AUTOMATED CODE GENERATION FOR DISCONTINUOUS GALERKIN METHODS

KRISTIAN B. ØLGAARD[†], ANDERS LOGG[‡], AND GARTH N. WELLS[§]

Abstract. A compiler approach for generating low-level computer code from high-level input for discontinuous Galerkin finite element forms is presented. The input language mirrors conventional mathematical notation, and the compiler generates efficient code in a standard programming language. This facilitates the rapid generation of efficient code for general equations in varying spatial dimensions. Key concepts underlying the compiler approach and the automated generation of computer code are elaborated. The approach is demonstrated for a range of common problems, including the Poisson, biharmonic, advection–diffusion and Stokes equations.

Key words. Variational forms, discontinuous Galerkin methods, finite element, form compiler, code generation.

AMS subject classifications. 65N30, 68N20.

1. Introduction. Discontinuous Galerkin methods in space have emerged as a generalisation of finite element methods for solving a range of partial differential equations. While historically used for first-order hyperbolic equations, discontinuous Galerkin methods are now applied to a range of hyperbolic, parabolic and elliptic problems. In addition to the usual integration over cell volumes that characterises the conventional finite element method, discontinuous Galerkin methods also involve the integration of flux terms over interior facets. Discontinuous Galerkin methods exist in many variants, and are generally distinguished by the form of the flux on facets. A sample of fluxes for elliptic problems can be found in [4].

We present here a compiler approach for generating computer code for discontinuous Galerkin forms. From a high-level input language which resembles conventional mathematical notation, low-level computer code is generated automatically. The generated code is called by an assembler to construct global sparse tensors, commonly known as the 'stiffness matrix' and the 'load vector'. The compiler approach affords a number of interesting possibilities. It permits the rapid prototyping and testing of new methods, as well as providing scope for producing optimised code. The latter can be achieved through the compiler by precomputing various terms which are traditionally evaluated at run time, and by deploying procedures for analysing the structure of forms which facilitates various *a priori* optimisations which may not be tractable when developing computer code in a conventional fashion. In addition, the representations of element tensors (element stiffness matrices) for a given variational form are not limited to the usual quadrature-loop approach. For many forms, computationally more efficient representations can be employed. In essence, the form compiler approach allows a high level of generality, while competing in terms of performance with specialised, dedicated code, as will be elaborated in this work.

The use of a form compiler is particularly attractive for mixed problems, where one may wish to work with a combination of continuous and discontinuous function

---

[†]Faculty of Civil Engineering and Geosciences, Delft University of Technology, Stevinweg 1, 2628 CN Delft, Netherlands (`k.b.oelgaard@tudelft.nl`)

[‡]Center for Biomedical Computing, Simula Research Laboratory / Department of Informatics, University of Oslo, P.O.Box 134, 1325 Lysaker, Norway (`logg@simula.no`)

[§]Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom (`gnw20@cam.ac.uk`)

spaces, and function spaces which differ on element interiors. Such problems become trivial in the context of a compiler, as the compiler can automatically generate a degree-of-freedom mapping, thereby alleviating a difficulty faced when using many legacy codes for mixed problems. It also bears emphasis that the compiler provides the necessary operators to generate code not only for discontinuous Galerkin methods, but also for a range of novel finite element methods that draw upon discontinuous Galerkin methods. These methods may not involve discontinuous function spaces but do involve integration over interior facets. Such examples can be found in [8, 22, 16].

The concepts presented in this work are implemented in the FEniCS Form Compiler (henceforth FFC). FFC is a component of the FEniCS project [23], which consists of a suite of tools which aim to automate computational mathematical modelling, and all components are released under a GNU public license. FFC is freely available at http://www.fenicsproject.org and will generate code for all examples presented in this work.

The rest of this work is arranged as follows. Section 2 considers aspects of the assembly of variational forms and the representation of finite element tensors. This is followed by key concepts for the assembly of discontinuous Galerkin forms in Section 3. We discuss the form compiler FFC and the performance of the code it generates in Section 4 and a number of examples are presented in Section 5.

2. Compiling and assembling finite element variational forms. In this section, we outline a general framework for compiling and assembling variational forms. We then extend the framework to discontinuous Galerkin methods in the following section, where the form compiler must also consider integrals of discontinuous integrands over interior facets[1] of the computational mesh.

2.1. Multilinear forms. Consider a general multilinear form,

$$a : V_h^1 \times V_h^2 \times \cdots \times V_h^r \to \mathbb{R}, \tag{2.1}$$

defined on the product space $V_h^1 \times V_h^2 \times \cdots \times V_h^r$ of a sequence $\{V_h^j\}_{j=1}^r$ of discrete function spaces on a triangulation $\mathcal{T}$ of a domain $\Omega \subset \mathbb{R}^d$.

Multilinear forms appear as the basic building blocks of finite element discretisations of partial differential equations. The canonical example is the standard variational formulation of Poisson's equation $-\Delta u = f$. It reads: find $u_h \in V_h$ such that

$$a(v, u_h) = L(v) \quad \forall\, v \in \hat{V}_h, \tag{2.2}$$

where $\hat{V}_h = V_h^1$ and $V_h = V_h^2$ is a pair of discrete finite element function spaces (the *test* and *trial* spaces). The *bilinear* form a is here given by

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x \tag{2.3}$$

and the *linear* form L is given by

$$L(v) = \int_\Omega v f \, \mathrm{d}x. \tag{2.4}$$

For this problem, $r = 2$ for the bilinear form a and $r = 1$ for the linear form L, but forms of higher arity also appear (see [18]).

---

[1] A *facet* is a topological entity of a computational mesh of dimension $D - 1$ (codimension 1) where D is the topological dimension of the cells of the computational mesh. Thus for a triangular mesh, the facets are the edges and for a tetrahedral mesh, the facets are the faces.

2.2. Finite element assembly. With $\{\phi_i\}_{i=1}^N$ a global basis for $\hat{V}_h = V_h$, one may obtain the solution $u_h = \sum_{i=1}^N U_i \phi_i$ of the variational problem (2.2) by solving a linear system $AU = b$ for the *degrees of freedom* U of the discrete solution u_h, where $A_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}x$ and $b_i = \int_\Omega \phi_i f \,\mathrm{d}x$. In general, we are concerned with the discretisation of the general multilinear form a of (2.1), that is, the computation of the (sparse) rank r tensor A obtained by applying the multilinear form to the basis functions $\{\phi_i^j\}_{i=1}^{N_j}$ of V_h^j for $j = 1, 2, \ldots, r$:

$$A_i = a(\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r), \tag{2.5}$$

where $i = (i_1, i_2, \ldots, i_r)$ is a multi-index.

