A Compiler Framework for Automatic Linearization and Efficient Discretization of Nonlinear Partial Differential Equations

by
Martin Sandve Alnæs

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
May 2009

Acknowledgements

This thesis is submitted in partial fulfilment of the degree Philosophiae Doctor to the Department of Informatics at the University of Oslo. The work has been conducted in the period from August 2006 through May 2009. I am grateful to the Research Council of Norway for their financial support under grant NFR 162730. The Center for Biomedical Computing at Simula Research Laboratory has provided excellent working conditions.

First of all, I wish to thank my supervisors Kent-André Mardal and Joakim Sundnes for their encouragement, advice and support. My colleages at Simula have provided an inspiring and social work environment. This thesis would not be the same without the good collaborative environment among my fellow developers in the FEniCS project. Finally, I wish to thank all my family and friends, for being who you are.

Martin Sandve Alnæs
Oslo, May 2009

Contents

Introduction **1**
1 On Computational Science . . . 1
2 Aim of the thesis . . . 2
3 The FEniCS project . . . 2
4 Overview of software abstractions . . . 4
5 Summary of papers . . . 6
6 Related work . . . 8
7 Future work . . . 9

Paper I: Application of symbolic finite element tools to nonlinear hyperelasticity **13**
1 Introduction . . . 15
2 Mathematical Model . . . 16
3 Numerical methods . . . 18
4 Implementation . . . 24
5 Test cases . . . 28
6 Discussion . . . 29

Paper II: Unified Framework for Finite Element Assembly **33**
1 Introduction . . . 35
2 Finite Element Discretization . . . 38
3 Finite Element Assembly . . . 41
4 Software Framework for Finite Element Assembly . . . 43
5 The UFC Interface . . . 48
6 Examples . . . 52
7 Discussion . . . 57
8 Conclusion . . . 58

Paper III: On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods **65**
1 Introduction . . . 67
2 Preliminaries . . . 69
3 Defining and compiling forms . . . 73
4 Examples Demonstrating Efficiency . . . 78
5 Discussion . . . 88
6 Conclusion . . . 91

Paper IV: The Unified Form Language **97**
1 Overview 100
2 Defining finite element spaces 103
3 Defining forms 104
4 Defining expressions 105
5 Form operators 112
6 Expression representation 116
7 Computing derivatives 120
8 Algorithms 126
9 Implementation issues 131
10 Future directions 132
11 Acknowledgements 133

List of Papers

- **Paper I**
  Application of symbolic finite element tools to nonlinear hyperelasticity
  M. S. Alnæs, K-A. Mardal and J. Sundnes. 2007.
  In *Fourth national conference on Computational Mechanics (MekIT07)*,
  B. Skallerud and H. Andersson, Eds.,
  Tapir Academic Press, NO-7005 Trondheim, 87101.

- **Paper II**
  Unified Framework for Finite Element Assembly
  M. S. Alnæs, A. Logg, K-A. Mardal, O. Skavhaug, and H. P. Langtangen. 2009.
  Accepted for publication in *International Journal of Computational Science and Engineering*.

- **Paper III**
  On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods
  M. S. Alnæs and K-A. Mardal. 2009.
  Accepted for publication in *ACM Transactions on Mathematical Software*.

- **Paper IV**
  The Unified Form Language
  M. S. Alnæs. 2009.
  Submitted for publication in the FEniCS book *Automated Scientific Computing*,
  A. Logg, K-A. Mardal and G. Wells, Eds.

Introduction

1 On Computational Science

Mathematical modeling allows the study of phenomena that cannot easily be inspected directly. Physical properties that are otherwise hard or expensive to measure can be inspected with the help of mathematical models in combination with the measurements that *can* be made. One example of such a property is the distribution of forces in vessel walls resulting from the flow of blood. Models can also allow us to make predictions, such as in weather forecasting and optimization of industrial processes. With the advent of increasingly powerful computers, the use of more detailed models has become feasible. This development is opening up many new possibilities.

The core idea of the scientific discipline called Computational Science is to gain new insight with the aid of computations. A computational scientist must relate to a range of scientific branches, making scientific computing a highly multidisciplinary field of research. Developing and understanding models of complex physical phenomena demands a thorough comprehension of the physics. Finding solutions to the model equations often requires sophisticated numerical solution methods. The actual computations are carried out by a computer, and a high performance software implementation is critical to obtain answers from the mathematical model quickly.

Computational biomechanics is one field of research where the sheer complexity of the physical phenomena demands a high level of sophistication in all aspects of scientific computing. Biological systems are often complex and detailed, and a good model must reflect all significant properties of the system. New and improved models are developed continuously, and simulation software for any particular model is hard to come by. Custom software must therefore often be developed to achieve state of the art research. Specialists in scientific disciplines such as mechanics or biology are rarely experts in computer science as well. Hence, time consuming software development is a major obstacle to efficient applied research within the field of scientific computing. To enable research with focus on the physical phenomena, the implementation of numerical software must be highly automated.

Many models of physical phenomena are written in the language of partial differential equations (PDEs). PDE based models can describe phenomena as diverse as, e.g., elastic and plastic deformations, fluid flow, and electromagnetism. Generic solution methods for PDEs usually involve converting the equations to some form that can be implemented on a computer, a process called discretization. One of the most flexible discretization methods for PDEs is the finite element method [5, 19]. Using the finite

element method, approximate solutions to partial differential equations can be found by following a well established mathematical framework step by step.

Although the finite element method provides well defined steps to follow, implementing an equation can still be tedious and hard work, prone to human error. This is true in particular when dealing with complex models. Any task that is well defined but complex and tedious to perform manually is a good candidate for automation. Ideally, the finite element method should be implemented as a fully automated machine, where any partial differential equation can be inserted together with suitable data and the solution is returned. In this thesis, contributions are made towards this goal.

2 Aim of the thesis

This thesis aims to simplify the implementation of partial differential equations, and nonlinear PDEs in particular. The main focus has been on the expression of PDEs using an abstract input format that is close to mathematical notation, and on automating the linearization and discretization of such equations. Although human efficiency has been the highest priority, computational efficiency does not have to be sacrificed. Both kinds of efficiency have been achieved by using a combination of high and low level languages together with symbolic computing and compiler techniques.

The work has been motivated by models of elastic biological tissue. Many types of biological tissue can be modeled with hyperelasticity [9]. The fiber structure typically seen in such tissue demands highly nonlinear anisotropic models. One example use of such models is to describe the deformation of the heart muscle wall during a heart cycle. Another example is modeling of vessel walls in fluid structure interaction simulations of blood flow. For each kind of biological tissue, many material descriptions have been developed over the years. Full models of vessel walls also need to describe the three-layer structure of such tissue [18]. Particular emphasis has thus been put on automating the parts of such equations that make them hard to implement.

3 The FEniCS project

A significant part of the work on this thesis has been the development of novel software. Most of the software is available as components in the FEniCS framework [6, 20]. The FEniCS project is a research software project with the long term vision of *Automation of Computational Mathematical Modeling*. The project is an international collaboration between many research institutions: University of Chicago, Argonne National Laboratory, Delft University of Technology, Royal Institute of Technology KTH, Simula Research Laboratory, Texas Tech University, and University of Cambridge. Each software component is freely available under some open source license.

The FEniCS software framework consists of several libraries which work together to form a complete problem solving environment for PDEs. A central concept is the combination of scripting languages (Python) with low level languages (C++) to achieve both human and computational efficiency. Applications based on FEniCS components can be implemented in either C++ or Python. A summary of the components follows:

- *UFL — The Unified Form Language*:
  A high level domain specific language for variational forms with automatic differentiation, embedded in Python. See Paper IV.

- *UFC — Unified Form-assembly Code*:
  A low level C++ interface between generic library code for finite element assembly and problem-specific generated code. See Paper II.

- *SyFi — Symbolic Finite Elements*:
  A C++ library providing basis functions for a range of finite elements, computed in a generic fashion using symbolic computing techniques. Also referred to as the SyFi kernel. See Paper III.

- *SFC — The SyFi Form Compiler*:
  A compiler which generates efficient implementations of UFC from the abstract UFL definitions of PDEs. Uses basis functions from the SyFi kernel. See Paper III.

- *FIAT — The FInite element Automatic Tabulator* [13, 14]:
  A Python library providing basis functions for a range of finite elements, computed in a generic fashion using numerical linear algebra techniques.

- *FFC — The FEniCS Form Compiler* [15, 16, 24]:
  A compiler which generates efficient implementations of UFC from the abstract UFL definitions of PDEs. Uses basis functions from FIAT.

- *FErari — Finite Element rearrangement to automatically reduce instructions* [10, 11, 12, 17]:
  An optimizing backend for FFC.

- *Instant*:
  A Python module using SWIG to compile and wrap C/C++ code on the fly inside a Python program.

- *Viper*:
  A Python module using VTK to provide easy visualization.

- *DOLFIN* [23]:
  A problem solving environment for PDEs. The main C++ library in FEniCS.

- *PyDOLFIN*:
  The Python interface to the DOLFIN C++ library with additional functionality to integrate with other FEniCS components dynamically and seamlessly.

The three most significant software projects resulting from the work on this thesis are UFL, SFC, and UFC. I am the main author of UFL and SFC, and one of the main authors of UFC. Together these projects improve upon the form compiler framework in the FEniCS project. See the Section 5 for more about these software projects. In addition, I have made contributions to the FEniCS software projects Instant and DOLFIN. My work on UFL and UFC has indirectly contributed to improvements in

FFC. Below, a brief overview is given of the main mathematical concepts implemented in the FEniCS framework and how they relate to specific software components.

4 Overview of software abstractions

The process of solving a PDE with the finite element method involves a series of steps. In the following these steps are summarized and relations to specific software components in the FEniCS framework are sketched. The goal is not to give a detailed overview of the finite element method, but to show which parts of the solution process that have been automated by methods and software in this thesis. More details about most of these operations are provided in the papers. Note that several of the mathematical abstractions are mirrored in more than one software component, each component implementing a different aspect of the mathematical concept.

Consider a physical model expressed as a PDE, here exemplified by the well known Poisson's equation:

$$-\Delta u = f, \quad \text{in} \quad \Omega, \tag{1}$$

$$u = 0, \quad \text{on} \quad \partial\Omega_0, \tag{2}$$

$$\frac{\partial u}{\partial n} = g, \quad \text{on} \quad \partial\Omega_1. \tag{3}$$

This formulation of the PDE is called the strong form of the equation. By multiplying Equation 1 with a test function v in an appropriate function space V, integrating over the domain $\Omega \subset \mathbb{R}^d$ and applying partial integration, we arrive at the weak form of Poisson's equation.

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\partial\Omega_1} g v \, ds, \quad \forall \, v \in V. \tag{4}$$

Many physical models are expressed using the strong form, but some are expressed directly in the weak form. When implementing PDEs in the FEniCS framework, the weak form is used. To keep the following general, we can write the weak form of the equations in abstract form as

$$a(v, u) = L(v; f), \quad \forall \, v \in V. \tag{5}$$

In the FEniCS framework, the definition of the forms a and L, such as in Equation (4), is expressed using the Unified Form Language (UFL). An alternative way to define discrete weak forms in UFL is through automatic differentiation of a functional or linear form, such as

$$f_h(x) = \sum_{k=1}^{|V_h|} f_k \phi_k(x), \tag{6}$$

$$M : V_h \to \mathbb{R}, \tag{7}$$

$$L(\phi_i; f_h) = \frac{\partial M(f_h)}{\partial f_i}, \tag{8}$$

$$a(\phi_i, \phi_j; f_h) = \frac{\partial L(\phi_i; f_h)}{\partial f_j}. \tag{9}$$

More details on this domain specific language and the automatic differentiation process is found in Paper IV.

The finite element method defines a way to convert Equation (5) to an algebraic system of equations. This process is called discretization. The first step is to approximate the geometric domain Ω with a triangulation Ω_h, which is a combination of polygonal shapes $\{K_i\}$,

$$\Omega \approx \Omega_h = \bigcup_{i=1}^{n} K_i. \tag{10}$$

The `Mesh` class [21] in DOLFIN is used to store, represent, and work with unstructured meshes.

Finite element spaces are defined by patching together local polynomial spaces V_h^K on each $K \in \Omega_h$ to form a global discrete function space V_h. Functions defined on Ω are approximated by discrete functions defined on Ω_h, expressed as linear combinations of basis functions for a space V_h.

$$u \in V \approx V_h = \text{span}\left\{\phi_k\right\}, \tag{11}$$

$$u(x) \approx u_h(x) = \sum_{k=1}^{|V_h|} u_k \phi_k(x). \tag{12}$$

The concepts of functions and function spaces occur in some form in almost all FEniCS libraries. The software components SyFi and FIAT provide basis functions of local polynomial spaces V_h^K for a range of common finite elements. The global discrete function space V_h on a particular mesh is represented in the DOLFIN library by the class `FunctionSpace`, which depends on a `Mesh` and generated finite element code. The interface between this generated code and the library code is a part of Unified Form-assembly Code (UFC), which is discussed in Paper II.

