A Form Compiler based on Symbolic Computations

Martin Sandve Alnæs and Kent-Andre Mardal, Simula Research Laboratory

January 31, 2008

1 Introduction

We have combined symbolic mathematics with 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 and user–friendly format. 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 [4].

Our efforts have resulted in the open source software package SyFi [2], which is part of the FEniCS project [8]. SyFi stands for symbolic finite elements and is implemented in C++ and Python. SyFi is built on top of the symbolic C++ library GiNaC [6] and uses its Python interface Swiginac [22]. SyFi is largely divided in two: a kernel and a form compiler. We refer to the form compiler as SFC.

The 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 [5], the Crouzeix-Raviart element [7], the standard Lagrange elements of arbitrary order, the Nedelec elements [19, 20], the Raviart-Thomas element [21], and the robust Darcy-Stokes element [17]. The SyFi kernel is roughly comparable to FIAT [9, 11, 12, 10], which tabulates finite elements using numerical techniques.

The SyFi Form Compiler (SFC) can be compared to FFC [15, 16, 13, 14]. It takes as input a symbolic description of a variational form and a set of finite elements and generates low level C++ code. The generated code complies with the UFC [1, 3] interface, and can be used in DOLFIN to assemble matrices and vectors. SFC supports Just-In-Time compilation, via the package Instant [18], such that we can define the variational form within Python, then generate C++ code, compile and link this code into a Python extension module, and load the module dynamically into Python.

We next present a short code example using the SyFi form compiler together with PyDOLFIN, before we show some benchmark results comparing

```
from sfc import *

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

# Define elements and form arguments
element = VectorElement("Lagrange", "tetrahedron", 2)
v = TestFunction(element)
w = Function(element)

# Generate and compile code
F_form = CallbackForm(basisfunctions  = [v],
                      coefficients    = [w],
                      cell_integrands = [convection_vector])
J_form = Jacobi(F_form)
compiled_F_form = compile_form(F_form)
compiled_J_form = compile_form(J_form)
```

the efficiency of computing some element tensors with other libraries.

2 Defining and compiling forms

We'll now step through the lines of the code in Figure (2). First, the code defines a function `convection_vector`. This function takes as arguments symbolic expressions for the test function v and coefficient function w, plus an "integral context object". The latter object can be queried to get a symbolic representation of the affine mapping G, its transposed inverse `GinvT`, and other geometric quantities like the normal vector $\mathbf{n}$ in boundary integrals. Using the operators dot and grad, the function computes the term $\mathbf{w} \cdot \nabla \mathbf{w} \cdot \mathbf{v}$ which is the integrand of our form.

Second, we define a finite element, namely a second order vector Lagrange element defined on a tetrahedron. Then we define a test function `v` and a coefficient function `w`, which are the arguments of the form. FFC users should notice that the syntax for elements and arguments is the same.

Third, we stitch the arguments and integrand function together to define the `F_form`, and define `J_form` as the Jacobi of the nonlinear form F. Notice that the Jacobi computation is completely automatic, utilizing symbolic differentiation tools.

Fourth, the form objects are passed to the code generation and compilation

```
# Assemble global vector and matrix
from dolfin import *
mesh = UnitCube(10,10,10)
asm = Assembler(mesh)

class MyFun(cpp_Function):
    def __init__(self, mesh):
        cpp_Function.__init__(self, mesh)
    def eval(self, v, x):
        v[0] = 0.3*x[0]
        v[1] = 0.2*x[1]
        v[2] = 0.1*x[2]
    def rank(self):
        return 1
    def dim(self,i):
        return 3

w_function = MyFun(mesh)
coefficients = ArrayFunctionPtr()
coefficients.push_back(w_function)

F = Vector()
J = Matrix()
F_dofmaps = DofMapSet(compiled_F_form, mesh)
J_dofmaps = DofMapSet(compiled_J_form, mesh)
asm.assemble(F, compiled_F_form, coefficients, F_dofmaps,
             None, None, None, True)
asm.assemble(J, compiled_J_form, coefficients, J_dofmaps,
             None, None, None, True)
```

phase, and we get back compiled C++ objects that are implementations of the UFC interface.

Finally, the UFC objects can be passed to PyDOLFIN to assemble the global matrix and vector, as seen in Figure (2). At the time of writing, this is possible in the development version of DOLFIN, but be aware that some details of the code shown here may change (in particular because of ongoing improvements to the PyDOLFIN interface).

