Representations of finite element tensors via automated code generation

KRISTIAN B. ØLGAARD
Faculty of Civil Engineering and Geosciences
Delft University of Technology
and
GARTH N. WELLS
Department of Engineering
University of Cambridge

We examine aspects of the computation of finite element matrices and vectors which are made possible by automated code generation. Given a variational form in a syntax which resembles standard mathematical notation, the low-level computer code for building finite element tensors, typically matrices, vectors and scalars, can be generated automatically via a form compiler. In particular, the generation of code for computing finite element matrices using a quadrature approach is addressed. For quadrature representations, a number of optimisation strategies which are made possible by automated code generation are presented. The relative performance of two different automatically generated representations of finite element matrices is examined, with a particular emphasis on complicated variational forms. It is shown that approaches which perform best for simple forms are not tractable for more complicated problems in terms of run time performance, the time required to generate the code or the size of the generated code. The approach and optimisations elaborated here are effective for a range of variational forms.

Categories and Subject Descriptors: G.4 [**Mathematical software**]: ; G.1.8 [**Numerical analysis**]: Partial differential equations—*Finite element methods*; D.1.2 [**Programming techniques**]: Automatic Programming

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Finite element method, Code generation

1. INTRODUCTION

The rapid development of solvers for a variety of partial differential equations while achieving optimal or near-optimal run time performance is a possibility offered by automated computer code generation. Rapid development and high performance can be reconciled by introducing a compiler that translates high-level mathematical representations of variational forms into low-level computer code. The FEniCS Form Compiler (henceforth FFC) is an example of one such compiler [Logg et al. 2008; Kirby and Logg 2006]. FFC takes as input a variational form, posed in a high-level mathematical language, and generates code for the computation of the element tensors (element matrices, vectors or scalars) in a low-level language, such

K.B. Ølgaard, Faculty of Civil Engineering and Geosciences, Delft University of Technology Stevinweg 1, 2628 CN Delft, Netherlands. Email: k.b.oelgaard@tudelft.nl.
G.N. Wells, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom. Email: gnw20@cam.ac.uk.

as C++. The generated code serves as input to a finite element assembler. FFC version 0.5.1 generates C++ code consistent with the UFC specification [Alnæs et al. 2008; Alnæs et al. 2009], and therefore the generated code can be used by any assembly library which supports the specification. DOLFIN [Logg et al. 2008] is an example of an assembly library which supports the UFC specification. However, FFC is designed such that a user can implement support for output in any format and in any language.

The automated generation of computer code for finite element tensors provides scope for various representations and optimisations which are not feasible via conventional code development approaches. A possibility is to adopt a 'tensor contraction' representation of element tensors, rather than the classical quadrature-loop representation [Kirby and Logg 2006; Ølgaard et al. 2008]. The approach is based on the multiplicative decomposition of an element tensor into two tensors, one of which depends only on the differential equation and the chosen finite element bases and can therefore be computed prior to run time. It has been proved for classes of problems that the tensor contraction representation is more efficient than the traditional quadrature approach, and the speed-ups can be quite dramatic [Kirby and Logg 2006]. Furthermore, strategies which analyse the structure of the tensor contraction representation can yield improved performance [Kirby et al. 2005; Kirby et al. 2006].

It has been our experience that the tensor contraction approach does not scale well for moderately complicated and complicated forms. This is manifest in three ways: the time required to generate low-level code for a variational form becomes prohibitive or may fail due to memory limitations or limitations of underlying libraries; the size of the generated code is such that the compilation of the generated low-level code is prohibitively slow and file size limitations of compilers acting on the low-level code may be exceeded; and the run time performance deteriorates rapidly relative to a quadrature approach. Complicated forms are by no means exotic. Many common nonlinear equations, when linearised, result in forms which involve numerous function products. It was when addressing these types of problems that we found automated code generation using the tensor contraction representation would frequently break down. Approaches to reduce the time required for the code generation phase when using the tensor contraction representation have been developed and implemented in FFC [Kirby and Logg 2007], although these cannot counter the inherently expensive nature of the approach for complicated forms. Various issues with automated code generation, particularly scaling, are only borne out when considering complicated forms. Naturally, automated code generation is most appealing when considering complicated variational forms which are time consuming to program, difficult to optimise and problematic to debug.

We address here issues pertinent to automated code generation for quadrature representations of finite element tensors, and in particular optimisations which are made possible by automation and could not be reasonably expected of a developer to program 'by hand'. We wish in particular to target complicated forms, for which the tensor contraction approach performs poorly. In assessing the tensor contraction and quadrature representations, we consider

(1) The run time performance of the generated code;

(2) The size of generated code; and
(3) The speed of code generation phase.

The relative importance of these points may well shift during a development cycle. During initial development, it is likely that the speed of the code generation phase and the size of the generated code are most important, whereas at the end of the development cycle run time performance is likely to be the most crucial consideration. Inevitably, and as we will show, the three are typically linked.

All developments which we present are implemented in FFC, which is freely available at `http://www.fenics.org/` under the GNU Public License. FFC is a component of the FEniCS project [FEniCS 2008], which consists of a suite of tools which aim to automate computational mathematical modelling, all of which are released under a GNU public license. The examples presented in this work can be 'compiled' using FFC version 0.5.1.

The remainder of this work is arranged as follows. We summarise automated code generation and representations of finite element tensors in Section 2. Then, we describe in Section 3 the optimisations for quadrature representations which we have applied. Examples and benchmarks on the performance of quadrature and tensor contraction representations are presented in Section 4, after which conclusions are drawn in Section 5.

2. FINITE ELEMENT TENSORS AND AUTOMATED COMPUTER CODE GENERATION

We review briefly in this section two representations, quadrature and tensor contraction, of an element stiffness matrix. We choose as a canonical example the bilinear form corresponding to the weighted Laplace equation $-\nabla \cdot (w\nabla u) = 0$, where u is unknown and w is prescribed. The bilinear form associated with the variational form of the weighted Laplacian reads

$$a(v,u) = \int_\Omega w\nabla v \cdot \nabla u \,\mathrm{d}x. \tag{1}$$

We assume that all of the above functions come from the finite element space

$$V_h = \left\{ v \in H^1(\Omega) : \; v|_K \in P_k(K) \,\forall K \in \mathcal{T} \right\}, \tag{2}$$

where $P_k(K)$ denotes the space of polynomials of degree k on the element K of the standard triangulation of Ω, which is denoted by $\mathcal{T}$. The local element matrix, often known as the 'stiffness matrix', for the cell K is given by

$$A^K_{i_1 i_2} = \int_K w\nabla\phi^K_{i_1} \cdot \nabla\phi^K_{i_2} \,\mathrm{d}x, \tag{3}$$

where $\left\{\phi^K_i\right\}$ are the local basis functions which span V_h on the element K.