Discretization of the weak form relies on the approximation $u_h \approx u$ and the linearity of $a(v, u)$ in u, such that

$$\sum_{j=1}^{|V_h|} a(\phi_i, \phi_j) u_j = L(\phi_i; f_h), \quad i = 1, \ldots, |V_h|. \tag{13}$$

Defining

$$A_{ij} = a(\phi_i, \phi_j), \tag{14}$$

$$b_i = L(\phi_i; f_h), \tag{15}$$

Equation (13) can be written as an algebraic system of equations

$$\mathbf{A}\mathbf{u} = \mathbf{b} \tag{16}$$

where $\mathbf{u}$ is the vector of function coefficients u_j.

The integral over Ω_h can be split into a sum of integrals over each $K \in \Omega_h$. Hence, the global matrix $\mathbf{A}$ or vector $\mathbf{b}$ can be accumulated from a number of element matrices $\mathbf{A}^K$

or vectors $\mathbf{b}^K$, one for each $K \in \Omega_h$. This generalizes to scalars and tensors of arbitrary rank, not just matrices and vectors. Since the basis functions for V_h have local support on each K, $\mathbf{A}^K$ is a small matrix and $\mathbf{A}$ is a large sparse matrix. The `Assembler` in DOLFIN implements the accumulation of local element tensors $\mathbf{A}^K$ into a global tensor $\mathbf{A}$. It assumes an implementation of UFC to compute the problem-specific element tensor $\mathbf{A}^K$, and handles the interaction with mesh and linear algebra components. A detailed algorithm for this computation can be found in Paper II.

One of the main concepts that separates FEniCS from most other finite element software is the *form compiler*. A form compiler (either SFC or FFC) is used to automate the translation of an abstract discrete weak form a to source code for computing the local element tensor $\mathbf{A}^K$. A compiler can use domain specific knowledge in optimizations to achieve high efficiency of the generated code. Paper III discusses the methods used in SFC, in a version from before UFL was designed.

The end result of the assembly process is usually a system of linear algebraic equations such as Equation (16). Computing the solution to these equations by numerical linear algebra techniques is handled by external libraries outside the FEniCS project. A variety of state-of-the-art libraries such as PETSc [3, 2, 4], uBLAS [25], Trilinos [8] and MTL [7] are supported, and interfacing other libraries is fairly easy. DOLFIN provides an easy to use interface to a subset of the available linear equation solvers with basic options that are suitable for simple equations. Direct access to the underlying linear algebra library allows the use of more advanced linear algebra algorithms when needed. Details on linear equation solvers are outside the scope of this thesis.

The versions of FEniCS components assumed in this introduction are released under the name *FEniCS 2009-04 (Automatic Augustin)*, except for updated versions of UFL (v0.3) and SyFi (v0.6.1). UFL is fully integrated in the current development versions of DOLFIN and FFC, and will be fully operational in next major release.

5 Summary of papers

5.1 Paper I: Application of symbolic finite element tools to nonlinear hyperelasticity

In this paper, my first attempts at combining symbolic computing and finite element methods in a semi-automatic code generation framework is discussed. The equations for large deformation elasticity with hyperelastic material properties are presented with two alternative constitutive laws. A brief overview of the numerical methods is given, including discretization by the finite element method and linearization of the weak form for solution of the nonlinear algebraic equations by Newtons method. An implementation of the manually linearized equations using SyFi is presented. The equations are assembled and solved in the Python based problem solving environment PyCC, which is an internal closed source code developed at Simula Research Laboratory. The methods applied in this paper were only partially successful, not scaling properly to higher order elements or more complicated material laws. Thus, the paper ends by a discussion of these limitations and possible improvements.

5.2 Paper II: Unified Framework for Finite Element Assembly

Finite element software is usually built from several library components. To which extent each library component depends on the other components varies between libraries. In this paper, a framework for finite element assembly is presented where we have tried to keep the dependencies minimal, without sacrificing generality or efficiency. The interface, called Unified Form-assembly Code (UFC), consists of only a single C++ header file with accompanying documentation. In particular, numbering conventions for cell entities are crucial and documented in detail. The main motivation for this interface is to have a fixed format for code generated by form compilers in the FEniCS project. Fixing the interface between generated code and library code such as DOLFIN allows more freedom in the development on either side of the interface. By implementing UFC, other libraries may combine code generated by FFC or SFC with their own mesh and linear algebra components without any other dependencies than a single header file. UFC supports a range of finite element methods including continuous and discontinuous Galerkin methods, Nedelec elements, and arbitrary mixed element hierarchies, but not hp methods. The paper explain the abstractions used in UFC, covering finite element evaluation, mapping of degrees of freedom, and integrals over arbitrary combinations of subdomains. The same interface is used for computing element tensors of any arity including in particular scalars, vectors, matrices, which corresponds to functionals, linear forms and bilinear forms respectively. The role of UFC in a complete problem solving environment is discussed.

5.3 Paper III: On the Efficiency of Symbolic Computations Combined with Code Generation for Finite Element Methods

In this paper, the combination of symbolic computing techniques and code generation is investigated for use with finite element methods. The methods discussed are implemented in a library called Symbolic Finite Elements (SyFi). Expressions for the basis functions of a finite element space can be computed by solving symbolic linear systems. This method is implemented in the SyFi kernel in C++, along with definitions for a collection of common finite elements. The use of symbolic libraries enable defining expressions abstractly with mathematical notation. The SyFi Form Compiler (SFC) compiles symbolic descriptions of variational forms implemented in Python to efficient low level C++ code implementing the UFC interface. The compiler takes as input a set of finite element declarations and user defined functions defining integrand expressions using symbolic computations. Expressions for basis functions are provided by the SyFi kernel. With the help of basic symbolic computing features and a set of common operators such as div, grad, curl, inner and dot, the user defines integrand expressions close to mathematical notation. The compiler can either integrate the integrand expressions analytically prior to code generation, or generate code with quadrature loops such that the integrals are evaluated when the code is executed.

A series of efficiency comparisons are provided, comparing the efficiency of the code generation approach with other finite element codes FFC, Deal.II and Diffpack. The

comparisons also investigate the benefits of analytic integration vs quadrature based code generation. Analytic integration provides orders of magnitude speedup for the simplest equations, but loses the advantage for more complex equations. The analytic integrals also become costly to compute for more complex equations and higher order elements, while quadrature based code can be generated efficiently. The quadrature based generated code is in general significantly faster than corresponding hand written quadrature code using Deal.II and Diffpack.

5.4 Paper IV: The Unified Form Language

The Unified Form Language (UFL) is a domain specific language for variational forms. The design goals set for UFL are threefold. The language should provide a common input language and compiler front-end for the form compilers FFC and SFC in the FEniCS project. The language should combine and improve upon the expressiveness of the form languages in previous versions of FFC and SFC. This includes in particular an integrated support for automatic differentiation. And finally, the implementation of UFL should improve the efficiency of the form compiler framework. All three goals have been met by this project.

This paper discusses UFL in two ways. The first part is an overview of the user interface and expressive capabilities of the language. This is intended to show users of the FEniCS project how they can declare their variational forms. The overview begins with explaining declaration of arbitrary mixed finite element hierarchies and declaration of forms and functionals on any combination of subdomains. How to declare functions and build expressions with a range of operators is covered in some detail. The operators offered by the language include common tensor algebra operators, and index notation with the Einstein summation convention. Differential operators are central in such a language, and several kinds of are provided including derivatives with respect to user defined variables. Form operators allow the automatic transformation of entire forms into related forms, such as the action of a bilinear form on a function, or the differentiation of nonlinear forms and functionals.

A more in depth technical overview of the software design is given in the second part of the paper. Particular emphasis is given to expression representation methods and automatic differentiation algorithms. Computational graphs are useful in the code generation performed by the form compilers, and UFL provides utilities to aid in this process. Some implementation details related to the embedding of UFL in Python are also discussed. Finally, some thoughts about future directions for UFL close the paper.

This paper has been submitted as a chapter to the upcoming FEniCS book.

6 Related work

Some of the work performed during the thesis period have not been included in this thesis. A collaboration with the University Hospital of North Norway on simulation of blood flow in the Circle of Willis resulted in a publication in the medical journal Stroke [1]. I have also contributed to the development of an internal software project at Simula called PyCC. Work on Python based problem solving environments for

PDE based simulations including PyCC and FEniCS has been discussed in a book chapter [22].

7 Future work

The contributions from this thesis may have significant impact on simplifying software development in several application areas. In particular, many applications relevant to research at the Center for Biomedical Computing at Simula Research Laboratory can both be implemented more efficiently and gain a significant speedup. Preliminary inexact efficiency comparisons with a Diffpack based hyperelasticity code shows promising results. Hyperelastic models of biological tissue will be used for heart wall mechanics, and extended with models of muscle contraction for full electro-mechanics simulations. In simulations of fluid structure interaction in blood vessels, similar models govern the elastic response of the vessel walls. Ongoing studies of turbulent flows have successfully applied automatic linearization to $k - \epsilon$ models, and the implementation of more advanced turbulence models will be significantly easier with this feature. The automatic differentiation of forms may also give important simplifications for the implementation of optimal control and inverse problems.

The FEniCS project is continuously evolving with the needs of its users and developers, who are mostly researchers with different goals and interests. Further improvements to the form compiler framework in FEniCS will be driven by the goals of each individual researcher, not limited to the applications mentioned above. Relevant improvements can be divided in two categories. The implementation of new applications may be simplified or made possible by increasing the expressiveness of the form language even further. In this category, introducing complex numbers in UFL could allow more natural expression of some PDEs. For other problems special functions, such as Bessel functions, could be useful. Space-time elements and time derivatives have also been suggested.

The other category is the improvement of the compiler technology. This involves both optimizing the compilation process and improving the generated code. The efficiency of the generated code is already quite good for a large class of equations but may be improved further for some of the more complex equations. One possible way to improve the generated code is to experiment with different Automatic Differentiation strategies for even better efficiency. Another prospect is to make the code generation process more aware of current hardware, by generating code to better utilize the CPU cache and minimize memory access.

Parallelism is clearly becoming an important issue for high performance software even for laptops and desktop computers in the near future. However, the compiler framework presented in this thesis does not need to handle parallelism. Instead, the form compilers provide efficient serial computational kernels, which a parallell application can distribute across the available computing resources. However, the concept of a compiler framework is well suited to adapt the computations to the hardware without user intervention. In particular, generating computational kernels that can run on the GPU, using for example OpenCL, is an idea with a lot of potential.

Bibliography

[1] M. S. Alnæs, J. Isaksen, K. Mardal, B. Romner, M. K. Morgan, and T. Ingebrigtsen, *Computation of hemodynamics in the circle of willis*, Stroke, 38 (2007), pp. 2500–2505.

[2] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc Web page*, 2009. http://www.mcs.anl.gov/petsc.

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

[5] D. Braess, *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*, Cambridge University Press, 2001.

[6] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and L. R. Scott, *The FEniCS project*, Tech. Rep. 2003–21, Chalmers Finite Element Center Preprint Series, 2003.

[7] P. Gottschling, D. S. Wise, and M. D. Adams, *Representation-transparent matrix algorithms with scalable performance*, in Proceedings of the 21st annual international conference on Supercomputing, Seattle, Washington, 2007, ACM, pp. 116–125.

[8] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[9] G. A. Holzapfel, *Nonlinear Solid Mechanics: A Continuum Approach for Engineering*, Wiley, 1st ed., Mar. 2000.

[10] R. Kirby, M. Knepley, A. Logg, and L. Scott, *Optimizing the evaluation of finite element matrices*, SIAM Journal on Scientific Computing, 27 (2006), pp. 741–758.

[11] R. Kirby, A. Logg, L. Scott, and A. Terrel, *Topological optimization of the evaluation of finite element matrices*, SIAM Journal on Scientific Computing, 28 (2007), pp. 224–240.

[12] R. Kirby and L. Scott, *Geometric optimization of the evaluation of finite element matrices*, SIAM Journal on Scientific Computing, 29 (2008), pp. 827–841.

[13] R. C. Kirby, *Algorithm 839: FIAT, a new paradigm for computing finite element basis functions*, ACM Trans. Math. Softw., 30 (2004), pp. 502–516.

[14] ——, *Optimizing FIAT with level 3 BLAS*, ACM Trans. Math. Softw., 32 (2006), pp. 223–235.

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

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

[17] R. C. Kirby and A. Logg, *Benchmarking Domain-Specific compiler optimizations for variational forms*, ACM Trans. Math. Softw., 35 (2008), pp. 1–18.

[18] M. Kroon and G. A. Holzapfel, *Modeling of saccular aneurysm growth in a human middle cerebral artery*, Journal of Biomechanical Engineering, 130 (2008), pp. 051012–10.

[19] H. P. Langtangen, *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*, Springer-Verlag New York, Inc., 2003.

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

[21] ——, *Efficient representation of computational meshes*, International Journal of Computational Science and Engineering, (2009). To appear.

