A Compiler for Variational Forms

Robert C. Kirby
The University of Chicago
and
Anders Logg
Toyota Technological Institute at Chicago

As a key step towards a complete automation of the finite element method, we present a new algorithm for automatic and efficient evaluation of multilinear variational forms. The algorithm has been implemented in the form of a compiler, the FEniCS Form Compiler FFC. We present benchmark results for a series of standard variational forms, including the incompressible Navier–Stokes equations and linear elasticity. The speedup compared to the standard quadrature-based approach is impressive; in some cases the speedup is as large as a factor 1000.

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: variational form, compiler, finite element, automation

1. INTRODUCTION

The finite element method provides a general mathematical framework for the solution of differential equations and can be viewed as a machine that automates the discretization of differential equations; given the variational formulation of a differential equation, the finite element method generates a discrete system of equations for the approximate solution.

This generality of the finite element method is seldom reflected in codes, which are often very specialized and can only solve one particular differential equation or a small set of differential equations.

There are two major reasons that the finite element method has yet to be fully automated; the first is the complexity of the task itself, and the second is that specialized codes often outperform general codes. We address both these concerns in this paper.

A basic task of the finite element method is the computation of the element

Robert C. Kirby, Department of Computer Science, University of Chicago, 1100 East 58th Street, Chicago, Illinois 60637, USA. *Email:* `kirby@cs.uchicago.edu`. This work was supported by the United States Department of Energy under grant DE-FG02-04ER25650.
Anders Logg, Toyota Technological Institute at Chicago, University Press Building, 1427 East 60th Street, Chicago, Illinois 60637, USA. *Email:* `logg@tti-c.org`.

stiffness matrix from a bilinear form on a local element. In many applications, computation of element stiffness matrices accounts for a substantial part of the total run-time of the code. This routine is a small amount of code, but it can be tedious to get it both correct and efficient. While the standard quadrature-based approach to computing the element stiffness matrix works on very general variational forms, it is well-known that precomputing certain quantities in multilinear forms can improve the efficiency of building finite element matrices.

The methods discussed in this paper for efficient computation of element stiffness matrices are based on ideas previously presented in [Kirby et al. 2005; Kirby et al. 2005], where the basic idea is to represent the element stiffness matrix as a tensor product. A similar approach has been implemented earlier in the finite element library DOLFIN [Hoffman et al. ; Hoffman and Logg 2002], but only for linear elements. The current paper generalizes and formalizes these ideas and presents an algorithm for generation of the tensor representation of element stiffness matrices and for evaluation of the tensor product. This algorithm has been implemented in the form of the compiler FFC for variational forms; the compiler takes as input a variational form in mathematical notation and automatically generates efficient code (C or C++) for computation of element stiffness matrices and their insertion into a global sparse matrix. This includes the generation of code both for the computation of element stiffness matrices and local-to-global mappings of degrees of freedom.

1.1 FEniCS and the Automation of CMM

FFC, the FEniCS Form Compiler, is a central component of FEniCS [Hoffman et al.], a project for the Automation of Computational Mathematical Modeling (ACMM). The central task of ACMM, as formulated in [Logg 2004], is to create a machine that takes as input a model $Au = f$, a tolerance $\mathrm{TOL} > 0$ and a norm $\|\cdot\|$ (or some other measure of quality), and produces as output an approximate solution $U \approx u$ that satisfies the accuracy requirement $\|U - u\| < \mathrm{TOL}$ using a minimal amount of work (see Figure 1). This includes an aspect of *reliability* (the produced solution should satisfy the accuracy requirement) and an aspect of *efficiency* (the solution should be obtained with minimal work).

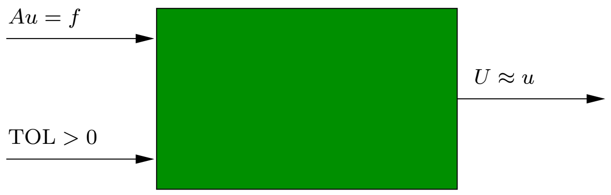

Fig. 1. The Automation of Computational Mathematical Modeling.

In many applications, several competing models are under consideration, and one would like to computationally compare them. Developing separate, special purpose codes for each model is prohibitive. Hence, a key step of ACMM is the *automation of discretization*, i.e., the automatic translation of a differential equation into a discrete system of equations, and as noted above this key step is automated by the finite element method. The FEniCS Form Compiler FFC may then be viewed as an important step towards the automation of the finite element method, and thus as an important step towards the Automation of CMM.

FEniCS software is free software. In particular, FFC is licensed under the GNU General Public License [Free Software Foundation]. All FEniCS software is available for download on the FEniCS web site [Hoffman et al.]. In Section 5.6, we return return to a discussion of the different components of FEniCS and their relation to FFC.

1.2 Current finite element software

Several emerging projects seek to automate important aspects of the finite element method. By developing libraries in existing languages or new domain-specific languages, software tools may be built that allow programmers to define variational forms and other parts of a finite element method with succinct, mathematical syntax. Existing C++ libraries for finite elements include DOLFIN [Hoffman et al. ; Hoffman and Logg 2002], Sundance [Long ; 2003] and deal.II [Bangerth et al.]. Projects developing domain-specific languages for finite element computation include FreeFEM [Pironneau et al.] and GetDP [Dular and Geuzaine]. A precursor to the FEniCS project, Analysa [Bagheri and Scott], was a Scheme-like language for finite element methods.

While these tools are effective at exploiting modern software engineering to produce workable systems, we believe that additional mathematical insight will lead to even more powerful codes with more general approximating spaces and more powerful algorithms.

1.3 Design goals

The primary design goal for FFC is to accept as input "any" multilinear variational form and "any" finite element, and to generate code that will run with close to optimal performance. We will make precise below in Section 3.2 which forms and which elements the compiler can handle.

A secondary goal for FFC is to create a new standard in form evaluation; hopefully FFC can become a standard tool for practitioners solving partial differential equations using the finite element method.

The primary output target of FFC is the C++ library DOLFIN. By default, FFC accepts as input a variational form and generates code for the evaluation of the variational form in DOLFIN, as illustrated in Figure 2. Although FFC is tightly integrated with other components of the FEniCS project, such as DOLFIN, it has been designed to allow for easy adoption to other systems.

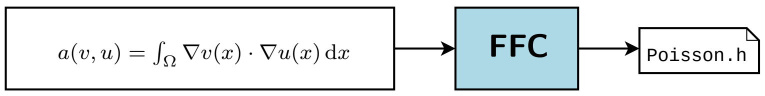
$a(v,u) = \int_\Omega \nabla v(x) \cdot \nabla u(x)\,\mathrm{d}x$
FFC
Poisson.h

Fig. 2. The form compiler FFC takes as input a variational form and generates code for evaluation of the form.

1.4 The compiler approach

It is widely accepted in developing software for scientific computing that there is a trade-off between generality and efficiency; a software component that is general in nature, i.e., it accepts a wide range of inputs, is often less efficient than another software component that performs the same job on a more limited set of inputs. As a result, most codes used by practitioners for the solution of differential equations are very specific.

However, by using a compiler approach, it is possible to combine generality and efficiency without loss of generality and without loss of efficiency. This is possible since our compiler works on a very small family of inputs (multilinear variational forms) with sharply defined mathematical properties. Our domain-specific knowledge allows us to generate much better code than if we used general-purpose compiler techniques.

1.5 Outline of this paper

Before presenting the main algorithm, we give a short background on the implementation of the finite element method and the evaluation of variational forms in Section 2. The main algorithm is then outlined in Section 3. In Section 4, we compare the complexity of form evaluation for the algorithm used by FFC with the standard quadrature-based approach. We then discuss the implementation of the form compiler in some detail in Section 5.

In Section 6, we compare the CPU time for evaluating a series of standard variational forms using code automatically generated by FFC and hand-coded quadrature-based implementations. The speedup is in all cases significant, in the case of cubic Lagrange elements on tetrahedra a factor 100 (Figure 3).

Finally, in Section 7, we summarize the current features and shortcomings of FFC and give directions for future development and research.

2. BACKGROUND

In this section, we present a quick background on the finite element method. The material is standard [Ciarlet 1976; Hughes 1987; Brenner and Scott 1994; Eriksson et al. 1996], but is included here to give a context for the presentation of the form compiler and to summarize the notation used throughout the remainder of this paper.

For simplicity, we consider here only linear partial differential equations and note that these play an important role in the discretization of nonlinear partial differential equations (in Newton or fixed-point iterations).