The task of the form compiler is to take an input which resembles the notation of equation (3) and generate low-level code. The FFC input for this problem is shown in Figure 1 for continuous piecewise cubic functions on tetrahedra as a basis for all functions in the form. Computer code is generated from the input shown in Figure 1 by simply running the compiler FFC on the input code. Options can be provided which affect aspects of the generated code. Relevant to the topic of this work are

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

v = TestFunction(element)
u = TrialFunction(element)
w = Function(element)

a = w*dot(grad(v), grad(u))*dx
```

Fig. 1. FFC input for the weighted Laplacian form on cubic tetrahedral elements.

the representation options '`-r quadrature`' for quadrature representation and '`-r tensor`' for tensor contraction representation, both of which are summarised in this section. FFC generates not only the code for computing the element matrix $A^K_{i_1 i_2}$, but also a degree-of-freedom mapping for use in assembly, as well as a number of utility functions, such as code for evaluating finite element functions at arbitrary points. The generated code conforms to the UFC specification, and can be used to assemble global matrices and vectors by an assembler which supports the UFC specification, such as the FEniCS component DOLFIN, which is a problem solving environment responsible for assembling the global system and solving the arising linear system of equations.

FFC implements support for basic differential and algebraic operators. The operators which are used in the examples in this work are: the gradient, `grad(v)`; the divergence, `div(v)`; and inner products `dot(v, w)`. Different types of integrals are also available. Presented examples will use integration over a cell, `*dx`, and integration over interior facets, `*dS`. The latter will be used in a discontinuous Galerkin example. Related to discontinuous Galerkin methods, the compiler offers the possibility of restricting functions evaluated on facets to either the plus side or the minus side of a given facet, which is expressed as `v('+')` and `v('-')`, respectively. Also relevant to discontinuous Galerkin methods are operators for the jump, `jump(v)`, and the average, `avg(v)`, of a function on a cell facet. Mixed elements with arbitrary combinations of functions and function spaces are also supported. FFC is built on top of Python, and therefore inherits Python syntax. This makes the addition of user-defined operators simple, and in combination with the language of FFC, makes it possible to define a wide range of variational forms simply and compactly.

2.1 Quadrature representation

Finite element codes typically deploy quadrature at run time for the numerical integration of local element tensors. Assuming the same local basis for all functions in equation (3) and a standard affine mapping $F_K : K_0 \to K$ from a reference element K_0 to any given element $K \in \mathcal{T}$ (recalling that $\mathcal{T}$ is the triangulation of the domain of interest Ω), a quadrature scheme reads

$$A^K_{i_1 i_2} = \sum_{q=1}^{N} \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\alpha_3=1}^{n} \Phi_{\alpha_3}(X^q) w_{\alpha_3} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial \Phi_{i_1}(X^q)}{\partial X_{\alpha_1}} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \Phi_{i_2}(X^q)}{\partial X_{\alpha_2}} \det F'_K W^q, \quad (4)$$

where a change of variables from the reference coordinates X to the real coordinates $x = F_K(X)$ has been used. In the above equation N denotes the number of integration points, d is the dimension of Ω, n is the number of degrees of freedom for the local basis of w, Φ_i denotes basis functions (shape functions) on the reference element, and W^q is the quadrature weight at integration point X^q.

When generating code automatically using FFC, exact quadrature is used by default. FFC computes the polynomial order of the form and uses a scheme based on the Gauss-Legendre-Jacobi rule mapped onto simplices [Kirby 2004]. This means that for exact integration of a second-order polynomial, FFC will use two quadrature points in each spatial direction i.e., $2^3 = 8$ points per cell in three dimensions. FFC does provide an option for a user to specify the number of quadrature points which then permits inexact quadrature.

2.2 Tensor contraction representation

In reviewing the tensor contraction representation approach, we follow the work of Kirby and Logg [2006]. Taking equation (4) as the point of departure, the tensor contraction representation of the element matrix for the weighted Laplacian is expressed by

$$A^K_{i_1 i_2} = \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\alpha_3=1}^{n} \det F'_K w_{\alpha_3} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \Phi_{\alpha_3} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, \mathrm{d}X. \quad (5)$$

Noteworthy is that the integral appearing in equation (5) is independent of the cell geometry and can therefore be evaluated prior to run time. The remaining terms, with the exception of w_{α_3}, depend only on the geometry of the cell. Exploiting this observation, the element tensor $A^K_{i_1 i_2}$ can then be expressed as a tensor contraction,

$$A^K_{i_1 i_2} = \sum_{\alpha} A^0_{i_1 i_2 \alpha} G^\alpha_K, \quad (6)$$

where the tensors $A^0_{i_1 i_2 \alpha}$ (the 'reference tensor') and G^α_K (the 'geometry tensor') are defined as

$$A^0_{i_1 i_2 \alpha} = \int_{K_0} \Phi_{\alpha_3} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, \mathrm{d}X, \quad (7)$$

$$G^\alpha_K = \det F'_K w_{\alpha_3} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta}. \quad (8)$$

We refer to Kirby and Logg [2007] for a generalisation of the approach.

Using FFC to generate computer code for the tensor contraction representation, the reference tensor $A^0_{i_1 i_2 \alpha}$ is precomputed and the contraction in equation (6) is unrolled. For a certain class of simple forms this can lead to a tremendous speed-up when evaluating the element matrices relative to a quadrature approach [Kirby and Logg 2006]. Note, however, that as the number of functions and derivatives present in the variational form increases, the rank of both the reference tensor and the geometry tensor increases, thereby increasing the complexity of the tensor contraction.

In contrast to the quadrature approach, the tensor contraction representation is somewhat specialised as it cannot be extended trivially to non-affine isoparametric mappings while maintaining efficiency, and it is not effective for classes of nonlinear problems which require the integration of functions that do not come from a finite element space. The attractive feature of the approach is the run time performance for classes of problems.