[22] A. Logg, K.-A. Mardal, M. S. Alnæs, H. P. Langtangen, and O. Skavhaug, *A Hybrid Approach to Efficient Finite Element Code Development*, Computational Science, Chapman and Hall, 2007, ch. 19.

[23] A. Logg and G. N. Wells, *DOLFIN: Automated finite element computing*, Submitted, (2009).

[24] K. B. Ølgaard, A. Logg, and G. N. Wells, *Automated code generation for discontinuous galerkin methods*, SIAM Journal on Scientific Computing, 31 (2008), pp. 849–864.

[25] J. Walter, M. Koch, et al., *uBLAS*, 2006. *http://www.boost.org/libs/numeric/ublas/*.

Paper I: Application of symbolic finite element tools to nonlinear hyperelasticity

Application of symbolic finite element tools to nonlinear hyperelasticity

M. S. Alnæs [1], K-A. Mardal [1,2], J. Sundnes[1,2]

[1] Center for Biomedical Computing, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway
[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

Abstract: The present paper addresses the use of high level languages, symbolic mathematical tools and code generation in an implementation of the finite element method, using a nonlinear hyperelasticity equation as example. Advantages of the software development method that will be demonstrated include closeness to the mathematics, enabling high human efficiency with easy to use high level languages, while still keeping a high computational efficiency by generating tailored inner loop code for the problem at hand. The application we have in mind for the equations presented here is the simulation of the passive elastic properties of heart and blood vessel tissue.

1 Introduction

Biomechanical modeling often involves coupling of many physical phenomena, including elastic tissue deformations, fluid flow, and electrochemical processes. This often results in complicated mathematical models, for which it is challenging to design and implement efficient numerical methods. Researchers involved in modeling and simulation of biomedical processes are also likely to have very varied backgrounds, and in many cases limited experience with numerical methods and programming. These are good arguments for increasing the abstraction level in the simulator development, by using high level languages and tools that make the computer program closely resemble the mathematical model. Developing the tools that combine these abilities with good numerics and efficient implementation has been an active research topic for decades. One approach that has led to recent progress in this area is to wrap efficient implementations in compiled languages with high level scripting languages such as Python [2], which are currently gaining momentum in the scientific computing communities [6, 21, 23, 17, 18, 5, 19, 22, 3, 24, 20, 16, 9, 8, 14, 12]

Characteristic of most biomaterials is that they can undergo large elastic deformations, often governed by complicated non-linear constitutive laws. In this paper we will demonstrate the implementation of equations for nonlinear anisotropic hyperelasticity with our software SyFi [5, 15] and PyCC [18]. The properties of hyperelastic materials are defined in terms of a strain energy function, see e.g. [10] for details. Within the limits of hyperelastic materials, we want to easily apply new material laws, by simply

specifying a new strain energy function. Through application of symbolic mathematics we can use automated differentiation of the material laws where applicable. As the engine for symbolic mathematics we use the C++ library GiNaC [1] and its python bindings swiginac [3]. A primary goal for the work is that the implementation should be close to the mathematics of the problem, in this case close to the weak formulation used for the finite element method. We assume the reader is familiar with continuum mechanics and the finite element method, but understanding the equations in detail should not be necessary to appreciate the ideas of this software development strategy.

The paper is outlined as follows. We first give a brief introduction to the mathematical model, followed by an outline of the applied numerical methods. We then present the corresponding code that specifies the same equations and solves them. This is followed by a brief explanation of the code generation techniques, and finally some preliminary results.

2 Mathematical Model

2.1 Momentum equations

The general momentum balance equation, or equilibrium equation, reads

$$\frac{\partial^2 \mathbf{u}}{\partial t^2} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{f}, \qquad \frac{\partial^2 u_j}{\partial t^2} = \frac{\partial \sigma_{ij}}{\partial x_i} + f_j, \tag{1}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor, $\mathbf{u}$ is the deformation vector field, and $\mathbf{f}$ is the body forces. In this paper we are primarily concerned with modelling the mechanical behavior of heart and blood vessel tissue, for which the acceleration term may be neglected. We thus focus on the stationary equation

$$\nabla \cdot \boldsymbol{\sigma} = -\mathbf{f}, \qquad \frac{\partial \sigma_{ij}}{\partial x_i} = -f_j, \tag{2}$$

which is here defined relative to the deformed geometry. Although Equation (2) describe a stationary situation, it is still relevant for describing the time dependent movement of the heart and blood vessels during normal physiological function. The equation must then be interpreted as a quasi-stationary equation, where the boundary conditions and source term change with time. More specifically, in a numerical time integration scheme, the work involves solving a number of problems of the kind (2), with a new source term and boundary conditions for each time step.

2.2 Finite hyperelasticity

In this section we describe a displacement based formulation of the equilibrium equations for a hyperelastic material. Since we allow large (finite) displacements, it is most convenient to use the Full Lagrangian formulation. With a Full Lagrangian formulation, the equations are formulated on a reference geometry instead of updating the geometry based on the deformation. Let the initial domain at time $t = 0$ be denoted Ω_0, and

the deformed domain at time t be Ω_t. If the coordinate of a particle in the reference geometry Ω_0 is $\mathbf{x}$, then the coordinate of the same particle in Ω_t is $\hat{\mathbf{x}}$. With this in mind we can define the displacement $\mathbf{u}$ as

$$\mathbf{u} = \hat{\mathbf{x}} - \mathbf{x}, \qquad u_i = \hat{x}_i - x_i. \tag{3}$$

In the reference coordinates, the equilibrium equation reads

$$\nabla \cdot \mathbf{P} = -\mathbf{f}, \qquad \frac{\partial P_{ij}}{\partial x_i} = -f_j, \tag{4}$$

where $\mathbf{P}$ is known as the first Piola-Kirchoff stress tensor, which is related to the Cauchy stress tensor through a geometry mapping. The rest of this section describes the components needed to formulate material laws for this stress tensor.

We will need the deformation gradient $\mathbf{F}$, defined as

$$\mathbf{F} = \frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{x}} = \mathbf{I} + \frac{\partial \mathbf{u}}{\partial \mathbf{x}} = \mathbf{I} + (\nabla \mathbf{u})^T, \qquad F_{ij} = \frac{\partial \hat{x}_i}{\partial x_j} = \delta_{ij} + \frac{\partial u_i}{\partial x_j}. \tag{5}$$

The deformation gradient can be decomposed into an orthonormal rotation tensor $\mathbf{R}$ and a stretch tensor $\mathbf{U}$, $\mathbf{F} = \mathbf{R}\mathbf{U}$. Since $\mathbf{R}$ is orthonormal it follows that that $\mathbf{F}^T\mathbf{F} = \mathbf{U}^T\mathbf{U}$, which is independent of rigid body rotation. We can thus use the right Cauchy-Green strain tensor

$$\mathbf{C} = \mathbf{F}^T\mathbf{F}, \qquad C_{ij} = F_{ki}F_{kj}, \tag{6}$$

as a measure of pure stretch or strain, without rigid body motions. From the right Cauchy-Green tensor another important measure of strain may be derived, namely the Green-Lagrange strain tensor

$$\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}), \qquad E_{ij} = \frac{1}{2}(C_{ij} - \delta_{ij}), \tag{7}$$

which fulfills $\mathbf{E} = \mathbf{0}$ for $\mathbf{u} = \mathbf{0}$. By assuming small deformations and neglecting higher order terms, the Green-Lagrange strain tensor can be shown to be equal to the strain tensor for small deformations $\boldsymbol{\epsilon}$. The right Cauchy-Green and the Green-Lagrange strain tensor are both defined relative to the reference geometry Ω_0.

It turns out the constitutive laws are easier formulated with the symmetric second Piola-Kirchoff stress tensor $\mathbf{S}$,

$$\mathbf{S} = \mathbf{F}^{-1}\mathbf{P}, \qquad S_{ij} = F^{-1}_{ik}P_{kj}, \tag{8}$$

since it is work-conjugate with the Green-Lagrange strain. This gives the final formulation of the equilibrium equations

$$\nabla \cdot (\mathbf{F}\mathbf{S}) = -\mathbf{f}, \qquad \frac{\partial (F_{ik}S_{kj})}{\partial x_i} = -f_j. \tag{9}$$

2.3 Material laws

As noted in the introduction, the constitutive law for a hyperelastic material is specified by a strain energy function Ψ. The second Piola-Kirchoff stress tensor is given as partial derivatives of the strain energy with respect to the Green-Lagrange strain tensor components

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}} = 2\frac{\partial \Psi}{\partial \mathbf{C}}, \qquad S_{ij} = \frac{\partial \Psi}{\partial E_{ij}} = 2\frac{\partial \Psi}{\partial C_{ij}}. \tag{10}$$

Many strain energy functions exist for different media, and it is of interest to be able to switch easily between these material laws in the software. In particular, constitutive laws for many biomaterials is an active research topic with few definite answers. Below we will use two different strain energy functions for demonstration.

A simple nonlinear material law for hyperelastic materials is the Saint Vernant-Kirchoff law

$$\Psi(\mathbf{E}) = \frac{1}{2}\lambda(trace\mathbf{E})^2 + \mu\mathbf{E} : \mathbf{E}. \tag{11}$$

One of the material laws that has been applied in modeling of passive heart tissue is the transversely isotropic Fung[7] type law

$$Q = b_{ff}E_{ff}^2 + b_{xx}(E_{nn}^2 + E_{ss}^2 + E_{sn}^2 + E_{ns}^2) + b_{fx}(E_{fn}^2 + E_{nf}^2 + E_{fs}^2 + E_{sf}^2), \tag{12}$$

$$\Psi(\mathbf{E}) = \frac{1}{2}K(e^Q - 1), \tag{13}$$

where f, s, and n refer to a coordinate system oriented with fibers in the material, in the fiber, sheet, and sheet normal directions respectively. See f.ex. [11] for more details. Examples of the implementation of both these constitutive equations and simple test cases are presented below.

3 Numerical methods

3.1 Weak formulation

The finite element method relies on an equivalent form of the PDE (9) called the weak form. Although this can also be derived from physical principles, we view it here as a purely mathematical step in the formulation of the numerical discretization. The weak form is obtained by multiplying with a test function $\mathbf{v}$ and integrating over Ω_0;

$$\int_{\Omega_0} (\nabla\cdot\mathbf{FS} + \mathbf{f}) \cdot \mathbf{v} d\mathbf{x} = 0, \qquad \int_{\Omega_0} (\frac{\partial (F_{ik}S_{kj})}{\partial x_i} + f_j)v_j = 0. \tag{14}$$

Assuming traction free or Dirichlet boundary conditions for simplicity, integration by parts now yields

$$\int_{\Omega_0} (\mathbf{FS} : \nabla\mathbf{v} - \mathbf{f} \cdot \mathbf{v}) d\mathbf{x} = 0, \qquad \int_{\Omega_0} (F_{ik}S_{kj}\frac{\partial v_j}{\partial x_i} - f_j v_j) = 0. \tag{15}$$

```
def finite_elasticity_B(v, u, f, G, GinvT, psi):
    nsd  = v.nops()
    I    = Id(nsd)
    symbol_names = ["F", "E", "S"]
    Fs, Es, Ss = symbolic_matrices(nsd, symbol_names)

    Du  = grad(u, GinvT)
    Dv  = grad(v, GinvT)
    DuT = Du.transpose()
    F   = I + DuT
    FTs = Fs.transpose()
    E   = (FTs * Fs - I) / 2
    p   = psi.value(Es)
    S   = diff(p, Es)
    integrand = contract(Fs*Ss, Dv) - inner(f, v)

    tokens = [ (Fs, F), (Es, E), (Ss, S) ]
    return (integrand, tokens)
```

Figure 1: Implementation of the integrand $((\mathbf{F}(\mathbf{u})\mathbf{S}(\mathbf{u})) : \nabla\mathbf{v} - \mathbf{f}\cdot\mathbf{v})$ for the weak formulation of finite elasticity.

To simplify the further discussion, we introduce the bilinear form

$$a(\mathbf{u},\mathbf{v}) = \int_{\Omega_0} \mathbf{FS} : \nabla\mathbf{v} d\mathbf{x}. \tag{16}$$

3.2 Discretization

We discretize the weak form (15) by the finite element method. Note that the symbol $\mathbf{u}$ will sometimes denote the physical deformation vector field, and sometimes the discrete solution vector from the linear system. The interpretation should be clear from the context. The vector field $\mathbf{u}$ is approximated as a superposition of n vector basis functions $\mathbf{v}^j$ with coefficients u^j:

$$\mathbf{u} = \sum_{j=1}^{n} u^j \mathbf{v}^j \tag{17}$$

Applying test functions $\mathbf{v}^i$ for $i = 1,\ldots,n$ we get a system of non-linear algebraic equations

$$\mathbf{B}(\mathbf{u}) = \mathbf{0}, \tag{18}$$

where component i is given by

$$\mathbf{B}_i(\mathbf{u}) = a(\mathbf{u},\mathbf{v}^i) - \int_{\Omega_0} \mathbf{f}\cdot\mathbf{v}^i \, d\mathbf{x} = 0, \qquad \text{for } i = 1,\ldots,n. \tag{19}$$

This system is to be solved for the unknown coefficients u^j in the vector $\mathbf{u}$.

```
def finite_elasticity_rhs(u, psi):
    nsd  = u.nops()
    I    = Id(nsd)

    Du = grad(u)
    DuT = Du.transpose()

    F  = I + DuT
    FT = F.transpose()

    E = (FT * F - I) / 2

    Es = symbolic_matrix(nsd, "E")
    p = psi.value(Es)
    S = diff(p, Es)

    for i in range(Es.nops()):
        S = S.subs( Es.op(i) == E.op(i) )

    f = -div( F*S )
    return f
```

Figure 2: Manufactured analytic right hand side computed from an analytic solution.