2.1 Variational forms

We work with the standard variational formulation of a partial differential equation: Find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \tag{1}$$

with $a : \hat{V} \times V \to \mathbb{R}$ a bilinear form, $L : \hat{V} \to \mathbb{R}$ a linear form, and $(\hat{V}, V)$ a pair of suitable function spaces. For the standard example, Poisson's equation $-\Delta u(x) = f(x)$ with homogeneous Dirichlet conditions on a domain Ω, the bilinear

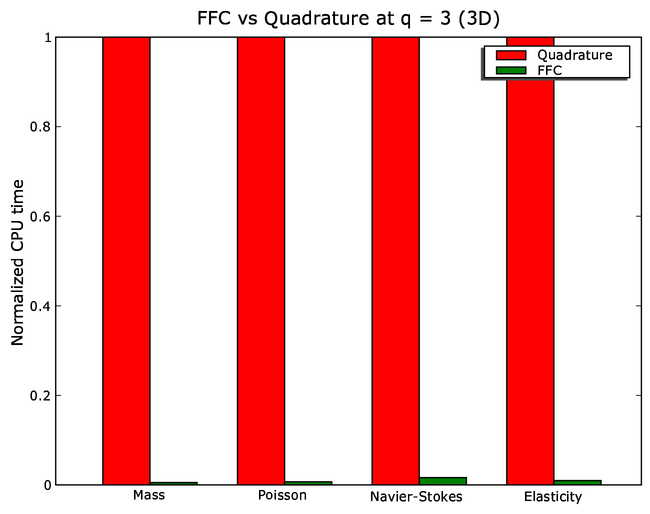

Fig. 3. Benchmark results for a series of standard variational forms (here compiled for cubic Lagrange elements on tetrahedra).

form a is given by $a(v,u) = \int_\Omega \nabla v(x) \cdot \nabla u(x) \,\mathrm{d}x$, the linear form L is given by $L(v) = \int_\Omega v(x) f(x) \,\mathrm{d}x$, and $\hat{V} = V = H^1_0(\Omega)$.

The finite element method discretizes (1) by replacing $(\hat{V}, V)$ with a pair of (piecewise polynomial) discrete function spaces. With $\{\hat{\varphi}_i\}_{i=1}^M$ a basis for the test space $\hat{V}$ and $\{\varphi_i\}_{i=1}^M$ a basis for the trial space V, we can expand the approximate solution U of (1) in the basis functions of V, $U = \sum_{j=1}^M \xi_j \varphi_j$, and obtain a linear system $A\xi = b$ for the *degrees of freedom* $\{\xi_j\}_{j=1}^M$ of the approximate solution U. The entries of the matrix A and the vector b defining the linear system are given by

$$\begin{aligned} A_{ij} &= a(\hat{\varphi}_i, \varphi_j), \quad i,j = 1,\dots,M, \\ b_i &= L(\hat{\varphi}_i), \quad i = 1,\dots,M. \end{aligned} \tag{2}$$

2.2 Assembly

The standard algorithm for computing the matrix A (or the vector b) is *assembly*; the matrix is computed by iteration over the elements K of a triangulation $\mathcal{T}$ of Ω, and the contribution from each local element is added to the global matrix A.

To see this, we note that if the bilinear form a is expressed as an integral over the domain Ω, we can write the bilinear form as a sum of element bilinear forms, $a(v,u) = \sum_{K \in \mathcal{T}} a_K(v,u)$, and thus

$$A_{ij} = \sum_{K \in \mathcal{T}} a_K(\hat{\varphi}_i, \varphi_j), \quad i,j = 1,\dots,M. \tag{3}$$

In the case of Poisson's equation, the element bilinear form a_K is defined by $a(v,u) = \int_K \nabla v(x) \cdot \nabla u(x) \,\mathrm{d}x$.

Let now $\{\hat{\varphi}_i^K\}_{i=1}^n$ be the restriction to K of the subset of $\{\hat{\varphi}_i\}_{i=1}^M$ supported on K and $\{\varphi_i^K\}_{i=1}^n$ the corresponding local basis for V. Furthermore, let $\hat{\iota}(\cdot,\cdot)$ be a mapping from the local numbering scheme to the global numbering scheme (local-to-global mapping) for the basis functions of $\hat{V}$, so that $\hat{\varphi}_i^K$ is the restriction to K of $\varphi_{\hat{\iota}(K,i)}$, and let $\iota(K,\cdot)$ be the corresponding mapping for V. We may now express an algorithm for computation of the matrix A (Algorithm 1).

Algorithm 1 $A = \text{Assemble}(a, \mathcal{T}, \hat{V}, V)$

```
$A = 0$
for $K \in \mathcal{T}$
    for $i = 1, \ldots, n$
        for $j = 1, \ldots, n$
            $A_{\hat{\iota}(K,i)\iota(K,j)} = A_{\hat{\iota}(K,i)\iota(K,j)} + a_K(\hat{\varphi}_i^K, \varphi_j^K)$
        end for
    end for
end for
```

Alternatively, one may define the *element matrix* A^K by

$$A_{ij}^K = a_K(\hat{\varphi}_i^K, \varphi_j^K) \quad i,j = 1, \ldots, n, \tag{4}$$

and separate the computation on each element K into two steps: computation of the element matrix A^K and insertion of A^K into A (Algorithm 2).

Algorithm 2 $A = \text{Assemble}(a, \mathcal{T}, \hat{V}, V)$

```
$A = 0$
for $K \in \mathcal{T}$
    Compute $A^K$ according to (4)
    Add $A^K$ to $A$ using the local-to-global mappings $(\hat{\iota}(K,\cdot), \iota(K,\cdot))$
end for
```

Separating the two concerns of computing the element matrix A^K and adding it to the global matrix A as in Algorithm 2 has the advantage that one may use an optimized library routine for adding the element matrix A^K to the global matrix A. Sparse matrix libraries such as PETSc [Balay et al. 2001; Balay et al. 2004; Balay et al. 1997] often provide optimized routines for this type of operation. Note that the cost of adding A^K to A may be substantial even with an efficient implementation of the sparse data structure for A [Kirby et al. 2005].

As we shall see below, we may also take advantage of the separation of concerns of Algorithm 2 to optimize the computation of the element matrix A^K. This step is automated by the form compiler FFC. Given a bilinear (or multilinear) form a, FFC automatically generates code for run-time computation of the element matrix A^K.

3. EVALUATION OF MULTILINEAR FORMS

In this section, we present the algorithm used by FFC to automatically generate efficient code for run-time computation of the element matrix A^K.

3.1 Multilinear forms

Let $\{V_i\}_{i=1}^r$ be a given set of discrete function spaces defined on a triangulation $\mathcal{T}$ of $\Omega \subset \mathbb{R}^d$. We consider a general multilinear form a defined on the product space $V_1 \times V_2 \times \cdots \times V_r$:

$$a : V_1 \times V_2 \times \cdots \times V_r \to \mathbb{R}. \tag{5}$$

Typically, $r = 1$ (linear form) or $r = 2$ (bilinear form), but the form compiler FFC can handle multilinear forms of arbitrary arity r. Forms of higher arity appear frequently in applications and include variable coefficient diffusion and advection of momentum in the incompressible Navier–Stokes equations.

Let now $\{\varphi_i^1\}_{i=1}^{M_1}, \{\varphi_i^2\}_{i=1}^{M_2}, \ldots, \{\varphi_i^r\}_{i=1}^{M_r}$ be bases of $V_1, V_2, \ldots, V_r$ and let $i = (i_1, i_2, \ldots, i_r)$ be a multiindex. The multilinear form a then defines a rank r tensor given by

$$A_i = a(\varphi_{i_1}^1, \varphi_{i_2}^2, \ldots, \varphi_{i_r}^r). \tag{6}$$

In the case of a bilinear form, the tensor A is a matrix (the stiffness matrix), and in the case of a linear form, the tensor A is a vector (the load vector).

