Algorithm 839: FIAT, A New Paradigm for Computing Finite Element Basis Functions

ROBERT C. KIRBY
The University of Chicago

Much of finite element computation is constrained by the difficulty of evaluating high-order nodal basis functions. While most codes rely on explicit formulae for these basis functions, we present a new approach that allows us to construct a general class of finite element basis functions from orthonormal polynomials and evaluate and differentiate them at any points. This approach relies on fundamental ideas from linear algebra and is implemented in Python using several object-oriented and functional programming techniques.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite element methods*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; D.1.5 [**Programming Techniques**]: Object-oriented Programming; G.4 [**Mathematical Software**]

General Terms: Algorithms, Languages, Reliability

Additional Key Words and Phrases: Finite elements, high order methods, linear algebra, Python

1. INTRODUCTION

In many ways, finite element codes fail to realize the generality afforded by the mathematical framework. One particularly notable shortcoming is the lack of implementations of many finite element spaces. For example, high order implementations of Nedelec's elements for electromagnetics are rare [Nédélec 1980], and implementations beyond the lowest order of Raviart-Thomas [Raviart and Thomas 1977a] elements are almost never found. A code allowing us to compute these interesting elements and explore their practical properties could open up many new discussions in the finite element community as well as possibilities for more effective applications.

Perhaps the lack of high order implementations is because finite element codes typically rely on explicit formulae for the basis functions. This difficulty can play out as follows: The programmer has a method and element in mind.

This work was supported in part by the Climate Systems Center at the University of Chicago under National Science Foundation Information Technology Research Program grant ATM-0121028.
Author's adddress: The University of Chicago, Department of Computer Science; 1100 E. 58th Street; Chicago, IL 60637; email: Kirby@cs.uchicago.edu.

For example, he may be using Galerkin's method for the Poisson equation with piecewise quadratic elements. Formulae for these functions on triangles or rectangles can be found in many standard finite element references [Zienkiewicz 1971]. The implicit assumption seems to be that these formulae will be copied into Fortran and evaluated at quadrature points to assemble the stiffness matrix. Many programmers will recognize that it is really no more difficult to write the loops for the stiffness matrix assembly so that a single routine will work for any basis functions of any orders of approximation, with the basis values, dimensions, and quadrature weights given as inputs. Then, separate routines evaluate linears, quadratics, and cubics at the quadrature points to pass into the matrix assembly. Still even very sophisticated codes that automate matrix assembly such as Dolfin [Logg and Hoffman 2002] and Deal [Bangerth and Kanschat 1999] are limited to the use of particular formulae for the basis functions. For elements with complicated degrees of freedom, such as Nedelec elements for electromagnetics [Nédélec 1980] or Raviart-Thomas elements for mixed methods for scalar elliptic equations [Raviart and Thomas 1977a], obtaining explicit formulae for the basis functions for these elements beyond the lowest order can be extremely tedious. Hence, with some notable exceptions [Rachowicz and Demkowicz 2002], there are very few high-order approximations using these more complicated elements.

One way to avoid these difficulties is to seek alternative formulations using simpler approximating spaces, such as discontinuous piecewise polynomials. For example, this is a large part of the interest in the porous media community in discontinuous Galerkin methods as a possible high-order alternative to mixed finite elements [Rivière and Wheeler 2000; Arnold et al. 2002]. However, these methods often have tradeoffs, such as lower convergence rates, nonsymmetric linear systems, or poor conditioning. As another example, Arnold and Winther [2002] have recently developed an element for the stress-displacement formulation of linear elasticity that is based on noncomposite polynomials, gives exactly symmetric stress tensors, and has optimal convergence rates, along with a very complicated approximating space with similarly complicated internal degrees of freedom. In many cases, these methods with more complicated approximating spaces have excellent properties, although it is not at all clear how one can leverage their power in practice.

The gap between theory and practice remains—the "mathematically powerful" elements remain largely theoretical, and practitioners usually make do with low order elements or seek alternative methods. We take a large step toward closing this gap here by presenting a new paradigm and associated model implementation that allows us to tabulate a very wide range of finite element basis functions on the reference triangle. We reinterpret the mathematical description of finite element families as a computational machine, applying some ideas from computer science as needed.

FIAT (Finite element automatic tabulator) computes general nodal basis functions as linear combinations of orthogonal polynomials. Since we have recurrence relations that allow us to stably evaluate these polynomials to very high order, and rules describing the degrees of freedom for the families of finite elements, we are able to create an effective code for tabulating arbitrary

order basis functions for nearly arbitrary finite elements. The implication is that there are no "hard" elements.

