomal basis functions at each point. Then, it is simple to see that

$$V_{i,j,k} = v_i(x_k)^j = C_{i,j,l}\Phi_{l,k}. \tag{7}$$

While this involves a higher rank tensor, this expression may naturally be recast as a matrix multiplication. To see this, we introduce the matrix

$$\tilde{C}_{\tilde{i},k} = C_{i,j,k}, \tag{8}$$

where $\tilde{i} = id + j$. The mapping $\tilde{\cdot}$ is an isomorphism between $\mathbb{R}^{N,d,|P|}$ and $\mathbb{R}^{Nd,|P|}$ and can be applied "in-place" by simply changing the way in which a block of memory is indexed. The `Numeric` module of Python supports this via the `reshape` function.

So, we compute the matrix multiplication

$$\tilde{V}_{i,k} = \tilde{C}_{i,l}\Phi_{l,k}, \tag{9}$$

and hence $V_{i,j,k}$ by the inverse of $\tilde{\cdot}$ (again, in place).

This same technique applies to general tensor-valued polynomials, as we can always re-index the tensors to be rank one-dimensional arrays with no data motion by an extension of the operation $\tilde{\cdot}$. Moreover, symmetric tensors (needed for the Arnold-Winther elements [Arnold and Winther 2002]) may be represented as some higher-level object (such as symmetric indexing into a linear array); as long as we are dealing with objects that are isomorphic to vectors, we can use this approach.

3.2.2 *Linear functionals.* With the L^2 inner product, all of our polynomial spaces are naturally considered as Hilbert spaces. Our representation in terms of orthonormal bases means that the L^2 inner product may be computed by a simple

Euclidean inner product (this is really just Parseval's relation $(p,q) = \mathcal{R}(p)_i \mathcal{R}(q)_i$). As with tabulating polynomials, it is possible to scale up dot products to matrix multiplication in building Vandermonde and constraint matrices needed throughout FIAT. After developing this idea for scalar-valued polynomials, we show how similar techniques used for tabulating vector-valued polynomials may be used for functionals and outer products.

We first develop an abstraction for Banach spaces and then return to the Hilbert space context. For a finite dimensional Banach space V with basis $\{\phi_i\}_{i=1}^{|V|}$, we may represent all members of its dual V' by vectors. As before, let $\mathcal{R}$ be the mapping from a member of V to its expansion coefficients in the basis. Then, if $\ell \in V'$, we may apply ℓ to some $v \in V$ by

$$\ell(v) = \ell(\mathcal{R}(v)_i \phi_i) = \mathcal{R}(v)_i \ell(\phi_i). \tag{10}$$

This motivates the introduction of a mapping $\mathcal{R}' : V' \to \mathbb{R}^{|P|}$ by $\mathcal{R}(\ell)_i = \ell(\phi_i)$, for then

$$\ell(v) = \mathcal{R}'(\ell)_i \mathcal{R}(v)_i, \tag{11}$$

which is just a dot product.

While this abstraction is sufficient to drive our computation, considering the Hilbert space context adds some additional insight. Finite elements often use integration against some set of polynomials as nodes (hence requiring an inner product). Let P be our polynomial space with basis $\{\phi_i\}_{i=1}^{|P|}$, and let $f \in P'$ be given by $f(p) = (p,q)$ for some fixed $q \in P$. Then, $f(p) = q_i p_i$, so that $\mathcal{R}'(f)_i = q_i$, but also $\mathcal{R}(q) = \mathcal{R}'(f)$ in the sense of $\mathbb{R}^{|P|}$. This is just what we expect from the Riesz Representation Theorem; the Hilbert space and its dual are isomorphic, so our mapping from either the space or its dual into Euclidean space give the same vector.

In FIAT, we model linear functionals with a class `Functional` that takes a polynomial space and a vector and can operate on any member of that polynomial space by computing a dot product. In order to implement new kinds of functionals, we must determine what they do to a given polynomial basis. This is typically implemented in a few lines of code. Moreover, we can model a set of linear functionals $\{\ell_i\}_{i=1}^{N}$ over a scalar-valued space of polynomials by including a reference to the polynomial space and a matrix $L_{i,j} = \mathcal{R}'(\ell_i)_j$.

This representation of polynomials and linear functionals over them allows us to phrase building functional outer products as matrix multiplication. For either Vandermonde or constraint matrices, we have a collection of linear functionals $\{\ell_i\}_{i=1}^{M}$ and a collection of polynomials $\{p_i\}_{i=1}^{N}$ and we must build a matrix $A_{i,j} = \ell_i(p_j)$. Once we have matrix $L_{i,j}$ and the coefficient matrix $C_{i,j}$, formation of A is straightforward, as

$$A_{i,j} = L_{i,k} C_{j,k}, \tag{12}$$

or $A = LC^t$, motivating the description of the operation as an outer product.

