On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods

MARTIN SANDVE ALNÆS
Simula Research Laboratory
and
KENT-ANDRÉ MARDAL
Simula Research Laboratory

Efficient and easy implementation of variational forms for finite element discretization can be accomplished with meta-programming. Using a high-level language like Python and symbolic mathematics makes an abstract problem definition possible, but the use of a low-level compiled language is vital for run-time efficiency. By generating low-level C++ code based on symbolic expressions for the discrete weak form, it is possible to accomplish a high degree of abstraction in the problem definition while surpassing the run-time efficiency of traditional hand written C++ codes. We provide several examples where we demonstrate orders of magnitude in speed-up.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Efficiency, algorithm design and analysis; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite Element Methods*; I.1.1 [**Computing Methodologies**]: Symbolic and algebraic manipulation—*Expressions and their representation*

General Terms: Performance, Algorithms

Additional Key Words and Phrases: Variational forms, code generation, compiler, finite element, automation, metaprogramming

1. INTRODUCTION

A cornerstone when developing finite element simulators is the task of implementing variational forms of partial differential equations (PDEs), and optimizing this implementation. A software development environment for this task should ideally be user-friendly and general, in the sense that implementations are close to the underlying mathematical concepts, and result in high computational efficiency without special effort from the user side. We have explored the combination of symbolic

M. S. Alnæs, Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway.
E-mail: `martinal@simula.no`. Alnæs has been partially supported by the Norwegian Research Council (grant 162730).
K.-A. Mardal, Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway, and Department of Informatics, University of Oslo, Oslo, Norway. E-mail: `kent-and@simula.no`. Mardal has been partially supported by the Norwegian Research Council (grant 170650).

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

mathematics and code generation to be able to specify finite element methods in a user-friendly environment while maintaining efficiency. By employing a symbolic engine in a high-level language we allow the user to specify the weak form of the PDE in an abstract format close to the mathematical formulation. Furthermore, the symbolic framework allow us to do certain calculations automatically that we earlier typically did by hand, e.g. the calculation of the Jacobian in the case of a nonlinear PDE, or differentiation of complex material laws for hyper-elastic materials [Alnæs et al. 2007].

The generated code is often, as will be demonstrated, significantly faster than traditional codes based on quadrature. This efficiency gain is due to a combination of the symbolic computations performed prior to the code generation and that the resulting C++ code is low-level and problem-specific.

Our efforts have resulted in the open source software package SyFi [Alnæs and Mardal 2008], which is part of the FEniCS project [FEniCS 2008]. SyFi stands for symbolic finite elements and is implemented in C++ and Python, building on the symbolic C++ library GiNaC [Bauer et al. 2002; Bauer et al. 2007] and its Python interface Swiginac [Skavhaug and Certik 2008]. SyFi is largely divided in two: a kernel and a form compiler.

The SyFi kernel consists of a collection of tools for symbolic computations on polynomial spaces and polygonal domains, and a collection of elements including Arnold-Falk-Winther element [Arnold et al. 2007], the Crouzeix-Raviart element [Crouzeix and Raviart 1973], the Hermite element, the standard Lagrange elements, the Nedelec elements [Nédélec 1980; 1986], the Raviart-Thomas element [Raviart and Thomas 1977], and the robust Darcy-Stokes element [Mardal et al. 2002]. The elements are implemented in a generic way by solving a symbolic linear system, defined by the degrees of freedom, in the construction of the element. Therefore the elements are defined for arbitrary order except for the Crouzeix-Raviart and Hermite elements. Another closely related approach of implementing a general finite element package is FIAT [Kirby 2004] which realizes polynomial spaces and degrees of freedom numerically.

The SyFi Form Compiler (SFC) takes as input a symbolic description of a variational form and a set of finite elements, and generates problem-specific C++ code. This code includes function(s) to compute the element tensor(s) for the given problem, evaluate basis functions and degrees of freedom for the specified finite elements, and tabulate the local to global mapping of degrees of freedom.

Symbolic computing to simplify finite element methods is not new, but the traditional way of programming expressions for the quadrature loop based on hand-made calculations dominates. We believe there are three reasons for this dominance; the extra effort of implementing a symbolic engine, assumed efficiency, and potential scalability problems. By reusing an existing symbolic library, the implementation effort is significantly reduced. The main subject of this paper is to demonstrate that efficiency does not need to be sacrificed when introducing symbolic computing. In fact, we will demonstrate that our approach often results in a significant speed-up. However, care must be taken to avoid various problems with complicated equations, which we'll discuss at the end.

There are many other software packages that enable a high-level way of specifying the variational form. Some projects like GetDp [Dular and Geuzaine 2006] and FreeFEM [Pironneau et al. 2006] implement domain specific languages. Other libraries like Diffpack [Bruaset et al. 2006; Langtangen 2003], Deal.II [Bangerth et al. 2007b; 2007a], Life [Prud'homme 2006] and Sundance [Long 2006] employ object-orientation and/or template metaprogramming. Another approach which is very similar to ours is taken by the FEniCS Form Compiler (FFC) [Kirby and Logg 2006; 2007; Logg 2008; Logg et al. 2008]. Both projects let the user specify the variational form in Python using a high-level description. This description is then parsed and efficient low-level C++ code is generated. The main difference between FFC and SFC is that SFC employs a general symbolic engine, while FFC exploits the geometric affine mapping to generate an efficient tensor representation of element matrices and vectors.

Generating code involves some overhead, and in our experience it is important to have a fixed interface between generated code and handwritten library code. Together with the developers of FFC we have developed a common interface for the code we generate, called UFC (Unified Form-assembly Code) [Alnæs et al. 2008; Alnæs et al. 2009]. This interface is well documented in the code and comes with a detailed manual. UFC is basically a small C++ header file with a few abstract classes that contain low-level signatures for the computation of element tensors (matrices, vectors and scalars), local to global mappings, and finite element evaluations etc. Using a fixed interface leads to a clear software separation between the finite element discretization of the PDE on one side, and the global linear algebra and mesh formats on the other. With this design, developers can combine the strengths of their favorite form compiler (at the moment FFC or SFC) with their favorite mesh and linear algebra libraries by writing the global tensor assembly loop. At the time of writing, global tensor assembly from UFC form objects is implemented in the problem solving environment DOLFIN, and an example assembler is sketched in the UFC documentation for interested developers.

Symbolic computations allow a definition of the equations that is close to mathematical notation. In our experience, thinking in terms of the same operators you would write on paper (such as div, grad, curl, dot) reduces both the time to implement new equations and the probability for bugs. Using such high-level syntax does not hinder computational efficiency because of the code generation. Furthermore, symbolic differentiation can also be a major work saving feature in the implementation of some forms, for instance in the differentiation of complicated constitutive laws, or automatic computation of the Jacobi matrix of a nonlinear equation.

Finally, it is worth noting that an advantage with a compiler is that it can detect user errors of a more abstract or mathematical kind than a standard C++ compiler since it is a special purpose compiler for finite element methods.

The purpose of this paper is to compare the efficiency of the element tensor computations as conventionally programmed by using hand-written quadrature with our approach using symbolic mathematics and code generation. Hand-written quadrature examples are programmed using Diffpack 4.0 and Deal.II 6.0.0. The efficiency of the code generated by SFC 0.5.1 is also compared to FFC 0.4.3 since FFC is known to produce very efficient code, as documented in [Kirby and Logg 2006; 2007].

While presenting the code examples we hope to demonstrate the user-friendliness of our software tools. The source examples work with SyFi release 0.5.1.

An outline is as follows. In Section 2 we introduce the necessary mathematical background and notation for discrete variational forms. We also introduce symbolic and numeric integration techniques, and show how this can be used as a basis for optimization. In Section 3 the corresponding software abstractions are presented, as well as the code generation process and issues related to generating efficient code. Section 4 contains a series of efficiency comparisons of the element tensor computations in Deal.II, Diffpack, FFC and SFC. Finally, in Section 5 we discuss limitations, advantages and future possibilities.