We start by developing the mathematical paradigm FIAT implements. This includes a mathematical definition of a finite element, and techniques for generating so-called prime bases for the function spaces. After this, we turn to questions of computation, describing the features of Python used in producing the implementation. While some of these details may be new to readers more familiar with Fortran or even C++, they help to make an effective, clean implementation of the ideas. After this, we show several examples of so-called "difficult" finite elements by tabulating some of the nodal basis functions with FIAT and plotting the results. Finally, we conclude by describing the potential impacts on scientific computing of this new technique.

2. MATHEMATICAL FORMULATION

While it is typically difficult to obtain explicit formulae for finite element basis functions, they may be otherwise represented through linear algebra. There are rules for stably evaluating orthogonal polynomial bases up to very high degree. Moreover, the rules for defining degrees of freedom for a family of finite elements give us exactly the information we need to build the nodal basis functions from the orthogonal polynomials.

2.1 Definition of a Finite Element

Ciarlet [1978] defines a finite element as a triple (K, P, N) such that

—K is a bounded domain in $\mathbf{R}^n$ with a piecewise smooth boundary,
—$P \subset C\left(\bar{K}\right)$ is a finite-dimensional function space over K, which may be vector-valued,
—$N = \{n_i\}_{i=1}^{\dim P}$ is a basis for the dual space P', called the set of *nodes*.

Practically speaking, finite element codes use domains K that are triangles or quadrilaterals in two dimensions and tetrahedral, prismatic, hexahedral, and so on in three dimensions. For now, FIAT does function spaces only on triangles, but as there is no conceptual difference, the code can readily be extended to other shapes. Most finite element spaces P consist of polynomials, though many interesting spaces are more complicated than just P_k. Generally, the spaces and basis functions are defined on some reference domain and mapped to each region in a mesh by an appropriate change of variables. The nodes N may include pointwise evaluation, pointwise differentiation, integration against other functions, and other functionals.

The *nodal basis* for the finite element (K, P, N) is the (unique) set $\{\psi_i\}_{i=1}^{\dim P}$ spanning P such that $n_i(\psi_j) = \delta_{i,j}$ for all $1 \leq i, j \leq \dim P$. The degrees of freedom of this basis are chosen such that, among other things, basis functions on adjacent elements can be patched together with the right continuity requirements.

As an example, we consider the quadratric Lagrangian element on the triangle K with vertices $(-1, -1), (1, -1), (-1, 1)$. In this case, the function space

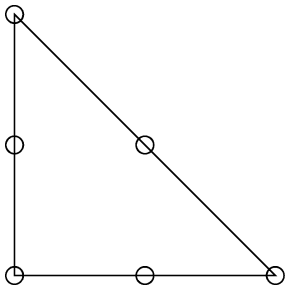

Fig. 1. Nodes for quadratic Lagrangian element.

is $P = P_k(K)$, and the nodes N consist of pointwise evaluation at the points shown in Figure 1.

More generally, *families* of finite elements $\{(K, P^k, N^k)\}_{k>k_0}$ are defined. In this case, we have a definition for each space (e.g. polynomials of total degree k) and a rule for the nodes (e.g. pointwise evaluation at the lattice of points of order k on K). As we shall see later, both the function spaces and the nodes may not be so simple.

2.2 Computing Nodal Bases from Prime Bases

In general, we do not know how to evaluate the nodal basis functions, but suppose that we can evaluate some other basis for the same space P. We denote this basis $\{\phi_i\}_{i=1}^{\dim P}$ and refer to it as the *prime basis*. When P is simply polynomials (or vectors or tensors thereof) of some degree k, we may use the basic orthogonal basis described for triangles in the following subsection as our prime basis. In many other situations, we can form a suitable prime basis from this orthogonal set. For the moment, we shall simply suppose that we have some prime basis and work with it.

We can express the nodal basis functions as linear combinations of the prime basis functions since the two sets span the same space. We look for coefficients $\alpha_j^{(i)}$ such that for $1 \leq i \leq \dim P$

$$\psi_i = \sum_{k=1}^{\dim P} \alpha_k^{(i)} \phi_k. \tag{1}$$

For each ψ_i, we have $\dim P$ degrees of freedom to determine. The criteria that $n_i(\psi_j) = \delta_{i,j}$ allow us to determine these coefficients. Letting a^i be the vector with $(a^i)_j = \alpha_j^{(i)}$ and e^i the canonical basis vector, we have

$$Va^i = e^i, \tag{2}$$

where $V_{i,j} = n_i(\phi_j)$ is a sort of generalized Vandermonde matrix. We may solve for all the coefficients simultaneously by posing a problem with multiple right hand sides as:

$$VA = I, \tag{3}$$

where the ith column of A is a^i. Then, the matrix V^{-1} contains all of the coefficients for all of the nodal basis functions. We remark that this generalizes the

techniques used by Hesthaven and Warburton [2002] for Lagrangian elements in spectral element methods.