3.3 Linearization

For solving of the nonlinear equations (19), we consider the well-known iterative Newton-Raphson method.

Algorithm 1 Newton's method.

Given $\mathbf{B}$, $\mathbf{u}$, and tolerance

$\epsilon = \|\mathbf{B}(\mathbf{u})\|$
while $\epsilon >$ tolerance
    $\mathbf{e} = \mathbf{J}^{-1}\mathbf{B}(\mathbf{u})$
    $\mathbf{u} = \mathbf{u} - \mathbf{e}$
    $\epsilon = \|\mathbf{B}(\mathbf{u})\|$

This requires the solution of linear systems of the form

$$\mathbf{J}\Delta\mathbf{u} = \mathbf{B}(\mathbf{u}), \qquad J_{ij}\delta u_j = B_i(u_k), \tag{20}$$

where $\mathbf{B}$ is the residual at the previous iteration, and $\mathbf{J}$ is the Jacobian with components

```
class StrainEnergy:
    def value(self, E):
        """Evaluate the strain energy Psi(E)."""
        pass

class SaintVenantKirchhoff(StrainEnergy):
    def __init__(self, lambd, mu):
        self.lambd = lambd
        self.mu = mu
    def value(self, E):
        return self.lambd * (trace(E)**2) / 2
                + self.mu * contract(E, E)

class Fung(StrainEnergy):
    def __init__(self, fiber, K, bff, bfx, bxx):
        self.fiber = fiber
        self.K   = K
        self.bff = bff
        self.bfx = bfx
        self.bxx = bxx
    def value(self, E):
        # Missing feature: assuming fiber == delta_ij
        Eff = E[0,0]
        ...
        Q = self.bff*Eff**2 + \
            self.bxx*(Enn**2 + Ess**2 + Esn**2 + Ens**2) + \
            self.bfx*(Efn**2 + Enf**2 + Efs**2 + Esf**2)
        return self.K * (exp(Q) - 1) / 2
```

Figure 3: Strain energy functions.

given by

$$\mathbf{J}_{ij} = \frac{\partial}{\partial u^j} a(\mathbf{u}, \mathbf{v}^i) = \int_{\Omega_0} \frac{\partial(\mathbf{FS})}{\partial u^j} : (\nabla \mathbf{v}^i)^T d\mathbf{x}. \tag{21}$$

Recall that the stresses occurring in (21) are defined as partial derivatives of the strain energy function Ψ. For biomaterials, as we will see below, this function can become rather complicated, and performing the differentiation in (21) is a tedious task. A commonly applied solution strategy is to differentiate Ψ by hand to obtain analytical expressions for the stresses S_{ks}, and then do the final differentiation numerically, either by expanding the Jacobian or by differentiating the complete integrand directly. We will apply a different approach, where we apply automatic differentiation software to generate the code for analytic expressions of the entries in the Jacobian. We start by

expanding the Jacobian into

$$J_{ij} = \int_{\Omega_0} \left(\frac{\partial \mathbf{F}}{\partial u^j} \mathbf{S} + \mathbf{F} \frac{\partial \mathbf{S}}{\partial u^j} \right) : (\nabla \mathbf{v}^i)^T d\mathbf{x}, \tag{22}$$

$$J_{ij} = \int_{\Omega_0} \left(\frac{\partial F_{rk}}{\partial u^j} S_{ks} + F_{rk} \frac{\partial S_{ks}}{\partial u^j} \right) \left(\frac{\partial v_s^i}{\partial x_r} \right) dx. \tag{23}$$

In the computation of the element tensor we will fix i and j as one permutation at a time, as a practical step to reduce the order of the tensors we have to deal with. Since the test function v^i is known, the term $\partial v_s^i / \partial x_r$ is trivial to compute in the software. The two terms inside the parenthesis require somewhat more care, and it is convenient to treat these separately. We assume u^j known, and first compute the quantities in $\frac{\partial F_{rk}}{\partial u^j} S_{ks}$ in the following order:

$$\mathbf{F} = \mathbf{I} + (\nabla \mathbf{u})^T, \qquad F_{rs} = \delta_{rs} + \frac{\partial u_r}{\partial x_s}, \tag{24}$$

$$\frac{\partial \mathbf{F}}{\partial u^j} = (\nabla \mathbf{v}^j)^T, \qquad \frac{\partial F_{rs}}{\partial u^j} = \frac{\partial v_r^j}{\partial x_s}, \tag{25}$$

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \mathbf{F} - \mathbf{I}), \qquad E_{rs} = \frac{1}{2}(F_{kr} F_{ks} - \delta_{ij}), \tag{26}$$

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}}, \qquad S_{rs} = \frac{\partial \Psi}{\partial E_{rs}}. \tag{27}$$

Having completed these steps, we turn our attention to the second term $F_{rk} \frac{\partial S_{ks}}{\partial u^j}$. The computation of this term is somewhat more complicated, and we introduce two helper variables $\mathbf{H}$ and $\mathbb{C}$, which are tensors of rank two and four, respectively. The computation is then made in the following order

$$\mathbf{H}^j = \frac{1}{2} \left(\mathbf{F}^T \frac{\partial \mathbf{F}}{\partial u^j} + \frac{\partial \mathbf{F}^T}{\partial u^j} \mathbf{F} \right), \qquad H_{rs}^j = \frac{1}{2} \left(F_{kr} \frac{\partial F_{ks}}{\partial u^j} + \frac{\partial F_{kr}}{\partial u^j} F_{ks} \right), \tag{28}$$

$$\mathbb{C}_{pqrs} = \frac{\partial^2 \Psi}{\partial E_{pq} \partial E_{rs}} = \frac{\partial S_{rs}}{\partial E_{pq}}, \tag{29}$$

$$\frac{\partial S_{rs}}{\partial u^j} = H_{pq}^j \mathbb{C}_{pqrs}. \tag{30}$$

In the implementation we do not compute the rank four tensor directly, but use a loop over two indices and handle the "subtensor" as a matrix. With this approach, each component in $\frac{\partial S_{rs}}{\partial u^j}$ can be computed as a contraction of rank 2 tensors. The calculations (29)-(30) are then replaced by

$$\forall (r,s): \quad \frac{\partial \mathbf{S}_{rs}}{\partial u^j} = \mathbf{H}^j : \frac{\partial \mathbf{S}_{rs}}{\partial \mathbf{E}}. \qquad \forall (r,s): \quad \frac{\partial S_{rs}}{\partial u^j} = H_{pq}^j \frac{\partial S_{rs}}{\partial E_{pq}}. \tag{31}$$

We have now computed all the components we need, and the integrand may be calculated with simple matrix products and a tensor contraction using equation (22).

```
def finite_elasticity_J(u, v, w, G, GinvT, psi):
    nsd = u.nops()
    I   = Id(nsd)
    symbol_names = ["F", "E", "H", "S", "dS", "dFS"]
    Fs, Es, Hs, Ss, dSs, dFSs = \
        symbolic_matrices(nsd, symbol_names)

    Du = grad(u, GinvT)
    Dv = grad(v, GinvT)
    Dw = grad(w, GinvT)
    DuT = Du.transpose()
    DwT = Dw.transpose()

    F   = I + DwT
    dF  = DuT
    FTs = Fs.transpose()
    dFT = dF.transpose()

    E = (FTs * Fs - I) / 2

    p = psi.value(Es)
    S = diff(p, Es)

    # stress increment
    dS = zeros(nsd, nsd)
    H = ((dFT * Fs) + (FTs * dF)) / 2
    for r in range(nsd):
        for s in range(nsd):
            dS[r, s] = contract(Hs, diff(S[r,s], Es))

    dFS = ((dF * Ss) + (Fs * dSs))
    integrand = contract( dFSs, Dv )

    tokens = [ (Fs, F), (Es, E),   (Ss, S),
               (Hs, H), (dSs, dS), (dFSs, dFS) ]
    return (integrand, tokens)
```

Figure 4: Implementation of linearized integrand for the application of the Newton-Raphson method to the weak formulation of finite elasticity.

4 Implementation

In our implementation of the full Lagrangian finite elasticity equations, we strive to stay close to the mathematics. A normal approach often referred to as the engineering formulation is to manually take into account symmetries and sparsity of the fourth order elasticity tensor C_{ijkl} to define a smaller nine by nine matrix, while representing second order tensors as vectors. This formulation is easy to implement in traditional finite element software, but has little similarity with the original mathematical model. With the formulation shown in (31), combined with the finite element library SyFi (Symbolic Finite elements [5, 15]), we avoid this step and stay closer to the mathematical model in the implementation of the weak forms. Furthermore, a part of the process of implementing finite hyperelasticity is often to compute $\frac{\partial \Psi}{\partial \mathbf{E}}$ and $\frac{\partial^2 \Psi}{\partial \mathbf{E} \partial \mathbf{E}}$ manually, or with the help of external symbolic applications.) The expressions are then manually written into the C/C++/Fortran code. We instead utilize a symbolic library to perform the differentiation as part of our application and generate code from these expressions automatically. Therefore, adding a new Ψ is as simple as writing the expression for Ψ. Code for the implementation of the Saint Vernant-Kirchoff and Fung type laws are seen in Figure 3. An implementation of a strain energy function can also be used to specify quantities to compute as part of the postprocessing stage, like strains and stresses.

4.1 Code generation

SyFi uses symbolic computations to construct basis functions for various finite elements. It has support for a large set of elements, but in this paper we will stick to regular Lagrange elements on tetrahedra. Based on the explicit basis functions expressions, we can construct symbolic expressions for the integrand of a weak form, using symbolic differentiation for the differential operators. To make the user code close to the mathematics, differential operators like $\nabla \cdot \mathbf{u}$, $\nabla \mathbf{u}$ and $\frac{\partial \Psi}{\partial \mathbf{E}}$ are available as `div(u)`, `grad(u)` and `diff(psi, E)`, and the products $\mathbf{u} \cdot \mathbf{v}$ and $\mathbf{A} : \mathbf{B}$ are simply `inner(u,v)`, `contract(A,B)`.

If the weak form only contains polynomials and regular differential operators, SyFi can also perform the integration over an element symbolically. Since some material laws use exponential and logarithmic functions, this feature cannot be applied to our problem, and the generated code will instead apply quadrature for the integration over a cell.

From the resulting symbolic expressions for the weak form, SyFi can generate C++ code for the computation of the element matrix and element vector. The generated code is in a format specified by the Unified Form-assembly Code [4] (UFC) project. UFC consists of a set of abstract classes in a single header file, providing a predefined interface to the computation of an element matrix, evaluating finite element basis functions, mapping degrees of freedom and related operations. An example of generated low-level code is seen in Figure 6. In this code excerpt, `tabulate_tensor` is a function from the UFC interface, `A` is the element vector for $\mathbf{B}(\mathbf{u})$, and the variable names `Fxx`, `Exx`, `Sxx` etc. should be recognizable from the mathematical formulation, even if the low level expressions are not.

```
# <... Imports and initialization>

# Define forms to be compiled:
def elasticity_fung_J(u, v, w, fiber, K, bff, bfx, bxx, G, Ginv):
    psi = Fung(fiber, K, bff, bfx, bxx)
    return finite_elasticity_J(u, v, w, G, Ginv, psi)

def elasticity_fung_B(v, w, f, fiber, K, bff, bfx, bxx, G, Ginv):
    psi = Fung(fiber, K, bff, bfx, bxx)
    return finite_elasticity_B(v, w, f, G, Ginv, psi)

form_J    = MatrixForm(elasticity_fung_J, name="J_fung_3D")
form_B    = VectorForm(elasticity_fung_B, name="B_fung_3D")

# Initialize SyFi
nsd      = 3
order    = 1
qorder   = 6
SyFi.initSyFi(nsd)
polygon  = SyFi.ReferenceTetrahedra()
u_fe     = SyFi.VectorLagrange(polygon, order)
fiber_fe = SyFi.TensorP0(polygon)
fe0      = SyFi.P0(polygon)

# Define the finite elements to use for each argument:
# (ref. arguments to elasticity_fung_* above)
fe_list_J = (u_fe, u_fe, u_fe, fiber_fe, fe0, fe0, fe0, fe0)
fe_list_B = (u_fe, u_fe, u_fe, fiber_fe, fe0, fe0, fe0, fe0)

# Generate code for the forms and compile it
compiled_elasticity_form_J = compile_form(form_J, fe_list_J,
             integration_mode='quadrature', quad_order=qorder)

compiled_elasticity_form_B = compile_form(form_B, fe_list_B,
             integration_mode='quadrature', quad_order=qorder)
```