3. A PRIORI OPTIMISATIONS FOR QUADRATURE REPRESENTATION

The automated generation of code provides scope for employing optimisations which are not practically feasible in hand-generated code. An example of such an approach which is pertinent to the tensor contraction representation involves the analysis of structures in the reference tensor in order to minimise the number of floating point operations required to compute an element matrix or vector [Kirby et al. 2005; Kirby et al. 2006; Kirby and Logg 2008]. For simple problems, this can lead to a significant reduction in the number of operations required to compute an element tensor. However, it is our experience that one is generally not well-rewarded for sophisticated optimisation strategies. Such strategies may not scale well in terms of the required computer time to perform the optimisations for moderately complex variational forms, proving to be prohibitive in terms of time and memory. This is in conflict with the goal of minimising development time (code generation phase), as described in the Introduction. Our experience indicates that simple optimisations, some of which are described in this section, offer the greatest rewards, even to the extent that the cost of evaluating element tensors becomes negligible relative to other aspects of a computation, such as insertion of entries into a sparse matrix.

We outline here three simple *a priori* approaches for optimising generated code for the quadrature representation of an element tensor. The central idea of all three methods is to implement low-cost strategies to reduce the number of floating point operations required to evaluate the local element tensor. By low-cost optimisations, we imply strategies which do not impact the time required for the code generation phase adversely. The optimisations which we have implemented are:

(1) Tabulation of basis functions: Basis functions are evaluated and tabulated at integration points. In conventional codes, this evaluation is often performed at run time. From an efficiency point of view it is better to tabulate the values of the basis functions and look up the values when required. The tabulation of basis functions is possible by hand for a particular basis and a particular quadrature scheme and is often done in specialised codes. However, this is not practically possible by hand in a generic problem solving environment where a variety of bases and quadrature schemes are employed.

(2) Eliminate floating point operations on zeros: Basis functions and derivatives of basis functions that are zero-valued at all integration points may be identified and eliminated during the generation phase, thereby reducing the dimension of the loops concerning these functions. In particular, when taking derivatives of basis functions on a reference element zeros often appear. This requires creating a mapping of indices in order to correctly access the basis values. This mapping results in memory not being accessed contiguously at run time and can therefore potentially lead to a performance drop, but in our experience this effect is outweighed by the reduction

of operations.

(3) Optimise nested loops: A naive implementation of a quadrature representation of equation (4) in which the summations are replaced by loops results in a set of nested loops where the number of required operations increase exponentially with the number of loops. However, few terms in equation (4) are dependent on the summation index in each of the sums. For instance, the value of the function w at a given quadrature point is simply computed as $w(X^q) = \sum_{\alpha_3=1}^{n} \Phi(X^q)_{\alpha_3} w_{\alpha_3}$. This means that the value for each entry of the element tensor $A^K_{i_1 i_2}$ in equation (4), for each combination of α_1 and α_2, can be computed in three operations, namely a sum and two multiplications. This can be seen in the generated code for the weighted Laplacian form which is shown in Figure 2.

The optimisations described above take place at the final stage of the code generation process where any given form is represented as simple loop and algebra instructions. Therefore, the optimisations are general and apply to all forms and elements that can be handled by FFC.

The generated code for the weighted Laplacian form in Figure 2 demonstrates the three optimisations described above. The values of basis functions have been tabulated in the variables `Psi_w`, which is the basis for the function w, and `Psi_vu`, which contains derivatives of the basis for the test function v and the trial function u. A zero in the table `Psi_vu` has been eliminated, which reduces the size of the loops over i and j, corresponding to i_1 and i_2 in equation (4), from three to two. Note also that for each combination of α_1 and α_2 in equation (4), we can evaluate the expression using only three operations. Therefore, increasing the number of functions and derivatives in the form will in general not lead to an as dramatic increase in the form representation complexity compared to the tensor contraction representation, although additional functions might lead to an increase in the number of quadrature points needed for exact integration. While the above optimisations are straightforward for simple forms and elements, their implementation using conventional programming approaches requires manual inspection of the form and the basis. This is often done in specialised codes, but the extension to non-trivial forms is difficult, time consuming and error prone. Furthermore, optimised code often bears little relation to the mathematical problem at hand. This makes maintenance and re-use of the hand-generated code problematic.

Our early attempts at generating code for the quadrature representation employed only the tabulation of basis functions as an optimisation strategy and led to disappointing performance results, both in terms of run time performance and the time required for code generation. Adding a run time test for operations on zeros led to a performance increase, but also led to a significant increase in the time required for the C++ compilation of the generated code. *A priori* elimination of operations on zeroes yielded run time improvements and a significant reduction in the time required to compile the generated code. For complicated forms, it was the optimisation of loops that led to dramatic performance improvements in both the code generation time and the run time performance. With the optimisation of the loops, for complicated forms we have observed improvements in the run time performance of several orders of magnitude over automatically generated code which did not optimise the quadrature loops.