Since we can write a rule for evaluating the prime basis to any order and we have a rule for evaluating the nodes of any order (e.g. formulae for the lattice points on the triangle), this procedure can be implemented on a computer in a language that supports higher order computation (i.e. functions may take function-like objects as arguments and have functions as return values).

2.3 A Prime Basis for P_k on the Reference Triangle

There is a well-known orthonormal basis for polynomials over a triangle due to Dubiner [1991]. This basis consists of special combinations of Jacobi polynomials, and has natural extensions to three-dimensional reference domains [Karniadakis and Sherwin 1999]. We will briefly describe this basis.

We let $P_k^{\alpha,\beta}$ denote the kth degree Jacobi polynomial with weights α and β. We define the mapping from (x, y) on the reference triangle K to coordinates (η_1, η_2) on the square $[-1, 1]^2$ by

$$\begin{cases} \eta_1 = 2\frac{1+x}{1-y} - 1 \\ \eta_2 = y \end{cases} \tag{4}$$

with $\eta_1 = -1$ when $y = 1$. Then, the functions

$$D_{i,j}(x, y) = P_i^{0,0}(\eta_1)\left(\frac{1-\eta_2}{2}\right)^i P_j^{2i+1,0}(\eta_2) \tag{5}$$

for $0 \le i, j \le k$ with $i + j \le k$ orthonormally span $P_k(K)$. Formulae for evaluating the partial derivatives of these functions are obtained by the product and chain rules and the formulae for differentiating the Jacobi polynomials.

The Dubiner basis may be ordered hierarchically, so that the first $(k+1)(k+2)/2$ functions span P_k. To do this, let the degree index d run from 0 to k. Then, for each d, take $D_{i,d-i}$ for $0 \le i \le d$. Any Dubiner polynomial of degree exactly k is orthogonal to all polynomials of degree $k-1$ or lower, much like the Legendre polynomials.

2.4 Generating New Prime Bases

Many finite elements of interest rely on function spaces that are not just polynomials of some degree k. For example, when p refinement is used, the function space on a particular domain may be P_k on the inside but constrained to be only P_{k-1} on a given edge, as in Figure 2, where our function space is P_2 constrained to be linear along the bottom edge.

Even apart from p refinement, some finite elements are defined with function space that require a modified prime basis. For example, the vector-valued Brezzi-Douglas-Fortin-Marini elements [Brezzi et al. 1987; Brezzi and Fortin 1991] use functions that are polynomials of degree k in each component but have normal components on the boundary that only have degree $k-1$. Later, we describe the Arnold-Winther stress element, the function space of which is symmetric tensors of polynomials of degree $k+2$ with divergences constrained to be degree k.

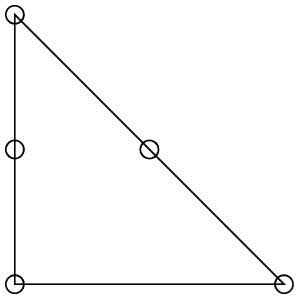

Fig. 2. Nodes for constrained Lagrangian quadratic.

In such situations, we can construct prime bases. Suppose that P lies in some larger finite-dimensional space $\bar{P}$ for which we do have a rule for evaluating a prime basis $\{\bar{\phi}_i\}_{i=1}^{\dim \bar{P}}$. Define $\dim \bar{P} - \dim P = d$. Suppose that there exists a collection of linear functionals on $\bar{P}$ with $L = \{\ell_i\}_{i=1}^{d}$ such that

$$P = \cap_{\ell \in L} \text{null}(\ell). \tag{6}$$

In our first example in this section, $P = \{p \in P_k : p|_\gamma \in P_{k-1}\}$. A natural choice for $\bar{P}$ is $\bar{P} = P_k$. Let μ_k be the Legendre polynomial of degree k. This polynomial is orthogonal in L^2 to all polynomials of degree $k-1$ or less. The set L consists of just a single functional ℓ:

$$\ell(f) = \int_\gamma f \mu_k \mathrm{d}s. \tag{7}$$

Now, we form the matrix $\mathrm{L}_{i,j} = \ell_i(\bar{\phi}_j)$. This matrix is d by $\dim \bar{P}$. The null space of this matrix will give us a prime basis for P. Let us compute the (full) singular value decomposition of L:

$$L = U_L \Sigma_L V_L^t. \tag{8}$$

It is a well-known fact of linear algebra [Golub and Van Loan 1996] that if L has a n dimensional null space, then the final n columns of V_L orthonormally span null(L). We let $\mathcal{V} = V_L(:, d+1 : \dim \bar{P})$. Then, the set

$$\left\{ \phi_i = \sum_{k=1}^{\dim \bar{P}} \mathcal{V}_{k,i} \bar{\phi}_k \right\}_{i=1}^{\dim P} \tag{9}$$

forms a prime basis for P. To see this, simply apply each ℓ_j to any member of the basis and recognize that the result is a row of L dotted with a column of $\mathcal{V}$ and hence vanishes. Then, since the new set has the same dimension as P, it must span P. Since we have an evaluation rule for the basis for $\bar{P}$, we also have an evaluation rule for our basis for P. With this prime basis for P, we may proceed to generate the nodal basis as before.

Remark. We note the similarity of this technique to the approximation-theoretic results of Dupont and Scott [1980], where error estimates for polynomial spaces are derived by characterizing the spaces in terms of what differential operators vanish on them.

In other situations, writing the space as the null space of a collection of functionals may be more difficult, but we could write down a set of functions that spans the space. This is the case of even order nonconforming elements and the Raviart-Thomas elements for $H(\mathrm{div})$. In these cases, we can project the list of functions onto a basis for some larger space and numerically obtain an orthonormal basis consisting of linear combinations of the basis functions of that larger space.

3. COMPUTING ISSUES

The primary goal of our code is to achieve as much of the mathematical structure described above as possible. While the code currently only does finite element spaces over triangles, almost any conceivable polynomial, noncomposite finite element can be implemented. Moreover, the system can be very easily extended to support other element shapes, even in three dimensions. All that needs to be changed is the prime polynomial basis and the quadrature rules, and these are well-documented for other shapes by Karniadakis and Sherwin [1999].

Our goal of general expressiveness and mathematical abstraction is somewhat unusual in numerical computing, where performance is typically not negotiable. However, a code that tabulates nodal basis functions is really a "run once" code. Once the family of basis functions has been generated and stored in a file, the code need not be run any more. Even if this were to take hours, it still represents a substantial savings in time over coming up with and then differentiating explicit formulae. In this section, we discuss several computing issues, from the use of Python to particular techniques and ideas employed in the code to enhance its expressiveness and avoid disastrous inefficiency.

3.1 Why Python?

The intersection of object-oriented and functional programming along with a strong extension module for linear algebra makes Python a good option for this kind of computing. Without the constraint of optimal performance, we were able to enumerate a set of language features that would make it easy to write, read, and extend the code, and then look at existing languages for a good fit. While Python is not the perfect language, it turned out to be a very practical and effective choice in this situation.

The following features seemed most relevant:

—automatic memory management,

—object orientation with support for overloaded operators,

—"higher-order" programming—function-like objects may be passed as arguments to or returned from functions,

—easy interaction with numerical linear algebra routines.

Memory management, becomes important when we consider that we will be making arrays of size only determined at run-time. Hence, we will dynamically be creating and freeing matrices as needed. This makes garbage-collected languages such as Java and Python attractive and much easier to debug than C or C++. On the other hand, object-orientation with overloaded operators allows for

data encapsulation and high-level syntax. Among other languages, these are supported by C++ and Python, but not Java. Third, as we want to pass around "basis functions" and "linear functionals", higher-order programming is important. In C++, this can be accomplished by making hierarchies of classes with the `operator()` method overloaded, or by using templates. However, Python not only allows class hierarchies, but also allows us simply to define some functionals as anonymous functions: "that function that takes a function as an argument and evaluates it at this point." Python has borrowed this feature from functional languages such as ML, Haskell, and Scheme, and it is quite useful in the code. Finally, thanks to the Numerical Python module [Ascher et al. 2001], Python supports matrix operations in a similar style as Matlab, with access to standard LAPACK linear algebra routines such as inversion and the singular value decomposition. Such extension modules are typically much less developed in other higher-order programming languages. Moreover, Python's list comprehension (borrowed from Haskell) can be combined with the Numeric module to give very natural syntax for forming the matrices necessary in the code. This is further described below.

3.2 What's in the Code?

Our code uses several techniques from computer science that may be of interest to scientific programmers. In this section, we describe some of these features and their use in Python. These include memoizing wrappers for basis functions, class hierarchies of our different types of function spaces, and list comprehension for forming our matrices. Additionally, we survey other features provided by the code such as numerical integration rules.