We implement functionals on vector-valued spaces with this approach as well. Just as before, we will consider P a space of polynomials of degree n with basis $\{\phi_i\}_{i=1}^{|P|}$ and to each $v \in (P)^m$ associate the tensor representation $v^i = \mathcal{R}(v)_{i,j} \phi_j$. For any functional $f \in ((P)^m)'$, we will specify a tensor representation $\mathcal{R}'(f)_{i,j}$ such

that $f(v) = \mathcal{R}'(f)_{i,j}\mathcal{R}(v)_{i,j}$ is computed by a contraction of two $m \times |P|$ tensors. We let e^j be the canonical basis vector such that $(e^j)_i = \delta_{i,j}$. To determine the representation $\mathcal{R}'(f)$, we write

$$\begin{aligned} f(v) &= f(\mathcal{R}_{i,j}\phi_j) \\ &= f(\mathcal{R}_{i,j}\delta_{i,k}e^k\phi_j) \\ &= f(\mathcal{R}_{i,j}e^i\phi_j) \\ &= \mathcal{R}_{i,j}f(e^i\phi_j), \end{aligned} \tag{13}$$

so that $\mathcal{R}'(f)_{i,j} = f(e^i\phi_j)$.

Now, we take $\{v_i\}_{i=1}^N$ be a set of vector-valued polynomials with coefficient tensor $C_{i,j,k}$ and let $\{\ell_i\}_{i=1}^M$ be a set of linear functionals. We may store all of the tensor representations in a rank three tensor $L_{i,j,k}$ such that $L_{i,j,k} = \mathcal{R}'(\ell_i)_{j,k}$. Application of the set of functionals to each polynomial entails a tensor contraction, and we must compute the matrix

$$A_{i,j} = L_{i,k,l}C_{j,k,l}. \tag{14}$$

This inolves contraction over the last two dimensions. However, we may reshape both tensors by an operator similar to $\tilde{\cdot}$. We let $\hat{L}_{i,\hat{j}} = L_{i,j,k}$ and $\hat{C}_{i,\hat{j}} = C_{i,j,k}$ with $\hat{j} = j|P| + k$. Then, our contraction becomes

$$A_{i,j} = \hat{L}_{i,k}\hat{C}_{j,k}. \tag{15}$$

As before, this requires reshaping the tensors by changing the indexing strategy, but no data rearrangement.

4. TOPOLOGICAL AND GEOMETRIC INFORMATION

Several kinds of topological and geometric information are necessary to specify point locations for functionals and what degrees of freedom are associate with vertices, edges, and so on. It is interesting that this can be presented to the programmer in a dimensionally-independent way with a very small programmer interface.

We make use of a "graded incidence relation" being developed by Knepley and Karpeev [Knepley and Karpeev] for presenting finite element meshes to the programmer. In this, we associate to each entity in the reference element a dimension and a number within that dimension. The vertices have dimension zero, edges have dimension one, triangles dimension two, and tetrahedra dimension three. For each reference domain, we have a particular ordering of the entities for each dimension.

This basic topological idea provides us with a formalism for associating particular finite element nodes with reference element entities. When building the mapping between local and global degrees of freedom, it is essential to know, for example, that linear Lagrange elements have nodes associated with vertices, as these must be shared by all triangles in a patch. In FIAT, we associate with our class for dual bases a nested associative array `entity_ids[d][e]` that maps the entity of number `e` of topological dimension `d` to the list of integers indicating the nodes asssociated with that entity. For quadratic Lagrange elements over triangles, if `d` is `2` and `e` is `1`, we are refering to the edge of the reference domain with number 2. There is one degree of freedom associated with each edge, and so we will have a list of one

node. We remark that a similar strategy is employed for automating connectivity information in Sundance [Long 2003].

Moreover, thinking about the mesh entities for the reference element in this way allows us a general mechanism for determining locations of sets of points. For example, the nodes for cubic Lagrange elements on a triangle are defined using nodal locations on the lattice of degree three (with ten points). In order to build our connectivity information, it is useful to order the vertex nodes first, followed by the edge nodes and then the internal node. Rather than enumerating the points in a lattice and then taking the lattice apart, we provide a function `make_points(shape,d,e,deg)` that returns the list of points on a lattice of degree `deg` that are associated with a particular topological entity of a reference element. As in the `entity_ids` array, the parameters `d` and `e` identify the entity of the reference element by its topological dimension and number within the set of entities of that dimension. This is a powerful abstraction for defining dual bases for finite elements dimensionally independently; we shall return to this point later, using the Lagrange elements as a simple example, but also discussing the BDM elements as well.