Figure 5: Compiling an element tensor.

```
void tabulate_tensor(double* A,
                     const double * const * w,
                     const ufc::cell& c) const
{
  ...
  static const double quad_weights[24] = {
      0.00665379170969, 0.00665379170969, ...
    };

  for(int iq=0; iq<24; iq++) {
    const double x = quad_points[iq][0];
    const double y = quad_points[iq][1];
    const double z = quad_points[iq][2];
    const double quad_weight_detG = quad_weights[iq] * detG;

    F00 = Ginv00*(-w[0][0]+w[0][3])+(-w[0][0]+w[0][9])*Ginv02
          +Ginv01*(w[0][6]-w[0][0])+1.0;
    F01 = Ginv12*(-w[0][0]+w[0][9])+...
    ...
    S22 = E22*exp(w[4][0]*((E01*E01)+(E10*E10)+...
    A[0] += ((S20*F12+S00*F10+S10*F11)*...
```

Figure 6: Excerpt of generated code for the computation of the element vector for $\mathbf{B}(\mathbf{u})$

4.2 Implementing the weak formulation

Figure 1 shows the implementation of the weak form for $\mathbf{B}(\mathbf{u})$. This user-defined function (`finite_elasticity_B`) will be called by the code generation tools in SyFi at a later stage. The function will then get as input a symbolic expression for a test function `v`, the deformation field `u` from the previous iteration as a superposition of symbolic basis functions,

$$\mathbf{u} = \sum_{k=1}^{n_e} u^k \mathbf{v}^k, \tag{32}$$

the body force `f` in the same representation as `u`, symbolic representations of the geometry mappings G and G^{-T} for mapping to a reference element, and a material law definition represented by a `StrainEnergy` object `psi`. Later during the finite element assembly, the coefficients u^k in the symbolic representation will be input values from a finite element vector restricted to one element. The return value is a symbolic representation of the integrand for a single entry in the element vector, along with a list of tokens. The `tokens` list holds symbol/value pairs for variables that will be generated code for, and which the integrand expression depends on. After calling this user-defined function, the code generation tools will generate code for assignments to these variables and wrap this code in a quadrature loop. An excerpt of this generated code is shown in Figure 6.

```
K, bff, bfx, bxx = 876, 18.48, 2.8, 3.58
fiber = (1,0,0, 0,1,0, 0,0,1)

# <... initialize mesh, vectors, matrix, etc.>

while eps > newton_tolerance:
    # Collect coefficients to the form J
    # (ref. fe_list_J in previous code example)
    coeffs_J = [u, fiber, K, bff, bfx, bxx]
    assembler.assemble_matrix(compiled_elasticity_form_J,
                              coeffs_J, J_before_bc)

    # Modify boundary rows and columns in J
    (J, BC, C) = dirichlet_boundarycondition(J_before_bc,
                                             boundary_dofs)
    B = BC*B_before_bc # set boundary components to zero

    # Find and apply the correction
    du.fill(0)
    du = conjgrad(J, du, F)
    u -= du

    coeffs_B = [u, f, fiber, K, bff, bfx, bxx]
    assembler.assemble_vector(compiled_elasticity_form_B,
                              coeffs_B, B_before_bc)

    eps = L2(B_before_bc)
    iter += 1
```

Figure 7: Newton loop with assembly of linear system in each iteration.

Stepping through the middle part of `finite_elasticity_B`, each line shows a clear resemblance with equations (24)-(27) and (15). The symbolic variables can be matrices and vectors, greatly reducing error prone index handling. Computing gradients in the reference domain is performed with `grad(u, GinvT)`. To reduce the size of the expressions, symbolic matrices are used for $\mathbf{F}$, $\mathbf{E}$ and $\mathbf{S}$ to represent their values in dependent expressions. Notice in particular how the strain energy function `psi` is evaluated with a symbolic matrix `Es`, and the stress tensor is differentiated with respect to the same symbolic matrix with `diff(p, Es)`.

In Figure 4, similar code is shown for the computation of $\mathbf{J}(\mathbf{u})$. Notice that the fourth order elasticity tensor is never explicitly constructed, it only exists as a step in the algorithm formulation in equation (31). Zeros and cancelling terms are automatically taken into account by the symbolic code generation tools, so that the resulting generated code for the computation of the element matrix and vector will be partially optimized before it is compiled.

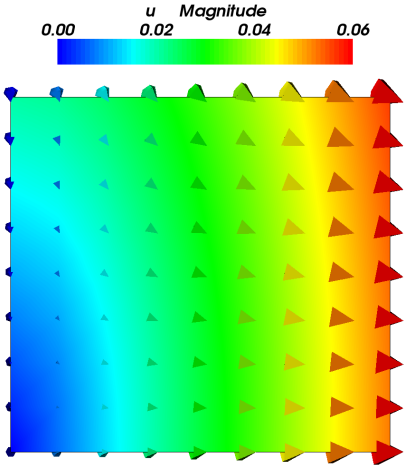

Figure 8: Testcase with SVK material law

4.3 Software verification

Another application of symbolic computations in finite element implementations is to verify the software with the method of manufactured solutions. First we define a set of (possibly unphysical) analytical solutions, and calculate the body force f required to obtain this solution using the strong formulation of the equilibrium equations (9). Figure 2 shows code for these calculations. Next we can solve the discrete equations with this calculated body force $\mathbf{f}$, and compare the computed solution with the original expression to find the error. This approach is particularly convenient when using complicated material laws like the Fung-law described previously.

4.4 Application code

PyCC (Python Computing Components [18]) is a high level Python framework for the implementation of PDE solvers in development at Simula. After defining the weak form of the equations like described above, and compiling the element matrix and element vector for $\mathbf{J}(\mathbf{u})$ and $\mathbf{B}(\mathbf{u})$ respectively, these compiled UFC forms can be loaded in a Python application and used by a PyCC Assembler object to assemble the global linear system inside a Newton-Raphson iteration, like shown in Figure 7. The linear equations in each iteration are solved with a conjugate gradient method implementation from PyCC. For simple visualization in the application script, a Python module called Viper is used, which is a thin layer on top of VTK [13]. Simulation results are written to file in VTK format, which were loaded in Paraview [20] (v2.9.9) to create the figures.

5 Test cases

As a simple test case, we apply Dirichlet boundary conditions to the x-component of $\mathbf{u}$ on two opposite sides of a cube, and leave the rest of the boundary traction-free. To

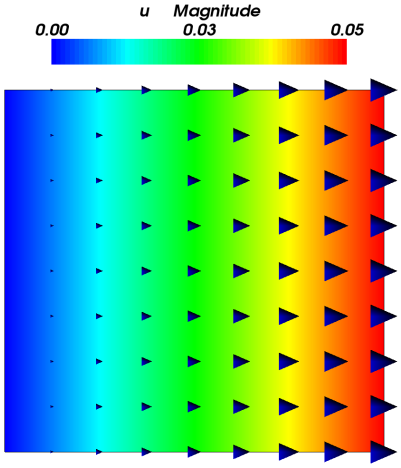

Figure 9: Testcase with Fung material law

remove the possibility of a rigid body translation, we must also fix all components of $\mathbf{u}$ in one point.

$$\mathbf{u} \cdot \mathbf{e}_x = 0, \qquad x = 0, \tag{33}$$
$$\mathbf{u} \cdot \mathbf{e}_x = \alpha \mathbf{i}, \qquad x = 1, \tag{34}$$
$$\mathbf{u} = 0, \qquad \mathbf{x} = \mathbf{0}, \tag{35}$$
$$\mathbf{t} = 0, \qquad 0 < x < 1. \tag{36}$$

Figures 8 and 9 show the resulting deformation fields as glyphs and color-coded magnitude of $\mathbf{u}$ from one side of the cube with its normal vector in y-direction. The Fung law here yields a fully compressible deformation, while with the SVK law we see the cube is compressed, and the color-coded magnitude shows where the fixed point is.

6 Discussion

One of the key advantages of this implementation is that it is easy to add new material laws. Since the implementations of the weak forms and postprocessing quantities (not shown) are close to mathematical model formulation, they should be easily readable for people without a background in numerics and programming. For those who are used to the traditional engineering formulation of the elasticity equations this point may not be very important. The user is still subjected to technical implementation details in SyFi, so there is still a need to work more on the user interface of the library. This is work in progress.

Generated code from SyFi is highly efficient for simple equations of similar complexity as mass matrices and stiffness matrices, competing with or even outperforming traditional quadrature-based implementations. But for more complex equations like the finite hyperelasticity presented here there are challenges to overcome in the code generation.

The generated code can grow quite large, and great care must be taken to keep the code size small. This problem grows for higher order elements.

In the current implementation and with the tests done so far, the time spent assembling the linear system dominates the Newton iterations for these equations. However, this is expected to improve significantly with future versions of SyFi, when more optimized code can be generated. Since the code generation tools has a more high level overview of the mathematical expressions than the C++ compiler will have at a later stage, it is possible to perform large optimizations by analyzing dependencies in the expressions. This role is traditionally filled by the human code implementer, who chooses the algorithms to use and tunes the code in a manual process. The current code generation tools in SyFi perform only very simple optimization steps, but improving this is work in progress. Quantifying the speedup cannot be done at this stage.

The software has not been tested with the most complicated material laws, only with unidirectional fiber directions and without compressibility constraints. There is also limited support for more advanced boundary conditions, which also must be addressed before truly relevant physiological applications can be attempted.

Bibliography

[1] *GiNaC*, 2006. http://www.ginac.de.

[2] *Python*, 2006. http://www.python.org/doc/.

[3] *Swiginac*, 2006. http://swiginac.berlios.de/.

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

[5] M. S. Alnæs and K.-A. Mardal, *Syfi user manual*. http://www.fenics.org/pub/documents/syfi/syfi-user-manual/syfi-user-manual.pdf.

[6] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant, *Numerical Python*. *http://www.pfdubois.com/numpy/*.

[7] Y. Fung, *Biomechanics: mechanical properties of living tissues*, Springer-Verlag New York, Inc., 1993.

[8] J. Hoffman, J. Jansson, C. Johnson, M. G. Knepley, R. C. Kirby, A. Logg, L. R. Scott, and G. N. Wells, *FEniCS*, 2006. *http://www.fenics.org/*.

[9] J. Hoffman, J. Jansson, A. Logg, and G. N. Wells, *DOLFIN*, 2006. *http://www.fenics.org/dolfin/*.

[10] G. Holzapfel, *Nonlinear Solid Mechanics, A Continuum Approach for Engineering*, John Wiley& Sons, Ltd, 2001.

[11] J. D. Humphrey, *Cardiovascular Solid Mechanics*, Springer-Verlag, 2002.

[12] R. C. Kirby, *FIAT*, 2006. *http://www.fenics.org/fiat/*.

[13] Kitware, *The Visualization ToolKit*, 2006. *http://www.vtk.org/*.

[14] A. Logg, *FFC*, 2006. *http://www.fenics.org/ffc/*.

[15] K.-A. Mardal, *Syfi - an element matrix factory*. To appear in the PARA'06 proceedings to be published in the Springer series Lecture Notes in Computer Science (LNCS).

[16] *MayaVi package*. *http://mayavi.sourceforge.net*.

[17] *PETSc software package*. *www.mcs.anl.gov/petsc/*.

[18] *PyCC*, 2007. Software framework under development. *http://www.simula.no/pycc/*.

[19] *PySE software package*. *http://pyfdm.sf.net*.

[20] Sandia National Laboratories, *ParaView*, 2006. *http://www.paraview.org/*.

[21] *SciPy software package. http://www.scipy.org.*

[22] *SWIG software package. http://www.swig.org.*

[23] *Trilinos software package. http://software.sandia.gov/trilinos.*

[24] *Vtk package. http://www.vtk.org.*

Paper II: Unified Framework for Finite Element Assembly

Unified Framework for Finite Element Assembly

M. S. Alnæs[1], A. Logg[1,2], K-A. Mardal[1,2], O. Skavhaug[1,2], and H. P. Langtangen[1,2]

[1] Center for Biomedical Computing, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

Abstract: At the heart of any finite element simulation is the assembly of matrices and vectors from discrete variational forms. We propose a general interface between problem-specific and general-purpose components of finite element programs. This interface is called Unified Form-assembly Code (UFC). A wide range of finite element problems is covered, including mixed finite elements and discontinuous Galerkin methods. We discuss how the UFC interface enables implementations of variational form evaluation to be independent of mesh and linear algebra components. UFC does not depend on any external libraries, and is released into the public domain.