3.2.1 *List Comprehension.* The list comprehension features of Python proved very useful. This syntactic feature constructs a list by evaluating some expression over the elements of another list. For example, suppose we wanted to form the list of squares of integers up to some number. Python provides a function `range(n)` that returns the list of integers from `0` to `n-1`. With this function, the list of squares of integers 0 through 9 may be expressed as:

```
[ i * i for i in range( 0 , 10 ) ]
```

We may also do nested list comprehension to form matrices. If we have a list of basis functions `us` and a list of linear functionals `nodes`, then we form the Vandermonde matrix by

```
v = array( [ [ node( u ) for u in us ] for node in nodes ] )
```

This does a few things. First, each row of the matrix is some functional applied to all the basis functions. This gives a list of lists. Second, the `array` function is a function imported from the `Numeric` module. It converts the list of lists into a matrix representation so that the linear algebra module can operate on it.

3.2.2 *Memoizing Wrapper System.* In the course of evaluating a nodal basis, the Dubiner functions (and others) may be repeatedly evaluated at the same set of points. For example, if the prime basis is a linear combination

of the Dubiner basis and the nodes include integration against some functions constructed from the Dubiner basis, building the Vandermonde may involve order $(\dim P)^2$ evaluations of each Dubiner basis function at each quadrature point, a disastrous inefficiency. While clever programming can circumvent this difficulty, the expressiveness of the code will be compromised. An alternative, which allows for more mathematical-looking code while avoiding the dramatic increase in algorithmic complexity, is memoization [Abelson et al. 1985]. Memoization simply means storing the result of each function call, and reusing the stored value on subsequent evaluation on the same arguments.

We created a class `MemoizedFunction` that is a container for callable objects mapping points to numbers. The instances each contain a reference to some callable object plus an associative array storing previously computed values. This class not only performs memoization, but two other important tasks for the code—operator overloading and differentiation.

When done correctly, operator overloading can allow for very clear, concise code. In our system, we have several different kinds of functions we want to operate on: Dubiner functions, algebraic combinations of functions, functions of barycentric coordinates, and so on. However, since all these different kinds of functions are wrapped into `MemoizedFunction` instances, we can just overload operators for that class.

Many callable objects in our system also have methods that return the function's derivative (as a function). The memoizing wrapper also has a method to invoke the wrapped object's derivative method. Differentiation is also cached in two senses: `MemoizedFunction` remembers if a function has been differentiated in a particular direction and so only creates a single instance of each partial derivative, and these derivative functions are also wrapped into `MemoizedFunction` objects.

Manipulating vectors and tensors as well as scalar-valued functions is also an important tool in a general code. In similar fashion, classes `MemoizedVectorFunction` and `MemoizedTensorFunction` and `MemoizedSymmetricTensorFunction` contain lists of `MemoizedFunction` instances. They may be indexed, evaluated, and differential operators (gradient, divergence, curl) are defined on them.

3.2.3 *Function Spaces.* FIAT uses a class hierarchy to implement the various kinds of function spaces described in our mathematical description. Most of the mathematics is done in a base class, and new classes are derived just by providing inputs to the constructors. This hides the need for users to delve into the mathematical details—they just provide the inputs needed.

For scalar-valued spaces, the base class `FunctionSpace` consists of a list of basis functions and a list of matrices representing partial differentiation in the various coordinate directions. A class implementing polynomials, spanned by the Dubiner basis, is derived directly from this class. Since most functions we will register with the system only have explicit rules for computing first order partial derivatives, it is important that we introduce these matrices for differentiation, as they can be used to construct higher order derivatives.

Also derived from `FunctionSpace` is a class called `LinCombFunctionSpace`. This class takes a function space and a matrix whose columns encode linear combinations of the basis functions of the input space. This is a natural base class for both constrained spaces and finite element spaces. We introduce a class `ConstrainedFunctionSpace` that takes a function space and a list of callable objects representing the constraints and forms the appropriate `LinCombFunctionSpace` by forming the constraint matrix and computing the singular value decomposition. Also derived from `LinCombFunctionSpace` is an `OrthonormalizedFunctionSpace`, which takes a function space, a list of functions, and an integration rule, and forms a new space by projecting the list onto the space and orthonormalizing them. Further, the class `FiniteElement FunctionSpace` takes the dual basis (containing a list of functionals) and a function space, forms the van der Monde matrix, and hence forms the `LinCombFunctionSpace`.