3 Benchmarks

Below we will present some benchmarks showing the efficiency of the generated code for computing the element tensor for a few forms. We will compare with quadrature based examples written in Diffpack and Deal.II (only on cubes), and

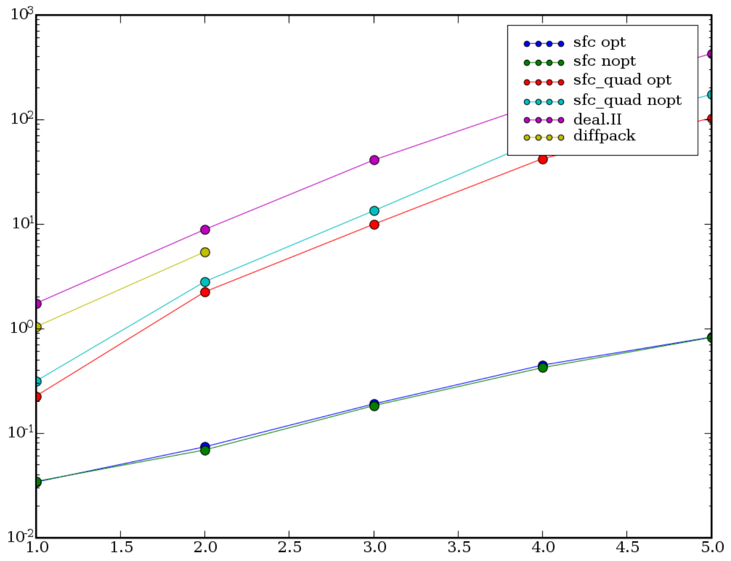

Figure 1: Time to compute the element tensor of the mass form on quadrilateral elements, in microseconds

code generated by FFC (only on simplices). The code generated by SyFi is run with both analytic integration and quadrature, and with and without further optimization.

Example 3.1 *Mass matrix*

$$A_{ij} = a(N_i, N_j) = \int_T N_i(\mathbf{x})\, N_j(\mathbf{x}) dx \tag{1}$$

After analytical integration, the computation of the mass matrix will consist of one floating point multiplication per entry in the matrix regardless of the choice of element and its order, while with the quadrature based implementation the number of quadrature points is a growing factor. The resulting speedup can be seen in Figure (1).

Example 3.2 *Convection vector*

$$A_i = a(\mathbf{N}_i; \mathbf{w}) = \int_T \mathbf{w} \cdot \nabla \mathbf{w}\, \mathbf{N}_i\, dx \equiv F_i(\mathbf{G}^{-T}, J, \mathbf{w}), \tag{2}$$

With analytical integration, $F_i(\mathbf{G}^{-T}, J, \mathbf{w})$ are polynomials that are linear in $\mathbf{G}^{-T}$ and J and quadratic in w_k^0. Figure (2) shows the benchmark results.

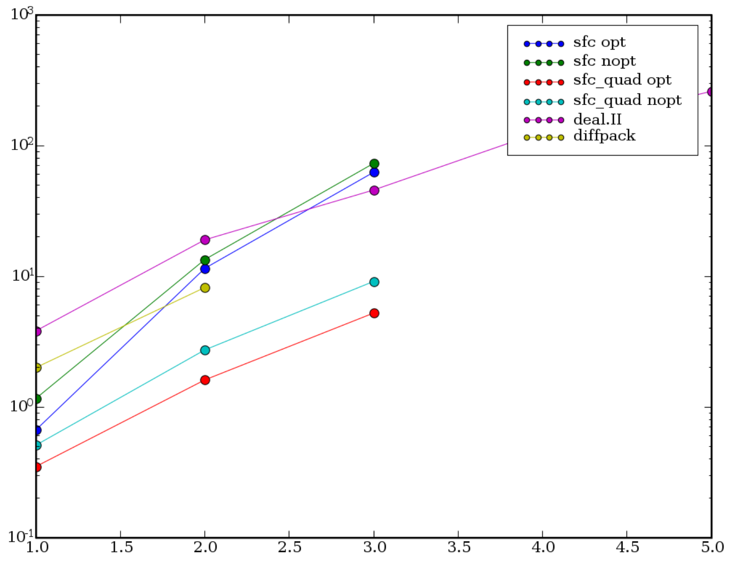

Figure 2: Time to compute the element tensor of the convection_vector form on quadrilateral elements, in microseconds

Example 3.3 *Jacobian of nonlinear convection*

$$\begin{aligned} A_{ij} &= \frac{d}{dw_i^0}\left[a(\mathbf{N}_j;\mathbf{w}^0)\right] = \frac{d}{dw_i^0}\int_T \mathbf{w}^0\cdot\nabla\mathbf{w}^0\,\mathbf{N}_j\,dx \\ &= \int_T (\mathbf{w}^0\cdot\nabla\mathbf{N}_j + \mathbf{N}_j\cdot\nabla\mathbf{w}^0)\cdot\mathbf{N}_i\,dx \equiv F_{ij}(\mathbf{G}^{-T},J,\mathbf{w}^0), \end{aligned} \tag{3}$$

In this case $F_{ij}(\mathbf{G}^{-T},J,w^0)$ are linear polynomials in $\mathbf{G}^{-T}$, J and w_k^0 after analytical integration. Figure (3) shows the benchmark results.