1 Introduction

Software for solving physical problems have traditionally been tailored to the problem at hand, often resulting in computationally very efficient special-purpose codes. However, experience has shown that such codes may be difficult and costly to extend to new problems. To decrease turn-over time from problem definition to its numerical solution, scientific code writers have to an increasingly larger extent tried to create general libraries, containing common numerical algorithms applicable to a wide range of problems. Such libraries can reduce the size of the application code dramatically and hide implementation details. In the field of finite element solution of partial differential equations, many general and successful libraries have emerged during the last couple of decades, e.g., Cactus, Cogito, COMSOL Multiphysics, Deal.II, Diffpack, DOLFIN (FEniCS), Getfem++, Kaskade, Sundance, and UG (see the reference list for papers and websites).

General finite element libraries implement many standard mathematical and numerical concepts, but the software components are often not as carefully designed as their mathematical counterparts. From a software engineering point of view it is important to achieve clear separation of the various software components that build up a finite element library, such that each component can be replaced separately. Not only does this offer greater flexibility for application and library developers, but it also makes the software easier to maintain, especially under changing requirements of several developers in long-term projects. These arguments have received much attention by developers of general finite element libraries in recent years (see, e.g., [12, 15]).

Well designed libraries provide clear interfaces to represent this separation. Typically, the application code uses functions or objects in the interface to perform basic "high level" steps of the solution process. Problem-specific details, such as the variational form, the mesh and coefficients are passed through the interface to the library to compute a solution. Such libraries and their interfaces are generally referred to as problem solving environments (PSEs).

However, one fundamental issue in designing such software libraries is how to separate problem-specific code from general library code. Some components, such as computational meshes and linear algebra, may be implemented as reusable components (e.g. as a set of C++ classes) with well-defined interfaces. However, other components, such as variational forms, are intrinsically problem-specific. As a result, those components must either be implemented and provided by the user or *generated* automatically by the library from a high-level description of the variational form. In either case, it becomes important to settle on a well-defined interface for how the library should communicate with those problem-specific components.

The design of such an interface is the subject of the present paper. We propose a C++ interface called UFC, which provides an interface between general reusable finite element libraries and problem-specific code. In other libraries, the implementation of finite elements and variational forms is usually tied to the specific mesh, matrix and vector format in use, while in UFC we have strived to decouple these concepts. Furthermore, the interface is designed to allow for a variety of elements such as continuous and discontinuous Lagrange, Nedelec and Raviart-Thomas elements.

To make a successful interface, one needs a sufficiently general framework for the underlying mathematical structures and operations. The software interface in the current paper relies on a more general view of variational forms and finite element assembly than commonly found in textbooks. We therefore precisely state the mathematical background and notation in Sections 2 and 3.

UFC is significantly inspired by our needs in the tools FFC, SFC, and DOLFIN, which are software units within FEniCS, see [20, 21, 9, 25]. The interplay between these tools and UFC is explained in Section 4, which provides additional and more specific motivation for the design of UFC. Highlights of the interface are covered in Section 5. Section 6 contains some examples of high-level specifications of variational forms with the form compilers FFC and SFC, which automatically generate code compatible with the UFC interface for computing element matrices and vectors. We also explain how the interface can be used with existing libraries.

We here note that as a result of the UFC interface, the two form compilers FFC and SFC may now be used interchangeably since both generate code conforming to the UFC interface. These form compilers were developed separately and independently. The work on UFC was initially inspired by our efforts to unify the interfaces for these form compilers.

Related Work

One major reason for the success of general finite element libraries is that many widely different physical problems can be solved by quite short application codes utilizing the

same library. The opposite strategy, i.e., one application utilizing different alternative libraries, has received less attention. For example, an application might want to use an adaptive mesh data structure and its functionality from one library, a very efficient assembly routine from another library, basic iterative methods from, e.g., PETSc, combined with a preconditioner from Trilinos or Hypre. To make this composition a true plug-and-play operation, the various libraries would need to conform to a unified interface to the basic operations needed in finite element solvers. Alternatively, low level interfaces can be implemented with thin wrapper code to connect separate software components.

In numerical linear algebra, the BLAS and LAPACK interfaces have greatly simplified code writing. By expressing operations in the application code in terms of BLAS and LAPACK calls, and using the associated data (array) formats, one program can be linked to different implementations of the BLAS and LAPACK operations. Despite the great success of this approach, the idea has to little extent been explored in other areas of computational science. One recent example is Easyviz ([31]), a thin unified interface to curve plotting and 2D/3D scalar- and vector-field visualization. This interface allows an application program to use a MATLAB-compatible syntax to create graphics, independently of the choice of graphics package (Gnuplot, Grace, MATLAB, VTK, VisIt, OpenDX, etc.). Another example is GLAS ([30]), a community initiative to specify a general interface for linear algebra libraries. GLAS can be viewed as an extension and modernization of the BLAS/LAPACK idea, utilizing powerful constructs offered by C++.

Within finite elements, DUNE ([15, 14]) is a very promising attempt to define unified interfaces between application code and libraries for finite element computing. DUNE provides interfaces to data structures and solution algorithms, especially finite element meshes and iterative solution methods for linear systems. In principle, one can write an application code independently of the mesh data structure and the matrix solution method. DUNE does not directly address interfaces between the finite element problem definition (element matrices and vectors), and the assembly process, which is the topic of the present paper. Another difference between DUNE and our UFC interface is the choice of programming technology used in the interface: DUNE relies heavily on inlining via C++ templates for efficient single-point data retrieval, while UFC applies pointers to chunks of data. However, our view of a finite element mesh can easily be adapted to the DUNE-Grid interface. The DUNE-FEM module (under development) represents interfaces to various discretization operators and serves some of the purposes of the UFC interface, though being technically quite different.

In the finite element world, there are many competing libraries, each with their own specialties. Thin interfaces offering only the least common denominator functionality do not support special features for special problems and may therefore be met with criticism. Thick interfaces, trying to incorporate "all" functionality in "all" libraries, become too complicated to serve the original purpose of simplifying software development. Obtaining community consensus for the thickness and syntax of a unified interface is obviously an extremely challenging process. The authors of this paper suggest another approach: a small group of people defines a thin (and hence efficient and easy-to-use) interface, they make the software publicly available together with a detailed documentation, and

demonstrate its advantages. This is our aim with the present paper.

2 Finite Element Discretization

2.1 The Finite Element

A finite element is mathematically defined as a triplet consisting of a polygon, a polynomial function space, and a set of linear functionals, see [17]. Given that the dimension of the function space and the number of the (linearly independent) linear functionals are equal, the finite element is uniquely defined. Hence, we will refer to a finite element as a collection of

- a polygon K,
- a polynomial space $\mathcal{P}_K$ on K,
- a set of linearly independent linear functionals, the *degrees of freedom*, $\ell_i : \mathcal{P}_K \to \mathbb{R}$, $i = 1, 2, \ldots, n_K$.

With this definition the basis functions $\{\phi_i^K\}_{i=1}^{n_K}$ are obtained by solving the following system of equations,

$$\ell_i(\phi_j^K) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n_K. \tag{1}$$

The computation of such a nodal basis can be automated, given (a basis for) the polynomial space $\mathcal{P}_K$ and the set of linear functionals $\{\ell_i\}_{i=1}^{n_K}$, see [19, 9].

2.2 Variational Forms

Consider the Poisson problem $-\nabla \cdot (w\nabla u) = f$ with Dirichlet boundary conditions on a domain $\Omega \subset \mathbb{R}^d$. Multiplying by a test function $v \in V_h$ and integrating by parts, one obtains the variational problem

$$\int_\Omega w\nabla v \cdot \nabla u_h \,\mathrm{d}x = \int_\Omega vf \,\mathrm{d}x, \quad \forall v \in V_h, \tag{2}$$

for the approximation $u_h \in V_h$. If $w, f \in W_h$ for some discrete function space[1] W_h we may thus write (2) as

$$a(v, u_h; w) = L(v; f) \quad \forall v \in V_h, \tag{3}$$

where the trilinear form $a : V_h \times V_h \times W_h \to \mathbb{R}$ is given by

$$a(v, u_h; w) = \int_\Omega w\nabla v \cdot \nabla u_h \,\mathrm{d}x \tag{4}$$

[1]It is assumed that any given function may be represented (exactly or approximately) in some finite element space. Alternatively, functions may be approximated by quadrature. Quadrature representation is not discussed here, but is covered by the UFC specification and implemented by the form compilers FFC and SFC.

and the bilinear form $L : V_h \times W_h \to R$ is given by

$$L(v; f) = \int_\Omega v f \, \mathrm{d}x. \tag{5}$$

Note here that a is *bilinear* for any given fixed $w \in W_h$ and L is *linear* for any given fixed $f \in W_h$.

In general, we shall be concerned with the discretization of finite element variational forms of general arity $r + n > 0$,

$$a : V_h^1 \times V_h^2 \times \cdots \times V_h^r \times W_h^1 \times W_h^2 \times \cdots \times W_h^n \to \mathbb{R}, \tag{6}$$

defined on the product space $V_h^1 \times V_h^2 \times \cdots \times V_h^r \times W_h^1 \times W_h^2 \times \cdots \times W_h^n$ of two sets $\{V_h^j\}_{j=1}^r, \{W_h^j\}_{j=1}^n$ of discrete function spaces on Ω. We refer to $(v_1, v_2, \ldots, v_r) \in V_h^1 \times V_h^2 \times \cdots \times V_h^r$ as *primary arguments*, and to $(w_1, w_2, \ldots, w_n) \in W_h^1 \times W_h^2 \times \cdots \times W_h^n$ as *coefficients* and write

$$a = a(v_1, \ldots, v_r; w_1, \ldots, w_n). \tag{7}$$

In the simplest case, all function spaces are equal but there are many important examples, such as mixed methods, where the arguments come from different function spaces. The choice of coefficient function spaces depends on the application; a polynomial basis simplifies exact integration, while in some cases evaluating coefficients in quadrature points may be required.

2.3 Discretization

To discretize the form a, we introduce a set of bases $\{\phi_i^1\}_{i=1}^{N^1}, \{\phi_i^2\}_{i=1}^{N^2}, \ldots, \{\phi_i^r\}_{i=1}^{N^r}$ for the function spaces $V_h^1, V_h^2, \ldots, V_h^r$ respectively and let $i = (i_1, i_2, \ldots, i_r)$ be a multiindex of length $|i| = r$. The form a then defines a rank r tensor given by

$$A_i = a(\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r; w_1, w_2, \ldots, w_n) \quad \forall i \in \mathcal{I}, \tag{8}$$

where $\mathcal{I}$ is the index set

$$\begin{aligned} \mathcal{I} = &\prod_{j=1}^{r} [1, |V_h^j|] = \\ &\{(1, 1, \ldots, 1), (1, 1, \ldots, 2), \ldots, (N^1, N^2, \ldots, N^r)\}. \end{aligned} \tag{9}$$

We refer to the tensor A as the *discrete operator* generated by the form a and the particular choice of basis functions. For any given form of arity $r + n$, the tensor A is a (typically sparse) tensor of rank r and dimension $|V_h^1| \times |V_h^2| \times \ldots \times |V_h^r| = N^1 \times N^2 \times \ldots \times N^r$.

Typically, the rank r is 0, 1, or 2. When $r = 0$, the tensor A is a scalar (a tensor of rank zero), when $r = 1$, the tensor A is a vector (the "load vector") and when $r = 2$, the tensor A is a matrix (the "stiffness matrix"). Forms of higher rank also appear, though they are rarely assembled as a higher-dimensional sparse tensor.

Note here that we consider the functions $w_1, w_2, \ldots, w_n$ as fixed in the sense that the discrete operator A is computed for a given set of functions, which we refer to as

coefficients. As an example, consider again the variational problem (2) for Poisson's equation. For the trilinear form a, the rank is $r = 2$ and the number of coefficients is $n = 1$, while for the linear form L, the rank is $r = 1$ and the number of coefficients is $n = 1$. We may also choose to directly compute the *action* of the form a obtained by assembling a vector from the form

$$a(v_1; w_1, w2) = \int_\Omega w_1 \nabla v_1 \cdot \nabla w_2 \,\mathrm{d}x, \tag{10}$$

where now $r = 1$ and $n = 2$.

We list below a few other examples to illustrate the notation.

Example 2.1. *Our first example is related to the divergence constraint in fluid flow. Let the form a be given by*

$$a(q, u) = \int_\Omega q \nabla \cdot u \,\mathrm{d}x, \quad q \in V_h^1, \quad u \in V_h^2, \tag{11}$$

where V_h^1 is a space of scalar-valued functions and where V_h^2 is a space of vector-valued functions. The form $a : V_h^1 \times V_h^2 \to \mathbb{R}$ has two primary arguments and thus $r = 2$. Furthermore, the form does not depend on any coefficients and thus $n = 0$.

Example 2.2. *Another common form in fluid flow (with variable density) is*

$$a(v, u; w, \varrho) = \int_\Omega \varrho \,(w \cdot \nabla u) \cdot v \,\mathrm{d}x. \tag{12}$$

Here, $v \in V_h^1$, $u \in V_h^2$, $w \in W_h^1$, $\varrho \in W_h^2$, where V_h^1, V_h^2, and W_h^1 are spaces of vector-valued functions, while W_h^2 is a space of scalar-valued functions. The form takes four arguments, where two of the arguments are coefficients,

$$a : V_h^1 \times V_h^2 \times W_h^1 \times W_h^2 \to \mathbb{R}. \tag{13}$$

Hence, $r = 2$ and $n = 2$.