3.2.4 *Linear Functionals and Quadrature Rules.* In addition to functions and function spaces, FIAT provides tools to define linear functionals and numerical integration. A module provides standard linear functionals for point evaluation, normal components of vectors or tensors, defining function moments, and so on. The integration rules are based on the Gauss-Legendre-Jacobi quadrature rules mapped onto the triangle [Karniadakis and Sherwin 1999]. One interesting point is that these quadrature rules are callable objects—for example, we can compute the L^2 inner product of two functions `f` and `g` by

```
integrate = quadrature.JacobiQuadrature2D( 3 )
l2ip = integrate( f * g ).
```

4. EXAMPLES

In this section, we present some examples of elements that are known in the literature, but have seldom (if ever) been used in computation, due in large part to the difficulty involved in evaluating the basis functions. We are interested at present in demonstrating that the approach outlined in this article allows us to tabulate such elements, hopefully enabling better computation by the larger finite element community.

4.1 Nonconforming Elements

The theory of nonconforming finite elements has been well-studied, and the lowest order element (Crouziex-Raviart) is quite practical in many situations. Nonconforming methods are closely related to the so-called primal hybrid method of Raviart and Thomas [1977b]. Raviart and Thomas define an entire family of function spaces, but the even order members are complicated by the presence of an extra function beyond P_k.

For the function space of order n, this extra function is

$$(\lambda_1 - \lambda_0)(\lambda_2 - \lambda_1)(\lambda_0 - \lambda_2)(\lambda_0\lambda_1)^{\frac{n-2}{2}}(\lambda_1\lambda_2)^{\frac{n-2}{2}}(\lambda_2\lambda_0)^{\frac{n-2}{2}}. \tag{10}$$

With the overloaded operators, and so on, we can create simple objects for the barycentric coordinates, append methods for doing the differentiation, and

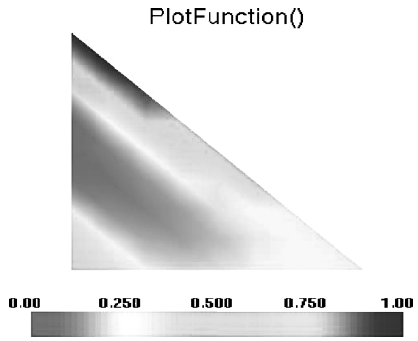

Fig. 3. Nonconforming P_2 basis function for an edge Gauss point.

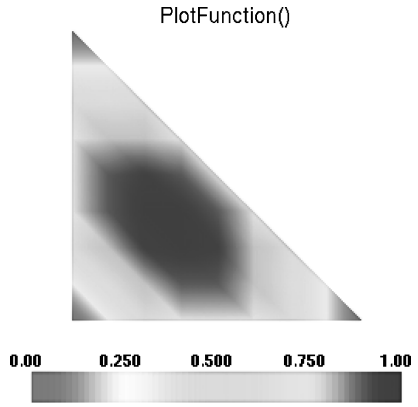

Fig. 4. Nonconforming P_2 basis function for the midpoint.

then build this function with the expression

```
( lam1 - lam0 ) * ( lam2 - lam1 ) * ( lam0 - lam2 )  \
      * ( ( lam0 * lam1 ) ** ( ( n - 2 ) / 2 )  \
          + ( lam1 * lam2 ) ** ( ( n - 2 ) / 2 ) \
          + ( lam0 * lam2 ) ** ( ( n - 2 ) / 2 ) ).
```

Then, we form an orthnormalized function space by projecting this function and the Dubiner basis of order k on P_{k+1}. The degrees of freedom for $k = 2$ are

—pointwise evaluation at the two Gauss-Legendre points on each edge of the triangle,
—pointwise evaluation at the center.

This is the Irons-Razzaque element [Irons and Razzaque 1972] Figures 3 and 4 show some of the basis functions for the second degree case. The first one has unit value at one of the Gauss points on the hypotenuse and vanishes at the other Gauss points around the boundary and at the center of the triangle. The second one vanishes at the Gauss points around the boundary and has unit value at the center.

While the function space is well-defined for all orders, suitable degrees of freedom for general even-order spaces have not been defined. For some even degrees (such as six), the barycenter is already in the natural set of interior degrees of freedom. Before computing with general even-order spaces, some general result for the extra degree of freedom should be obtained. Our code will allow empirical verification of these degrees of freedom (check that the van der Monde matrix is numerically nonsingular), and it seems that specifying the function's average value may be a suitable candidate.