As discussed in the previous section, to compute the tensor A by assembly, we need to compute the *element tensor* A^K on each element K of the triangulation $\mathcal{T}$ of Ω. Let $\{\varphi_i^{K,1}\}_{i=1}^{n_1}$ be the restriction to K of the subset of $\{\varphi_i^1\}_{i=1}^{M_1}$ supported on K and define the local bases on K for $V_2, \ldots, V_r$ similarly. The rank r element tensor A^K is then defined by

$$A_i^K = a_K(\varphi_{i_1}^{K,1}, \varphi_{i_2}^{K,2}, \ldots, \varphi_{i_r}^{K,r}). \tag{7}$$

3.2 Evaluation by tensor representation

The element tensor A^K can be efficiently computed by representing A^K as a special tensor product. Under some mild assumptions which we shall make precise below, the element tensor A^K can be represented as the tensor product of a *reference tensor* A^0 and a *geometry tensor* G_K:

$$A_i^K = A_{i\alpha}^0 G_K^\alpha, \tag{8}$$

or more generally a sum $A_i^K = A_{i\alpha}^{0,k} G_{K,k}^\alpha$ of such tensor products, where i and α are multiindices and we use the convention that repetition of an index means summation over that index. The rank of the reference tensor is the sum of the rank $r = |i|$ of the element tensor and the rank $|\alpha|$ of the geometry tensor G_K. As we shall see, the rank of the geometry tensor depends on the specific form.

Our goal is to develop an algorithm that converts an abstract representation of a multilinear form into (i) the values of the reference tensor A^0 and (ii) an expression for evaluating the geometry tensor G_K for any given element K. Note that A^0 is fixed and independent of the element K and may thus be precomputed. Only G_K has to be computed on each element. As we shall see below in Section 4, for a wide range of multilinear forms, this allows for computation of the element tensor

A^K using far fewer floating-point operations than if the element tensor A^K were computed by quadrature on each element K.

To see how to obtain the tensor representation (8), we fix a small set of operations, allowing only multilinear forms that can be expressed through these operations, and observe how the tensor representation (8) transforms under these operations. As we shall see below, this covers a wide range of multilinear forms (but not all).

As basic elements, we take the local basis functions $\{\varphi_\gamma\}_\gamma = \cup_i \{\varphi_j^{K,i}\}_{j=1}^{n_i}$ for a set of finite element spaces V_i, $i = 1, 2, \ldots$, including the finite element spaces $V_1, V_2, \ldots, V_r$ on which the multilinear form is defined. Allowing addition $\varphi_1 + \varphi_2$ and multiplication with scalars $\alpha\varphi$, we obtain a vector space $\mathcal{A}$ of linear combinations of basis functions. Since $\varphi_1 - \varphi_2 = \varphi_1 + (-1)\varphi_2$ and $\varphi/\alpha = (1/\alpha)\varphi$, we can easily equip the vector space with subtraction and division by scalars.

We next equip our vector space with multiplication between elements of the vector space. We thus obtain an algebra $\mathcal{A}$ of linear combinations of products of basis functions. Finally, we extend $\mathcal{A}$ by adding differentiation $\partial/\partial x_i$ with respect to a coordinate direction x_i, $i = 1, \ldots, d$, on K, to obtain

$$\mathcal{A} = \{v : v = \sum c_{(\cdot)} \prod \frac{\partial^{|(\cdot)|}\varphi_{(\cdot)}}{\partial x_{(\cdot)}}\}, \tag{9}$$

where $(\cdot)$ represents some multiindex.

To summarize, $\mathcal{A}$ is the algebra of linear combinations of products of basis functions or derivatives of basis functions that is generated from the set of basis functions through addition $(+)$, subtraction $(-)$, multiplication $(\cdot)$, including multiplication with scalars, division by scalars $(/)$, and differentiation $\partial/\partial x_i$. Note that if the basis functions are vector-valued (or tensor-valued), the algebra is generated from the set of scalar components of the basis functions.

We may now state precisely the multilinear forms that the form compiler FFC can handle, namely those multilinear forms that can be expressed as integrals over K (or the boundary of K) of elements of the algebra $\mathcal{A}$. Note that not all integrals over K of elements of $\mathcal{A}$ are multilinear forms; in particular, each product needs to be linear in each argument of the form.

The tensor representation (8) now follows by a standard change of variables using an affine mapping $F_K : K_0 \to K$ from a reference element K_0 to the current element K (see Figure 4). With $\{\Phi_\gamma\}_\gamma$ the basis functions on the reference element corresponding to $\{\varphi_\gamma\}_\gamma$, defined by $\Phi_\gamma = \varphi_\gamma \circ F_K$, we obtain the following representation of the element tensor A^K corresponding to $v_i = \left(\sum c_{(\cdot)} \prod \frac{\partial^{|(\cdot)|}\varphi_{(\cdot)}}{\partial x_{(\cdot)}}\right)_i$:

$$\begin{aligned}
A_i^K &= a_K(\varphi_{i_1}^{K,1}, \varphi_{i_2}^{K,2}, \ldots, \varphi_{i_r}^{K,r}) = \int_K v_i \, \mathrm{d}x \\
&= \left(\int_K \sum c_{(\cdot)} \prod \frac{\partial^{|(\cdot)|}\varphi_{(\cdot)}}{\partial x_{(\cdot)}} \, \mathrm{d}x\right)_i = \sum \left(c_{(\cdot)} \int_K \prod \frac{\partial^{|(\cdot)|}\varphi_{(\cdot)}}{\partial x_{(\cdot)}} \, \mathrm{d}x\right)_i \\
&= \sum \left(c_{(\cdot)} \det F_K' \prod \frac{\partial X_{(\cdot)}}{\partial x_{(\cdot)}}\right)_\alpha \left(\int_{K_0} \prod \frac{\partial^{|(\cdot)|}\Phi_{(\cdot)}}{\partial X_{(\cdot)}} \, \mathrm{d}X\right)_{i\alpha} \\
&= A_{i\alpha}^{0,k} G_{K,k}^\alpha,
\end{aligned} \tag{10}$$

where

$$A^{0,k}_{i\alpha} = \left(\int_{K_0} \prod \frac{\partial^{|(\cdot)|}\Phi_{(\cdot)}}{\partial X_{(\cdot)}} \, \mathrm{d}X \right)_{i\alpha}, \tag{11}$$

$$G^{\alpha}_{K,k} = \left(c_{(\cdot)} \det F'_K \prod \frac{\partial X_{(\cdot)}}{\partial x_{(\cdot)}} \right)_{\alpha}. \tag{12}$$

Note that the expression for the geometry tensor $G_{K,k}$ implicitly contains a summation if an index is repeated twice. Also note that the geometry tensor contains any variable coefficients appearing in the form.

As we shall see below in Section 5, the representation of a multilinear form as an integral over K of an element of $\mathcal{A}$ is automatically available to the form compiler FFC, since a multilinear form must be specified using the basic operations that generate $\mathcal{A}$.

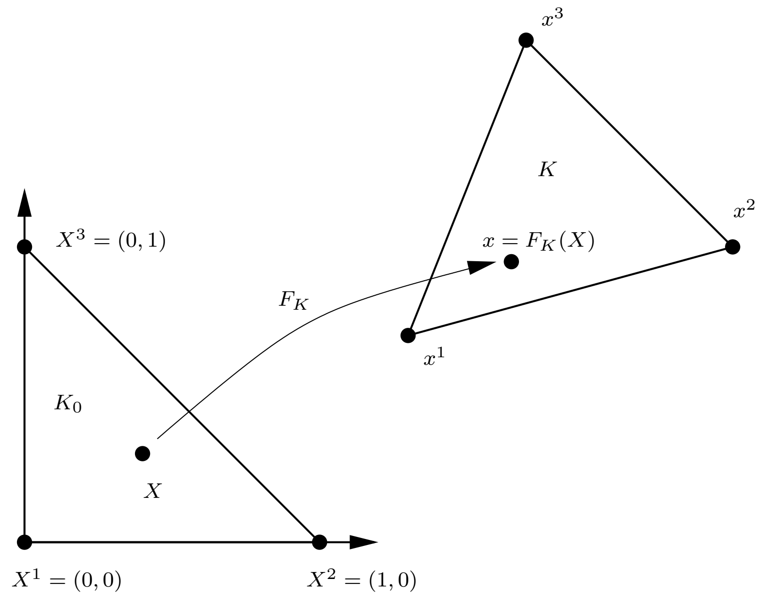

Fig. 4. The affine mapping F_K from the reference element K_0 to the current element K.

3.3 Test cases

To make these ideas concrete, we write down the explicit tensor representation (8) of the element tensor A^K for a series of standard forms. We return to these test cases below in Section 6 when we present benchmark results for each test case.