2. PRELIMINARIES

2.1 Variational Forms and Functionals in the Continuous Case

We will consider variational forms and functionals on the following form

$$a_\Omega(u^0, \dots, u^{n-1}; w^0, \dots, w^{m-1}) \to \mathbb{R}, \tag{1}$$

where $0 \leq n \leq 2$, u^0 is the test function, u^1 is the trial function and $w^0, \dots, w^{m-1}$ are the coefficient functions (or prescribed functions). Furthermore, $u^k \in U_k$ and $w^l \in W_l$ where U_k and W_l typically are some Sobolev spaces. We only handle forms that map to real numbers. Some examples (to illustrate the notation) are:
1) The bilinear form of a second order elliptic PDE with a coefficient μ,

$$a^1_\Omega(v, u; \mu) = \int_\Omega \mu \nabla u \cdot \nabla v \, d\mathbf{x},$$

2) the linear form of a typical right hand side with a given coefficient function f,

$$a^2_\Omega(v; f) = \int_\Omega f v \, d\mathbf{x},$$

3) and finally the inner product of the given coefficient functions f and g

$$a^3_\Omega(; f, g) = \int_\Omega f g \, d\mathbf{x}.$$

Neither the trial nor test function need to be present, but if the trial function is present then so is the test function. The variational forms or functionals may take any number of coefficients.

In the case of a form based on a nonlinear PDE, we may compute the Jacobian matrix as follows. Let the nonlinear differential operator be

$$\mathcal{L}(w^0, \dots, w^k, \dots, w^{m-1}).$$

The weak form is then:

$$a_\Omega(v; w^0, \dots, w^{m-1}) = \int_\Omega \mathcal{A}(v, w^0, \dots, w^{m-1}) \, d\mathbf{x} + \int_{\partial\Omega} \mathcal{B}(v, w^0, \dots, w^{m-1}) \, d\text{ß},$$

where $\mathcal{A}(\dots)$ and $\mathcal{B}(\dots)$ are obtained by multiplying $\mathcal{L}(\dots)$ by the test function v and performing integration by parts. The form a_Ω is linear in the first argument (the test function v) and nonlinear in w^k. If we then assume that $W_k = \text{span}(\{\phi^k_j\})$[1]

[1]The Sobolev space must be separable.

and $w^k = \sum_j w_j^k \phi_j^k$, then the Jacobian with respect to coefficient number k is the derivative of $a_\Omega(\phi_i^0; \ldots)$ with respect to $\{w_j^k\}$,

$$J_{ij}^k = \frac{\partial}{\partial w_j^k} a_\Omega(\phi_i^0; w^0, \ldots, w^{m-1}), \qquad \forall i, j$$

where $\{\phi_i^0\}$ spans U_0. This produces a bilinear form

$$a_\Omega^J(\phi_i^0, \phi_j^1; w^0, \ldots, w^{m-1}) = J_{ij}^k,$$

where $\{\phi_j^1\}$ spans $U_1 = W_k$ and the functions $w^0, \ldots, w^{m-1}$ are fixed.

2.2 Variational Forms and Functionals in the Discrete Case Using Finite Elements

In the finite element method the variational form (1) is split up as a sum of variational forms over a set of simple polygons $\{T_i\}$ such that $\sum_i T_i = \Omega_h \approx \Omega$. Hence, we need to be able to compute

$$a_T(u^0, \ldots, u^{n-1}; w^0, \ldots, w^{m-1}) \to \mathbb{R}. \tag{2}$$

on a generic polygon T. Here $u^0, \ldots, u^{n-1}$ and $w^0, \ldots, w^{m-1}$ are functions in the finite element spaces $\{U_k\}$ and $\{W_l\}$, respectively. The element tensor (matrix, vector or scalar) is computed as

$$A^T_{\iota_0,\ldots,\iota_{r-1}} = a_T(N^0_{\iota_0}, \ldots, N^{r-1}_{\iota_{r-1}}; w^0, \ldots, w^{m-1}),$$

where $\iota = \iota_0, \ldots, \iota_{r-1}$ is a multi-index with r indices and $\dim \iota_i = \dim U_i$. We will refer to the element tensor as a rank r tensor, i.e. rank 2 is a matrix, rank 1 is a vector, and rank 0 is a scalar. Each element tensor index ι_i corresponds to the degree of freedom/basis function number ι_i in the finite element space U_i, i.e., $N^0_{\iota_0}$ represents basis function number ι_0 in the space of test functions U_0 (rank $\geq$ 1) and $N^1_{\iota_1}$ represents basis function number ι_1 in the space of trial functions U_1 (rank=2). The coefficient functions w^k are given[2] functions, represented by a set of degrees of freedom $\{w_j^k\}$. In the case of integration by quadrature, the coefficients may be represented as the point-wise evaluation of a function in quadrature points, but in the general case $\{w_j^k\}$ will be defined by the degrees of freedom of the finite element space used to represent the coefficient, with $w^k = \sum_j w_j^k N_j^{w^k}$. See [Alnæs et al. 2008; Alnæs et al. 2009] for more details.

To clarify the distinction between forms of various ranks we consider a few examples. The rank 2 form without any coefficients

$$A_{ij} = a(N_i^0, N_j^1) = \int_T N_i^0 \, N_j^1 \, d\mathbf{x}$$

produces a matrix (the mass matrix), while the rank 1 form with one coefficient

$$A_i = a(N_i^0; w^0) = \int_T N_i^0 \, w^0 \, d\mathbf{x}$$

[2]From an implementation point of view they are typically prescribed at run-time prior to the element tensor computations.

produces a vector (the load vector or source vector), and the rank 0 form with two coefficients

$$A = a(;w^0, w^1) = \int_T w^0\, w^1\, d\mathbf{x}$$

produces a scalar (the L_2 inner product of w^0 and w^1). Of course, we can relate the forms of rank 2, 1 and 0 obtained from the same variational form a as above by $A_i = \sum_j A_{ij} w_j^0$ and $A = \sum A_i w_i^1$.

Finally, we note that the element Jacobian matrix of a rank 1 form with respect to its k'th coefficient is

$$A^T_{\iota_0,\iota_1} = J^k_{\iota_0,\iota_1} = \frac{\partial}{\partial w^k_{\iota_1}} a_T(N^0_{\iota_0}; w^0, \ldots, w^k, \ldots, w^{m-1}).$$

This is precisely what is needed when using Newtons method to solve a nonlinear PDE using the finite element method.

2.3 Integration Techniques

Computing the element tensor A^T involves integration of some expressions over an element, which can be handled in two different ways. In this section we will first discuss the mapping from a global element to a reference element, before we describe traditional integration by quadrature as well as our analytical integration approach.

On a general polygon T, the variational form and the finite elements are usually defined in terms of a mapped reference element[3]. This is done as follows. Let T be a polygon and $\hat{T}$ the corresponding reference polygon. Between the coordinates $\mathbf{x} \in T$ and $\boldsymbol{\xi} \in \hat{T}$ we use the mapping

$$\mathbf{x} = \mathbf{G}(\boldsymbol{\xi}) + \mathbf{x}_0, \tag{3}$$

and define the Jacobian determinant of this mapping as

$$J(\mathbf{x}) = \left| \frac{\partial \mathbf{G}(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \right|. \tag{4}$$

The basis functions are defined in terms of the basis function on the reference element as

$$N_j(\mathbf{x}) = \hat{N}_j(\boldsymbol{\xi}), \tag{5}$$

where $\hat{N}_j$ is basis function number j on the reference element. The integral can then be performed on the reference polygon,

$$\int_T f(\mathbf{x})\, d\mathbf{x} = \int_{\hat{T}} f(\boldsymbol{\xi})\, J d\boldsymbol{\xi}, \tag{6}$$

and the spatial derivatives are defined by the derivatives on the reference element

[3]This is not possible for all elements, e.g. the Rannacher-Turek [Rannacher and Turek 1992] element has better properties on anisotropic meshes when defined globally. The SyFi kernel supports both approaches, but only mapped elements are currently implemented in the form compiler.