The tensor A may be computed efficiently by an algorithm known as *assembly* [24, 7, 17]. This algorithm computes the tensor A by iterating over the elements of the triangulation $\mathcal{T} = \{K\}$ and adding the local contribution from each local element K to the global tensor A. We refer to the local contribution from each element as the *element tensor* A^K [18]. For $r = 2$, this is normally referred to as the 'element stiffness matrix'. With $\{\phi_i^{K,j}\}_{i=1}^{n_j}$ a local basis for V_h^j on element K, the element tensor A^K is given by

$$A_i^K = a_K(\phi_{i_1}^{K,1}, \phi_{i_2}^{K,2}, \ldots, \phi_{i_r}^{K,r}), \tag{2.6}$$

where a_K is the local contribution to the multilinear form from element K. In the case of the bilinear form for Poisson's equation, this contribution is given by $a_K(v, u) = \int_K \nabla v \cdot \nabla u \,\mathrm{d}x$.

2.3. Tensor representation. In [11, 15, 12, 13], it was demonstrated that by generating low-level code from a special tensor representation of the element tensor A^K, one may generate (compile) very efficient code for assembly of the corresponding global matrix A. We return to the issue of the efficiency of the generated code in Section 4.3.

We demonstrate here how to derive this tensor representation for a basic example below and refer to [13] for a general representation theorem. Consider the bilinear form for the weighted Laplacian $-\nabla \cdot (w\nabla u) = f$,

$$a(v, u) = \int_\Omega w\nabla v \cdot \nabla u \,\mathrm{d}x. \tag{2.7}$$

The corresponding element tensor A^K is given by

$$A_i^K = \int_K w\nabla\phi_{i_1}^K \cdot \phi_{i_2}^K \,\mathrm{d}x. \tag{2.8}$$

For simplicity, we consider here the case where $V_h^1 = V_h^2 = V_h$. Now, let $F_K : K_0 \to K$ be the standard affine mapping from a reference element K_0 to any given element $K \in \mathcal{T}$. Using a change of variables from the reference coordinates X to the real coordinates $x = F_K(X)$, we find that

$$A_i^K = \sum_{\alpha_1=1}^d \sum_{\alpha_2=1}^d \sum_{\alpha_3=1}^n \det F_K' w_{\alpha_3} \sum_{\beta=1}^d \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \Phi_{\alpha_3} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \,\mathrm{d}X, \tag{2.9}$$

where Φ denotes basis functions on the reference element, n is the number of degrees of freedom for the local basis of w, d is the dimension of the domain Ω, and we
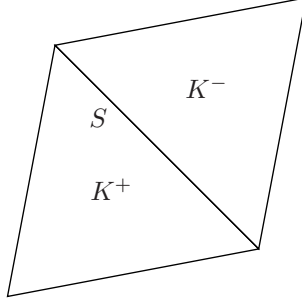
FIG. 3.1. *Two cells K^+ and K^- sharing a common facet S.*

have expanded w in the local nodal basis of V_h. By defining the two tensors $A^0 = \int_{K_0} \Phi_{\alpha_3} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, dX$ and $G_K = \det F'_K w_{\alpha_3} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta}$, we may express the element tensor A^K as the tensor contraction

$$A_i^K = \sum_\alpha A_{i\alpha}^0 G_K^\alpha. \tag{2.10}$$

We refer to A^0 as the *reference tensor* and to G_K as the *geometry tensor*.

The main point of this representation is that the reference tensor A^0 is independent of the triangulation $\mathcal{T}$ and may thus be precomputed. During assembly, one may then iterate over all elements of the triangulation and on each element K compute the geometry tensor G_K, compute the tensor contraction (2.10) and then add the resulting element tensor A^K to the global sparse matrix A. The form compiler FFC automatically generates the tensor representation (2.10); i.e., it precomputes the reference tensor A^0 at compile-time and generates code for evaluating the geometry tensor G_K and the tensor contraction.

3. Extending the framework to discontinuous Galerkin methods. To extend the above framework for finite element assembly to discontinuous Galerkin methods, we need to consider variational forms expressed as integrals over the *interior facets* of a finite element mesh. Consider for example the following bilinear form which may appear as a term in a discontinuous Galerkin formulation:

$$a(v, u) = \sum_{S \in \partial_i \mathcal{T}} \int_S [\![v]\!] [\![u]\!] \, ds, \tag{3.1}$$

where $\partial_i \mathcal{T}$ denotes the set of all interior facets of the triangulation $\mathcal{T}$ and where $[\![v]\!]$ denotes the *jump* in the function value of v across the facet S:

$$[\![v]\!] = v^+ - v^-. \tag{3.2}$$

Here, v^+ and v^- denote the values of v on the facet S as seen from the two cells K^+ and K^- incident with S, respectively (see Figure 3.1). We note that each interior facet is incident to exactly two cells, and we may label these two cells K^+ and K^-.

3.1. A general assembly algorithm. To assemble the global sparse tensor A for variational forms that contain integrals over interior facets as in (3.1), one may extend the standard assembly algorithm over the cells of the computational mesh by including an iteration over the interior facets of the mesh. Similarly, one may extend

the assembly to include the *exterior facets* (the set of facets incident with $\partial\Omega$) to account for terms that involve integrals over the boundary of the mesh.

A general assembly algorithm for the computation of the global sparse tensor A is given in [1]. This algorithm iterates first over all the cells of the mesh to compute the local contribution from each cell to the global sparse tensor. Above, we referred to this contribution as the element tensor A^K. In the context of the general assembly algorithm we shall refer to the contribution from each cell as the *cell tensor* A^K. Similarly, one may iterate over the exterior and interior facets of the mesh and add the local contribution from each facet to the global sparse tensor. We refer to these local contributions, denoted by A^S, as the *exterior* or *interior facet tensors*.