```
virtual void tabulate_tensor(double* A, const double* const* w,
                             const ufc::cell& c) const
{
  ...
  // Quadrature weight
  const static double W0 = 0.5;

  // Tabulated basis functions and arrays of non-zero columns
  const static double Psi_w[1][3] =\
                 {{0.33333333333, 0.33333333333, 0.33333333333}};
  const static double Psi_vu[1][2] = {{-1, 1}};
  static const unsigned int nzc0[2] = {0, 1};
  static const unsigned int nzc1[2] = {0, 2};

  // Geometry constants
  const double G0 = Jinv_00*Jinv_10*W0*det;
  const double G1 = Jinv_01*Jinv_11*W0*det;
  const double G2 = Jinv_00*Jinv_00*W0*det;
  const double G3 = Jinv_01*Jinv_01*W0*det;
  const double G4 = Jinv_10*Jinv_10*W0*det;
  const double G5 = Jinv_11*Jinv_11*W0*det;

  // Loop integration points
  for (unsigned int ip = 0; ip < 1; ip++)
  {
    // Compute function value
    double F0 = 0;
    for (unsigned int r = 0; r < 3; r++)
      F0 += Psi_w[ip][r]*w[0][r];
    const double Gip0 = (G0 + G1)*F0;
    const double Gip1 = (G2 + G3)*F0;
    const double Gip2 = (G4 + G5)*F0;
    for (unsigned int i = 0; i < 2; i++)
    {
      for (unsigned int j = 0; j < 2; j++)
      {
        A[nzc0[i]*3 + nzc0[j]] += Psi_vu[ip][i]*Psi_vu[ip][j]*Gip1;
        A[nzc0[i]*3 + nzc1[j]] += Psi_vu[ip][i]*Psi_vu[ip][j]*Gip0;
        A[nzc1[i]*3 + nzc0[j]] += Psi_vu[ip][i]*Psi_vu[ip][j]*Gip0;
        A[nzc1[i]*3 + nzc1[j]] += Psi_vu[ip][i]*Psi_vu[ip][j]*Gip2;
      }
    }
  }
}
```

Fig. 2. Part of the generated code for the weighted Laplacian using linear elements in two dimensions. The variables like `Jinv_00` are components of the inverse of the Jacobian matrix and `det` is the determinant of the Jacobian. `A` holds the values of the local element tensor and `w` contains nodal values of the weighting function w.

As stated in Section 2.1, a quadrature scheme based on the Gauss-Legendre-Jacobi rule is applied. A further optimisation would be to construct the quadrature rules directly for simplices, in which case the required number of integration points for exact quadrature could be reduced, although we recall that a user may specify the number of integration points to be used.

4. PERFORMANCE COMPARISONS

We compare now generated tensor contraction and quadrature-based code in terms of the metrics outlined in the Introduction, namely the run time performance, the size of generated code and the speed of the code generation phase. The aim is to elucidate features of the two representations for various problems with the goal of finding a guiding principle for selecting the most appropriate representation for a given problem.

We set the scene by first considering some typical forms of differing complexity and nature to illustrate some trends and differences between the representations. We then proceed with a systematic comparison using some very simple forms for which we expect the tensor contraction representation to prove superior, before increasing the complexity of the forms in order to investigate the cross-over point at which the quadrature representation becomes the better representation in terms of run time performance. Exact quadrature is used for all examples.

All tests were performed on an Intel Core 2 X6800 CPU at 2.93GHz with 3.2GB of RAM running Ubuntu 8.04.1 with Linux kernel 2.6.24. We used Python version 2.5.2 and NumPy version 1.0.4 (both pertinent to FFC), and g++ version 4.2.3 with the '-O2' optimisation flag to compile the generated C++ code. For tests which involve compressed sparse matrices, we use DOLFIN to assemble the global sparse matrix. DOLFIN provides various linear algebra backends, and we have used PETSc [Balay et al. 2001] as the backend for the assembly tests. The non-zero structure of the compressed sparse matrix is initialised and no special reordering of degrees of freedom has been used in the assembly tests.

4.1 Performance for a selection of forms

We set out by comparing the two representations to demonstrate the strengths and weaknesses for different 'real' forms. The first form considered is a mixed Poisson formulation using fifth-order Brezzi-Douglas-Marini (BDM) elements [Brezzi and Fortin 1991], automation aspects of which have been addressed by Rognes et al. [2008]. The FFC input for this form is shown in Figure 3. We also consider a discontinuous Galerkin method for the biharmonic equation [Ølgaard et al. 2008] which involves both cell and interior facet integrals, and is shown in Figure 4. The third example is a complicated form which has arisen in modelling temperature-dependent multiphase flow through porous media [Wells et al. 2008]. It involves standard simple Lagrange basis functions of low order but the products of many functions. The input for the form is shown in Figure 5. Due to the origins of this form, we denote it as the 'pressure equation'.

The three forms have been compiled with FFC using the tensor contraction and quadrature representations. In Table I, the time required to generate the code, the size of the generated code and the time required to compile the C++ code are reported for each form. Results are presented for the tensor contraction case, together

```
BDM = FiniteElement("Brezzi-Douglas-Marini", "triangle", 5)
DG  = FiniteElement("Discontinuous Lagrange", "triangle", 5 - 1)

mixed_element = BDM + DG

(tau, w)   = TestFunctions(mixed_element)
(sigma, u) = TrialFunctions(mixed_element)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma))*dx
```

Fig. 3. FFC input for the stiffness matrix of the mixed Poisson problem using BDM elements of order five.

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

v = TestFunction(element)
u = TrialFunction(element)

n = FacetNormal("triangle")
h = MeshSize("triangle")

alpha = 10.0

a = dot(div(grad(v)), div(grad(u)))*dx \
   - dot(avg(div(grad(v))), jump(grad(u), n))*dS \
   - dot(jump(grad(v), n), avg(div(grad(u))))*dS \
   + alpha/h('+')*dot(jump(grad(v),n), jump(grad(u),n))*dS
```

Fig. 4. FFC input for the stiffness matrix of a discontinuous Galerkin formulation for the biharmonic equation in two-dimensional elements of order three.

Table I. Timings and code size for the compilation phase for the various variational forms. 'generation' is the time required by FFC to generate the tensor contraction code; 'size' is the size of the generated tensor contraction code; and 'C++' is the time to compile the generated C++ code. The ratio q/t is the ratio between quadrature and tensor contraction representations.

Form	generation [s]	q/t	size [kB]	q/t	C++ [s]	q/t
mixed Poisson	3.9	1.00	1500	0.92	15.7	0.76
DG biharmonic	16.2	0.35	3200	0.13	47.5	0.19
pressure equation	35.1	1.03	2600	0.19	41.4	0.22

with the ratio of the time/size for the quadrature representation case divided by the time/size required for the tensor contraction representation case, denoted by q/t. In measuring the C++ compile time and the run time performance, the generated code has been compiled against the library DOLFIN. Noteworthy from the results in Table I is that the generation phase for the quadrature representation is at least as fast as the tensor contraction representation generation phase (the difference for

```
scalar_p = FiniteElement("Lagrange","triangle",2)
scalar   = FiniteElement("Lagrange","triangle",1)
dscalar  = FiniteElement("Discontinuous Lagrange","triangle",0)
vector   = VectorElement("Discontinuous Lagrange", "triangle", 1)

q   = TestFunction(scalar_p)
p   = TrialFunction(scalar_p)

f0  = Function(scalar)
f1  = Function(scalar)
f2  = Function(scalar)
f3  = Function(scalar)
f4  = Function(scalar)
f5  = Function(scalar)
f6  = Function(scalar)
f7  = Function(dscalar)
f8  = Function(dscalar)
f9  = Function(dscalar)
f10 = Function(dscalar)
f11 = Function(dscalar)
f12 = Function(dscalar)
f13 = Function(dscalar)
f14 = Function(dscalar)
f15 = Function(vector)
f16 = Function(vector)
f17 = Function(vector)

S0  = f14*f7*dot(f15, grad(q))
S1  = f14*f8*dot(f16, grad(q))
S2  = f14*f9*dot(f17, grad(q))
S   = S0 + S1 + S2

a0  = f10*f0*f9*1/f11*p*q*dx\
            - f7*(1-f12)*dot(f15, grad(p))*q*dx\
            - f8*(1-f12)*dot(f16, grad(p))*q*dx\
            - f9*(1-f12)*dot(f17, grad(p))*q*dx\
            - f13*f1/f2*(1-f12)*dot(grad(p),grad(q))*dx\
            - f13*f3/f4*(1-f12)*dot(grad(p), grad(q))*dx\
            - f13*f5/f6*(1-f12)*dot(grad(p), grad(q))*dx

a1  = f10*f0*f9*1/f11*p*S*dx\
        - f7*(1-f12)*dot(f15, grad(p))*S*dx\
        - f8*(1-f12)*dot(f16, grad(p))*S*dx\
        - f9*(1-f12)*dot(f17, grad(p))*S*dx\
        + f13*f1/f2*(1-f12)*div(grad(p))*S*dx\
        + f13*f3/f4*(1-f12)*div(grad(p))*S*dx\
        + f13*f5/f6*(1-f12)*div(grad(p))*S*dx

a   = a0 + a1
```