	Triangle		Tetrahedron		Quadrilateral		Hexahedral	
p	n_q	q	n_q	q	n_q	q	n_q	q
1	3	(2)	4	(2)	4	(3)	8	(3)
2	6	(4)	11	(4)	9	(5)	27	(5)
3	12	(6)	24	(6)	16	(7)	64	(7)
4	16	(8)	43	(8)	25	(9)	125	(9)
5	25	(10)	126	(11)	36	(11)	216	(11)

Table I. For each element order p, the number of quadrature points used in a quadrature rule of order $q = 2p$ or $q = 2p + 1$.

and the geometry mapping simply by using the chain rule,

$$\frac{\partial N}{\partial x_i} = \frac{\partial N}{\partial \xi_j}\frac{\partial \xi_j}{\partial x_i}. \tag{7}$$

If we let G_T denote the set of variables depending on the geometry of the cell T (e.g. $\mathbf{x}_0$, $\mathbf{G}$, and $\mathbf{n}$), and $W_T = \{w_j^i\}$ denote the set of degrees of freedom for all coefficients, we can write the expressions for the element tensor entries as

$$A_\iota = \int_{\hat{T}} f_\iota(\boldsymbol{\xi}, G_T, W_T)\, d\boldsymbol{\xi}. \tag{8}$$

The traditional approach is to perform numerical integration by quadrature, which means approximating the integral with a weighted sum of the integrand evaluated in certain points,

$$\int_{\hat{T}} f_\iota(\boldsymbol{\xi}, G_T, W_T)\, d\boldsymbol{\xi} \approx \sum_{i=0}^{n_q-1} \omega_i f_\iota(\boldsymbol{\xi}_i^q, G_T, W_T),$$

where the points $\boldsymbol{\xi}_i^q$ and weights ω_i together form a quadrature rule. For triangles and tetrahedrons we use the economical Gauss rules found in [Solin et al. 2004], and for quadrilaterals and hexahedrons we use simple tensor products of $1D$ Gauss rules. The order of the quadrature rule can be specified as an option when compiling the form. Table I shows the number of quadrature points in some of the these rules.

Alternatively, analytical integration can be applied. Symbolic integration is slower than integration by quadrature, but this can be done as a preprocessing step prior to code generation by the form compiler. This way the dependency on $\mathbf{x}$ or $\boldsymbol{\xi}$ is integrated away in the expressions we generate code from, and we obtain a set of explicit expressions for the integrals

$$\int_{\hat{T}} f_\iota(\boldsymbol{\xi}, G_T, W_T)\, d\boldsymbol{\xi} = F_\iota(G_T, W_T), \tag{9}$$

where the functions $F_\iota(G_T, W_T)$ are often much simpler than $f_\iota(\boldsymbol{\xi}, G_T, W_T)$. This is particularly the case when the geometry mapping is affine, the order of finite element spaces for test and trial functions is greater than 1 or 2, and with forms where the dependencies on G_T and W_T are simple. Thus we can expect the analytic integration to be more beneficial when using high order basis functions and less so with coefficient functions that are of high order or occur in complex nonlinear terms. We will demonstrate this in the efficiency comparisons described below.

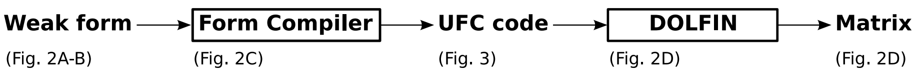

Fig. 1. Information flow from weak form of the PDE to global matrix assembly. The form compiler (SFC) takes a symbolic representation of the weak form and produces a C++ program which implements the UFC interface. This program is used as a computational kernel in the DOLFIN Assembler to produce the global matrix.

```
from sfc import *
# A) Define elements and arguments
element = FiniteElement("Lagrange", "triangle", 1)
v = TestFunction(element)
u = TrialFunction(element)

# B) Define integrand
def mass(v, u, itg):
    return inner(u, v)

# C) Generate and compile code
form = Form(basisfunctions = [v, u])
form.add_cell_integral(mass)
compiled_form = compile_form(form)

# D) Assemble global vector and matrix
from dolfin import *
mesh = UnitSquare(10, 10)
M = assemble(compiled_form, mesh)
```

Fig. 2. Code for defining and assembling a mass matrix

Symbolic integration requires $f\iota$ to be possible to integrate automatically. Because of limitations in GiNaC, they must be polynomials. Polynomials can always be obtained by taking the Taylor series, using the moment basis in GiNaC, but this may not always be stable or efficient. Other symbolic engines may manage to integrate some more complicated expressions, but in general a similar limitation will still apply. Additionally, the coefficients must be represented by a polynomial such as a field over a finite element space, whereas when using quadrature the coefficients can be evaluated point-wise in quadrature points. Thus we wish to apply analytic integration when it improves performance and quadrature where dictated by practical and theoretical limitations.

3. DEFINING AND COMPILING FORMS

The SyFi Form Compiler (SFC) is a Python module which takes as input a symbolic definition of a variational form or functional and a choice of finite element spaces. The compiler produces as output C++ code which can compute the element tensor given cell and coefficient data, as well as code for the finite elements and local to global mapping. SFC uses the Python interface Swiginac of the symbolic C++ library GiNaC as the base of its input language. On top of this general symbolic engine, common operators for PDEs are defined, the most important being the

```
/// Tabulate the tensor for the contribution from a local cell
void cell_itg__mass__Lagrange_1_2D::tabulate_tensor(double* A,
                                                    const double * const * w,
                                                    const ufc::cell& c) const
{
    // geometric quantities
    double x0 = c.coordinates[0][0];  double y0 = c.coordinates[0][1];
    double x1 = c.coordinates[1][0];  double y1 = c.coordinates[1][1];
    double x2 = c.coordinates[2][0];  double y2 = c.coordinates[2][1];
    double G00 = x1-x0;
    double G01 = x2-x0;
    double G10 = y1-y0;
    double G11 = -y0+y2;
    double t6 = G11*G00;
    double t7 = -G10*G01;
    double t8 = t7+t6;
    double detG = fabs(t8);

    // local_tokens, product of optimization
    const double t10 = 8.3333333333333329e-02*detG;
    const double t11 = 4.1666666666666664e-02*detG;
    A[3*0 + 0] = t10;
    A[3*0 + 1] = t11;
    A[3*0 + 2] = t11;
    A[3*1 + 0] = t11;
    A[3*1 + 1] = t10;
    A[3*1 + 2] = t11;
    A[3*2 + 0] = t11;
    A[3*2 + 1] = t11;
    A[3*2 + 2] = t10;
}
```

Fig. 3. Snippet of generated C++ code for computing the mass matrix

differential operators `grad`, `div`, and `curl`, as well as operators like `dot` and `inner`[4]. The basis functions of the finite elements are provided by the SyFi kernel.

Figure 1 shows the information flow from the hand written user implementation of a weak form, through the form compiler to C++ code which is used to assemble the global matrix by the DOLFIN library. In the following we first step through an example Python code and show the resulting generated code, before we go into some more details about the code generation process.

Figure 2 shows a Python script which defines and computes the mass matrix using linear Lagrange elements on triangles. We will explain each step in the script below, but refer to the SyFi manual for more details about the user interface. Note that step D uses PyDOLFIN.

In step A we define a finite element in terms of its family name, the order and polygon type. Using this element, we construct the arguments of the variational form, namely the test and trial functions.

[4]Notice that the inner product of vectors or the contraction of matrices is denoted by `inner`, while matrix vector multiplication (or the inner product of vectors) is denoted by `dot`.

Step B defines the integrand in terms of a callback function `mass`. This function takes as input explicit symbolic expressions (swiginac objects) for its arguments and returns the corresponding integrand for a single element tensor entry.

In step C a form is defined with the test and trial functions from step A as arguments. When calling `add_cell_integral`, the integrand of each element matrix entry A^T_{ij} is computed by one call to `mass` with the test function $v = \phi^0_i$ and trial function $u = \phi^1_j$, and the resulting symbolic expression for each integrand is integrated analytically. In the call to `compile_form`, corresponding C++ code is generated by the form compiler. This C++ code is an implementation of the UFC interface. The generated C++ code is compiled and linked into a Python extension module, which is loaded dynamically into Python, a kind of Just-In-Time compilation[5]. The code for computing this element matrix is shown in Figure 3.

Finally, to assemble a global tensor, the UFC implementation must be combined with other software components like a mesh library and a linear algebra library, tied together with an assembly algorithm. See [Alnæs et al. 2009] for a discussion of how the UFC interface is intended to be combined with other software components. Step D shows how this step looks in the PyDOLFIN [Hoffman et al. 2008a; 2008b] problem solving environment.

3.1 Generating Efficient Code

There are two main reasons for the speed-up when comparing SFC generated code with conventional quadrature codes: SFC 1) performs some computations prior to the code generation and 2) generates low-level and problem-specific code. In this section we will describe the techniques used in SFC, before several examples demonstrating significant speed-up is shown in the next section.