3.2. Computing the interior facet tensor. To define the interior facet tensor A^S for a given multilinear form a expressed as an integral over the set of interior facets $\partial_i\Omega$ such as in (3.1), we write

$$A_i = a(\phi_{i_1}, \phi_{i_2}) = \sum_S a_S(\phi_{i_1}, \phi_{i_2}), \tag{3.3}$$

where the summation is carried out only over those interior facets where both ϕ_{i_1} and ϕ_{i_2} are nonzero. In the case of (3.1), we have $a_S(v, u) = \int_S [\![v]\!][\![u]\!]\,\mathrm{d}s$. To assemble the global sparse tensor A efficiently, one may introduce a *local-to-global* mapping that maps the basis functions on the local facet S to the set of global basis functions. To construct this mapping, consider again two cells K^+ and K^- sharing a common facet S as in Figure 3.1. As above, let $\{\phi_i\}_{i=1}^N$ be a global (possibly discontinuous) basis for V_h. For ease of notation, we consider the assembly of the global tensor (matrix) for a bilinear form for $V_h^1 = V_h^2 = V_h$ and drop the index j (see equation (2.5)). We thus assume here that all discretising function spaces are equal, but note that this is not necessary. In particular, our implementation in FFC does not make this assumption and is able to generate code for assembly of tensors of arbitrary ranks for arbitrary combinations of finite element function spaces.

Furthermore, let $\{\phi_i^{K^+}\}_{i=1}^n$ be the local finite element basis on K^+ and let similarly $\{\phi_i^{K^-}\}_{i=1}^n$ be the local finite element basis on K^-. We now extend these local basis functions to the macro cell $\bar{K} = K^+ \cup K^-$ by the following construction:

$$\bar{\phi}_i^{\bar{K}}(x) = \begin{cases} \phi_i^{K^+}(x), & i = 1, 2, \ldots, n, & x \in K^+, \\ 0, & i = 1, 2, \ldots, n, & x \in K^-, \\ 0, & i = n+1, n+2, \ldots, 2n, & x \in K^+, \\ \phi_{i-n}^{K^-}(x), & i = n+1, n+2, \ldots, 2n, & x \in K^-. \end{cases} \tag{3.4}$$

We thus extend the local basis functions on K^+ and K^- to $\bar{K}$ by zero to obtain a local finite element space on $\bar{K}$ of dimension $2n$. For each $K \in \mathcal{T}$, we further let

$$\iota_K : [1, n] \to [1, N] \tag{3.5}$$

be a standard *local-to-global* mapping, that is, a mapping from a local enumeration of the basis functions on cell K to the corresponding global basis functions such that $\phi_i^K = \phi_{\iota_K(i)}|_K$ for $i = 1, 2, \ldots, n$. By the construction (3.4), we obtain a local-to-global mapping for $\bar{K}$ (or S). Thus, $\iota_{\bar{K}}(1) = \iota_{K^+}(1), \ldots, \iota_{\bar{K}}(n) = \iota_{K^+}(n), \iota_{\bar{K}}(n+1) = \iota_{K^-}(1), \ldots, \iota_{\bar{K}}(2n) = \iota_{K^-}(n)$.

We may now proceed to define the interior facet tensor A^S. Consider first the case when $\iota_{\bar{K}}$ is an injective mapping and note that $\iota_{\bar{K}}$ is injective when the ranges of

ι_{K^+} and ι_{K^-} are disjoint (which is the case for discontinuous elements). Continuing from (3.3), we then obtain

$$A_i = \sum_S a_S(\phi_{i_1}, \phi_{i_2}) = \sum_S a_S\left(\bar{\phi}^{\bar{K}}_{\iota_{\bar{K}}^{-1}(i_1)}, \bar{\phi}^{\bar{K}}_{\iota_{\bar{K}}^{-1}(i_2)}\right) = \sum_S A^S_{\iota_{\bar{K}}^{-1}(i_1), \iota_{\bar{K}}^{-1}(i_2)}, \qquad (3.6)$$

where the interior facet tensor A^S is thus defined by

$$A^S_i = a_S(\bar{\phi}^{\bar{K}}_{i_1}, \bar{\phi}^{\bar{K}}_{i_2}), \quad i_1, i_2 = 1, 2, \ldots, 2n. \qquad (3.7)$$

It follows that the global tensor A may be computed by Algorithm 1.

Algorithm 1 Assembly algorithm over interior facets.

$A = 0$
for $S \in \partial_i \mathcal{T}$
      for $i_1, i_2 = 1, 2, \ldots, 2n$
            $A^S_i = a_S(\bar{\phi}^{\bar{K}}_{i_1}, \bar{\phi}^{\bar{K}}_{i_2})$
      end for
      for $i_1, i_2 = 1, 2, \ldots, 2n$
            $A_{\iota_{\bar{K}}(i_1), \iota_{\bar{K}}(i_2)} \mathrel{+}= A^S_{i_1, i_2}$
      end for
end for

Now, if $\iota_{\bar{K}}$ is not injective (two local basis functions are restrictions of the same global basis function), which may happen if the basis functions are continuous, we may still assemble the global tensor A by Algorithm 1 and compute the interior facet tensor as in (3.7). To see this, assume that $\iota_{\bar{K}}(i_1) = \iota_{\bar{K}}(i_1') = I$ for some $i_1 \neq i_1'$. It then follows that the entry $A_{I, \iota_{\bar{K}}(i_2)}$ will be a sum of the two terms $A^S_{i_1, i_2}$ and $A^S_{i_1', i_2}$ (and possibly other terms). Since a_S is bilinear, we have

$$A^S_{i_1, i_2} + A^S_{i_1', i_2} = a_S(\bar{\phi}^{\bar{K}}_{i_1}, \bar{\phi}^{\bar{K}}_{i_2}) + a_S(\bar{\phi}^{\bar{K}}_{i_1'}, \bar{\phi}^{\bar{K}}_{i_2}) = a_S(\bar{\phi}^{\bar{K}}_{i_1} + \bar{\phi}^{\bar{K}}_{i_1'}, \bar{\phi}^{\bar{K}}_{i_2}) = a_S(\phi_I, \bar{\phi}^{\bar{K}}_{i_2}), \quad (3.8)$$

where by the construction (3.4) ϕ_I is the global basis function that both $\bar{\phi}^{\bar{K}}_{i_1}$ and $\bar{\phi}^{\bar{K}}_{i_1'}$ are mapped to.