Example 2.3. *We next consider the following form appearing in nonlinear convection-diffusion with a power-law viscosity,*

$$a(v; w, \mu, \varrho) = \int_\Omega \varrho (w \cdot \nabla w) \cdot v + \mu |\nabla w|^{2q} \nabla w : \nabla v \,\mathrm{d}x. \tag{14}$$

Here, $v \in V_h^1$, $w \in W_h^1$, $\mu \in W_h^2$, $\varrho \in W_h^3$, where V_h^1, and W_h^1 are spaces of vector-valued functions, while W_h^2 and W_h^3 are spaces of scalar-valued functions. The form takes four arguments, where three of the arguments are coefficients,

$$a : V_h^1 \times W_h^1 \times W_h^2 \times W_h^3 \to \mathbb{R}. \tag{15}$$

Hence, $r = 1$ and $n = 3$.

Example 2.4. *The $H^1(\Omega)$ norm of the error $e = u - u_h$ squared is*

$$a(;u,u_h) = \int_\Omega (u-u_h)^2 + |\nabla(u-u_h)|^2 \,\mathrm{d}x. \tag{16}$$

The form takes two arguments and both are coefficients,

$$a : W_h^1 \times W_h^2 \to \mathbb{R}. \tag{17}$$

Hence, $r = 0$ and $n = 2$.

Defining variational forms for coupled PDEs can be performed in two ways in the above described framework. One approach is to couple the variational forms on the linear algebra level, using block vectors and block matrices and defining one form for each block. Alternatively, a single form for the coupled system may be defined using mixed finite elements.

3 Finite Element Assembly

The standard algorithm for computing the global sparse tensor A is known as *assembly*, see [34, 18]. By this algorithm, the tensor A may be computed by assembling (summing) the contributions from the local entities of a finite element mesh. To express this algorithm for assembly of the global sparse tensor A for a general finite element variational form of rank r, we introduce the following notation and assumptions.

Let $\mathcal{T} = \{K\}$ be a set of disjoint *cells* (a triangulation or tessellation) partitioning the domain $\Omega = \cup_{K\in\mathcal{T}} K$. Further, let $\partial_e\mathcal{T}$ denote the set of *exterior facets* (the set of cell facets on the boundary $\partial\Omega$), and let $\partial_i\mathcal{T}$ denote the set of *interior facets* (the set of cell facets not on the boundary $\partial\Omega$). For each discrete function space V_h^j, $j = 1, 2, \ldots, r$, we assume that the global basis $\{\phi_i^j\}_{i=1}^{N^j}$ is obtained by patching together local function spaces $\mathcal{P}_K^j$ on each cell K as determined by a local-to-global mapping.

We shall further assume that the variational form (6) may be expressed as a sum of integrals over the cells $\mathcal{T}$, the exterior facets $\partial_e\mathcal{T}$ and the interior facets $\partial_i\mathcal{T}$. We shall allow integrals expressed on disjoint subsets $\mathcal{T} = \cup_{k=1}^{n_c}\mathcal{T}_k$, $\partial_e\mathcal{T} = \cup_{k=1}^{n_e}\partial_e\mathcal{T}_k$ and $\partial_i\mathcal{T} = \cup_{k=1}^{n_i}\partial_i\mathcal{T}_k$ respectively.

We thus assume that the form a is given by

$$\begin{aligned} &a(v_1,\ldots,v_r;w_1,\ldots,w_n) = \\ &\quad\sum_{k=1}^{n_c}\sum_{K\in\mathcal{T}_k}\int_K I_k^c(v_1,\ldots,v_r;w_1,\ldots w_n)\,\mathrm{d}x \\ &+\sum_{k=1}^{n_e}\sum_{S\in\partial_e\mathcal{T}_k}\int_S I_k^e(v_1,\ldots,v_r;w_1,\ldots,w_n)\,\mathrm{d}s \\ &+\sum_{k=1}^{n_i}\sum_{S\in\partial_i\mathcal{T}_k}\int_S I_k^i(v_1,\ldots,v_r;w_1,\ldots,w_n)\,\mathrm{d}s. \end{aligned} \tag{18}$$

We refer to an integral over a cell K as a *cell integral*, an integral over an exterior facet S as an *exterior facet integral* (typically used to implement Neumann and Robin

type boundary conditions), and to an integral over an interior facet S as an *interior facet integral* (typically used in discontinuous Galerkin methods).

For simplicity, we consider here initially assembly of the global sparse tensor A corresponding to a form a given by a single integral over all cells $\mathcal{T}$, and later extend to the general case where we must also account for contributions from several cell integrals, interior facet integrals and exterior facet integrals.

We thus consider the form

$$\begin{aligned} &a(v_1, \ldots, v_r; w_1, \ldots, w_n) = \\ &\quad \sum_{K \in \mathcal{T}} \int_K I^c(v_1, \ldots, v_r; w_1, \ldots, w_n) \, \mathrm{d}x, \end{aligned} \tag{19}$$

for which the global sparse tensor A is given by

$$A_i = \sum_{K \in \mathcal{T}} \int_K I^c(\phi^1_{i_1}, \ldots, \phi^r_{i_r}; w_1, \ldots, w_n) \, \mathrm{d}x. \tag{20}$$

To see how to compute the tensor A by summing the local contributions from each cell K, we let $n^j_K = |\mathcal{P}^j_K|$ denote the dimension of the local finite element space on K for the jth primary argument $v_j \in V^j_h$ for $j = 1, 2, \ldots, r$. Furthermore, let

$$\iota^j_K : [1, n^j_K] \to [1, N^j] \tag{21}$$

denote the local-to-global mapping for V^j_h, that is, on any given $K \in \mathcal{T}$, the mapping ι^j_K maps the number of a local degree of freedom (or, equivalently, local basis function) to the number of the corresponding global degree of freedom (or, equivalently, global basis function). We then define for each $K \in \mathcal{T}$ the collective local-to-global mapping $\iota_K : \mathcal{I}_K \to \mathcal{I}$ by

$$\iota_K(i) = (\iota^1_K(i_1), \iota^2_K(i_2), \ldots, \iota^r_K(i_r)) \quad \forall i \in \mathcal{I}_K, \tag{22}$$

where $\mathcal{I}_K$ is the index set

$$\begin{aligned} \mathcal{I}_K &= \prod_{j=1}^{r} [1, |\mathcal{P}^j_K|] \\ &= \{(1, 1, \ldots, 1), (1, 1, \ldots, 2), \ldots, (n^1_K, n^2_K, \ldots, n^r_K)\}. \end{aligned} \tag{23}$$

Furthermore, for each V^j_h we let $\{\phi^{K,j}_i\}_{i=1}^{n^j_K}$ denote the restriction to an element K of the subset of the basis $\{\phi^j_i\}_{i=1}^{N^j} \subset \mathcal{P}^j_K$ of V^j_h supported on K.

We may now compute A by summing the contributions from the local cells,

$$\begin{aligned} A_i &= \sum_{K \in \mathcal{T}_i} \int_K I^c(\phi^1_{i_1}, \ldots, \phi^r_{i_r}; w_1, \ldots, w_n) \, \mathrm{d}x \\ &= \sum_{K \in \mathcal{T}_i} \int_K I^c(\phi^{K,1}_{(\iota^1_K)^{-1}(i_1)}, \ldots, \phi^{K,r}_{(\iota^r_K)^{-1}(i_r)}; w_1, \ldots, w_n) \, \mathrm{d}x \\ &= \sum_{K \in \mathcal{T}_i} A^K_{\iota_K^{-1}(i)}, \end{aligned} \tag{24}$$

where A^K is the local *cell tensor* on cell K (the "element stiffness matrix"), given by

$$A_i^K = \int_K I^c(\phi_{i_1}^{K,1}, \ldots, \phi_{i_r}^{K,r}; w_1, \ldots, w_n)\,\mathrm{d}x, \tag{25}$$

and where $\mathcal{T}_i$ denotes the set of cells on which all basis functions $\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r$ are supported. Similarly, we may sum the local contributions from the exterior and interior facets in the form of local *exterior facet tensors* and *interior facet tensors*.

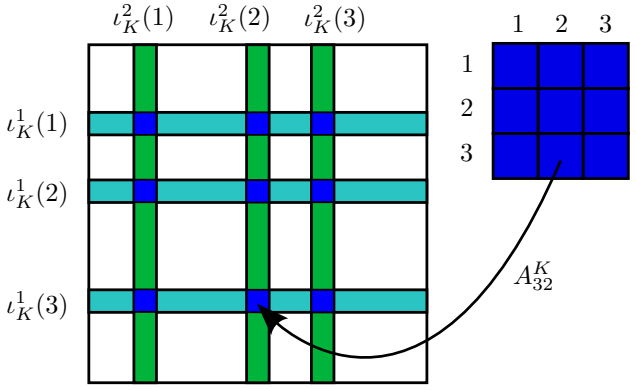

Figure 1: Adding the entries of a cell tensor A^K to the global tensor A using the local-to-global mapping ι_K, illustrated here for a rank two tensor (a matrix).

In Algorithm 2, we present a general algorithm for assembling the contributions from the local cell, exterior facet and interior facet tensors into a global sparse tensor. In all cases, we iterate over all entities (cells, exterior or interior facets), compute the local cell tensor A^K (or exterior/interior facet tensor A^S) and add it to the global sparse tensor as determined by the local-to-global mapping, see Figure 1.

4 Software Framework for Finite Element Assembly

In a finite element application code, typical input from the user is the variational (weak) form of a PDE, a choice of finite elements, a geometry represented by a mesh, and user-defined functions that appear as coefficients in the variational form. For a linear PDE, the typical solution procedure consists of first assembling a (sparse) linear system $AU = b$ from given user input and then solving that linear system to obtain the degrees of freedom U for the discrete finite element approximation u_h of the exact solution u of the PDE. Even when the solution procedure is more involved, as for a nonlinear problem requiring an iterative procedure, each iteration may involve assembling matrices and vectors. It is therefore clear that the assembly of matrices and vectors (or in general tensors) is an important task for any finite element software framework. We refer to the software component responsible for assembling a global tensor from given user input consisting of a variational form, finite element function spaces, mesh and coefficients as the *Assembler*.

Algorithm 2 Assembling the global tensor A from the local contributions on all cells, exterior and interior facets. For assembly over exterior facets, $K(S)$ refers to the cell $K \in \mathcal{T}$ incident with the exterior facet S, and for assembly over interior facets, $K(S)$ refers to the "macro cell" consisting of the pair of cells K^+ and K^- incident with the interior facet S.

$A = 0$

(i) *Assemble contributions from all cells*

for each $K \in \mathcal{T}$

> **for** $j = 1, 2, \ldots, r$:
> Tabulate the local-to-global mapping ι_K^j
>
> **for** $j = 1, 2, \ldots, n$:
> Extract the values of w_j on K
>
> Take $0 \leq k \leq n_c$ such that $K \in \mathcal{T}_k$
> Tabulate the cell tensor A^K for I_k^c
> Add A_i^K to $A_{\iota_K^1(i_1), \iota_K^2(i_2), \ldots, \iota_K^r(i_r)}$ for $i \in I_K$

(ii) *Assemble contributions from all exterior facets*

for each $S \in \partial_e \mathcal{T}$

> **for** $j = 1, 2, \ldots, r$:
> Tabulate the local-to-global mapping $\iota_{K(S)}^j$
>
> **for** $j = 1, 2, \ldots, n$:
> Extract the values of w_j on $K(S)$
>
> Take $0 \leq k \leq n_e$ such that $S \in \partial_e \mathcal{T}_k$
> Tabulate the exterior facet tensor A^S for I_k^e
> Add A_i^S to $A_{\iota_{K(S)}^1(i_1), \iota_{K(S)}^2(i_2), \ldots, \iota_{K(S)}^r(i_r)}$ for $i \in I_{K(S)}$

(iii) *Assemble contributions from all interior facets*

for each $S \in \partial_i \mathcal{T}$

> **for** $j = 1, 2, \ldots, r$:
> Tabulate the local-to-global mapping $\iota_{K(S)}^j$
>
> **for** $j = 1, 2, \ldots, n$:
> Extract the values of w_j on $K(S)$
>
> Take $0 \leq k \leq n_i$ such that $S \in \partial_i \mathcal{T}_k$
> Tabulate the interior facet tensor A^S for I_k^i
> Add A_i^S to $A_{\iota_{K(S)}^1(i_1), \iota_{K(S)}^2(i_2), \ldots, \iota_{K(S)}^r(i_r)}$ for $i \in I_{K(S)}$

As demonstrated in Algorithm 2, the Assembler needs to iterate over the cells in the mesh, tabulate degree of freedom mappings, extract local values of coefficients, compute the local element tensor, and add each element tensor to the global tensor which is the final output. Thus, the Assembler is a software component where many other components are combined. It is therefore important that the software components on which the Assembler depends have well-defined interfaces. We discuss some issues relating to the design of these software components below and then demonstrate how these software components together with the Assembler may be combined into a general software framework for finite element assembly.