The code generation in SFC relies on a few simple principles. The overall code structure is provided by templates for each class in the UFC interface. These templates are distributed with UFC. Formatting of a symbolic expression as a C expression is provided by GiNaC. A variable in the generated code is represented before code generation as a symbolic (symbol, expression) tuple we call a token, which can be formatted as a variable declaration, definition, assignment or accumulation using some simple helper functions. A block of the program in SSA form (Single Static Assignment, where each variable is assigned a value only once) is represented as a list of tokens. Generating code for a sequence of variable assignments or declarations can thus be done using a single function call, given a symbolic SSA form (a list of tuples of symbols and expressions). In other words, the symbolic expressions are directly inlined in the generated C++ code.

There are many ways a symbolic engine can be used for doing computations prior to the code generation and we have only tested a few techniques, so far. Our greatest success comes from employing analytical integration on the entries in the element tensor. By doing this we remove the spatial dependency of the integrand function, as seen in equation (9), and hence the quadrature loop. With this technique it is also easy to estimate the speed-up, as will be done in the next section.

In section 3 we saw that the form compiler gets the user code for an integrand as

[5]Using the package Instant [Mardal and Westlie 2008], which caches the compiled modules and compares MD5 sums of the source code to avoid recompilation if the source code doesn't change.

a callback function. In the simplest case, the form compiler calls the user code to compute the integrand expression for each combination of basis functions, integrates the expression analytically, and stores the resulting expressions in symbolic SSA form from which we can easily generate C++ code. In the case of quadrature code, the tokens can be analyzed to split them in one SSA form to be computed outside the quadrature loop and one inside the quadrature loop based on their dependencies.

We can also apply further optimizations on the symbolic SSA form. Constant propagation and removal of unused variables is fairly easy in symbolic SSA form. Common Subexpression Elimination (CSE) is implemented in two steps. The first step uses the knowledge that many element tensors inhibit symmetries, and detects equal element tensor entries directly by inserting their expressions in a hashmap. The second step is a single sweep over subexpressions, creating new variables for each operation and reusing them where possible. This is not a particularly good algorithm, since it scales poorly and does not identify all common subexpressions. Neither does it consider the effect of too many temporary variables, and as a result it may actually degrade the performance in some cases. A similar technique is likely used by the C++ compiler, since we compile the code with optimization (-O2). Still, as will be shown, it may produce a significant speed-up when applied prior to code generation.

The second reason for our efficiency is that we generate low-level and problem-specific code. This is in some sense similar to template metaprogramming in C++, where user-level abstractions are removed during the C++ compilation stage and code is inlined. With either technique parts of expressions can be removed at compile time. For example, as pointed out in [Prud'homme 2006], when writing expressions like dot(u, v) then the terms associated with zeros in the vectors u and v may cancel automatically, yielding a smaller expression. Instead of exploiting the template engine of the C++ compiler, we generate explicitly inlined C++ expressions. Notice further that (problem-specific) code generation offers greater flexibility than template metaprogramming in C++. For example, when one or more of the coefficients are piecewise constants, computations related to these coefficients may be performed outside the quadrature loop.

Other traditional code optimization techniques are inlining and loop unrolling, which are both natural results of our code generation approach since we produce explicit expressions for each element tensor entry.

Finally, the generated code is low-level without abstractions and external dependencies. Therefore the C++ compiler is not hindered by abstraction barriers like virtual functions when optimizing the machine code.

4. EXAMPLES DEMONSTRATING EFFICIENCY

Below we will look at a few examples of variational forms, and present comparisons of the time taken to run the generated code versus more traditional hand-written quadrature based code. Note that all the examples use an affine geometry mapping.

In the performed tests we have measured the time to compute a single element tensor by computing element tensors in a loop, without the overhead of matrix insertion and mesh iteration found in the actual assembly algorithm. Therefore, the speed-up presented here is only a part of the assembly of a global tensor. The actual speed-up seen in an application depends on the mesh and linear algebra library in use, and is limited by the fraction of the assembly time spent on the element tensor computation in the first place. An approximate timing of the Deal.II codes show us that the element tensor computation for computing the stiffness matrix take from 30% for linear elements to 86% for fifth order elements. With SFC/Dolfin a similar test shows that the element tensor computations vary from 2% for linear elements to 6% for fifth order elements. The Deal.II assembly process is about 30% faster than SFC/Dolfin on linear elements, but more than twice as slow for fifth order elements. Deal.II uses quadrilateral elements while SFC/Dolfin uses triangular elements and in these experiments the number of degrees of freedom is the same. Since Dolfin uses triangles the mesh then consists of twice as many cells.

All codes are compiled using g++ 4.1 with the optimization flag -O2, and run on a Dell XPS M1710 with an Intel T2600 @ 2.16 GHz CPU (using a single core only). Measured times varied about 5% between runs, which is well within the accuracy of interest. A subset of the tests has been run on a different computer, achieving similar relative times.

Code is generated by SFC using both analytic integration and quadrature, and for comparison with external software we have chosen FFC, Deal.II and Diffpack. Deal.II and Diffpack are C++ libraries for the finite element method using quadrature. FFC has an approach similar to the analytic integration in SFC, known to produce very efficient code [Kirby and Logg 2007]. Note that FErari [Kirby et al. 2005; Kirby et al. 2006], an optimizing backend used by FFC, was not available in FFC during these tests.

Not all libraries could be compared in each testcase. FFC only supports simplex elements, and Deal.II only supports hypercube elements, while SyFi supports both. Diffpack supports both polygon types, but only low order elements. Optimized versions of SFC code (with CSE applied) are included in the benchmarks only in the cases optimization resulted in a definite speed-up.

Below we will present mathematical definitions of the forms used as testcases, along with an analysis and discussion of the test results. Note that while we use Lagrange elements in all examples, the conclusions are independent of the actual element type. SFC code which defines the integrands of these forms is shown in Figure 4.

For each test case we measured code both generation time (time spent by SFC alone) and total compile times (time spent by SFC and GCC). In most cases presented here the total code generation and compilation time is between 10 s and a minute (when the code is not cached). For each test case we discuss the cases where compilation becomes slower or breaks down.

4.1 Example: Mass Matrix

The element mass matrix is

$$A_{ij} = a(N_i, N_j) = \int_T N_i(\mathbf{x})\, N_j(\mathbf{x}) d\mathbf{x}. \tag{10}$$

Applying analytic integration results in

$$A_{ij} = M_{ij} J, \tag{11}$$

```
def mass(v, u, itg):
    return inner(u, v)

def rhs_vector(v, f, itg):
    return inner(f, v)

def stiffness(v, u, itg):
    GinvT = itg.GinvT()
    Du = grad(u, GinvT)
    Dv = grad(v, GinvT)
    return inner(Du, Dv)

def convection_vector(v, w, itg):
    GinvT = itg.GinvT()
    Dw = grad(w, GinvT)
    return dot(dot(w, Dw), v)

def convection_jacobi(v, w, itg):
    GinvT = itg.GinvT()
    Du = grad(u, GinvT)
    Dw = grad(w, GinvT)
    return dot(dot(w, Du) + dot(u, Dw), v)

def power_functional(w, itg):
    p = 2
    return w**p
```

Fig. 4. Definition of example forms with SFC

where M_{ij} are real numbers and J is the Jacobian determinant of the affine geometry mapping. Hence, the computation of the element mass matrix will consist of computing J, plus one floating point multiplication per entry in the matrix regardless of the choice of element and its order. Thus the computational cost of one entry in the element matrix is constant, and dominated by the cost of memory access. In contrast, when using numerical integration the cost per entry is proportional to the number of quadrature points. Figure 3 shows the code generated by SFC for computing this element matrix. The timing results presented in Table II, Table III, and Figure 7 clearly show the large speed-up resulting from analytic integration. This speed-up ranges from a factor 50 to a factor 800 compared to Deal.II. It is also evident that the generated code using quadrature is several times faster than the handwritten C++ codes.