Test case 1: the mass matrix. As a first simple example, we consider the computation of the mass matrix M with $M_{i_1 i_2} = a(\varphi^1_{i_1}, \varphi^2_{i_2})$ and the bilinear form a given by

$$a(v,u) = \int_\Omega v(x)u(x)\,\mathrm{d}x. \tag{13}$$

By a change of variables given by the affine mapping $F_K : K_0 \to K$, we obtain

$$A_i^K = \int_K \varphi_{i_1}^{K,1}(x)\varphi_{i_2}^{K,2}(x)\,\mathrm{d}x = \det F_K' \int_{K_0} \Phi_{i_1}^1(X)\Phi_{i_2}^1(X)\,\mathrm{d}X = A_i^0 G_K, \tag{14}$$

where $A_i^0 = \int_{K_0} \Phi_{i_1}^1(X)\Phi_{i_2}^2(X)\,\mathrm{d}X$ and $G_K = \det F_K'$. In this case, the reference tensor A^0 is a rank two tensor (a matrix) and the geometry tensor G_K is a rank zero tensor (a scalar). By precomputing the reference tensor A^0, we may thus compute the element tensor A^K on each element K by just multiplying the precomputed reference tensor with the determinant of (the derivative of) the affine mapping F_K.

Test case 2: Poisson's equation. As a second example, consider the bilinear form for Poisson's equation,

$$a(v,u) = \int_\Omega \nabla v(x) \cdot \nabla u(x)\,\mathrm{d}x. \tag{15}$$

By a change of variables as above, we obtain the following representation of the element tensor A^K:

$$\begin{aligned} A_i^K &= \int_K \nabla\varphi_{i_1}^{K,1}(x) \cdot \nabla\varphi_{i_2}^{K,2}(x)\,\mathrm{d}x \\ &= \det F_K' \frac{\partial X_{\alpha_1}}{\partial x_\beta}\frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \frac{\partial \Phi_{i_1}^1(X)}{\partial X_{\alpha_1}}\frac{\partial \Phi_{i_2}^2(X)}{\partial X_{\alpha_2}}\,\mathrm{d}X = A_{i\alpha}^0 G_K^\alpha, \end{aligned} \tag{16}$$

where $A_{i\alpha}^0 = \int_{K_0} \frac{\partial \Phi_{i_1}^1(X)}{\partial X_{\alpha_1}}\frac{\partial \Phi_{i_2}^2(X)}{\partial X_{\alpha_2}}\,\mathrm{d}X$ and $G_K^\alpha = \det F_K' \frac{\partial X_{\alpha_1}}{\partial x_\beta}\frac{\partial X_{\alpha_2}}{\partial x_\beta}$. We see that the reference tensor A^0 is here a rank four tensor and the geometry tensor G_K is a rank two tensor (one index for each derivative appearing in the form).

Test case 3: Navier–Stokes. Consider now the nonlinear term $u \cdot \nabla u$ of the incompressible Navier–Stokes equations,

$$\begin{aligned} \dot{u} + u\cdot\nabla u - \nu\Delta u + \nabla p &= f, \\ \nabla \cdot u &= 0. \end{aligned} \tag{17}$$

To solve the Navier–Stokes equations by fixed-point iteration (see for example [Eriksson et al. 2003]), we write the nonlinear term in the form $u \cdot \nabla u = w \cdot \nabla u$ with $w = u$ and consider w as fixed. We then obtain the following bilinear form:

$$a(v,u) = a_w(v,u) = \int_\Omega v(x) \cdot (w(x)\cdot\nabla)u(x)\,\mathrm{d}x. \tag{18}$$

Note that we may alternatively think of this as a trilinear form, $a = a(v,u,w)$.

Let now $\{w_\alpha^K\}_\alpha$ be the expansion coefficients for w in a finite element basis on the current element K, and let $v[i]$ denote component i of a vector-valued function v. Furthermore, assume that u and w are discretized using the same discrete space $V = V_2$. We then obtain the following representation of the element tensor A^K:

$$\begin{aligned} A_i^K &= \int_K \varphi_{i_1}^{K,1}(x) \cdot (w(x)\cdot\nabla)\varphi_{i_2}^{K,2}(x)\,\mathrm{d}x \\ &= \det F_K' \frac{\partial X_{\alpha_3}}{\partial x_{\alpha_1}} w_{\alpha_2}^K \int_{K_0} \Phi_{i_1}^1[\beta](X)\Phi_{\alpha_2}^2[\alpha_1](X)\frac{\partial \Phi_{i_2}^2[\beta](X)}{\partial X_{\alpha_3}}\,\mathrm{d}X = A_{i\alpha}^0 G_K^\alpha, \end{aligned} \tag{19}$$

where $A^0_{i\alpha} = \int_{K_0} \Phi^1_{i_1}[\beta](X)\Phi^2_{\alpha_2}[\alpha_1](X)\frac{\partial \Phi^2_{i_2}[\beta](X)}{\partial X_{\alpha_3}}\,\mathrm{d}X$ and $G^\alpha_K = \det F'_K \frac{\partial X_{\alpha_3}}{\partial x_{\alpha_1}} w^K_{\alpha_2}$. In this case, the reference tensor A^0 is a rank five tensor and the geometry tensor G_K is a rank three tensor (one index for the derivative, one for the function w, and one for the scalar product).

Test case 4: linear elasticity. Finally, consider the strain-strain term of linear elasticity,

$$\begin{aligned} a(v,u) &= \int_\Omega \frac{1}{4}(\nabla v + (\nabla v)^\top) : (\nabla u + (\nabla u)^\top)\,\mathrm{d}x \\ &= \int_\Omega \frac{1}{4}\frac{\partial v_i}{\partial x_j}\frac{\partial u_i}{\partial x_j}\,\mathrm{d}x + \frac{1}{4}\frac{\partial v_i}{\partial x_j}\frac{\partial u_j}{\partial x_i}\,\mathrm{d}x + \frac{1}{4}\frac{\partial v_j}{\partial x_i}\frac{\partial u_i}{\partial x_j}\,\mathrm{d}x + \frac{1}{4}\frac{\partial v_j}{\partial x_i}\frac{\partial u_j}{\partial x_i}\,\mathrm{d}x. \end{aligned} \tag{20}$$

Note that because of symmetry, the second and third terms may be grouped together. Considering here only the first term, a change of variables leads to the following representation of the element tensor $A^{K,1}$:

$$\begin{aligned} A^{K,1}_i &= \int_K \frac{1}{4}\frac{\partial \varphi^{K,1}_{i_1}[\beta_1](x)}{\partial x_{\beta_2}}\frac{\partial \varphi^{K,2}_{i_2}[\beta_1](x)}{\partial x_{\beta_2}}\,\mathrm{d}x \\ &= \frac{1}{4}\det F'_K \frac{\partial X_{\alpha_1}}{\partial x_{\beta_2}}\frac{\partial X_{\alpha_2}}{\partial x_{\beta_2}}\int_{K_0}\frac{\partial \Phi^1_{i_1}[\beta_1](X)}{\partial X_{\alpha_1}}\frac{\partial \Phi^2_{i_2}[\beta_1](X)}{\partial X_{\alpha_2}}\,\mathrm{d}X = A^{0,1}_{i\alpha}G^\alpha_{K,1}, \end{aligned} \tag{21}$$

where $A^{0,1}_{i\alpha} = \int_{K_0}\frac{\partial \Phi^1_{i_1}[\beta_1](X)}{\partial X_{\alpha_1}}\frac{\partial \Phi^2_{i_2}[\beta_1](X)}{\partial X_{\alpha_2}}\,\mathrm{d}X$ and $G^\alpha_{K,1} = \frac{1}{4}\det F'_K\frac{\partial X_{\alpha_1}}{\partial x_{\beta_2}}\frac{\partial X_{\alpha_2}}{\partial x_{\beta_2}}$. Here, each reference tensor $A^{0,k}$ is a rank four tensor and each geometry tensor $G_{K,k}$ is a rank two tensor (one index for each derivative).

3.4 Extensions

The current version of FFC supports affinely mapping function spaces. It is interesting to consider the generalization of our approach to other kinds of function spaces such as Raviart-Thomas [Raviart and Thomas 1977] elements for $H(\mathrm{div})$ and curvilinear mappings such as arise in isoparametric elements.