3.3. Tensor representation and precomputation on facets. In Section 2, we described how the cell tensor (element tensor) may be computed from the tensor representation

$$A^K_i = \sum_\alpha A^0_{i\alpha} G^\alpha_K. \qquad (3.9)$$

Similarly, one may use the affine mappings F_{K^+} and F_{K^-} to obtain a tensor representation for the interior facet tensor A^S. However, depending on the topology of the macro cell $\bar{K}$, one obtains different tensor representations. For a triangular mesh, each cell has three facets (edges) and there are thus $3 \times 3 = 9$ different topologies to consider; there are nine different ways in which two edges can meet. Similarly, for a tetrahedral mesh, there are $4 \times 4 = 16$ different topologies to consider.[2] We thus

---

[2]FFC assumes a particular local ordering of the entities of the computational mesh as described in [19]. If no particular ordering of the mesh entities is assumed, one needs to consider $3 \times 3 \times 2 = 18$ different topologies for triangles and $4 \times 4 \times 6 = 96$ topologies for tetrahedra. This is because there are two different ways to superimpose two edges, and there are six ways to superimpose two faces.

obtain a tensor representation of the form

$$A_i^S = \sum_{\alpha} A_{i\alpha}^{0,f^+(S),f^-(S)} G_{\tilde{K}(S)}^{\alpha}, \tag{3.10}$$

where f^+ and f^- denote the local numbers of the two facets that meet at S relative to the two cells K^+ and K^- respectively. We note that the geometry tensor G_K^{α} in (3.9) involves the mapping from the reference cell and differs from the geometry tensor $G_{\tilde{K}(S)}^{\alpha}$ in (3.10), which may involve the mapping from the reference cell and the mapping from the reference facet. The form compiler FFC precomputes the reference tensor A^{0,f^+,f^-} for each facet–facet combination (f^+, f^-) and a run-time decision must be made as to which reference tensor should be contracted with the geometry tensor.

4. Implementation. We give here a short introduction to the FFC language before putting the compiler into context with respect to the other components of FEniCS. Then we discuss the performance in terms of the efficiency of the generated code as well as the benefits of automated code generation in general.

4.1. The Form Compiler FFC. The form compiler FFC computes the tensor representation (2.10) from a high-level description of the variational form, and generates efficient C++ code for the computation of the element tensor based on this representation. It is also possible to generate code that uses the conventional quadrature representation. Code can be generated which is consistent with the UFC [1] specification, although any format in any language can be implemented.

The bilinear form for the weighted Laplacian (2.7) can be expressed in the FFC form language by `a = w*dot(grad(v), grad(u))*dx`. Integration over a cell is denoted by `*dx`, integration over an exterior facet is denoted by `*ds` and integration over an interior facet is denoted by `*dS`. The FFC form language is equipped with basic differential operators including partial derivatives, `v.dx(i)`; the gradient, `grad(v)`; the divergence, `div(v)`; and the curl, `curl(v)`. Basic linear algebra operators like inner products `dot(v, w)` and matrix-vector multiplications `mult(A, v)` are also implemented. Functions which are evaluated on facets can be 'restricted' to the plus or minus sides of the facet. A function v evaluated on the plus side of a facet is expressed as `v('+')`, and the same function evaluated on the minus side is expressed as `v('-')`. Typical discontinuous Galerkin operators are available such as `jump(v)`, which is equivalent to $v^+ - v^-$; `jump(v, n)`, which is equivalent to $v^+ \boldsymbol{n}^+ + v^- \boldsymbol{n}^-$ or $\boldsymbol{v}^+ \cdot \boldsymbol{n}^+ + \boldsymbol{v}^- \cdot \boldsymbol{n}^-$; and `avg(v)`, which is equivalent to $(v^+ + v^-)/2$.

FFC is written in Python, and the interface to FFC also uses the Python syntax which makes the addition of user-defined operators simple. This will be demonstrated in Section 5, as will be the use of the operators introduced above.

4.2. FEniCS. The form compiler FFC is a core component of FEniCS [23], a software system aiming at automation of various aspects of computational mathematical modelling, in particular the solution of partial differential equations. Other core components of FEniCS include FIAT [9, 10], SyFi [3, 21], UFC [2] and DOLFIN [20].

FIAT is a tool for generating and tabulating finite element basis functions for a range of finite element spaces. FFC calls FIAT at compile-time to evaluate basis functions on the reference element as described in Section 3.3. FFC supports the generation of C++ code which is consistent with the Unified Form-assembly Code (UFC) specification. Any library which supports the UFC interface can use the automatically generated code to assemble forms.

Finally, DOLFIN is a consistent high-level problem-solving environment (PSE) for the solution of partial differential equations. DOLFIN handles the communication between the core components of FEniCS. Amongst other things, DOLFIN manages meshes (it does not generate meshes) and provides various linear algebra solvers and tools, as well as an interface to specialised linear algebra libraries. DOLFIN supports the UFC interface and uses the code generated by FFC to assemble and solve the global linear systems.

4.3. Performance. As claimed above, the form compiler FFC generates efficient code that may compete with and in some cases outperforms hand-written and optimised code. This was demonstrated in [12] where speedups as large as a factor 1000 were demonstrated for a set of standard test cases.[3] These speedups are a consequence of the reduced complexity of computing the tensor contraction (2.10) compared to a run-time iteration over quadrature points for certain forms. It was further demonstrated in [11, 15, 14] that these results may be further improved by finding *a priori* so-called *complexity-reducing relations* between subtensors of the reference tensor A^0.

As an example, consider here the operation count for evaluating the tensor-contraction for the Laplacian,

$$A_i^K = \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \det F'_K \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} \, dX. \tag{4.1}$$

With n the dimension of the local finite element function space, each of the n^2 entries of the element tensor A^K may be evaluated in roughly $T_T \sim d^2(d+1) + d^2 \sim d^3$ operations; first $d+1$ operations to evaluate each entry of the $d \times d$ geometry tensor G_K and then d^2 operations to perform the tensor contraction.

If instead we use a loop over N quadrature points at run-time, each entry of A^K may be evaluated in CN operations, where C is the cost of evaluating the integrand at each quadrature point. It is difficult to estimate exactly the size of C, but a reasonable estimate is $2d^2$ (transforming the gradient of both the test and trial functions from the reference element by a matrix-vector product with the inverse transpose of the Jacobian matrix). It is assumed that the values of the gradients of all basis functions have been pretabulated at the quadrature points on the reference element. Now, the number of quadrature points needed for exact integration is $N \sim (2(k-1)+1)^d = (2k-1)^d$, where k is the polynomial degree of the finite element basis functions. It follows that the complexity of quadrature is $T_Q \sim CN \sim 2d^2(2k-1)^d$. We thus find that the speedup of the tensor contraction (2.10) compared to quadrature is

$$\frac{T_Q}{T_T} \sim \frac{2d^2(2k-1)^d}{d^3} = 2(2k-1)^d/d. \tag{4.2}$$

It follows that the speedup may be substantial for moderate sized values of k. For $k = 1$, it is not clear that the speedup is positive, although it was demonstrated in [12] that the speedup in this case is a factor ~ 10.