Compilation times are generally low for this simple form, but grows to about a minute for a 5th order tetrahedron element and blows up faster for hexahedron elements, from several minutes for a 4th order element to about an hour for the 5th order element.

```
from sfc import *

# Define elements and arguments
element = VectorElement("Lagrange", "triangle", 1)
v = TestFunction(element)
w = Function(element)

# Define integrand
def convection_vector(v, w, itg):
    GinvT = itg.GinvT()
    Dw = grad(w, GinvT)
    return dot(dot(w, Dw), v)

# Define forms
F_form = Form(basisfunctions = [v], coefficients   = [w])
F_form.add_cell_integral(convection_vector)
J_form = Jacobi(F_form)

# Generate and compile code
compiled_F_form = compile_form(F_form)
compiled_J_form = compile_form(J_form)
```

Fig. 5. Code for defining and compiling forms for a convection vector and its Jacobi matrix

```
# Assemble global vector and matrix using PyDOLFIN
from dolfin import *

class W(cpp_Function):
    def __init__(self, mesh):
        cpp_Function.__init__(self, mesh)
    def rank(self):
        return 1
    def dim(self, i):
        return 2
    def eval(self, v, x):
        v[0], v[1] = x[0], x[1]

mesh = UnitSquare(10, 10)
w_function = W(mesh)

# Assemble global vector and matrix
F = assemble(compiled_F_form, mesh, coefficients = [w_function])
J = assemble(compiled_J_form, mesh, coefficients = [w_function])
```

Fig. 6. PyDOLFIN code for assembling the compiled convection vector and Jacobi matrix forms defined in Figure 5

4.2 Example: Right Hand Side Vector

The element right hand side vector, often called the load vector or source vector, is

$$A_i = a(N_i; w) = \int_T N_i(\mathbf{x})\, w(\mathbf{x})\, d\mathbf{x} = \int_{\hat{T}} \hat{N}_i(\boldsymbol{\xi}) \sum_{k=0}^{N-1} w_k \hat{N}_k(\boldsymbol{\xi})\, J\, d\boldsymbol{\xi}, \qquad (12)$$

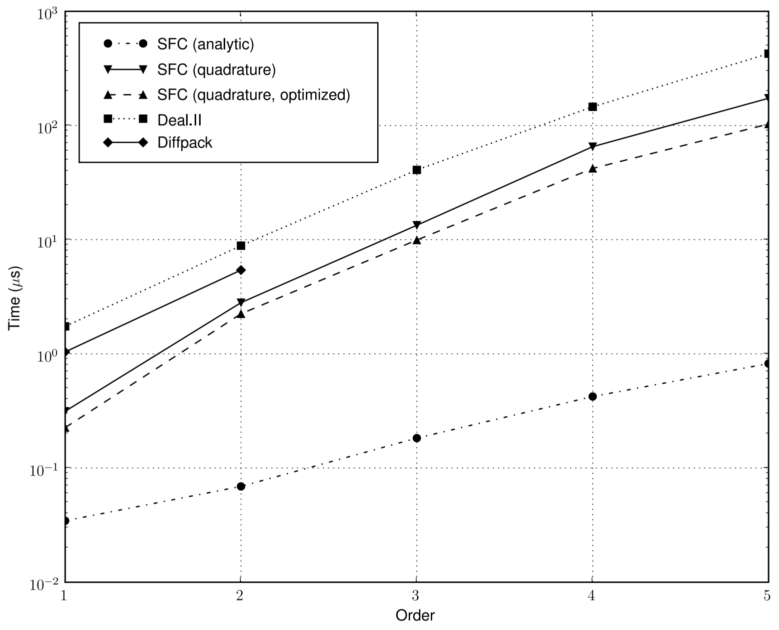

Fig. 7. Time to compute the element tensor of the mass form on quadrilateral elements, in μs

	Triangle					Tetrahedron			
Order	1	2	3	4	5	1	2	3	4
Timescales (μs)	0.024	0.039	0.083	0.16	0.29	0.051	0.095	0.28	0.82
SFC	1.0	1.0	1.0	1.0	1.0	1.00	1.0	1.0	1.0
SFC (quad.)	7.3	26.8	61.8	114.2	177.9	6.6	48.1	161.8	333.2
SFC (quad., opt.)	5.4	18.7	46.0	71.4	111.0	5.3	36.6	104.1	198.5
FFC	1.0	1.1	1.0	1.1	1.1	0.9	1.0	1.0	1.1
Diffpack	19.0	56.4	–	–	–	15.1	–	–	–

Table II. Time to compute the element tensor of the mass form, relative to a symbolic integration for each order.

where the coefficient w is assumed to be a finite element field, i.e., $w = \sum_{k=0}^{N-1} w_k N_k$. Analytic integration results in

$$A_i = J \sum_k M_{ik} w_k \equiv F_i(J, w_k), \tag{13}$$

where F_i will be linear polynomials in $\{w_k\}$. The number of floating point operations per element vector entry is proportional to the degrees of freedom per element, N when using analytical integration, while it is proportional to the number of quadrature points in quadrature based implementations. This reduces the benefit of analytical vs numerical integration, as seen in the testcase using quadri-

	Quadrilateral					Hexahedron			
Order	1	2	3	4	5	1	2	3	4
Timescales (μs)	0.035	0.069	0.18	0.42	0.82	0.072	0.49	5.45	21.8
SFC	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
SFC (quad.)	9.1	40.7	73.2	154.4	210.8	32.6	204.9	264.2	–
SFC (quad., opt.)	6.5	32.5	54.3	99.5	125.4	25.0	120.1	201.5	379.5
Deal.II	50.4	128.4	223.5	345.2	518.3	160.8	404.5	453.2	811.2
Diffpack	30.1	78.6	–	–	–	88.3	228.2	–	–

Table III. Time to compute the element tensor of the mass form, relative to a symbolic integration for each order.

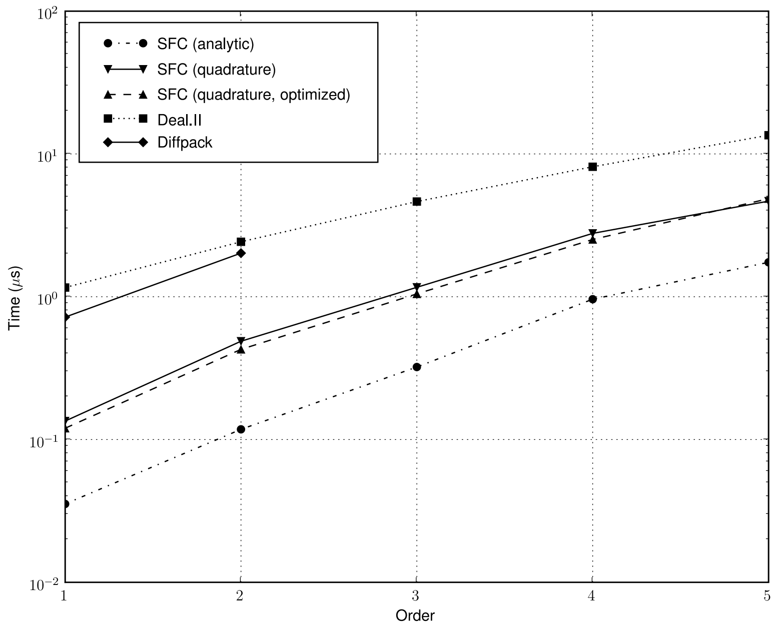

Fig. 8. Time to compute the element tensor of the rhs_vector form on quadrilateral elements, in μs

lateral elements presented in Figure 8. Here the speed-up of SFC with analytic integration vs Deal.II varies from a factor 32 for linear elements to a factor almost 8 for 5th order elements, while the speed-up vs generated quadrature code varies from a factor 3.5 to 2.5.

Although the number of element vector entries grows slower than the number of entries in the element mass matrix (n vs n^2), the complexity of the integrand increases faster. As a result of this complexity, the code generation is a bit slower than for the mass matrix, and grows from a couple of minutes for 3rd order hexahedron elements to several hours for 4th order hexahedron elements.

	Triangle					Tetrahedron			
Order	1	2	3	4	5	1	2	3	4
Timescale (in μs)	0.057	0.10	0.28	0.8	1.5	0.14	0.7	2.6	11.3
SFC (analytic)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
SFC (quadrature)	2.7	9.6	17.0	17.3	24.9	2.1	8.6	20.4	36.5
FFC	0.8	0.7	0.7	0.6	0.7	0.7	0.4	0.5	0.8
Diffpack	10.5	31.0	–	–	–	8.2	–	–	–

Table IV. Time to compute the element tensor of the stiffness form for each order respectively on triangle and tetrahedron elements.

	Quadrilateral					Hexahedron			
Order	1	2	3	4	5	1	2	3	4
Timescale (in μs)	0.073	0.24	0.52	1.2	3.2	0.36	2.3	20.6	75.8
SFC (analytic)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
SFC (quadrature)	4.5	11.7	26.1	41.7	68.5	7.8	52.2	83.8	209.6
Deal.II	31.6	52.8	109.3	169.2	186.1	42.3	123.0	166.9	332.4
Diffpack	18.1	37.1	–	–	–	27.5	105.4	–	–

Table V. Time to compute the element tensor of the stiffness form for each order respectively on quadrilateral and hexahedron elements.