Implementing $H(\mathrm{div})$ or $H(\mathrm{curl})$ elements requires two kinds of generalizations to FFC. First of all, the basis functions are mapped to the reference element by the Piola transform [Brezzi and Fortin 1991] rather than the standard change of coordinates. With $F_K : K_0 \to K$ the standard affine mapping for an element K, F'_K the Fréchet derivative of the mapping and $\det F'_K$ its determinant, the Piola mapping is defined by $\mathcal{F}_K(\Psi) = \frac{1}{\det F'_K}F'_K(\Psi \circ (F_K)^{-1})$. Since our tools already track Jacobians and determinants for differentiating through affine mappings, it should be straightforward to support the Piola mapping. Second, defining the mapping between local and global degrees of freedom becomes more complicated, as we must contend with the directions of vector components.

As an example of using the Piola transform, we consider the Raviart-Thomas elements of the lowest order with the standard (vector-valued) nodal basis $\{\Psi_i\}_{i=1}^d$ on the reference element. We compute the mass matrix M with $M_{i_1 i_2} = a(\psi_{i_1}, \psi_{i_2})$ and the bilinear form a given by

$$a(v,u) = \int_\Omega v(x)\cdot u(x)\,\mathrm{d}x. \tag{22}$$

On K, the basis functions are given by $\psi_i^K = \mathcal{F}_K(\Psi_i)$ and computing the element tensor A^K, we obtain

$$\begin{aligned} A_i^K &= \int_K \psi_{i_1}^K(x) \cdot \psi_{i_2}^K(x) \, \mathrm{d}x \\ &= \frac{1}{\det F_K'} \frac{\partial x_\beta}{\partial X_{\alpha_1}} \frac{\partial x_\beta}{\partial X_{\alpha_2}} \int_{K_0} \Psi_{i_1}[\alpha_1] \Psi_{i_2}[\alpha_2] \, \mathrm{d}X = A^0_{i\alpha} G_K^\alpha, \end{aligned} \tag{23}$$

where $A^0_{i\alpha} = \int_{K_0} \Psi_{i_1}[\alpha_1] \Psi_{i_2}[\alpha_2] \, \mathrm{d}X$ and $G_K^\alpha = \frac{1}{\det F_K'} \frac{\partial x_\beta}{\partial X_{\alpha_1}} \frac{\partial x_\beta}{\partial X_{\alpha_2}}$. We see that, like for Poisson, the reference tensor A^0 is rank four and the geometry tensor G_K is rank two.

The possibility of extending our techniques to isoparametric elements is less obvious. While the Piola mapping is more complicated than the standard affine mapping, it still leads to geometric data that is constant over each cell and can be pulled out of the integrals to set up tensor products. In the case of isoparametric elements, however, F_K varies spatially within a cell. Moreover, the determinant factor often appears in a denominator, leading to a rational function. It is not clear how to proceed, but it may be possible to make suitably accurate polynomial approximations and hence recover the structure of tensor products.

3.5 Optimization

We consider here three different kinds of optimization that could be built into FFC in the future. For one, the current code is generated entirely straightline as a sequence of arithmetic and assignment. It should is possible to store the tensor A^0 in a contiguous array. Moreover, each G_K may be considered as a tensor or flattened into a vector. In the latter case, the action of forming the element matrix for one element may be written as a matrix-vector multiply using the level 2 BLAS. Once this observation is made, it is straightforward to see that we could form several G_K vectors and make better use of cache by computing several element matrices at once by a matrix-matrix multiply and the level 3 BLAS.

This corresponds to a coarse-grained optimization. In other work [Kirby et al. 2005; Kirby et al. 2005], we have seen that for many forms, the entries in A^0 are related in such ways that various entries of the element matrices may be formed in fewer operations. For example, if two entries of A^0 are close together in Hamming distance, then the contraction of one entry with G_K can be computed efficiently from the other. As our code for optimization, FErari, evolves, we will integrate it with FFC as an optimizing backend. It will be simple to compare the output of FErari to the best performance using the BLAS, and let FFC output the best of the two (which may be highly problem-dependent).

Finally, optimizations that arise from the variational form itself will be fruitful to explore in the future. For example, it should be possible to detect when a variational form is symmetric within FFC, as this leads to fewer operations to form the associated matrix. Moreover, for forms over vector-valued elements that have a Cartesian product basis (each basis function has support in only one component), other kinds of optimizations are appropriate. For example, the viscosity operator for Navier-Stokes is the vector Laplacian, which can be written as a block-diagonal matrix with one axis for each spatial dimension. By "taking apart" the basis functions, we hope to uncover this block structure, which will lead to more efficient compilation and hopefully more efficient code.

4. COMPLEXITY OF FORM EVALUATION

We now compare the proposed algorithm based on tensor representation to the standard quadrature-based approach. As we shall see, tensor representation can be much more efficient than quadrature for a wide range of forms.

4.1 Basic assumptions and notation

To analyze the complexity of form evaluation, we make the following simplifying assumptions:

—the form is bilinear, i.e., $r = |i| = 2$;

—the form can be represented as one tensor product, i.e., $A^K_i = A^0_{i\alpha} G^\alpha_K$;

—the basis functions are scalar;

—integrals are computed exactly, i.e., the order of the quadrature rule must match the polynomial order of the integrand.

We shall use the following notation: q is the polynomial order of the basis functions on every element, p is the total polynomial order of the integrand of the form, d is the dimension of Ω, n is the dimension of the function space on an element, and N is the number of quadrature points needed to integrate polynomials of degree p exactly.

Furthermore, let n_f be the number of functions appearing in the form. For test cases 1–4 above, $n_f = 0$ in all cases except test case 3 (Navier–Stokes) where $n_f = 1$. We use n_D to denote the number of differential operators. For test cases 1–4, we have $n_D = 0$ in case 1 (the mass matrix), $n_D = 2$ in case 2 (Poisson), $n_D = 1$ in case 3 (Navier–Stokes), and $n_D = 2$ in case 4 (linear elasticity).

4.2 Complexity of tensor representation

The element tensor A^K has n^2 entries. The number of basis functions n for polynomials of degree q in d dimensions is $\sim q^d$. To compute each entry A^K_i of the element tensor A^K using tensor representation, we need to compute the tensor product between $A^0_{i\cdot}$ and G_K. The geometry tensor G_K has rank $n_f + n_D$ and the number entries of G_K is $n^{n_f} d^{n_D}$. The cost for computing the n^2 entries of the element tensor A^K using tensor representation is thus

$$T_T \sim n^2 n^{n_f} d^{n_D} \sim (q^d)^2 (q^d)^{n_f} d^{n_D} \sim q^{2d+n_f d} d^{n_D}. \tag{24}$$

Note that there is no run-time cost associated with computing the tensor representation (8), since this is computed at compile-time. Also note that we have not taken into account any of the optimizations discussed in Section 3.5. These optimizations can in some cases significantly reduce the operation count.

4.3 Complexity of quadrature

To compute each entry A^K_i of the element tensor A^K using quadrature, we need to evaluate an integrand of total order p at N quadrature points. The number of quadrature points needed to integrate polynomials of order p exactly in d dimensions is $N \sim p^d$. Since the form is bilinear with basis functions of order q, the total order is $p = 2q + n_f q - n_D$. It is difficult to estimate precisely the cost of evaluating the integrand at each quadrature point, but a reasonable estimate is $n_f + n_D d + 1$. Note that we assume that all basis functions and their derivatives have been pretabulated at all quadrature points on the reference element.

We thus obtain the following estimate of the total cost for computing the n^2 entries of the element tensor A^K using quadrature:

$$\begin{aligned} T_Q &\sim n^2 N(n_f + n_D d + 1) \sim (q^d)^2 p^d (n_f + n_D d + 1) \\ &\sim q^{2d}(2q + n_f q - n_D)^d (n_f + n_D d + 1). \end{aligned} \tag{25}$$

4.4 Tensor representation vs. Quadrature

Comparing tensor representation with quadrature, the speedup of tensor representation is

$$T_Q/T_T \sim \frac{(2q + n_f q - n_D)^d (n_f + n_D d + 1)}{q^{n_f d} d^{n_D}}. \tag{26}$$

We immediately note that there can be a significant speedup for $n_f = 0$, since T_T/n^2 is then independent of the polynomial degree q. In particular, we note that for the mass matrix ($n_f = n_D = 0$) the speedup is $T_Q/T_T \sim (2q)^d$, and for Poisson's equation ($n_f = 0$, $n_D = 2$) the speedup is $T_Q/T_T \sim (2q-2)^d(2d+1)/d^2$. As we shall see below, the speedup for test cases 1–4 is significant even for $q = 0$.

On the other hand, we note that quadrature may be more efficient if n_f is large. One may thus imagine an intelligent system that automatically makes the choice between tensor representation and quadrature in each specific situation.