Fig. 5. FFC input for the 'pressure equation' in two dimensions.

Table II. Run time performance for the various variational forms.

Form	flops	q/t	run time [s]	q/t
mixed Poisson	12866	73.90	4.5	60.33
DG biharmonic	26420	1.19	25.0	1.27
pressure equation	160752	0.17	190.0	0.17

the pressure equation is only slight). Our experience is that the difference grows in favour of the quadrature representation for complicated forms. In all cases the size of the generated quadrature code is smaller than the tensor contraction code, which is reflected in the C++ compile time. The differences in the C++ compile time are substantial for the biharmonic and pressure equations (approximately a factor of five), which is important during the code development phase with frequent recompilations.

Timings and operation counts for the three forms are presented in Table II. We define the number of floating point operations (flops) as the sum of all '+' and '$*$' operators in the code for computing the element matrix. Although multiplications are generally more expensive than additions, this definition provides a good measure for the performance of the generated code. The compound operator '+=' is counted as one operation. For the run time performance, the time required to compute the element tensors N times is recorded. For the mixed Poisson problem $N = 4.5\times10^5$, for the discontinuous Galerkin biharmonic problem $N = 1\times10^6$ and for the pressure equation $N = 2.5\times10^6$. Table II presents the timings and operation counts for tensor contraction representation, together with the ratio of the quadrature representation case and the tensor contraction representation case, q/t. The run time performance is indicative of an aspect of the two representations; there can be significant performance difference depending on the nature of the differential equation. For the mixed Poisson problem, the tensor contraction representation is close to a factor of sixty faster than the quadrature representation, whereas for the pressure equation the quadrature representation is close to a factor of six faster than the tensor contraction case. This observation of dramatic difference in run time performance highlights the desirability of a strategy to determine the best representation, without generating the code for each case. Such concepts have been successfully developed in digital signal processing [Püschel et al. 2005].

4.2 Performance for common, simple forms

We consider now the performance of the two representations for two canonical examples: the scalar 'mass' matrix and the 'elasticity-like' stiffness matrix. The input for the mass matrix form is shown in Figure 6 and the input for the elasticity-like stiffness matrix is shown in Figure 7. The performance of the two representations are compared for two- and three-dimensional cases on simplicies and for various polynomial orders. Code is generated using FFC, and we report the number of floating point operations required to form the element matrix for all cases. In addition to reporting the number of floating point operations, the time required to compute the element matrix N times is also presented, which we expect in most cases to be strongly correlated to the floating point operations count. As before,

```
element = FiniteElement("Lagrange", "triangle", 2)

v = TestFunction(element)
u = TrialFunction(element)

a = dot(v, u)*dx
```

Fig. 6. FFC input for the mass matrix in two dimension with element order $q = 2$.

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

v = TestFunction(element)
u = TrialFunction(element)

def eps(v):
    return grad(v) + transp(grad(v))