4.3 Example: Stiffness Matrix

The element stiffness matrix is

$$\begin{aligned} A_{ij} = a(N_i, N_j) &= \int_T \nabla N_i(\mathbf{x}) \cdot \nabla N_j(\mathbf{x})\, d\mathbf{x} \\ &= \int_{\hat{T}} \mathbf{G}^{-T} \nabla \hat{N}_i(\boldsymbol{\xi}) \cdot \mathbf{G}^{-T} \nabla \hat{N}_j(\boldsymbol{\xi})\, J\, d\boldsymbol{\xi}. \end{aligned} \tag{14}$$

Analytic integration results in

$$A_{ij} = F_{ij}(\mathbf{G}^{-T}, J), \tag{15}$$

where the polynomials $F_{ij}(\mathbf{G}^{-T}, J)$ are quadratic in $\mathbf{G}^{-T}$ and linear in J. Tables IV and V and Figure 9 shows the timing results. Here we see a growing speed-up with element order similar to the behavior in the mass matrix testcase, but to a lesser extent because the integrated expressions are more complicated. The highest observed speed-up is 332.

Code generation times grow to about ten minutes for hexahedron 3rd order and tetrahedron 5th order elements, and several hours for 4th order hexahedron elements. The time spent is dominated by analytic integration of the large number of element tensor entries.

4.4 Example: Nonlinear Convection Vector

The nonlinear convection vector is

$$A_i = a(\mathbf{N}_i; \mathbf{w}) = \int_T \mathbf{w} \cdot \nabla \mathbf{w}\, \mathbf{N}_i\, d\mathbf{x}. \tag{16}$$

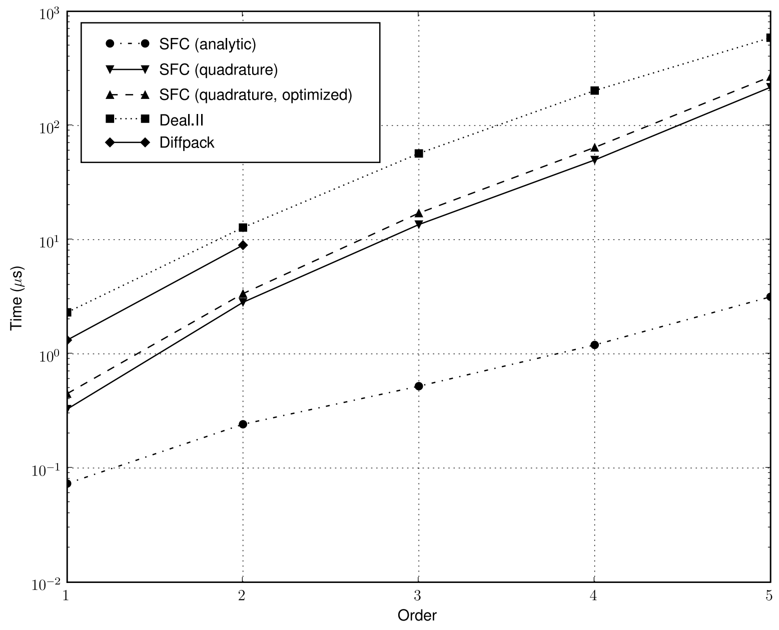

Fig. 9. Time to compute the element tensor of the stiffness form on quadrilateral elements, in μs

Analytic integration results in

$$A_i = F_i(\mathbf{G}^{-T}, J, \mathbf{w}), \tag{17}$$

where $F_i(\mathbf{G}^{-T}, J, \mathbf{w})$ are polynomials that are linear in $\mathbf{G}^{-T}$ and J and quadratic in $\mathbf{w}$. Figure 5 shows definition and computation of this form in SFC, and Figure 10 shows the timing results. A PyDOLFIN example performing assembly of the global tensors is shown in Figure 6.

The reason for the poor performance of analytic integration in this case is the product of the coefficient function $\mathbf{w}$ with its gradient. To perform the analytic integration, the functions $\mathbf{w}$ and $\nabla\mathbf{w}$ are expanded in their polynomial finite element basis. While integration still removes the spatial dependencies, the resulting expressions can be written on the form

$$F_i = J \sum_{j=1}^{n} \sum_{k=1}^{n} M_{jk}(\mathbf{G}^{-T}) w_j w_k.$$

Thus the number of operations to compute the length n element vector is proportional to n^3, with $n = |W^k|$. A good factorization algorithm capable of factoring multiple expressions at once would be needed to optimize this.

In contrast, the quadrature code can compute $\mathbf{w}$ and $\nabla\mathbf{w}$ once per quadrature point, and the operation count per element vector entry is constant, yielding a number of operations proportional to $n_q n$, with n_q being the number of quadrature

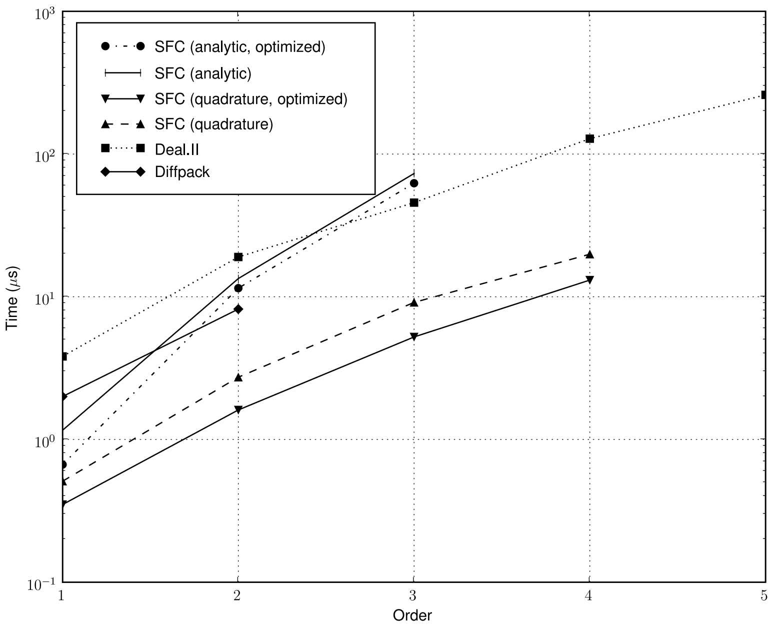

Fig. 10. Time to compute the element tensor of the convection_vector form on quadrilateral elements, in μs

points.

Note that quadrature rules of order $q = 2p$ or $q = 2p+1$ have been used, where p is the element order. The polynomial order of the integrand here is actually $3p$, which means we are under-integrating. Increasing q here to achieve exact integration with quadrature will increase the computation time with less than a factor 3 which still leaves the analytical integration slower than the generated quadrature code.

For quadrilateral elements of orders 1 and 2, the convection vector form compiled in less than a minute. However, the generation time exploded to a couple of hours for cubic elements and failed to complete within a day for higher order. For 4th and 5th order elements on triangles the generation times were about ten minutes and five hours respectively. For tetrahedra this blowup occurred at cubic elements and for hexahedral elements only linear elements finished within a day.