---

[3]It should be noted that these speedups concern the evaluation of the element tensor on each local element. Insertion into the global sparse matrix and solution of the linear system are not included. The global speedup is smaller and depends on the efficiency of linear algebra and mesh data structures, as well as the problem at hand, the efficiency of the linear solver, and choice of preconditioner.

If instead we consider the weighted Laplacian (2.7), it is less clear that the tensor contraction outperforms quadrature. As discussed in [12], the relative performance of quadrature improves with an increasing number of coefficient functions (weights) in the variational form. Because of this, the form compiler FFC supports a mode where quadrature code is generated instead of code based on the tensor representation (2.10). Even with quadrature, various simple *a prior* optimisations are possible when using precomputation, such as the elimination of operations on zero entries and loop unrolling.

In summary, the advantage of code generation is not that the tensor representation (2.10) always leads to more efficient code than hand-written code, in particular since (2.10) may (with some effort) also be coded manually. Instead, the merit lies in: (i) the potential to employ sophisticated strategies for evaluation of the element tensor as in [12]; (ii) the potential to employ sophisticated compile-time optimisation techniques as in [11, 15, 14]; (iii) the generation of architecture-specific code (in particular for multicore processors); and (iv) the reduction in development time through simple and compact coding of finite element variational forms while retaining efficiency. In particular, the simplicity with which forms can be coded is demonstrated in the following section. For all examples in the following section, the time required for FFC to generate C++ code is of the order of seconds. The quantity of automatically produced code is highly dependent on the complexity of the considered form.

5. Examples. We demonstrate the compilation of variational forms for discontinuous Galerkin methods through a number of examples.

5.1. Poisson's equation. Consider the function space V_h,

$$V_h = \left\{ v \in L^2(\Omega) : \ v|_K \in P_k(K) \, \forall K \in \mathcal{T} \right\}, \tag{5.1}$$

where $P_k(K)$ denotes the space of polynomials of degree k on the element K. Setting $V_h^1 = V_h^2 = V_h$, the bilinear and linear forms for the Poisson equation with homogeneous Dirichlet boundary conditions read [4]

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u \ \mathrm{d}x - \int_{\Gamma^0} [\![v]\!] \cdot \langle \nabla u \rangle \ \mathrm{d}s - \int_{\Gamma^0} \langle \nabla v \rangle \cdot [\![u]\!] \ \mathrm{d}s$$
$$- \int_{\partial\Omega} [\![v]\!] \cdot \nabla u \ \mathrm{d}s - \int_{\partial\Omega} \nabla v \cdot [\![u]\!] \ \mathrm{d}s + \int_{\Gamma^0} \frac{\alpha}{h} [\![v]\!] \cdot [\![u]\!] \ \mathrm{d}s + \int_{\partial\Omega} \frac{\alpha}{h} vu \ \mathrm{d}s \tag{5.2}$$

and

$$L(v) = \int_\Omega vf \ \mathrm{d}x, \tag{5.3}$$

where Γ^0 denotes interior facets, α is a penalty parameter and h is a measure for the average of the mesh size defined as $h = (h^+ + h^-)/2$ for the two cells K^+ and K^- incident with the given interior facet. The size of a cell is defined here as twice the circumradius. The jump $[\![\cdot]\!]$ and average $\langle\cdot\rangle$ operators are defined as $[\![v]\!] = v^+\boldsymbol{n}^+ + v^-\boldsymbol{n}^-$ and $\langle\nabla v\rangle = (\nabla v^+ + \nabla v^-)/2$ on Γ^0 and $[\![v]\!] = v\boldsymbol{n}$ on $\partial\Omega$. Here, $\boldsymbol{n}^+$ and $\boldsymbol{n}^-$ denote the outward unit normal to the given facet as seen from the two cells K^+ and K^- respectively. The corresponding FFC input for this problem is shown in Table 5.1 for 5th order polynomials on triangular elements. The form and syntax of the compiler input resembles closely the mathematical notation in (5.2) and (5.3). Note that in the code we need to restrict the (potentially) multi-valued function h to either K^+ or K^- (here `h('+')`) even if the function in this particular case is single-valued ($h = (h^+ + h^-)/2$).

TABLE 5.1
FFC input for the interior penalty method applied to the Poisson equation using $k = 5$.

```
element = FiniteElement("Discontinuous Lagrange", "triangle", 5)

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

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

alpha = 32.0

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

L = v*f*dx
```

5.2. Advection–diffusion equation. We consider now the advection–diffusion equation with homogeneous Dirichlet boundary conditions on inflow boundaries and full upwinding of the advective flux at element facets. Setting $V_h^1 = V_h^2 = V_h$, where V_h is defined as in (5.1), the bilinear and linear forms read

$$a(v,u) = \int_\Omega \nabla v \cdot (\kappa \nabla u - \boldsymbol{b} u) \ \mathrm{d}x + \int_{\Gamma^0} [\![v]\!] \cdot \boldsymbol{b} u^\star \ \mathrm{d}s + \int_{\partial\Omega} [\![v]\!] \cdot \boldsymbol{b} u^\star \ \mathrm{d}s$$
$$- \int_{\Gamma^0} \kappa [\![v]\!] \langle \nabla u \rangle \ \mathrm{d}s - \int_{\Gamma^0} \kappa \langle \nabla v \rangle \cdot [\![u]\!] \ \mathrm{d}s - \int_{\partial\Omega} \kappa [\![v]\!] \cdot \nabla u \ \mathrm{d}s$$
$$- \int_{\partial\Omega} \kappa \nabla v \cdot [\![u]\!] \ \mathrm{d}s + \int_{\Gamma^0} \frac{\kappa\alpha}{h} [\![v]\!] \cdot [\![u]\!] \ \mathrm{d}s + \int_{\partial\Omega} \frac{\kappa\alpha}{h} vu \ \mathrm{d}s \quad (5.4)$$

and

$$L(v) = \int_\Omega vf \ \mathrm{d}x, \quad (5.5)$$

where the vector $\boldsymbol{b}$ is a given velocity field, $u^\star$ is equal to u restricted to the upwind side of a facet,

$$u^\star = \begin{cases} u^+ & \boldsymbol{b} \cdot \boldsymbol{n}^+ \geq 0, \\ u^- & \boldsymbol{b} \cdot \boldsymbol{n}^+ < 0, \end{cases} \quad (5.6)$$

and κ is the diffusion coefficient. The definitions of the jump and average operators are the same as for the Poisson equation. The FFC input for this problem is depicted in Table 5.2, and is again a reflection of the mathematical formulation. In this particular implementation, the same basis has been used for the solution and components of the advective velocity field, although different orders can be used. The value of the discontinuous function 'of' (outflow facet) in Table 5.2 is either 1 or 0 on a side of