a = 0.25*dot(eps(v), eps(u))*dx
```

Fig. 7. FFC input for the elasticity-like matrix in three dimensions with element order $q = 3$.

values are reported for the tensor contraction representation case together with the ratio of the quadrature value over the tensor contraction value. We also report the time required for insertion into a sparse matrix, which is independent of the element matrix representation. The total assembly time is the 'run time' plus the 'insertion' time, which provides a picture of the overall assembly performance. The ratio of the total assembly time for the quadrature representation over the total assembly time for the tensor contraction representation, denoted by $\mathrm{a_q}/\mathrm{a_t}$, is also presented. When taking this into account, for some forms the difference in performance between different representations appears less drastic.

The various timings for the mass matrix problem are reported in Table III for the two-dimensional case and in Table IV for the three-dimensional case. What is clear from these results is that tremendous speed-ups for computing the element matrices can be achieved using the tensor contraction representation, particularly as the element order is increased. This is perhaps not surprising considering that the geometry tensor for this case is simply a scalar, therefore the entire matrix is essentially precomputed. The speed-up is mitigated, however, by the time required to insert terms into a sparse matrix. For the case of $q = 4$ in three dimensions, the tensor contraction representation is a factor of 358 faster for computing the element matrix, but when insertion is included an overall speed-up factor of only 2.76 is observed. A factor of 2.76 is not trivial, but obviously to reap the full benefits of the tensor contraction approach for these types of problems, matrix insertion must be addressed. The various timings for the elasticity-like stiffness matrix are presented in Table V for the two-dimensional case and in Table VI for the three-dimensional case. Compared to the mass matrix, the differences in performance of the tensor contraction representation relative to quadrature representation are less

Table III. Timings for the mass matrix in two dimensions for varying polynomial order basis q.

	flops	q/t	run time [s]	q/t	insertion [s]	a_q/a_t
$q = 1$ $(N = 1 \times 10^7)$	10	11	0.13	6	8	1.07
$q = 2$ $(N = 1 \times 10^7)$	25	39	0.27	19	23	1.21
$q = 3$ $(N = 1 \times 10^7)$	89	54	0.67	37	59	1.40
$q = 4$ $(N = 1 \times 10^6)$	214	79	0.12	71	13	1.64
$q = 5$ $(N = 1 \times 10^6)$	442	108	0.20	112	24	1.91

Table IV. Timings for the mass matrix in three dimensions for varying polynomial order basis q.

	flops	q/t	run time [s]	q/t	insertion [s]	a_q/a_t
$q = 1$ $(N = 1 \times 10^7)$	17	23	0.31	8	11	1.19
$q = 2$ $(N = 1 \times 10^7)$	101	80	0.65	60	72	1.52
$q = 3$ $(N = 1 \times 10^6)$	281	273	0.18	202	34	2.06
$q = 4$ $(N = 1 \times 10^6)$	1226	375	0.60	358	121	2.76

Table V. Timings for the elasticity-like matrix in two dimensions for varying polynomial order basis q.

	flops	q/t	run time [s]	q/t	insertion [s]	a_q/a_t
$q = 1$ $(N = 1 \times 10^7)$	236	1.1	0.64	3	19	1.06
$q = 2$ $(N = 1 \times 10^7)$	728	7	1.65	17	100	1.25
$q = 3$ $(N = 1 \times 10^6)$	2728	13	0.69	33	29	1.74
$q = 4$ $(N = 1 \times 10^5)$	7724	20	0.41	22	7	2.16

dramatic, but nonetheless substantial, especially for higher-order functions in three dimensions.

4.3 Performance for forms of increasing complexity

The complexity of the forms investigated in the previous section is now increased in order to examine under which circumstances the quadrature representation will be more favourable in terms of run time performance. The comparison is based on the floating point operation count as this is a good indicator of performance and the size of the generated file for a large class of problems. We consider the 'complexity' of a variational form to increase when the number of function products increases and when the number of derivatives present increases. Increasing the number of derivatives and/or the numbers of functions appearing in a form leads to higher rank tensors for the tensor contraction representation. Also, increases in the polynomial order of the basis of a function leads to an increase in complexity of the geometry tensor. We initially restrict ourselves to manipulating the number of function multiplications in the forms and the polynomial order of these functions, before introducing products of derivatives.

To generate forms of greater complexity than those in the previous section, we take the mass matrix and elasticity-like problems with a Lagrange basis of order q, and premultiply the forms with n_f functions of order p. An example is shown in Figure 8 for the mass matrix where $q = 2$, $n_f = 2$ and $p = 3$. A comparison of the representations for the mass matrix with a different number of premultiplying

Table VI. Timings for the elasticity-like matrix in three dimensions for varying polynomial order basis q.

	flops	q/t	run time [s]	q/t	insertion [s]	a_q/a_t
$q = 1$ ($N = 1 \times 10^6$)	1098	1.1	0.27	3	8	1.07
$q = 2$ ($N = 1 \times 10^5$)	8622	11	0.36	15	9	1.54
$q = 3$ ($N = 1 \times 10^4$)	44010	38	0.19	60	6	2.81
$q = 4$ ($N = 1 \times 10^3$)	155529	90	0.08	115	2	5.38

```
element   = FiniteElement("Lagrange", "triangle", 2)
element_f = FiniteElement("Lagrange", "triangle", 3)

v = TestFunction(element)
u = TrialFunction(element)

f = Function(element_f)
g = Function(element_f)