5. IMPLEMENTATION

We now discuss a number of important aspects of the implementation of the form compiler FFC. We also write down the forms for the test cases discussed above in Section 3.3 in the language of the form compiler FFC. Basically, we can consider FFC's functionality broken into three phases. First, it takes an expression for a multilinear form and generates the tensor A^0. While doing this, it derives an expression for evaluation of the element tensor G_K from the affine mapping and the coefficients of the form. Finally, it generates code for evaluating G_K and contracting it with A^0, and for constructing the local-to-global mapping.

5.1 Parsing of forms

The form compiler FFC implements a domain-specific language (DSL) for variational forms, using Python as the host language. A language of variational forms is obtained by overloading the appropriate operators, including addition `+`, subtraction `-`, multiplication (`*`), and differentiation `.dx(·)` for a hierarchy of classes corresponding to the algebra $\mathcal{A}$ discussed above in Section 3.2. FFC thus uses the built-in parser of Python to process variational forms.

5.2 Generation of the tensor representation

The basic elements of the algebra are objects of the class `BasisFunction`, representing (derivatives of) basis functions of some finite element space. Each `BasisFunction` is associated with a particular finite element space and different `BasisFunctions` may be associated with different finite element spaces. Products of scalars and (derivatives of) basis functions are represented by the class `Product`, and sums of such products are represented by the class `Sum`. In addition, we include a class `Function`, representing linear combinations of basis functions (coefficients). In the diagrams of Tables I and II, we summarize the basic unary and binary operators respectively implemented for the hierarchy of classes.

Note that by declaring a common base class for `BasisFunction`, `Product`, `Sum`, and `Function`, some of the operations can be grouped together to simplify the implementation. As a results, the result of most operators will directly yield a `Sum`. Also note that the algebra of `Sums` is closed under the operations listed above.

op	B	F	P	S
-	P	S	P	S
.dx(·)	P	S	S	S

Table I. Unary operators and their results for the classes `BasisFunction` (B), `Function` (F), `Product` (P), and `Sum` (S).

+/-	B	F	P	S
B	S	S	S	S
F	S	S	S	S
P	S	S	S	S
S	S	S	S	S

*	B	F	P	S
B	P	S	P	S
F	S	S	S	S
P	P	S	P	S
S	S	S	S	S

Table II. Binary operators and their results for the classes `BasisFunction` (B), `Function` (F), `Product` (P), and `Sum` (S).

By associating with each object one or more *indices*, implemented by the class `Index`, an object of type `Sum` automatically represents a tensor, and by differentiating between different types of indices, an object of type `Sum` automatically encodes the tensor representation (8). FFC differentiates between four different types of indices: *primary*, *secondary*, *auxiliary*, and *fixed*. A primary index (i) is associated with the multiindex of the element tensor A^K, a secondary index (α) is associated with the multiindex of the geometry tensor G_K, and thus the secondary indices indicate along which dimensions to compute the tensor product $A^0_{i\alpha}G^{\alpha}_K$. Auxiliary indices ($\beta$) are internal indices within the reference tensor A^0 or the geometry tensor G_K and must be repeated exactly twice; summation is performed over each auxiliary index β before the tensor product is computed by summation over secondary indices α. Finally, a fixed index is a given constant index that cannot be evaluated. Fixed indices are used to represent for example a derivative in a fixed coordinate direction.

Implicit in our algebra is a *grammar* for multilinear forms. We could explicitly write an EBNF grammar and use tools such as `lex` and `yacc` to create a compiler for a domain-specific language. However, by limiting ourselves to overloaded operators, we successfully embed our language as a high-level library in Python.

To make this concrete, consider test case 2 of section 3.3, Poisson's equation. The tensor representation $A_i^K = A^0_{i\alpha} G_K^\alpha$ is then given by

$$\begin{aligned} A^0_{i\alpha} &= \int_{K_0} \frac{\partial \Phi^1_{i_1}(X)}{\partial X_{\alpha_1}} \frac{\partial \Phi^2_{i_2}(X)}{\partial X_{\alpha_2}} \,\mathrm{d}X, \\ G_K^\alpha &= \det F_K' \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta}. \end{aligned} \tag{27}$$

There are here two primary indices (i_1 and i_2), two secondary indices (α_1 and α_2), and one auxiliary index (β).

5.3 Evaluation of integrals

Once the tensor representation (8) has been generated, FFC computes all entries of the reference tensor(s) by quadrature on the reference element. The quadrature rule is automatically chosen to match the polynomial order of each integrand. FFC uses FIAT [Kirby 2004; 2005] as the finite element back-end; FIAT generates the set of basis functions, the quadrature rule, and evaluates the basis functions and their derivatives at the quadrature points.

Although FIAT supports many families of finite elements, the current version of FFC only supports general order continuous/discontinuous Lagrange finite elements and first order Crouzeix–Raviart finite elements on triangles and on tetrahedra (or any other finite element with nodes given by pointwise evaluation). Support for other families of finite elements will be added in future versions.

Computing integrals is the most expensive step in the compilation of a form. The typical run-time (of the compiler) ranges between 0.1 and 30 seconds, depending on the type of form and finite element.

5.4 Generation of code

When a form has been parsed, the tensor representation has been generated, and all integrals computed, code is generated for evaluation of geometry tensors and tensor products. The form compiler FFC has been designed to allow for generation of code in multiple different languages. Code is generated according to a specific *format* (which is essentially a Python dictionary) that controls the output code being generated, see Figure 5. The current version of FFC supports three output formats: C++ (DOLFIN [Hoffman et al. ; Hoffman and Logg 2002]), LaTeX (for verification and presentation purposes) and a raw format that just lists the values of the reference tensors. FFC can be easily extended with new output formats, including for example Python, C, or Fortran.

5.5 Input/output

FFC can be used either as a Python package or from the command-line. We here give a brief description of how FFC can be called from the command-line to generate C++ code for DOLFIN. To use FFC from the command-line, one specifies the form in a text file in a special language for variational forms, which is simply Python with equipped with the hierarchy of classes and operators discussed above in Section 5.1. In Table III we give the complete code for the specification of test case 2, Poisson's equation. Note that FFC uses tensor-notation, and thus the summation over the

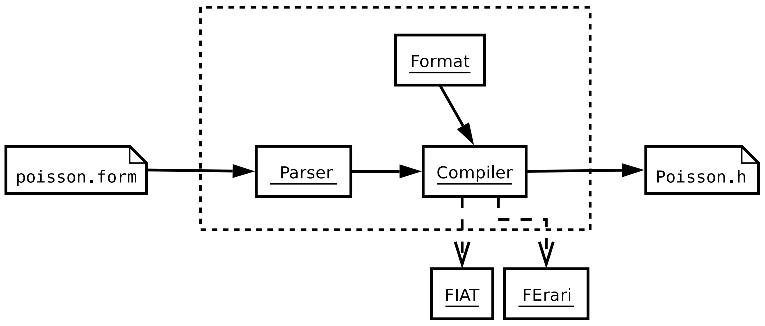

Fig. 5. Diagram of the components of the form compiler FFC.

index `i` is implicit. Also note that the integral over an element K is denoted by `*dx`.

```
element = FiniteElement("Lagrange", "tetrahedron", 3)

v = BasisFunction(element)
u = BasisFunction(element)
f = Function(element)

a = v.dx(i)*u.dx(i)*dx
L = v*f*dx
```

Table III. The complete code for specification of test case 2, Poisson's equation, with piecewise cubics on tetrahedra in the language of FFC.

Assuming that the form has been specified in the file `Poisson.form`, the form can be compiled using the command `ffc Poisson.form`. This generates the C++ file `Poisson.h` to be included in a DOLFIN program. In Table IV, we include part of the output generated by FFC with input given by the code from Table III. In addition to this code, FFC generates code for the mapping $\iota(\cdot,\cdot)$ from local to global degrees of freedom for each finite element space associated with the form. Note that the values of the 10×10 element tensor A^K (in the case of cubics on triangles) are stored as one contiguous array (`block`), since this is what the linear algebra back-end of DOLFIN (PETSc) requires for assembly.