4.5 Example: Convection Jacobian Matrix

The Jacobian element matrix of the nonlinear convection form from Example 4.4 is

$$\begin{aligned} A_{ij} &= \frac{d}{dw_i}\left[a(\mathbf{N}_j;\mathbf{w})\right] = \frac{d}{dw_i}\int_T \mathbf{w}\cdot\nabla\mathbf{w}\,\mathbf{N}_j\,d\mathbf{x} \\ &= \int_T (\mathbf{w}\cdot\nabla\mathbf{N}_j + \mathbf{N}_j\cdot\nabla\mathbf{w})\cdot\mathbf{N}_i\,d\mathbf{x}. \end{aligned} \tag{18}$$

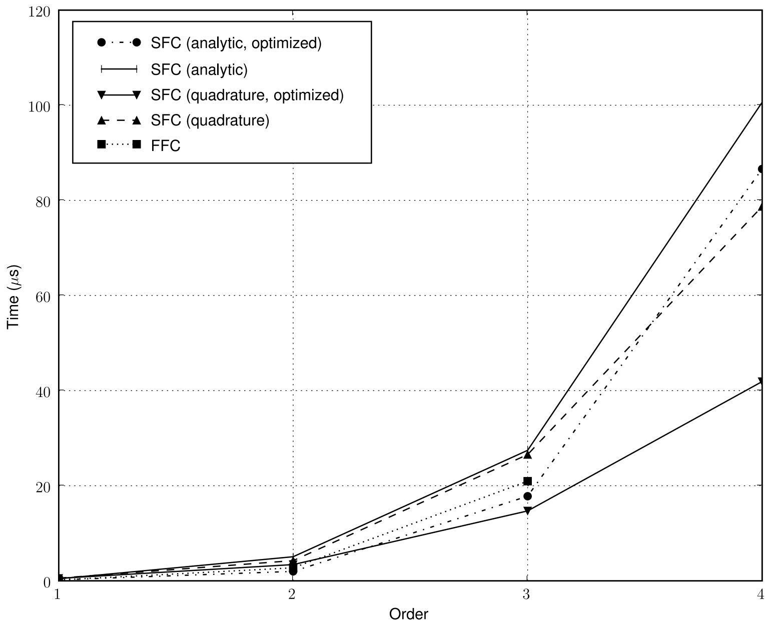

Fig. 11. Time to compute the element tensor of the convection_jacobi form on triangle elements, in μs

Applying analytic integration results in

$$A_{ij} = F_{ij}(\mathbf{G}^{-T}, J, \mathbf{w}), \tag{19}$$

where $F_{ij}(\mathbf{G}^{-T}, J, \mathbf{w})$ are linear polynomials in $\mathbf{G}^{-T}$, J and $\mathbf{w}$. The code in Figure 5 shows definition and computation of this form in SFC, and Figure 11 shows the timing results.

In this case the differences in computational time are much smaller, with only about a factor 2.5 between the extreme cases. Analytic integration with SFC and the tensor representation in FFC are roughly equivalent, while the optimized quadrature code is faster than the other approaches with a factor 2 - 2.5 when using cubic elements.

Note that the quadrature rules are the same ones used for nonlinear convection, and in this case using exact rules would make the quadrature code slower than the code using analytic integration.

Compilation and generation times for the convection Jacobi form relate to the convection vector similarly to the relation between the mass matrix and the rhs vector. The matrix has more entries but the vector expressions are more complicated. Generation and compilation times blow up for the convection Jacobi as well, but at somewhat higher element orders.

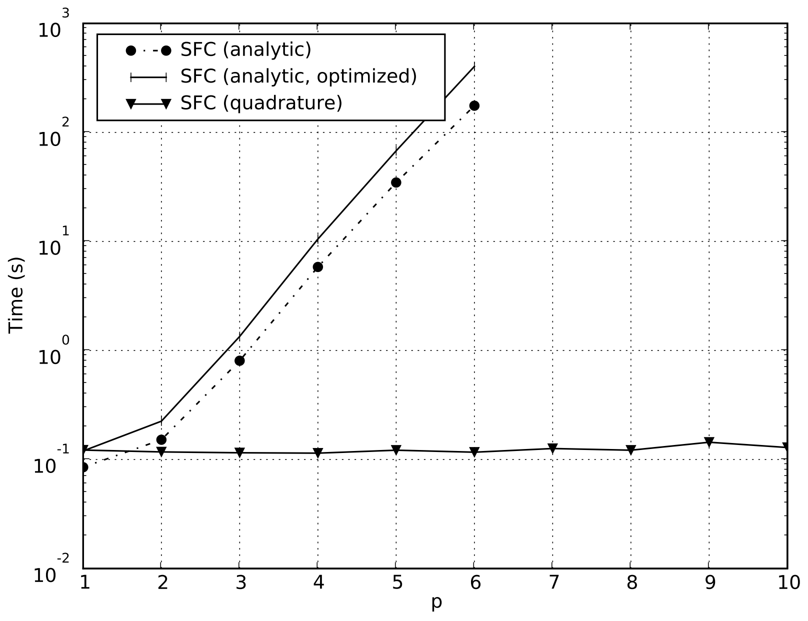

Fig. 12. Time to generate code for the power functional on quadratic tetrahedron elements, in s

4.6 Example: Power Functional

By the power functional we mean

$$a(;u) = \int_\Omega u^p \, dx, \tag{20}$$

with $u \in V_h$. In the following V_h is taken to be a Lagrange finite element space of degree q on a cell K, and the integer exponent p can be varied over a suitable range. Inserting the finite sum of basis functions and degrees of freedom for u, the integrand polynomial on a cell K becomes

$$F(u) = u^p = \left(\sum_{i=1}^{n_q} u_i \phi_i(x) \right)^p, \tag{21}$$

where $n_q = |V_h^K|$ is the dimension of the local finite element space. To integrate this expression, the polynomial F is expanded into monomials, which results in an exponential growth in the number of terms as p grows, roughly estimated as $O(n_q^p)$.

To demonstrate the limits of the symbolic integration approach clearly, we show run time and code generation time for this functional with increasing p. An exact quadrature rule of order $q * p$ is chosen for this comparison[6].

[6]Unlike the other tests, these were run on a computer with an Intel Xeon L5420 2.5 GHz CPU (using a single core) and 8 Gb RAM.

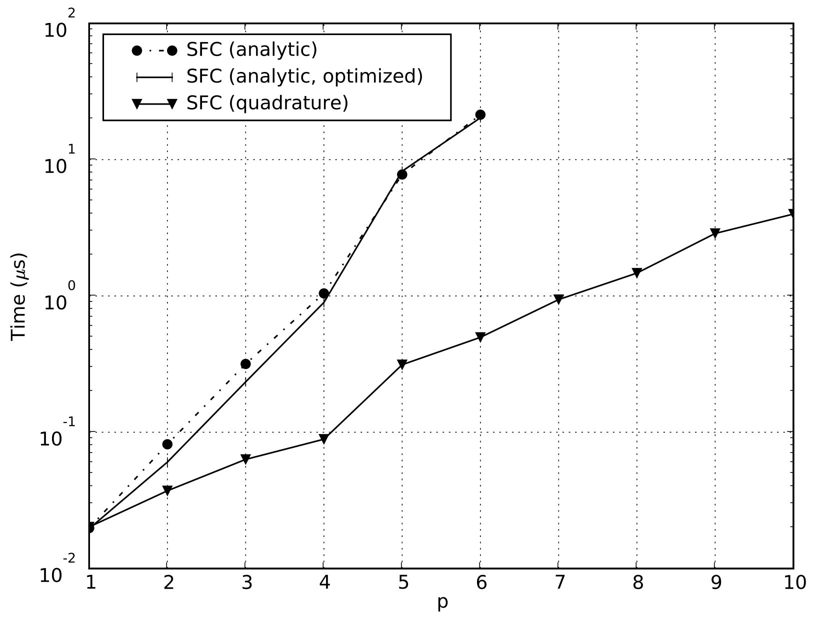

Fig. 13. Time to compute the power functional on quadratic tetrahedron elements, in μs

We measured code generation time and time to compute the functional on a single cell on triangles and tetrahedra with $q = 1, 2, 3$ and p in the range $[1, 10]$. With $q = 1$ no problems occurred, but there was no significant efficiency gain and generated quadrature code was faster for $p > 2$. With $q = 3$ the symbolic computations break down for $p > 4$.

Results from the power functional tests with $q = 2$ on tetrahedra are presented in more detail here. Figure 12 shows code generation time for analytically integrated and quadrature based code over varying p. While quadrature code generation time is a constant fraction of a second independent of p, the analytic integration approach gives exponentially growing time with increasing p as anticipated. For $p > 6$ the code generation broke down because of excessive memory usage. Looking at Figure 13, we see that the time to compute the functional is not improved by this precomputation. In fact, the generated quadrature code is faster for all $p > 1$ and shows a slower growth in run time with increasing p, so there is no reason to choose the analytic approach here.

5. DISCUSSION

5.1 Computational Efficiency

As demonstrated, symbolic computations combined with code generation can often lead to high computational efficiency. The code generated with analytical integration often outperforms integration by quadrature with orders of magnitude. Furthermore, the generated code based on quadrature is several times more efficient than corresponding code in Deal.II and Diffpack, even though the codes essentially performs the same operations in the same language. Writing the same kind of low-level code by hand would be tedious, error prone, and very inflexible, which is why FEM libraries provide abstractions in the first place. By positioning the user-level abstractions prior to the compilation phase, the low-level code is automatically tailored to the problem at hand with no additional manual work.