4 Conclusions

In cases where analytic integration is feasible, it can result in a major speedup in the computation of the element tensor. For elements defined on simplices, the performance of SyFi is roughly the same as FFC (without FErari). One point to notice is that if coefficient functions from high order finite element spaces are involved in nonlinear terms in the form, the expressions blow up and the gain may be lost. A good factorization algorithm could maybe remedy this problem. Preliminary attempts to optimize the symbolic expressions prior to code generation have been only partially successfull.

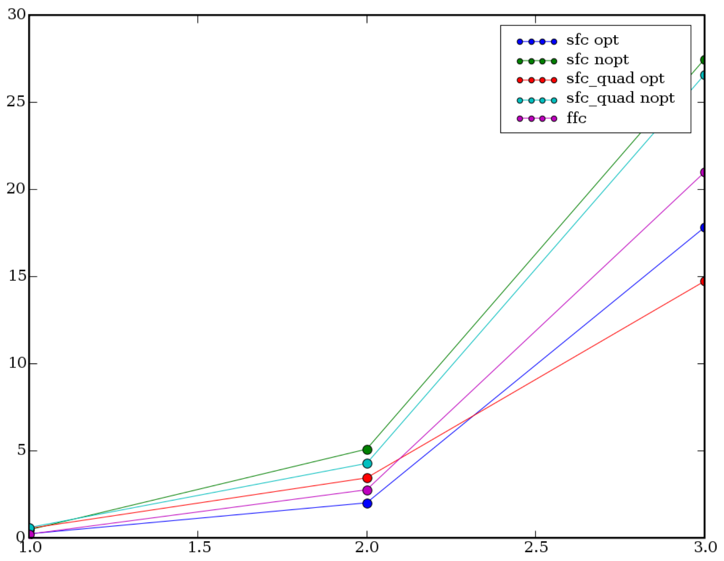

Figure 3: Time to compute the element tensor of the convection_jacobi form on triangle elements, in microseconds

Moreover, the code generation process can be quite intensive in terms of memory usage and computing time. The code generation based on quadrature is less demanding, and the resulting code is (as demonstrated) much more efficient than the implementations in Diffpack and Deal.II, the reason being that Diffpack and Deal.II provide their abstractions at the C++ level. Thus, both the efficiency gain caused by generating low level code and the advantages of working with a more abstract input format based on symbolic tools are retained independent of integration method chosen.

References

[1] M. Alnæs, H.-P. Langtangen, A.Logg, K.-A. Mardal, and O. Skavhaug, *UFC Specification and User Manual*, 2007. http//www.fenics.org/ufc/.

[2] M. Alnæs and K.-A. Mardal, *SyFi*, 2006. http://www.fenics.org/syfi/.

[3] M. S. Alnæs, H. P. Langtangen, A. Logg, K.-A. Mardal, and O. Skavhaug, *UFC*, 2007. http://www.fenics.org/ufc/.

[4] M. S. Alnæs, K.-A. Mardal, and J. Sundnes, *Application of symbolic finite element tools to nonlinear hyperelasticity*, in Fourth national conference on Computational Mechanics (MekIT'07), B. Skallerud and H. Andersson, eds., NO-7005 Trondheim, 2007, Tapir Academic Press, pp. 87–101.

[5] D. N. Arnold, R. S. Falk, and R. Winther, *Mixed finite element methods for linear elasticity with weakly imposed symmetry*, Submitted to Math. Comp., (2006).

[6] C. Bauer, C. Dams, A. Frink, V. V. Kisil, R. Kreckel, A. Sheplyakov, and J. Vollinga, *GiNaC*, 2007. http://www.ginac.de.

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

[8] FEniCS, *FEniCS project*. http//www.fenics.org/, 2007.

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

[10] ——, *FIAT*, 2006. http://www.fenics.org/fiat/.

[11] R. C. Kirby, *Optimizing FIAT with Level 3 BLAS*, to appear in ACM Transactions on Mathematical Software, (2006).

[12] R. C. Kirby, *Optimizing FIAT with the level 3 BLAS*, to appear in ACM Trans. Math. Software, (2006).

[13] R. C. Kirby and A. Logg, *A compiler for variational forms*, ACM Transactions on Mathematical Software, 32 (2006), pp. 417–444.

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

[15] A. Logg, *FFC user manual*, 2007. http//www.fenics.org/ffc/.

[16] A. Logg et al., *FFC*. http//www.fenics.org/ffc/, 2006.

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

[18] K.-A. Mardal and M. Westlie, *Instant*. http//www.fenics.org/instant, 2007.

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

[20] ——, *A new family of mixed finite elements in R^3*, Numer. Math., 50 (1986), pp. 57–81.

[21] P. A. Raviart and J. M. Thomas, *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, 1977, pp. 295–315.

[22] O. Skavhaug and O. Certik, *Swiginac*, 2006. http://swiginac.berlios.de/.