4.2 Vector-Valued *H*(div) Elements: Raviart-Thomas

On triangles, there are three fairly well-known elements discretizing $H(\mathrm{div})$. These are due to Raviart and Thomas (hence RT) [Raviart and Thomas 1977a], Brezzi, Douglas and Marini (hence *BDM*) [Brezzi et al. 1985], and Brezzi, Douglas, Fortin and Marini (hence *BDFM*) [Brezzi et al. 1987; Brezzi and Fortin 1991]. All three families are defined and studied for all orders, with optimal approximation properties, and so on. Yet almost all calculations use only the lowest order RT space, even when higher orders of approximation would be appropriate. This is due in large part to the perceived difficulty of tabulating the elements. Here, we show that a proper understanding of the definition of these spaces and the right programming tools allows us to tabulate these elements. We present RT here as an example, but the other families are also implemented in a module.

The family of Raviart-Thomas elements uses the function spaces:

$$RT_k(T) = P_k(T, \mathbf{R}^2) + xP_k. \tag{11}$$

That is, the spaces are vectors of polynomials of degree k plus the position vector times scalar polynomials of degree k. This space is the smallest space V that the divergence maps onto P_k. If D_k denotes the list of Dubiner polynomials of degree k or less, then $D_k - D_{k-1}$ is the set of $k+1$ Dubiner polynomials that are exactly degree k. A non-orthonormal prime basis for RT_k is then

$$\{(p, 0)\}_{p\in D_k} \cup \{(0, p)\}_{p\in D_k} \cup \{(xp, yp)\}_{p\in(D_k - D_{k-1})}. \tag{12}$$

We can numerically project these functions onto a basis for $P_{k+1}(T, \mathbf{R}^2)$ and orthonormalize them (we use the SVD). This gives a prime basis, so it is only necessary to describe the nodes. These are

—normal component at $k+1$ points on each edge,
—moments against a basis for $P_{k-1}(T, \mathbf{R}^2)$.

We choose the edge points to be equispaced on the interior of each edge, and use the vectors consisting of a Dubiner polynomial in one component and zero in the other as a basis for the interior nodes. These are all easy to specify in FIAT.

In Figures 5 and 6, we visualize two nodal basis functions for RT_3. The first one has a unit normal component at a point on the hypotenuse of the triangle. Note how the normal component vanishes at the other nodal points on the edge. The second corresponds to the degree of freedom for integrating the y

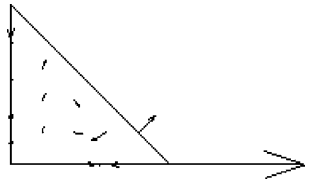

Fig. 5. RT_3 basis function corresponding to point normal at an edge.

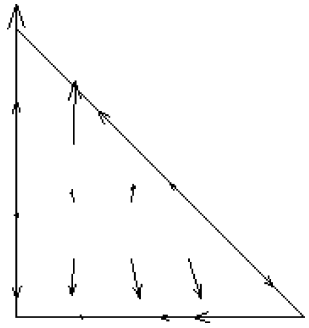

Fig. 6. RT_3 basis function with a unit linear moment of the y component.

component against one of the linear Dubiner functions. Note the strong linear variation in the y component, while the normal component vanishes around the boundary.

4.3 A Tensor-Valued Element Due to Arnold and Winther

Developing a stable, noncomposite pair of elements for the stress/displacement formulation of linear elasticity proved a daunting task for many decades. Recently, Arnold and Winther [2002] formulated and analyzed a suitable family of elements. While the vector-valued displacement spaces are just discontinuous polynomials of degree k for $k \geq 1$, the tensor space for stress is more complex. Denoting by $P_k(T, \mathbf{S})$ the set of symmetric 2×2 tensors with components in P_k, the space is

$$\Sigma_T = \{\tau \in P_{k+2}(T, \mathbf{S}) : \nabla \cdot \tau \in P_k(T, \mathbf{R}^2)\}. \tag{13}$$

In order to build this function space, we start with symmetric tensors of polynomials of degree $k+2$ and specify some constraints. Divergences of $P_{k+2}(T, \mathbf{S})$ naturally lie in $P_{k+1}(T, \mathbf{R}^2)$. The Dubiner polynomials of degree exactly $k+1$ are orthogonal to polynomials of degree k or less. So, our linear constraints on $P_{k+2}(T, \mathbf{S})$ are just integration of each component against these $k+2$ functions. Once these constraints are specified in a list, we can form a constrained function space.

Specifying the nodes is somewhat more involved, but now straightforward. The degrees of freedom are

—point values of the three unique components at each vertex,
—normal components of each row at $k+1$ points on each edge,

—moments against symmetric parts of gradients of $P_k(\mathbf{R}^2)$, and

$$\{\tau \in P_{k+2}(\mathbf{S}) : \nabla \cdot \tau = 0 \text{ and } \tau n = 0 \text{ on } \partial \hat{T}\}.$$

While implementing this family of elements is somewhat more involved than other cases, FIAT makes it relatively straightforward. Having a tool to relatively quickly tabulate these functions let us learn something about them. For one, they may be poorly behaved. The basis function with xx component taking unit value at $(-1, -1)$ has different scales in its three components. The xx component varies between 1 and about -20. The off-diagonal component takes values between 1 and -2. The yy component varies on a much wider scale, however, taking values between about 200 and -900. While a systematic study has not been done, FIAT makes it possible to tabulate and study these basis functions, as well as put them into a solver in order to experiment with them.