TABLE 5.2
FFC input for the advection–diffusion equation using $k = 3$.

```
scalar   = FiniteElement("Discontinuous Lagrange", "triangle", 3)
vector   = VectorElement("Lagrange", "triangle", 3)
constant = FiniteElement("Discontinuous Lagrange", "triangle", 0)

v  = TestFunction(scalar)
u  = TrialFunction(scalar)

b  = Function(vector)
f  = Function(scalar)
n  = FacetNormal("triangle")
h  = MeshSize("triangle")
of = Function(constant)

kappa = 0.2
alpha = 20.0

def upwind(b, u):
  return [b[i]('+')*(of('+')*u('+') + of('-')*u('-')) for i in range(len(b))]

a =  dot(grad(v), mult(kappa, grad(u)) - mult(b, u))*dx \
   + dot(jump(v, n), upwind(b, u))*dS \
   + dot(mult(v, n), mult(b, of*u))*ds \
   - kappa*dot(jump(v, n), avg(grad(u)) )*dS \
   - kappa*dot(avg(grad(v)), jump(u, n))*dS \
   - kappa*dot(mult(v, n), grad(u))*ds \
   - kappa*dot(grad(v), mult(u, n))*ds \
   + kappa*alpha/h('+')*dot(jump(v, n), jump(u, n))*dS \
   + kappa*alpha/h*v*u*ds

L = v*f*dx
```

the interior facet which is being considered. When performing a computation, we compute this value in DOLFIN according to the definition in (5.6) and pass it to the form as a function.

If a facet is an outflow facet relative to the element K^+, i.e., $\boldsymbol{b} \cdot \boldsymbol{n}^+ \geq 0$, then the value of `of('+')` is 1, while the value of `of('-')` is 0 and vice versa. As a consequence, the return value of the 'upwind' function is equal to $\boldsymbol{b}u^\star$. The 'upwind' function is an example of how one can extend the FFC language with user-defined functions written in Python.

5.3. The Stokes equations. We consider next the Stokes equations with a mixture of continuous and discontinuous functions, as well as basis functions with possibly varying polynomial orders. Consider the function spaces $\boldsymbol{V}_h$ and Q_h,

$$\boldsymbol{V}_h = \left\{ \boldsymbol{v} \in \left(L^2\left(\Omega\right) \right)^d : \ v_i \in P_k\left(K\right) \forall K \in \mathcal{T}, \ 1 \leq i \leq d \right\}, \tag{5.7}$$

$$Q_h = \left\{ q \in H^1\left(\Omega\right) : \ q \in P_j\left(K\right) \forall K \in \mathcal{T} \right\}. \tag{5.8}$$

TABLE 5.3
FFC input for the Stokes equation using $k = 1$.

```
V = VectorElement("Discontinuous Lagrange", "triangle", 1)
Q = FiniteElement("Lagrange", "triangle", 1)

element = V + Q

(v, q) = TestFunctions(element)
(u, p) = TrialFunctions(element)

f = Function(V)
n = FacetNormal("triangle")
h = MeshSize("triangle")

alpha = 4.0

a =  dot(grad(v), grad(u))*dx + dot(v, grad(p))*dx - dot(grad(q), u)*dx \
    + dot(q('+'), jump(u, n))*dS \
    + q*dot(u, n)*ds \
    - dot(mult(avg(grad(v)), n('+')), jump(u))*dS \
    - dot(jump(v), mult(avg(grad(u)), n('+')))*dS \
    - dot(mult(grad(v), n), u)*ds \
    - dot(v, mult(grad(u), n))*ds \
    + alpha/h('+')*dot(jump(v), jump(u))*dS \
    + alpha/h*dot(v, u)*ds

L = dot(v, f)*dx
```

Setting $V_h^1 = V_h^2 = \boldsymbol{V}_h \times Q_h$ and $\boldsymbol{u} = 0$ on $\partial\Omega$, particular bilinear and linear forms for the Stokes equation read [5]

$$a(\boldsymbol{v}, q; \boldsymbol{u}, p) = \int_\Omega \nu \nabla\boldsymbol{v} : \nabla\boldsymbol{u} \ \mathrm{d}x + \int_\Omega \boldsymbol{v} \cdot \nabla p \ \mathrm{d}x - \int_\Omega \nabla q \cdot \boldsymbol{u} \ \mathrm{d}x$$
$$+ \int_{\Gamma^0} q[\![\boldsymbol{u} \cdot \boldsymbol{n}]\!] \ \mathrm{d}s + \int_{\partial\Omega} q\boldsymbol{u} \cdot \boldsymbol{n} \ \mathrm{d}s$$
$$- \int_{\Gamma^0} \nu[\![\boldsymbol{v}]\!] : \langle\nabla\boldsymbol{u}\rangle \ \mathrm{d}s - \int_{\Gamma^0} \nu\langle\nabla\boldsymbol{v}\rangle : [\![\boldsymbol{u}]\!] \ \mathrm{d}s - \int_{\partial\Omega} \nu[\![\boldsymbol{v}]\!] : \nabla\boldsymbol{u} \ \mathrm{d}s$$
$$- \int_{\partial\Omega} \nu\nabla\boldsymbol{v} : [\![\boldsymbol{u}]\!] \ \mathrm{d}s + \int_{\Gamma^0} \frac{\nu\alpha}{h}[\![\boldsymbol{v}]\!] : [\![\boldsymbol{u}]\!] \ \mathrm{d}s + \int_{\partial\Omega} \frac{\nu\alpha}{h}[\![\boldsymbol{v}]\!] : [\![\boldsymbol{u}]\!] \ \mathrm{d}s, \quad (5.9)$$

and

$$L(\boldsymbol{v}, q) = \int_\Omega \boldsymbol{v} \cdot \boldsymbol{f} \ \mathrm{d}x. \tag{5.10}$$