We have in this paper mainly considered two techniques, analytical integration and common subexpression elimination (CSE). Usually, approximate speed-up can be predicted by counting the necessary operations in the code. However, it is worth noting that none of these techniques always produce efficient code. For instance, in the case with complex material laws for hyper-elastic materials considered in [Alnæs et al. 2007], analytic integration can produce huge expressions and corresponding C++ code, which is not always possible to compile. Furthermore, in cases where analytic integration and CSE gave speed-up, combining the techniques did not necessarily improve the performance. The reason for this lack of improvement can be due to the fact that our CSE routine is fairly primitive, and also that the C++ compiler performs similar optimizations. Benchmarks of code optimized like this have only been shown for the cases it resulted in a speed-up.

5.2 Metaproblems

Our approach is a form of metaprogramming, since the program we write using symbolic computing has a C++ program as its output data. Metaprogramming carries its own set of problems and disadvantages.

Debugging a C++ application can be difficult enough with all the tools a programmer has at his disposal. When a bug is located in generated C++ code, it cannot (or should not) be fixed directly, since it reflects a bug in the form compiler or the metaprogram. Thus the problem must be possible to trace back to the form compiler and fixed at the actual source. The C++ code generated by SFC is fairly easy to follow, which makes this process manageable.

For complicated problems and high order elements, code generation, optimization, and compilation of the resulting code can carry high memory requirements and take much time. For example, analytic integration of the element mass matrix using 4th order hexahedron elements can take several hours to complete, because each of the 15625 (125^2) element tensor entries is integrated separately. The power functional example in subsection 4.6 involves the integration of only one expression, and demonstrates how the cost of even a single integral may be too high for practical usage.

Finally, techniques like loop-unrolling and inlining must be used with care. In Table III, timing of the element mass matrix with 4th order hexahedron elements and quadrature is left out because g++ required too much memory to compile the code using -O2. For similar reasons, the results seen in Figure 10 are truncated because of problems with code size and memory usage. This is usually a result of too much explicit inlining in the code generation.

5.3 Limitations

The technology presented here has several limitations, both theoretical and implementation specific. Analytic integration is not even possible for all nonlinear operators, and doesn't scale performance wise to more complicated forms. For some simple forms the run time performance is excellent for high order elements but the integration time becomes too high for most practical use.

By using quadrature we can still get performance benefits from code generation. The limitations in compilation of quadrature based code seen in the convection vector example is caused by flaws in our current implementation, mainly explicit inlining of all expressions in the code generation.

We have used SFC with quadrature for isotropic hyperelasticity with a Fung type material law, as presented in [Alnæs et al. 2007]. However, adding orthotropy or higher order elements makes the equations too complicated for the current framework.

Higher order geometries are useful for accurate description of smooth domains. These are not feasible for analytic integration since they contain the inverse of a geometry mapping. However, if implemented using quadrature this should not affect the code generation. Unfortunately, our software does not support this.

Automatic linearization using symbolic differentiation does not scale well. This is the approach taken by SFC when using analytic integration, since the element tensor entries are represented as monolithic symbolic expressions. A better approach to differentiation of programs is called automatic differentiation (AD) [Long 2004; Griewank 1989; Tadjouddine 2008] and should be applied instead. When using quadrature, the linearization implementation in the current SFC version is similar to a forward mode AD algorithm but with symbolic differentiation of partial expressions.

Work is in progress to fix the most pressing of these issues by avoiding explicit inlining in the code generation, and improved implementations of AD, with the goal of providing a form compiler that is robust w.r.t. more complicated equations.

6. CONCLUSION

Code generation from an abstract (user-friendly) problem definition allows domain specific optimizations exploiting knowledge of the problem that the C++ compiler does not have. In our case we have shown that employing a symbolic engine inside a finite element form compiler can lead to speed-up of several orders of magnitude in addition to a user-friendly and time saving problem solving environment. Our efforts have resulted in the open source package SyFi which generates UFC code that is directly importable in DOLFIN and other libraries implementing this thin interface.

REFERENCES

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

Alnæs, M. and Mardal, K.-A. 2008. *SyFi - Symbolic Finite Elements.* URL: `http://www.fenics.org/syfi/`.

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*. Accepted for publication. Preprint: `http://simula.no/research/scientific/publications/Simula.SC.96`.

Alnæs, M. S., Mardal, K.-A., and Sundnes, J. 2007. Application of symbolic finite element tools to nonlinear hyperelasticity. In *Fourth national conference on Computational Mechanics*

(MekIT'07), B. Skallerud and H. Andersson, Eds. Tapir Academic Press, NO-7005 Trondheim, 87–101.

Arnold, D. N., Falk, R. S., and Winther, R. 2007. Mixed finite element methods for linear elasticity with weakly imposed symmetry. *Math. Comp. 76*, 1699–1723.

Bangerth, W., Hartmann, R., and Kanschat, G. 2007a. deal.II — a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw. 33,* 4.

Bangerth, W., Hartmann, R., and Kanschat, G. 2007b. `deal.II` *Differential Equations Analysis Library, Technical Reference.* URL: `http://www.dealii.org`.

Bauer, C., Dams, C., Frink, A., Kisil, V. V., Kreckel, R., Sheplyakov, A., and Vollinga, J. 2007. GiNaC. URL: `http://www.ginac.de`.

Bauer, C., Frink, A., and Kreckel, R. 2002. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *Journal of Symbolic Computation 33*, 1–12.

Bruaset, A. M., Langtangen, H. P., et al. 2006. *Diffpack.* URL: `http://www.diffpack.com/`.

Crouzeix, M. and Raviart, P. 1973. Conforming and non–conforming finite element methods for solving the stationary Stokes equations. *RAIRO Anal. Numér. 7*, 33–76.

Dular, P. and Geuzaine, C. 2006. GetDP: a General environment for the treatment of Discrete Problems. URL: `http://www.geuz.org/getdp/`.

FEniCS. 2008. FEniCS project. URL: `http//www.fenics.org/`.

Griewank, A. 1989. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, M. Iri and K. Tanabe, Eds. Kluwer Academic Publishers, 83–108.

Hoffman, J., Jansson, J., Logg, A., and Wells, G. N. 2008a. DOLFIN. `http://www.fenics.org/dolfin/`.

Hoffman, J., Jansson, J., Logg, A., and Wells, G. N. 2008b. *DOLFIN User Manual.* URL: `http://www.fenics.org/dolfin/`.

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

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

Kirby, R. C., Logg, A., Scott, L. R., and Terrel, A. R. 2006. Topological optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput. 28,* 1, 224–240.

Langtangen, H. P. 2003. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming.* Springer-Verlag. 2nd edition, 855 pages.

Logg, A. 2008. FFC user manual. URL: `http//www.fenics.org/ffc/`.

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

Long, K. 2004. Efficient discretization and differentiation of partial differential equations through automatic functional differentiation. http://www.autodiff.org/ad04/abstracts/Long.pdf.

Long, K. 2006. Sundance. URL: `http://software.sandia.gov/sundance/`.

Mardal, K.-A., Tai, X.-C., and Winther, R. 2002. A robust finite element method for Darcy–Stokes flow. *SIAM J. Numer. Anal. 40*, 1605–1631.

Mardal, K.-A. and Westlie, M. 2008. Instant. URL: `http//www.fenics.org/instant`.

Nédélec, J.-C. 1980. Mixed finite elements in R^3. *Numer. Math. 35,* 3 (Oct.), 315–341.

Nédélec, J.-C. 1986. A new family of mixed finite elements in R^3. *Numer. Math. 50,* 1 (Nov.), 57–81.

Pironneau, O., Hecht, F., Hyaric, A. L., and Ohtsuka, K. 2006. FreeFEM. URL: `http://www.freefem.org/`.

Prud'homme, C. 2006. A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming 14,* 2, 81–110.

Rannacher, R. and Turek, S. 1992. A simple nonconforming quadrilateral Stokes element. *Numerical Methods for Partial Differential Equations 8*, 97–111.

Raviart, P. A. and Thomas, J. M. 1977. A mixed finite element method for 2-order elliptic problems. In *Mathematical Aspects of Finite Element Methods*. Lecture Notes in Mathematics, No. 606. Springer Verlag, 295–315.

Skavhaug, O. and Certik, O. 2008. Swiginac. URL: `http://swiginac.berlios.de/`.

Solin, P., Segeth, K., and Dolezel, I. 2004. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC. 855 pages.

Tadjouddine, E. M. 2008. Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal 51*, 688–699.