5.6 Completing the toolchain

With the FEniCS project [Hoffman et al.], we have the beginnings of a working system realizing (in part) the Automation of Computational Mathematical Modeling, and the form compiler FFC is just one of several components needed to complete the toolchain. FIAT automates the generation of finite element spaces and FFC automates the evaluation of variational forms. Furthermore, PETSc [Balay et al. 2001; Balay et al. 2004; Balay et al. 1997], automating the solution of discrete sys-

```
void eval(double block[], const AffineMap& map) const
{
  // Compute geometry tensors
  double G0_0_0 = map.det*(map.g00*map.g00 + map.g01*map.g01);
  double G0_0_1 = map.det*(map.g00*map.g10 + map.g01*map.g11);
  double G0_1_0 = map.det*(map.g10*map.g00 + map.g11*map.g01);
  double G0_1_1 = map.det*(map.g10*map.g10 + map.g11*map.g11);

  // Compute element tensor
  block[0]  =  4.249999999999996e-01*G0_0_0 + 4.249999999999995e-01*G0_0_1 +
               4.249999999999995e-01*G0_1_0 + 4.249999999999995e-01*G0_1_1;
  block[1]  = -8.749999999999993e-02*G0_0_0 - 8.749999999999995e-02*G0_0_1;
  block[2]  = -8.750000000000005e-02*G0_1_0 - 8.750000000000013e-02*G0_1_1;
  ...
  block[99] =  4.049999999999997e+00*G0_0_0 + 2.024999999999998e+00*G0_0_1 +
               2.024999999999998e+00*G0_1_0 + 4.049999999999995e+00*G0_1_1;
}
```

Table IV. Part of the code generated by FFC for the input code from Table III.

tems, is used as the solver back-end of FEniCS. A common C++ interface to the different FEniCS components is provided by DOLFIN.

A complete automation of CMM, as outlined in [Logg 2004], is a major task and we hope that by a modular approach we can contribute to this automation.

6. BENCHMARK RESULTS

As noted above, the speedup for the code generated by the form compiler FFC can in many cases be significant. Below, we present a comparison with the standard quadrature-based approach for the test cases discussed above in Section 3.3.

The forms were compiled for a range of polynomial degrees using FFC version 0.1.6. This version of FFC does not take into account any of the optimizations discussed in Section 3.5, other than not generating code for multiplication with zero entries of the reference tensor.

For the quadrature-based code, all basis functions and their derivatives were pretabulated at the quadrature points using FIAT. Loops for all scalar products were completely unrolled.

In all cases, we have used the "collapsed-coordinate" Gauss-Jacobi rules described by Karniadakis and Sherwin [Karniadakis and Sherwin 1999]. These take tensor-product Gaussian integration rules over the square and cube and map them to the reference simplex. These rules are not the best known (see, for example, [Dunavant 1985]), but they are fairly simple to generate for arbitrary degree. Eventually, these rules will be integrated with FIAT, but even if we reduce the number of quadrature points by a factor of five, FFC still outperforms quadrature.

The codes were compiled with gcc (g++) version 3.3.6 and the benchmark results presented below were obtained on an Intel Pentium 4 (CPU 3.0 GHz, 2GB RAM) running Debian GNU/Linux. The times reported are for the computation of each entry of the element tensor on one million elements (scaled). The total time can be obtained by multiplying with n^2, the number of entries of the element tensor. The complete source-code for the benchmarks can be obtained from the FEniCS web site [Hoffman et al.].

6.1 Summary of results

In Table V, we summarize the results for test cases 1–4. In all cases, the speedup T_Q/T_T is significant, ranging between a factor 10–1500.

From Section 4, we know that the speedup for the mass matrix should grow as q^d, but from Table V it is clear that the speedup is not quadratic for $d = 2$ and for $d = 3$, an optimum seems to be reached around $q = 8$.

The reason that the predicted speedups are not obtained in practice is that the complexity estimates presented in Section 4 only account for the number of floating-point operations. When the polynomial degree q grows, the number of lines of code generated by the form compiler grows. FFC unrolls all loops and generates one line of code for each entry of the element tensor to be computed. For a bilinear form, the number of entries is $n^2 \sim q^{2d}$. With $q = 8$, the number of lines of code generated is about $25,000$ for the mass matrix and Poisson in 3D, see Figure 6. As the number of lines of code grows, memory access becomes more important and dominates the run-time. Using BLAS to compute tensor products as discussed above might lead to more efficient memory traffic.

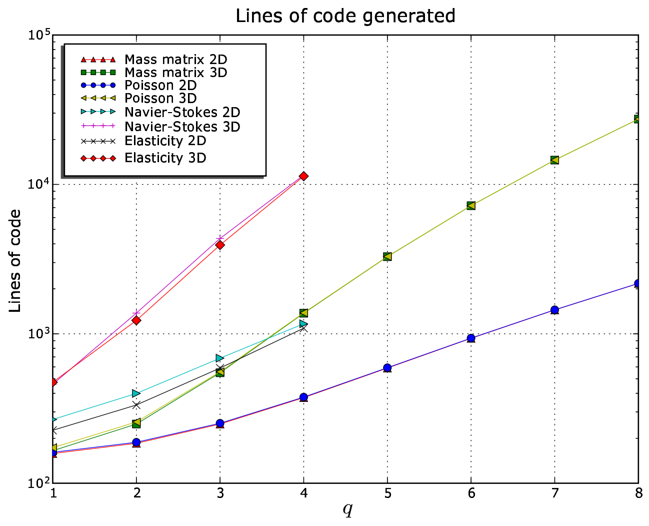

Fig. 6. Lines of code generated by the form compiler FFC as function of the polynomial degree q.

Note however that although the optimal speedup is not obtained, the speedup is in all cases significant, even at $q = 1$.

Form	$q=1$	$q=2$	$q=3$	$q=4$	$q=5$	$q=6$	$q=7$	$q=8$
Mass 2D	12	31	50	78	108	147	183	232
Mass 3D	21	81	189	355	616	881	1442	1475
Poisson 2D	8	29	56	86	129	144	189	236
Poisson 3D	9	56	143	259	427	341	285	356
Navier–Stokes 2D	32	33	53	37	—	—	—	—
Navier–Stokes 3D	77	100	61	42	—	—	—	—
Elasticity 2D	10	43	67	97	—	—	—	—
Elasticity 3D	14	87	103	134	—	—	—	—

Table V. Speedups T_Q/T_T for test cases 1–4 in 2D and 3D.

6.2 Results for test cases

In Figures 7–10, we present the results for test cases 1–4 discussed above in Section 3.3. In connection to each of the results, we include the specification of the form in the language used by the form compiler FFC.

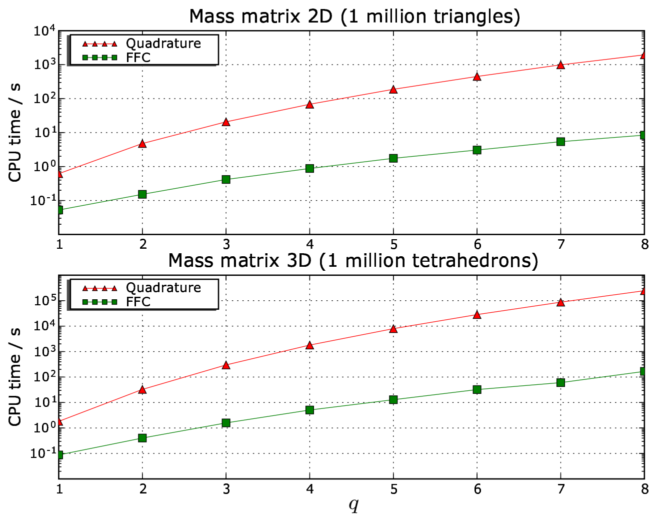

Fig. 7. Benchmark results for test case 1, the mass matrix, specified in FFC by `a = v*u*dx`.

7. CONCLUDING REMARKS AND FUTURE DIRECTIONS

We have demonstrated a proof-of-concept form compiler that for a wide range of variational forms can generate code that gives significant speedups compared to the standard quadrature-based approach.

The form compiler FFC is still in its early stages of development but is already in production use. A number of basic modules based on FFC have been implemented

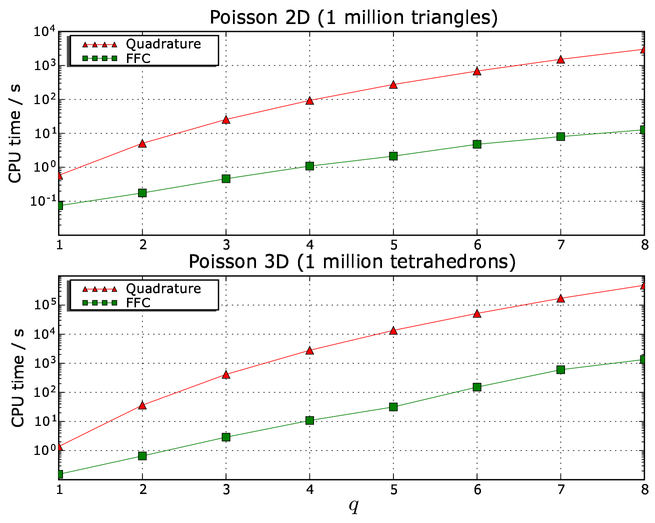

Fig. 8. Benchmark results for test case 2, Poisson's equation, specified in FFC by `a = v.dx(i)*u.dx(i)*dx`.