5. EXAMPLES

Here, we describe what is involved with implementing the Lagrange and BDM elements of general degree over both triangles and tetrahedra. This makes use of the `points` and `shapes` modules in FIAT, as well as batch evaluation of orthonormal polynomials. After describing the implementation of both elements, we present some timing results indicating the vast performance gains the new framework provides.

5.1 Lagrange Elements

First, we consider the Lagrange elements over the reference simplex in two and three space dimensions. For degree n, our function space is simply P_n, and we must specify the functionals.

We showed the vector representation of pointwise evaluation in (5); the coefficients are merely the values of the orthonormal basis functions at the given point. Since all the nodes in the dual basis are pointwise evaluation, it makes sense to do all of the evaluation with a single tabulation. We can then read off the rows of the resulting matrix and construct linear functionals for each one. Since this is a common operation, we have provided a function `make_point_evaluations` that takes a basis and a list of points, and returns the list of functionals corresponding to evaluating the members of the basis at each of the points.

FIAT includes a short snippet (eight lines) that computes the `entity_ids` for any degree on all simplices. Here, we give an example of the `entity_ids` associative array for cubic Lagrange on triangles. We used the `make_points` function to generate the appropriate points on each entity.

```
{ 0: { 0: [ 0 ] ,
       1: [ 1 ] ,
       2: [ 2 ] } ,
  1: { 0: [ 3, 4 ] ,
```

```
      1: [ 5, 6 ] ,
      2: [ 7, 8 ] } ,
2: { 0: [ 9 ] } }
```

5.2 BDM Elements

$H(\mathrm{div})$-conforming elements are frequently discussed when locally conservative formulations of elliptic problems are desired, such as porous media. Given the difficulty of constructing bases for $H(\mathrm{div})$ elements, most implementations of mixed methods for Poisson-type problems use only the lowest order Raviart-Thomas elements.

The BDM elements of order k over the $d-$dimensional simplex use the function space $(P_k)^d$. The nodes consist of specifying the normal component on the $(d-1)$-dimensional mesh entities (edges on triangles, faces on tetrahedra), integral moments against gradients of polynomials of degree $k-1$, and integral moments against vector polynomial of degree k that are divergence-free and have vanishing normal components on the boundary.

The graded incidence relation used by `make_points` helps us to specify the appropriate sets of points on the boundary abstractly. Consider just the lowest order case of linear elements. On triangles, we need two points on the interior of each edge to specify the normal component of linears. Taking two points per edge means taking the internal edge points of a lattice of degree three. We can compute these points on edge `e` by `make_points(TRIANGLE,1,e,3)` On tetrahedra, we must specify three points per face. The three internal nodes lie on a lattice of degree four, which we may compute for face `f` by `make_points(TETRAHEDRON,2,f,4)` A simple calculation can show that the single call `make_points(shape,d-1,i,d+k)` will generate the appropriate points on boundary entity `i`, so that we can write one function (without a branch) to generate the nodes for both triangular and tetrahedral elements.

As in [Brezzi and Fortin 1991] and denote the space of divergence-free polynomials of degree k with vanishing normal component by Φ_k. In two space dimensions, the space Φ_k has a closed-form representation as $\{\mathrm{curl}(pb) : p \in P_{k-2}\}$, where b represents a cubic function that vanishes along the triangle boundary. While no such formulation exists in three dimensions, we construct a basis for this space in FIAT by means of constraint functionals, as more carefully described in [Kirby 2004]. In that work, we typically used constraint functionals to build bases for element spaces, but here we use them to build a function space needed for the dual space.

To say that a vector-valued polynomial of degree k has zero divergence, we may say that the integral of its divergence against each member of P_{k-1} vanishes. Hence, we construct a set of linear functionals that specify $(\nabla \cdot v, p)$ for p in some basis for P_{k-1}. Additionally, we construct functionals evaluating the normal component at appropriate points on each edge/face of the element boundary. The vector representations of these functionals on $(P_k)^d$ give a matrix of which we compute the null space and hence a basis for Φ_k. Relying on our mechanism for dimensional abstraction and the modules for constructing constrained spaces and functionals, specifying Φ_k in FIAT takes about twenty lines of code, which works for any degree polynomial over triangles and tetrahedra.