The jump $[\![\cdot]\!]$ and average $\langle\cdot\rangle$ operators are defined as $[\![\boldsymbol{v}]\!] = \boldsymbol{v}^+ \otimes \boldsymbol{n}^+ + \boldsymbol{v}^- \otimes \boldsymbol{n}^-$, $[\![\boldsymbol{v} \cdot \boldsymbol{n}]\!] = \boldsymbol{v}^+ \cdot \boldsymbol{n}^+ + \boldsymbol{v}^- \cdot \boldsymbol{n}^-$ and $\langle\nabla\boldsymbol{v}\rangle = (\nabla\boldsymbol{v}^+ + \nabla\boldsymbol{v}^-)/2$ on Γ^0 and $[\![\boldsymbol{v}]\!] = \boldsymbol{v} \otimes \boldsymbol{n}$ on $\partial\Omega$. The FFC input for this problem with $k = j = 1$, as proposed in [5], and the kinematic viscosity $\nu = 1.0$ is shown in Table 5.3.

5.4. Biharmonic equation. Classically, Galerkin methods for the biharmonic equation seek approximate solutions in a subspace of $H^2(\Omega)$. However, such functions are difficult to construct in a finite element context. Based on discontinuous Galerkin

```
element = FiniteElement("Lagrange", "tetrahedron", 4)

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

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

alpha = 16.0

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

L = v*f*dx
```

principles, methods have been developed which utilise functions from $H^1(\Omega)$ [6, 22]. Rather than considering jumps in functions across element boundaries, terms involving the jump in the normal derivative across element boundaries are introduced. Unlike fully discontinuous approaches, this method does not involve double-degrees of freedom on element edges and therefore does not lead to the significant increase in the number of degrees of freedom relative to conventional methods. Consider the continuous function space

$$V_h = \left\{ v \in H_0^1(\Omega): \ v \in P_k(K) \, \forall K \in \mathcal{T} \right\}. \qquad (5.11)$$

Setting $V_h^1 = V_h^2 = V_h$, the bilinear and linear forms for the biharmonic equation, with the boundary conditions $u = 0$ on $\partial\Omega$ and $\nabla^2 u = 0$ on $\partial\Omega$, read

$$a(v, u) = \int_\Omega \nabla^2 v \nabla^2 u \ \mathrm{d}x - \int_{\Gamma^0} [\![\nabla v]\!] \cdot \langle \nabla^2 u \rangle \ \mathrm{d}s - \int_{\Gamma^0} \langle \nabla^2 v \rangle \cdot [\![\nabla u]\!] \ \mathrm{d}s$$
$$+ \int_{\Gamma^0} \frac{\alpha}{h} [\![\nabla v]\!] \cdot [\![\nabla u]\!] \ \mathrm{d}s, \qquad (5.12)$$

$$L(v) = \int_\Omega v f \ \mathrm{d}x. \qquad (5.13)$$

The jump $[\![\cdot]\!]$ and average $\langle \cdot \rangle$ operators are defined as $[\![\nabla v]\!] = \nabla v^+ \cdot \boldsymbol{n}^+ + \nabla v^- \cdot \boldsymbol{n}^-$ and $\langle \nabla^2 v \rangle = (\nabla^2 v^+ + \nabla^2 v^-)/2$ on Γ^0. The FFC input for this problem with $k = 4$ is shown in Table 5.4.

For the biharmonic equation we consider an example on the domain $\Omega = [0,1] \times [0,1] \times [0,1]$ with $f = 9\pi^4 \sin(\pi x) \sin(\pi y) \sin(\pi z)$, in which case the exact solution $u = \sin(\pi x) \sin(\pi y) \sin(\pi z)$. The observed convergence behaviour is illustrated in Figure 5.1 for various polynomial orders. As predicted by *a priori* estimates, a convergence rate of $k + 1$ is observed for $k > 2$ [6], and a rate of k for polynomial order $k = 2$ [22]. The example demonstrates that the step from a two-dimensional problem to a three-dimensional problem is trivial when using the compiler.
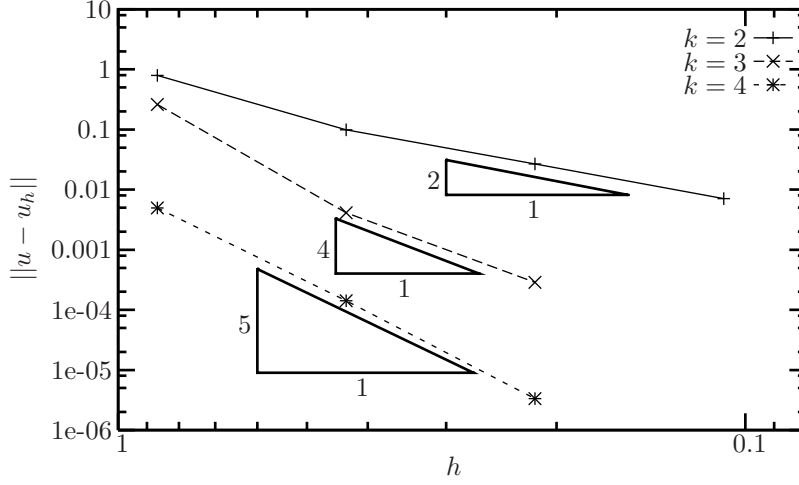
FIG. 5.1. *Error in the L^2 norm for the biharmonic equation with penalty parameters $\alpha = 4$, $\alpha = 16$ and $\alpha = 16$, for $k = 2$, $k = 3$ and $k = 4$ respectively.*

TABLE 5.5
Computation of the error in the L^2 norm (squared).

```
element_u  = FiniteElement("Lagrange", "triangle", 15)
element_uh = FiniteElement("Discontinuous Lagrange", "triangle", 5)

u   = Function(element_u)
u_h = Function(element_uh)

e = u - u_h
M = e*e*dx
```

5.5. Evaluating functionals. FFC can also be used to generate the required code for functionals (forms of rank zero), which is useful for computing the error when the exact solution is known or for evaluating various functionals of the computed solution. For a problem which has been solved using 5th order Lagrange basis functions, given the exact solution u and the finite element solution u_h, the error $e = u - u_h$ and the FFC input for computing the L^2 norm of the error is shown in Table 5.5. The exact solution has been approximated by interpolating the exact solution using a continuous 15th order polynomial. The FFC input for computing the mesh-dependent semi-norm of the error

$$\||e\||^2 = \int_\Omega \nabla e \cdot \nabla e \, \mathrm{d}x + \int_{\Gamma^0} [\![e]\!] \cdot [\![e]\!] \, \mathrm{d}s, \tag{5.14}$$

is shown in Table 5.6. Here, $\Gamma^0 = \cup \partial_i \mathcal{T}$ denotes the union of all interior facets of the mesh.