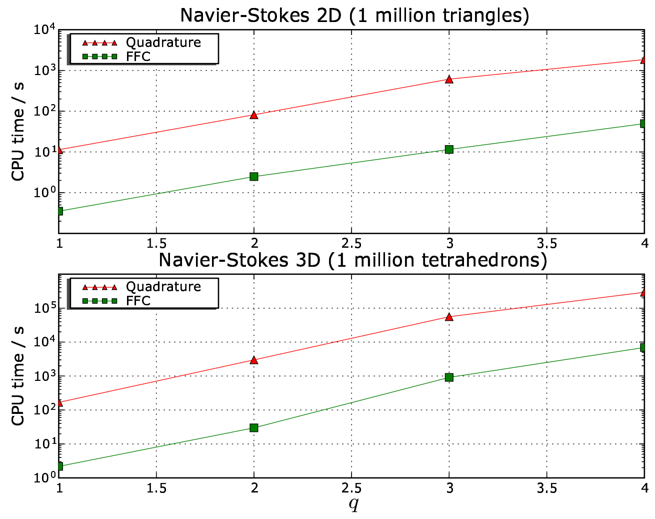

Fig. 9. Benchmark results for test case 3, the nonlinear term of the incompressible Navier–Stokes equations, specified in FFC by `a = v[i]*w[j]*u[i].dx(j)*dx`.

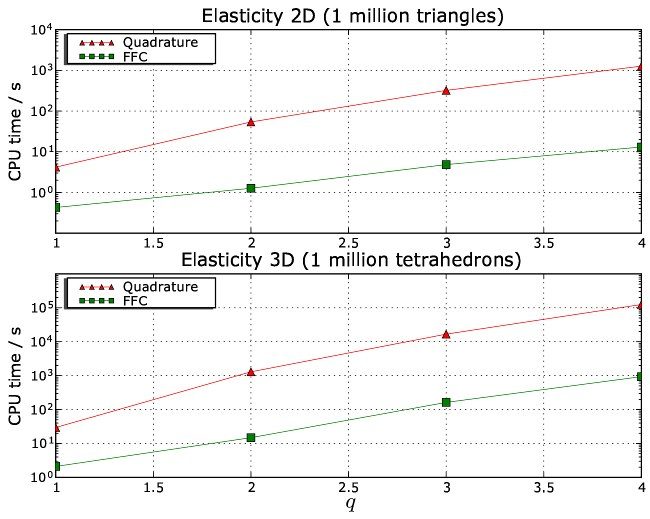

Fig. 10. Benchmark results for test case 4, the strain-strain term of linear elasticity, specified in FFC by `a = 0.25*(v[i].dx(j) + v[j].dx(i)) * (u[i].dx(j) + u[j].dx(i)) * dx`.

in DOLFIN and others are currently being developed (Navier–Stokes and updated elasticity). This will serve as a test bed for future development of FFC.

Future plans for FFC include adding support for integrals over the boundary (adding the operator `*ds` to the language), support for automatic differentiation of nonlinear forms and automatic generation of dual problems and a posteriori error estimators [Eriksson et al. 1995; Becker and Rannacher 2001], optimization through FErari [Kirby et al. 2005; Kirby et al. 2005], adding support for new families of finite elements, including elements that require non-affine mappings from the reference element. In addition to general order continuous/discontinuous Lagrange finite elements and Crouzeix–Raviart [Crouzeix and Raviart 1973] finite elements, the plan is to add support for Raviart–Thomas [Raviart and Thomas 1977], Nedelec [Nédélec 1980], Brezzi–Douglas–Marini [Brezzi et al. 1985], Brezzi–Douglas–Fortin–Marini [Brezzi and Fortin 1991], Arnold–Winther [Arnold and Winther 2002], and Taylor–Hood [Boffi 1997; Brenner and Scott 1994] elements.

We also plan to investigate the use of BLAS for evaluation of tensor products as an alternative to generating explicit unrolled code. Other topics of interest include automatic verification of the correctness of the code generated by the form compiler [Kirby et al.].

ACKNOWLEDGMENT

We wish to thank the FEniCS team, in particular Johan Hoffman, Johan Jansson, Claes Johnson, Matthew Knepley, and Ridgway Scott, for substantial suggestions and comments regarding this paper.

REFERENCES

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

Bagheri, B. and Scott, R. Analysa. `http://people.cs.uchicago.edu/~ridg/al/aa.html`.

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2004. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2001. PETSc. `http://www.mcs.anl.gov/petsc/`.

Balay, S., Eijkhout, V., Gropp, W. D., McInnes, L. C., and Smith, B. F. 1997. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 163–202.

Bangerth, W., Hartmann, R., and Kanschat, G. `deal.II` Differential Equations Analysis Library. `http://www.dealii.org`.

Becker, R. and Rannacher, R. 2001. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica 10.*

Boffi, D. 1997. Three-dimensional finite element methods for the Stokes problem. *SIAM J. Numer. Anal. 34,* 2, 664–670.

Brenner, S. C. and Scott, L. R. 1994. *The Mathematical Theory of Finite Element Methods.* Springer-Verlag.

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. 1976. *Numerical Analysis of the Finite Element Method.* Les Presses de l'Universite de Montreal.

Crouzeix, M. and Raviart, P. A. 1973. Conforming and nonconforming finite element methods for solving the stationary stokes equations. *RAIRO Anal. Numr. 7,* 33–76.

Dular, P. and Geuzaine, C. GetDP: a General environment for the treatment of Discrete Problems. `http://www.geuz.org/getdp/`.

Dunavant, D. A. 1985. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *Internat. J. Numer. Methods Engrg. 21,* 6, 1129–1148.

Eriksson, K., Estep, D., Hansbo, P., and Johnson, C. 1995. Introduction to adaptive methods for differential equations. *Acta Numerica*, 105–158.

Eriksson, K., Estep, D., Hansbo, P., and Johnson, C. 1996. *Computational Differential Equations.* Cambridge University Press.

Eriksson, K., Estep, D., and Johnson, C. 2003. *Applied Mathematics: Body and Soul.* Vol. III. Springer-Verlag.

Free Software Foundation. GNU GPL. `http://www.gnu.org/copyleft/gpl.html`.

Hoffman, J., Jansson, J., Johnson, C., Knepley, M., Kirby, R. C., Logg, A., and Scott, L. R. *FEniCS*. `http://www.fenics.org/`.

Hoffman, J., Jansson, J., and Logg, A. DOLFIN. `http://www.fenics.org/dolfin/`.

Hoffman, J. and Logg, A. 2002. DOLFIN: Dynamic Object oriented Library for FINite element computation. Tech. Rep. 2002–06, Chalmers Finite Element Center Preprint Series.

Hughes, T. J. R. 1987. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Prentice-Hall.

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. FIAT: A new paradigm for computing finite element basis functions. *ACM Trans. Math. Software 30*, 502–516.

Kirby, R. C. 2005. Optimizing FIAT with the level 3 BLAS. *submitted to ACM Trans. Math. Software*.

Kirby, R. C., Knepley, M., Logg, A., and Scott, L. R. 2005. Optimizing the evaluation of finite element matrices. *To appear in SIAM J. Sci. Comput.*.

Kirby, R. C., Knepley, M., and Scott, L. R. 2005. Evaluation of the action of finite element operators. *submitted to BIT*.

Kirby, R. C., Strout, M. M., Hovland, P., and Scott, L. R. Verification of scientific code using rationality analysis. *in preparation*.

Logg, A. 2004. Automation of computational mathematical modeling. Ph.D. thesis, Chalmers University of Technology, Sweden.

Long, K. Sundance. `http://csmr.ca.sandia.gov/~krlong/sundance.html`.

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.

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

Pironneau, O., Hecht, F., and Hyaric, A. L. FreeFEM. `http://www.freefem.org/`.

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.