a = f*g*dot(v, u)*dx
```

Fig. 8. FFC input for the mass matrix in two dimension with with $q = 2$, premultiplied by two functions ($n_f = 2$) of order $p = 3$.

Table VII. The number of operations and the ratio between number of operations for the two representations for the mass matrix in two dimensions as a function of different polynomial orders and numbers of functions.

	$n_f = 1$		$n_f = 2$		$n_f = 3$		$n_f = 4$	
	flops	q/t	flops	q/t	flops	q/t	flops	q/t
$p = 0$, $q = 1$	10	11.30	11	10.36	12	9.58	13	8.92
$p = 0$, $q = 2$	25	39.28	26	37.81	27	36.44	28	35.18
$p = 0$, $q = 3$	89	54.12	90	53.55	91	52.96	92	52.39
$p = 0$, $q = 4$	214	78.98	215	78.61	216	78.25	217	77.90
$p = 1$, $q = 1$	48	2.91	171	2.21	558	0.79	1773	0.51
$p = 1$, $q = 2$	183	5.70	474	4.15	1917	1.09	5448	0.63
$p = 1$, $q = 3$	431	11.43	1442	5.46	5381	1.50	15368	0.77
$p = 1$, $q = 4$	1128	15.14	3819	6.50	12006	2.09	36549	0.94
$p = 2$, $q = 1$	81	4.56	555	1.56	4143	0.40	26007	0.11
$p = 2$, $q = 2$	258	7.57	2412	1.40	14724	0.36	94428	0.08
$p = 2$, $q = 3$	950	8.26	6800	1.73	42998	0.39	251876	0.10
$p = 2$, $q = 4$	2457	10.10	15987	2.15	95247	0.48	585567	0.10
$p = 3$, $q = 1$	181	2.44	1715	1.02	20991	0.16	218767	0.03
$p = 3$, $q = 2$	550	3.78	6992	0.78	73596	0.11	754084	0.02
$p = 3$, $q = 3$	1910	4.21	20100	0.84	202900	0.11	2038820	0.02
$p = 3$, $q = 4$	4285	5.86	44099	1.04	452775	0.13	4538983	0.02

functions and a range of orders p and q are presented in Table VII for the two-dimensional case and in Table VIII for the three-dimensional case. What is clear from Table VII is that with few premultiplying functions, the tensor contraction

Table VIII. The number of operations and the ratio between number of operations for the two representations for the mass matrix in three dimensions as a function of different polynomial orders and numbers of functions.

	$n_f = 1$		$n_f = 2$		$n_f = 3$		$n_f = 4$	
	flops	q/t	flops	q/t	flops	q/t	flops	q/t
$p = 1$, $q = 1$	116	4.00	528	3.43	2224	0.92	9200	0.59
$p = 1$, $q = 2$	608	13.77	3084	6.62	12412	1.69	52124	0.81
$p = 1$, $q = 3$	2660	29.11	12432	12.26	46528	3.30	205424	1.30
$p = 1$, $q = 4$	7955	57.90	38007	20.99	155751	5.14	622679	2.04
$p = 2$, $q = 1$	314	6.02	3336	1.75	34984	0.40	359984	0.08
$p = 2$, $q = 2$	1838	11.21	20100	2.13	202900	0.39	2034140	0.06
$p = 2$, $q = 3$	7610	20.07	79800	3.36	765592	0.57	8039600	0.08
$p = 2$, $q = 4$	23285	34.29	239415	5.33	2451775	0.78	24538775	0.11
$p = 3$, $q = 1$	644	3.77	13584	1.21	279984	0.13	5759984	0.02
$p = 3$, $q = 2$	3752	5.83	80700	1.03	1564572	0.09	`FFC failure`	
$p = 3$, $q = 3$	14684	10.57	315216	1.40	6372120	0.11	-	-
$p = 3$, $q = 4$	47795	16.80	979575	1.96	19594199	0.14	-	-

approach is generally more efficient, even for relatively high order premultiplying functions. The situation changes quite dramatically for $p > 0$ as the number of premultiplying functions increases, and as the polynomial order of the premultiplying functions increases. The cases with numerous premultiplying functions are typical of the Jacobian resulting from the linearisation of a nonlinear differential equation in a practical simulation, and are therefore important. Obviously, the selection of the representation can have a tremendous performance impact. The relative performance of the representations in three dimensions is shown in Table VIII. The number of operations has increased relative to the two-dimensional case, which corresponds to an increase in the size of the generated code. For the more complex forms, compilation of the generated C++ code for the tensor contraction representation is no longer feasible, and in some cases simply not possible due to compiler limitations. For the most complicated cases, FFC was unable to generate tensor contraction code due to memory being exhausted. In practice, time is the limiting factor as a memory error is usually only encountered after many hours of code generation for the tensor contraction case. FFC was able to generate quadrature representation code for all cases.

Interestingly, for complicated forms the operation count is not always a good indicator of performance. For the three-dimensional mass matrix case with $p = 1$, $q = 4$ and $n_f = 4$, we would expect from the operation count that the tensor contraction representation would be faster. However, when computing the element tensor 48000 times, we observed a ratio of q/t = 0.78, indicating that the quadrature representation is faster. Noteworthy for this case is that the size of the generated code for tensor contraction representation is 11 MB, while the size of the generated quadrature code is only 362 kB. This size difference leads not only to a significant difference in the C++ compile time, but also appears to result in a drop in run time performance. The performance drop could be attributed to the increased memory traffic noted by Kirby and Logg [2006]. Also, it may be that the compiler is unable to perform effective optimisations on the unrolled code, or that the compiler is

Table IX. The number of operations and the ratio between number of operations for the two representations for the elasticity-like tensor in two dimensions as a function of different polynomial orders and numbers of functions.

	$n_f = 1$		$n_f = 2$		$n_f = 3$	
	flops	q/t	flops	q/t	flops	q/t
$p = 1$, $q = 1$	888	0.34	3060	0.36	10224	0.11
$p = 1$, $q = 2$	3564	1.42	11400	1.01	35748	0.33
$p = 1$, $q = 3$	10988	3.23	34904	1.82	100388	0.63
$p = 1$, $q = 4$	26232	5.77	82548	2.87	254304	0.93
$p = 2$, $q = 1$	888	1.20	8220	0.31	54684	0.09
$p = 2$, $q = 2$	7176	1.59	41712	0.49	284232	0.11
$p = 2$, $q = 3$	22568	2.80	139472	0.71	856736	0.17
$p = 2$, $q = 4$	54300	4.36	337692	1.01	2058876	0.23
$p = 3$, $q = 1$	3044	0.36	30236	0.16	379964	0.02
$p = 3$, $q = 2$	12488	0.92	126368	0.26	1370576	0.03
$p = 3$, $q = 3$	36664	1.73	391552	0.37	4034704	0.05
$p = 3$, $q = 4$	92828	2.55	950012	0.49	9566012	0.06
$p = 4$, $q = 1$	3660	0.68	73236	0.11	1275624	0.01
$p = 4$, $q = 2$	17652	1.16	296712	0.16	4628460	0.02
$p = 4$, $q = 3$	57860	1.71	903752	0.22	13716836	0.02
$p = 4$, $q = 4$	138984	2.46	2133972	0.29	32289984	0.03

Table X. The number of operations and the ratio between number of operations for the two representations for the elasticity-like tensor in three dimensions as a function of different polynomial orders and numbers of functions.

	$n_f = 1$		$n_f = 2$		$n_f = 3$	
	flops	q/t	flops	q/t	flops	q/t
$p = 1$, $q = 1$	5508	0.26	25200	0.40	112176	0.09
$p = 1$, $q = 2$	40176	2.42	169020	1.95	597564	0.55
$p = 1$, $q = 3$	201348	8.37	735408	5.44	3422160	1.16
$p = 1$, $q = 4$	708291	19.78	2958831	9.25	11728143	2.33
$p = 2$, $q = 1$	13986	0.70	158256	0.22	1691676	0.05
$p = 2$, $q = 2$	103518	3.17	1059804	0.74	11132244	0.14
$p = 2$, $q = 3$	450882	8.86	5417136	1.44	`FFC failure`	
$p = 2$, $q = 4$	1836225	14.90	18941967	2.50	-	-
$p = 3$, $q = 1$	11160	0.89	443376	0.19	13218516	0.01
$p = 3$, $q = 2$	186624	1.76	4402620	0.35	`FFC failure`	
$p = 3$, $q = 3$	1035684	3.86	21777552	0.62	-	-
$p = 3$, $q = 4$	3681171	7.43	`FFC failure`		-	-
$p = 4$, $q = 1$	49311	0.69	1940256	0.09	`FFC failure`	
$p = 4$, $q = 2$	364275	2.14	13527684	0.20	-	-

particularly effective at optimising the loops in the generated quadrature code.

A similar comparison is made for elasticity-like forms and the results are presented in Table IX for the two-dimensional case and in Table X for the three-dimensional case. Similar trends to those observed for the mass matrix hold. In three dimensions FFC fails to generate code for a number of the more complex forms using the tensor contraction representation. Code generation using the quadrature representation is successful in all cases. Also, file size considerations,