5. CONCLUSIONS

We have developed a mathematical paradigm and practical computer implementation for tabulating general finite element basis functions of possibly high order. This allows us to tabulate the basis functions for complicated elements, including for vector-valued and tensor-valued spaces. This appears to be the first systematic approach to computing basis functions that cuts across different families of elements.

Having an easy way of getting basis functions frees people developing finite element code to use complicated, possibly high-order elements when appropriate. Elements with excellent theoretical properties now also become practical options. We are currently looking at ways of incorporating these techniques into finite element codes. One subject of current investigation is building a run-time C library for orthogonal polynomials on the various reference domains. Then, FIAT can output the coefficients of the nodal basis functions in the orthonormal expansions, and the run-time library can evaluate and differentiate the basis functions as needed.

Finally, this code hopefully serves as an example of the power that higher-order programming can offer. Given what this code is able to accomplish with relative ease, it will be interesting to explore what other opportunities higher order programming offers to scientific computing.

REFERENCES

ABELSON, H., SUSSMAN, G. J., AND SUSSMAN, J. 1985. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

ARNOLD, D. N., BREZZI, F., COCKBURN, B., AND MARINI, L. D. 2002. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM J. Numer. Anal. 39*, 5, 1749–1779 (electronic).

ARNOLD, D. N. AND WINTHER, R. 2002. Mixed finite elements for elasticity. *Numer. Math. 92*, 3, 401–419.

ASCHER, D., DUBOIS, P. F., HINSON, K., HUGUNIN, J., AND OLIPHANT, T. 2001. Numerical Python. Tech. Rep. UCRL-MA-128569. Lawrence Livermore National Laboratory, Livermore, CA 94566.

BANGERTH, W. AND KANSCHAT, G. 1999. Concepts for Object-Oriented Finite Element Software the `deal.II` *Library*. Preprint 99-43 (SFB 359). IWR Heidelberg.

BREZZI, F., DOUGLAS, JR., J., FORTIN, M., AND MARINI, L. D. 1987. Efficient rectangular mixed finite elements in two and three space variables. *RAIRO Modél. Math. Anal. Numér. 21*, 4, 581–604.

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.

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

DUPONT, T. AND SCOTT, L. 1980. Polynomial approximation of functions in Sobolev spaces. *Math. Comp. 34*, 441–463.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix computations*, Third ed. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD.

HESTHAVEN, J. S. AND WARBURTON, T. 2002. Nodal high-order methods on unstructured grids. I. Time-domain solution of Maxwell's equations. *J. Comput. Phys. 181*, 1, 186–221.

IRONS, B. M. AND RAZZAQUE, A. 1972. Experience with the patch test for convergence of finite elements. In *The mathematical foundations of the finite element method with applications to partial differential equations (Proc. Sympos., Univ. Maryland, Baltimore, Md., 1972)*. Academic Press, New York, 557–587.

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

LOGG, A. AND HOFFMAN, J. 2002. Dolfin: Dynamic object oriented library for finite element computation. Tech. Rep. 2002-07, Chalmers Finite Element Center Preprint Series.

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

RACHOWICZ, W. AND DEMKOWICZ, L. 2002. An *hp*-adaptive finite element method for electromagnetics. II. A 3D implementation. *Internat. J. Numer. Methods Engrg. 53*, 1, 147–180. *p* and *hp* finite element methods: mathematics and engineering practice (St. Louis, MO, 2000).

RAVIART, P.-A. AND THOMAS, J. M. 1977a. 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.

RAVIART, P.-A. AND THOMAS, J. M. 1977b. Primal hybrid finite element methods for 2nd order elliptic equations. *Math. Comp. 31*, 138, 391–413.

RIVIÈRE, B. AND WHEELER, M. F. 2000. Locally conservative algorithms for flow. In *The mathematics of finite elements and applications, X, MAFELAP 1999 (Uxbridge)*. Elsevier, Oxford, 29–46.

ZIENKIEWICZ, O. C. 1971. *The finite element method in engineering science*. McGraw-Hill, London. The second, expanded and revised, edition of *The finite element method in structural and continuum mechanics*.

Received October 2003; revised April 2004; accepted June 2004