Developing the `entity_id` dictionary for BDM elements is fairly straightforward.

In this case, all degrees of freedom are associated with the boundary or the interior - entities of dimension d or $d-1$, as there are no vertex degrees of freedom for triangles or tetrahedra or edge degrees of freedom for tetrahedra.

5.3 Timing results

First, we consider the costs of instantiating (calling the class constructor) for the Lagrange and BDM elements in the old implementation provided in [Kirby 2004] and in our new implementation. We compare triangular Lagrange and BDM elements in the new and old systems, and also include both element families on tetrahedra in the new system. All computations were run on a Macintosh G4 Powerbook with a 1GHz processor and 1.25 GB of memory.

The time in seconds to create the element (including building the dual basis, Vandermonde matrix, and inverting it), is plotted in log-scale against the polynomial degree in Figure 1. Notice that asymptotically, instantiating triangular Lagrange elements is about ten times faster in the new system than the old. BDM elements give more striking speedups, with over two orders of magnitude improvement. Notice that for higher degree, the new system allows instantiation of tetrahedral BDM elements in a tenth of the time that the old system took for the triangular case. These vast speedups reflect not only vastly improved algorithmics but also open the possibility of using FIAT in an online rather than offline mode.

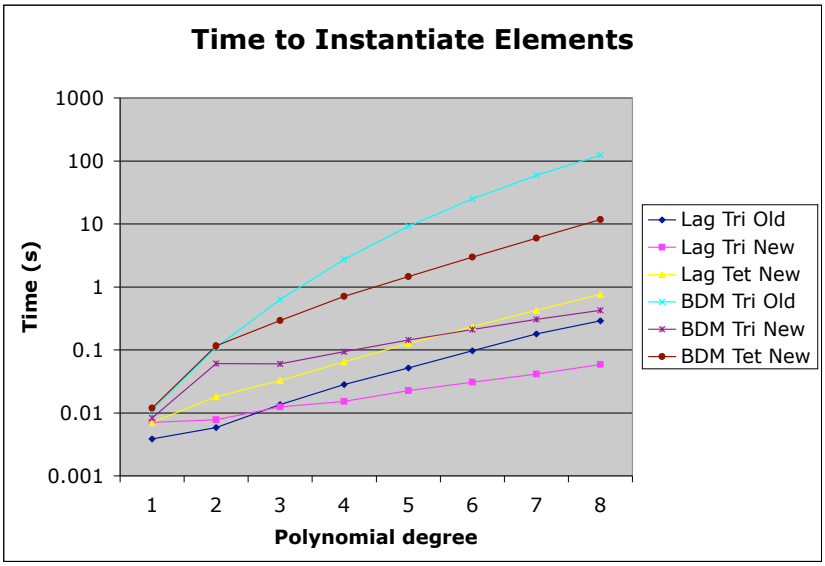
The performance gains in tabulating the nodal bases for these elements is also significant. We consider the cost of tabulating the nodal basis functions at a collection of quadrature points. For degree d, we tabulate all the nodal basis functions at the rule obtained by mapping Gauss-Jacobi quadrature rules with d points per direction on the reference square/cube to the reference triangle/tetrahedron. As before, we consider triangular elements of both families in both systems, and also include tetrahedral versions in the new family. Note that tetrahedral elements require tabulation at a much larger number of points than triangular elements. We plot the time in seconds to perform the tabulation in log-scale against the polynomial degree in Figure 2. We remark that the time differential between BDM and Lagrange elements for triangles is negligible (perhaps the tabulation of the orthonormal basis dominates the rather large matrix multiplication). For higher degree BDM elements, the new system allows tabulation about a thousand times faster than the old. Even high degree tetrahedral elements take less than a second to tabulate.

6. CONCLUSIONS AND FUTURE WORK

We have seen that the mathematical structure of finite element spaces lends itself to an abstract linear algebraic formulation. Moreover, this formulation can be effectively expressed as matrix multiplication, inversion, and decomposition, leading to high-performance implementation as evidenced by timing comparisons with the old version described in [Kirby 2004].

Current work is focused on using FIAT as a back-end for systems which automate the evaluation of variational forms over finite element spaces. In the case of FFC [Logg 2005], FIAT is being called at run-time to produce code for evaluating local stiffness matrices. As FFC is written in Python, this integration is simple. In the case of Sundance [Long 2003], we are developing C++ bindings to FIAT so

Fig. 1. Time to instantiate various elements in old and new versions of FIAT.



that it may be called at run-time. Our experimental results in the previous section indicate that FIAT will not create a new bottleneck in Sundance.