```
element   = VectorElement("Lagrange", "triangle", 2)
element_f = VectorElement("Lagrange", "triangle", 3)

v = TestFunction(element)
u = TrialFunction(element)

f = Function(element_f)
g = Function(element_f)

a = div(f)*div(g)*dot(grad(v), grad(u))*dx
```

Fig. 9. FFC input for the vector-valued Poisson problem in two dimension with with $q = 2$, premultiplied by the divergence of two vector valued functions ($n_f = 2$) of order $p = 3$.

Table XI. The number of operations and the ratio between number of operations for the two representations for the vector-valued Poisson problem in two dimensions as a function of different polynomial orders and numbers of functions.

	$n_f = 1$		$n_f = 2$	
	flops	q/t	flops	q/t
$p = 1$, $q = 1$	686	0.33	6126	0.07
$p = 1$, $q = 2$	2180	1.22	18372	0.18
$p = 1$, $q = 3$	8068	2.23	66372	0.29
$p = 1$, $q = 4$	22526	3.38	183870	0.43
$p = 2$, $q = 1$	1390	0.17	24558	0.06
$p = 2$, $q = 2$	7768	0.36	162744	0.05
$p = 2$, $q = 3$	24872	0.73	512872	0.07
$p = 2$, $q = 4$	60190	1.27	1246478	0.10
$p = 3$, $q = 1$	2094	0.42	96750	0.04
$p = 3$, $q = 2$	11640	0.55	541800	0.03
$p = 3$, $q = 3$	44776	0.73	1697576	0.03
$p = 3$, $q = 4$	110774	1.09	4099406	0.04

especially in the three-dimensional cases, will rule out the tensor contraction representation for a number of forms where, based on the ratio, it would be expected to outperform the quadrature representation. It is more difficult in these cases to make broad generalisation as to the best representation. This again demonstrates the desirability of a method for automatically determining the best representation based on inspection of the form.

Finally, we investigate the influence of premultiplying a vector-valued Poisson form by the divergence of vector-valued functions. The form for the case $n_f = 2$, $p = 3$ and $q = 2$ is shown in Figure 9. A comparison of tensor contraction and quadrature representations is performed, as in the previous cases, and the results are shown in Table XI. Premultiplying forms with derivatives of functions clearly increases the complexity to such a degree that the tensor contraction representation involves fewer operations for only a very limited number of the considered cases.

5. CONCLUSIONS

We have presented two representations, namely the tensor contraction and quadrature representations, for the computation of element tensors arising in the finite element method. The generation of code for these representations is automated and permits both the rapid development of solvers for broad classes of problems and the application of specialised performance optimisations. In particular, we have addressed strategies for optimising automatically the quadrature representation code. The strategies introduce negligible overhead in the code generation phase, but can yield substantial run time speed-ups. The presented techniques are possible with conventional 'hand' coding, and in fact commonly employed in specialised codes and simple problems. Automation makes the approach generic and allows the application of simple but tedious to implement by hand strategies to an unlimited range of problems.

The relative performance of two representations of finite element tensors has been investigated for a range of different problems. Numerical experiments have shown that the relative performance of the two representations can differ substantially depending on the nature of the considered variational form. Furthermore, small modifications of a form can mean that the most efficient representation changes. In general, however, the quadrature representation is significantly faster for more complicated forms. Also, relative to the tensor contraction representation, the time required to generate code for the quadrature representation is less and the size of the generated code is smaller. Automation is most attractive for complicated forms as they are time consuming to implement, implementations are error prone and performance is more elusive. In addressing the quadrature representation in the context of automated code generation, we have extended the applicability of automated modelling and of FFC to complicated variational forms. In practice, a sophisticated solver will often involve the assembly of various forms of differing complexity, so having both tensor contraction and quadrature representations as part of the computational arsenal allows the most appropriate representation for a given form to be used.

A particular challenge is the automated selection of the best representation. FFC presently computes the operation count for the code which is generated, on the basis of which a choice could be made, but this involves generating computer code for each case which can be time consuming. Ideally, the form compiler would select the best representation based on an inspection of the form. It turns out, however, that this is a non-trivial task if the goal is a general approach which holds for any form which FFC can handle and is a topic of ongoing investigation.

ACKNOWLEDGMENT

KBØ acknowledges the support of the Netherlands Technology Foundation STW, the Netherlands Organisation for Scientific Research and the Ministry of Public Works and Water Management. Valuable discussions with Anders Logg are gratefully acknowledged.

REFERENCES

Alnæs, M. S., Logg, A., Mardal, K.-A., Skavhaug, O., and Langtangen, H. P. 2008. *UFC Specification and User Manual*. `http://www.fenics.org/ufc/`.

Alnæs, M. S., Logg, A., Mardal, K.-A., Skavhaug, O., and Langtangen, H. P. 2009. Unified framework for finite element assembly. *International Journal of Computational Science and Engineering*. To appear.

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

Brezzi, F. and Fortin, M. 1991. *Mixed and Hybrid Finite Element Methods*. Springer Series in Computational Mathematics, vol. 15. Springer, New York.

FEniCS. 2008. FEniCS Project. `http//www.fenics.org/`.

Kirby, R. C. 2004. Algorithm 839: FIAT, A new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software 30,* 4, 502–516.

Kirby, R. C., Knepley, M. G., Logg, A., and Scott, L. R. 2005. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing 27,* 3, 741–758.

Kirby, R. C. and Logg, A. 2006. A compiler for variational forms. *ACM Transactions on Mathematical Software 32,* 3, 417–444.

Kirby, R. C. and Logg, A. 2007. Efficient compilation of a class of variational forms. *ACM Transactions on Mathematical Software 33,* 3, 17.

Kirby, R. C. and Logg, A. 2008. Benchmarking domain-specific compiler optimizations for variational forms. *ACM Transactions on Mathematical Software 35,* 2, 10. Article 10, 18 pages.

Kirby, R. C., Logg, A., Scott, L. R., and Terrel, A. R. 2006. Topological optimization of the evaluation of finite element matrices. *SIAM Journal on Scientific Computing 28,* 1, 224–240.

Logg, A. et al. 2008. *FFC*. `http://www.fenics.org/ffc/`.

Logg, A., Wells, G. N., et al. 2008. *DOLFIN*. `http://www.fenics.org/dolfin/`.

Ølgaard, K. B., Logg, A., and Wells, G. N. 2008. Automated code generation for discontinuous Galerkin methods. *SIAM Journal on Scientific Computing 31,* 2, 849–864.

Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE 93,* 2, 232– 275.

Rognes, M. E., Kirby, R. C., and Logg, A. 2008. Efficient assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming finite elements. Submitted for publication.

Wells, G. N., Hooijkaas, T., and Shan, X. 2008. Modelling temperature effects on multiphase flow through porous media. *Philosophical Magazine* 28–29, 3265–3279.