6. Conclusions. An approach for automated code generation for discontinuous Galerkin forms has been presented. The concept is manifest in the form of a compiler which translates discontinuous Galerkin forms expressed in a high-level language into efficient low-level code.

A special representation for element tensors for discontinuous Galerkin methods

TABLE 5.6
Computation of the error in a mesh-dependent semi-norm (squared).

```
element_u  = FiniteElement("Lagrange", "triangle", 15)
element_uh = FiniteElement("Discontinuous Lagrange", "triangle", 5)

u   = Function(element_u)
u_h = Function(element_uh)

e = u - u_h
M = dot(grad(e), grad(e))*dx + dot(jump(e), jump(e))*dS
```

has also been presented. This representation involves a tensor contraction and permits the separation of terms which can be computed *a priori* and terms which are computed at run time. Such a representation can lead to improved performance relative to conventional quadrature approaches for a variety of forms. However, the approach is not generally tractable by hand and necessitates automated code generation techniques. The compiler for discontinuous Galerkin forms facilitates rapid implementation of new and existing approaches and reduces the time required for code testing through generality, while delivering efficient code and providing scope for various automated optimisations.

REFERENCES

[1] M. S. ALNÆS, A. LOGG, K.-A. MARDAL, O. SKAVHAUG, AND H.-P LANGTANGEN, *UFC Specification and User Manual*, 2008. URL: http://www.fenicsproject.org/.

[2] MARTIN SANDVE ALNÆS, ANDERS LOGG, KENT-ANDRE MARDAL, OLA SKAVHAUG, AND HANS PETTER LANGTANGEN, *Unified framework for finite element assembly*, International Journal of Computational Science and Engineering, 4 (2009), pp. 231–244.

[3] M. S. ALNÆS AND K.-A. MARDAL, *SyFi: Symbolic Finite Elements*, 2008. http://www.fenicsproject.org/.

[4] D. N. ARNOLD, F. BREZZI, B. COCKBURN, AND L. D. MARINI, *Unified analysis for discontinuous Galerkin methods for elliptic problems*, SIAM Journal on Numerical Analysis, 39 (2002), pp. 1749–1779.

[5] G. A. BAKER, W. N. JUREIDINI, AND O. A. KARAKASHIAN, *Piecewise solenoidal vector fields and the Stokes problem*, SIAM Journal on Numerical Analysis, 27 (1990), pp. 1466–1485.

[6] G. ENGEL, K. GARIKIPATI, T. J. R. HUGHES, M. G. LARSON, AND R. L. TAYLOR, *Continuous/discontinuous finite element approximations of fourth-order elliptic problems in structural and continuum mechanics with applications to thin beams and plates, and strain gradient elasticity*, Computer Methods in Applied Mechanics and Engineering, 191 (2002), pp. 3669–3750.

[7] T. J. R. HUGHES, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Dover Inc., 2000.

[8] T. J. R. HUGHES, G. SCOVAZZI, P. B. BOCHEV, AND A. BUFFA, *A multiscale discontinuous Galerkin method with the computational structure of a continuous Galerkin method*, Computer Methods in Applied Mechanics and Engineering, 195 (2006), pp. 2761–2787.

[9] R. C. KIRBY, *FIAT: A new paradigm for computing finite element basis functions*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 502–516.

[10] ———, *Optimizing FIAT with Level 3 BLAS*, ACM Transactions on Mathematical Software (TOMS), 32 (2006), pp. 223–235.

[11] R. C. KIRBY, M. G. KNEPLEY, A. LOGG, AND L. R. SCOTT, *Optimizing the evaluation of finite element matrices*, SIAM Journal on Scientific Computing, 27 (2005), pp. 741–758.

[12] R. C. KIRBY AND A. LOGG, *A compiler for variational forms*, ACM Transactions on Mathematical Software (TOMS), 32 (2006), pp. 417–444.

[13] ———, *Efficient compilation of a class of variational forms*, ACM Transactions on Mathematical Software (TOMS), 33 (2007).

[14] R. C. KIRBY AND A. LOGG, *Benchmarking domain-specific compiler optimizations for variational forms*, ACM Transactions on Mathematical Software, 35 (2008), pp. 1–18.

[15] R. C. KIRBY, A. LOGG, L. R. SCOTT, AND A. R. TERREL, *Topological optimization of the evaluation of finite element matrices*, SIAM Journal on Scientific Computing, 28 (2006), pp. 224–240.

[16] R. J. LABEUR AND G. N. WELLS, *A Galerkin interface stabilisation method for the advection-diffusion and incompressible Navier-Stokes equations*, Computer Methods in Applied Mechanics and Engineering, 196 (2007), pp. 4985–5000.

[17] H. P. LANGTANGEN, *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, Lecture Notes in Computational Science and Engineering, Springer, 1999.

[18] A. LOGG, *Automating the finite element method*, Archives of Computational Methods in Engineering, 14 (2007), pp. 93–138.

[19] ———, *FFC User Manual*, 2007. URL: http://www.fenics.org/ffc/.

[20] A. LOGG, G. N. WELLS, J. HOFFMAN, J. JANSSON, ET AL., *DOLFIN: A general-purpose finite element library. http://www.fenicsproject.org/.*

[21] K.-A. MARDAL, *SyFi - an element matrix factory*, in Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing, PARA'06, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 703–711.

[22] G. N. WELLS AND N. T. DUNG, *A C^0 discontinuous Galerkin formulation for Kirchhoff plates*, Computer Methods in Applied Mechanics and Engineering, 196 (2007), pp. 3370–3380.

[23] *The FEniCS Project*. http://www.fenicsproject.org.

[24] O. C. ZIENKIEWICZ, R. L. TAYLOR, AND J. Z. ZHU, *The Finite Element Method: Its Basis and Fundamentals*, Butterworth-Heinemann, sixth ed., 2005.