The pressing question that is raised by this integration is how to go from reference elements to physical elements and hence an assembled global space. For Lagrange elements, this is not terribly difficult, but more general elements present additional considerations. For example, the normal component degrees of freedom for $H(\mathrm{div})$ are always specified as outward on the reference element, but a sign convention must be adopted in the code so that a unique normal is used for each boundary. Also, Hermite and Arnold-Winther elements only map to interpolation-equivalent elements. After mapping, we must apply some transformation to acquire the nodal basis on each element. In both these situations, we are considering how to use FIAT to generate the appropriate assembly algorithms or at least specify type information regarding the nodes and how they couple or require special treatment.

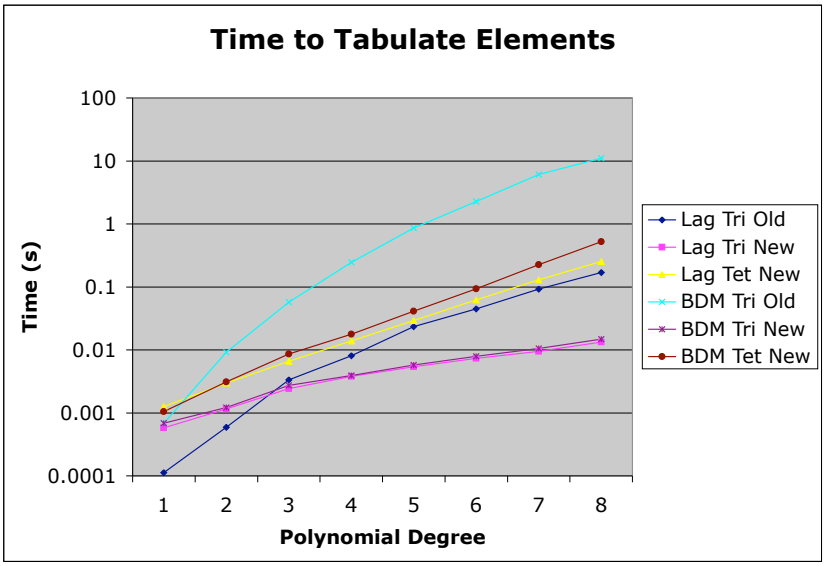
REFERENCES

Arnold, D. N. and Winther, R. 2002. Mixed finite elements for elasticity. *Numer. Math. 92,* 3, 401–419.

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software 31,* 1 (Mar.), 1–26.

Brenner, S. C. and Scott, L. R. 2002. *The Mathematical Theory of Finite Element Methods, 2nd Edition.* Springer-Verlag.

Fig. 2. Time to tabulate various elements at quadrature points in old and new versions of FIAT.



Brezzi, F., Douglas, Jr., J., and Marini, L. D. 1985. Two families of mixed finite elements for second order elliptic problems. *Numer. Math. 47,* 2, 217–235.

Brezzi, F. and Fortin, M. 1991. *Mixed and hybrid finite element methods*. Springer Series in Computational Mathematics, vol. 15. Springer-Verlag, New York.

Ciarlet, P. G. 1978. *The finite element method for elliptic problems*. North-Holland.

Dongarra, J. J., Croz, J. D., Duff, I. S., and Hammarling, S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16*.

Dubiner, M. 1991. Spectral methods on triangles and other domains. *J. Sci. Comput. 6,* 4, 345–390.

Karniadakis, G. E. and Sherwin, S. J. 1999. *Spectral/hp element methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, New York.

Kirby, R. C. 2004. Algorithm 839:FIAT, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Software 30*, 502–516.

Knepley, M. and Karpeev, D. A flexible representation for computational meshes. in preparation.

Laboratory, A. N. Petsc 3 web page. `http://www-unix.mcs.anl.gov/petsc/petsc-3/`.

Logg, A. 2005. FFC: the FEniCS Form Compiler. `http://www.fenics.org/ffc`.

Long, K. 2003. Sundance, a rapid prototyping tool for parallel PDE-constrained optimization. In *Large-Scale PDE-Constrained Optimization*. Lecture notes in computational science and engineering. Springer-Verlag.

Raviart, P.-A. and Thomas, J. M. 1977. A mixed finite element method for 2nd order elliptic problems. In *Mathematical aspects of finite element methods (Proc. Conf., Consiglio Naz. delle Ricerche (C.N.R.), Rome, 1975)*. Springer, Berlin, 292–315. Lecture Notes in Math., Vol. 606.