4.1 Variational Forms

Implementations of discrete variational forms in a general finite element library usually consist of programming expressions for the integrands I_k^c, I_k^e, I_k^i (see Equation (18)), eventually writing a quadrature loop and a loop over element matrix indices depending on the abstraction level of the library (see [12, 24]). An alternative approach is to apply exact integration instead of quadrature. In either case, the result of this computation may be communicated through the UFC interface.

The motivation behind the UFC interface is to separate the implementation of the form from other details of the assembly such as the mesh and the linear algebra libraries in use.

In the FEniCS finite element software framework, a high-level form language embedded in Python is used to define the variational form and finite elements. This reduces the distance from the mathematical formulation of a PDE to an implementation of a PDE solver, removes tedious and error-prone tasks such as coding PDE-specific assembly loops, and enables rapid prototyping of new models and methods. To retain computational efficiency, we generate efficient low-level code from the abstract form description, using exact integration where possible. Code generation adds another complexity layer to the software, and it becomes even more important to keep a clear separation between software components such that the interface between generated code and library code is well defined. This is achieved by generating implementations of the UFC interface.

4.2 Mesh Libraries

Many different representations of computational meshes exist. Typically, each finite element library provides its own internal implementation of a computational mesh. We do not wish to tie the UFC interface to one particular mesh representation or one particular library. Still, several operations like the element tensor computation depends on local mesh data. For this reason, the UFC interface provides a low-level data structure to communicate single cell data. In addition, a small data structure is used to communicate global mesh dimensions which are necessary for computing the local-to-global mapping. Assemblers implemented on top of the UFC interface must therefore be able to copy/translate cell data from the mesh library being used to the UFC data structure (involving a minimal overhead). This makes it possible to achieve separation between the mesh representation and the element tensor computation.

The Assembler implemented in FEniCS (as part of DOLFIN) is implemented for one particular mesh format, see [26], but an Assembler component could easily be written for other mesh libraries like the PETSc *Sieve*, see [22] and the DUNE-Grid interface ([15, 14]).

4.3 Linear Algebra Libraries

It is desirable to reuse existing high-performance linear algebra libraries like PETSc and Trilinos. It is therefore important that the Assembler is able to assemble element tensors into global matrices and vectors implemented by external libraries. Aggregation into matrix and vector data structures are fairly similar operations, typically consisting of passing array pointers to existing functions in the linear algebra libraries. By implementing a common interface for assembly into tensors of arbitrary rank, the same assembly routine can be reused for any linear algebra library without changes. This avoids duplication of assembly code, and one may easily change the output format of the assembly procedure. The details of these interfaces are beyond the scope of the current paper. At the time of writing, we have written assembly routines with support for matrices and vectors from Epetra (Trilinos), PETSc, PyCC, and uBLAS in addition to scalars.

With components available for finite element variational forms, mesh representation, and linear algebra, we may use the UFC interface to combine these components to build a PSE for partial differential equations. The central component of this PSE is the Assembler. As illustrated in Figure 2, the Assembler takes as input a variational form, communicated through the UFC interface, a mesh and a set of functions (the coefficients), and assembles a tensor.

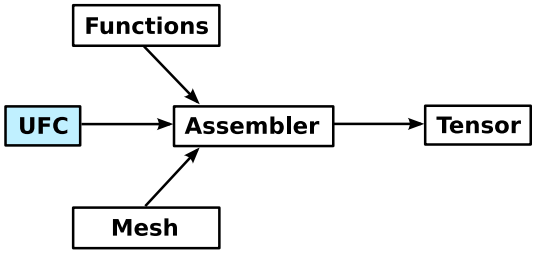

Figure 2: Assembling a tensor from a given UFC, mesh and functions (coefficients).

4.4 High-level Interfaces

In FEniCS, we have additional application-level abstractions for expressing variational forms, meshes, functions and linear algebra objects to achieve a consistent high-level user-interface. The generation of the UFC code may then be hidden from the user, who just provides a high-level description of the form. The PSE may then automatically generate the UFC at run-time, functioning as just-in-time (JIT) compiler, and call the Assembler with the generated UFC. Below, we demonstrate how this may be done in the Python interface of DOLFIN. The user here defines a finite element function space,

and a pair of bilinear and linear forms $a(v,u) = \int_\Omega \nabla v \cdot \nabla u + vu \, \mathrm{d}x$ and $L = \int_\Omega vf \, \mathrm{d}x$, from which a matrix and vector may be assembled by calls to the function `assemble`. A linear system solver may then be invoked to compute the degrees of freedom U of the solution.

```
element = FiniteElement("CG", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element, mesh, 100.0)

a = dot(grad(v), grad(u))*dx + v*u*dx
L = v*f*dx

A = assemble(a, mesh)
b = assemble(L, mesh)

U = solve(A, b)
```

While FEniCS provides an integrated environment, including the PSE DOLFIN and the two form compilers FFC (FEniCS Form Compiler) and SFC (SyFi Form Compiler), the fact that these components comply with the UFC interface means that they may also be used interchangeably in heterogeneous environments together with other libraries (that implement or use the UFC interface). This is illustrated in the flow diagram of Figure 3 where alternate routes from mathematical description to matrix assembly are demonstrated.

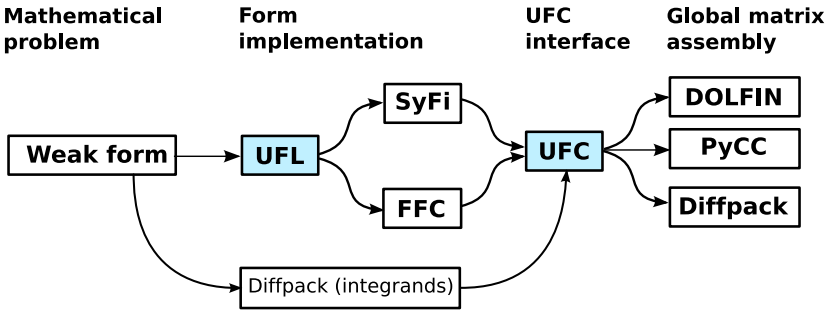

Figure 3: Alternate routes from mathematical description to matrix assembly enabled by the UFC interface (Note that the Diffpack example is fictional).

In Figure 3, we have also included another interface UFL (Unified Form Language) which provides a unified way to express finite element variational forms. The UFL interface is currently in development. Together, UFL and UFC provide a unified interface for the input and output of form compilers, see Figure 4.

Figure 4: An abstract definition (UFL) of a finite element variational form is given as input to a form compiler, which generates UFC code as output.

5 The UFC Interface

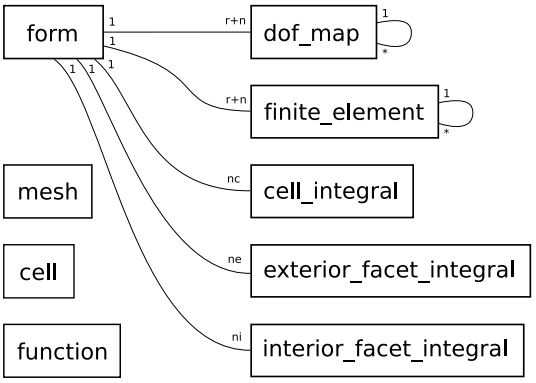

Figure 5: UML diagram of the UFC class relations

The UFC interface consists of a small collection of abstract C++ classes that represent common components for assembling tensors using the finite element method. These classes are accompanied by a well documented ([8]) set of conventions for numbering of cell data and other arrays. We have strived to make the classes as simple as possible while not sacrificing generality or efficiency. Data is passed as plain C arrays for efficiency and minimal dependencies. Most functions are declared `const`, reflecting that the operations they represent should not change the outcome of future operations.[2] Other initialization of implementation-specific data should ideally be performed in constructors.

One can ask why the UFC interface consists of classes and not plain functions. There are three reasons for this. First, we want to handle each form as a self contained "black box", which can be passed around easily in an application. Many functions belong together conceptually, thus making it natural to collect them in a class "namespace". Second, we need multiple versions of each function in the software representation of variational forms, in particular to represent multiple variational forms and multiple finite element function spaces. This is best achieved by making each such function a member function of a class and having multiple instances of that class. Third, UFC function implementations may need access to stored data, and with a plain function-based interface these data would then need to be global variables. In particular, when existing libraries or applications want to implement the UFC interface, it may be necessary for

[2]The exceptions are the functions to initialize a `dof_map`.

the subclasses of UFC classes to inherit from existing classes or to have pointers to other objects.

5.1 Class Relations

Figure (5) shows all the classes and their relations. The classes `mesh`, `cell`, and `function` provide the means for communicating mesh and coefficient function data as arguments. Each argument of the form (both primary arguments and coefficients) is represented by a `finite_element` and `dof_map` object. The integrals are represented by one of the classes `cell_integral`, `exterior_facet_integral`, or `interior_facet_integral`. An object of the class `form` gives access to all other objects in a particular implementation. In this paper, we will not describe all the functions of these classes in detail. A complete specification can be found in the manual ([8]).

At the core of UFC is the class `form`, which represents the general variational form a of Equation (18). Subclasses of `form` must implement factory functions which may be called to create `cell_integral`, `exterior_facet_integral` and `interior_facet_integral` objects. These objects in turn know how to compute their respective contribution from a cell or facet during assembly. A code fragment from the `form` class declaration is shown below.

```
class form
{
public:

  ...

  /// Create cell integral on sub domain i
  virtual cell_integral*
  create_cell_integral(unsigned int i)
  const = 0;

  /// Create exterior facet integral on sub domain i
  virtual exterior_facet_integral*
  create_exterior_facet_integral(unsigned int i)
  const = 0;

  /// Create interior facet integral on sub domain i
  virtual interior_facet_integral*
  create_interior_facet_integral(unsigned int i)
  const = 0;
};
```

The `form` class also specifies functions for creating `finite_element` and `dof_map` objects for the finite element function spaces $\{V_h^j\}_{j=1}^r, \{W_h^j\}_{j=1}^n$ of the variational form. The `finite_element` object provides functionality such as evaluation of degrees of

freedom and evaluation of basis functions and their derivatives. The `dof_map` object provides functionality such as tabulating the local-to-global mapping of degrees of freedom on a single element, as well as tabulation of subsets associated with particular mesh entities, used to apply Dirichlet boundary conditions and build connectivity information.

Both the `finite_element` and `dof_map` classes can represent mixed elements, in which case it is possible to obtain `finite_element` and `dof_map` objects for each sub-element in a hierarchical manner. Vector elements composed of scalar elements are in this context seen as special cases of mixed elements where all sub-elements are equal. Thus, e.g., from a `dof_map` representing a $P_2 - P_1$ Taylor-Hood element, it is possible to extract one `dof_map` for the quadratic vector element and one `dof_map` for the linear scalar element. From the vector element, a `dof_map` for the quadratic scalar element of each vector component can be obtained. This can be used to access subcomponents from the solution of a mixed system.

5.2 Stages in the Assembly Algorithm

```
enum shape {interval, triangle, quadrilateral,
            tetrahedron, hexahedron};

class cell {
public:
  shape cell_shape;
  unsigned int topological_dimension;
  unsigned int geometric_dimension;

  /// Array of global indices for the mesh entities of the cell
  unsigned int** entity_indices;

  /// Array of coordinates for the vertices of the cell
  double** coordinates;
};
```

Figure 6: Data structure for communicating single cell data.

Next, we focus on a few key parts of the interface and explain how these can be used to implement the assembly algorithm (Algorithm 2). This algorithm consists of three stages: (i) assembling the contributions from all cells, (ii) assembling the contributions from all exterior facets, and (iii) assembling the contributions from all interior facets.

Each of the three assembly stages (i)–(iii) of Algorithm 2 is further composed of five steps. In the first step, the polygon K is fetched from the mesh, typically implemented by filling a `cell` structure (see Figure 6) with coordinate data and global numbering of the mesh entities in the cell. This step depends on the specific mesh being used.

Secondly, the local-to-global mapping of degrees of freedom is tabulated for each of the function spaces. That is, for each of the discrete finite element spaces on K, we tabulate (or possibly compute) the global indices for the degrees of freedom on $\{V_h^j\}_{j=1}^r$ and $\{W_h^j\}_{j=1}^n$.

The class `dof_map` represents the mapping between local and global degrees of freedom for a finite element space. A `dof_map` is initialized with global mesh dimensions by calling the function `init_mesh(const mesh& m)`. If this function returns `true`, the `dof_map` should be additionally initialized by calling the function `init_cell(const mesh& m, const cell& c)` for each cell in the global mesh, followed by `init_cell_finalize` after the last cell. After the initialization stage, the mapping may be tabulated at a given cell by calling a function with the following signature.

```
void dof_map::tabulate_dofs(unsigned int* dofs,
                            const mesh& m,
                            const cell& c) const
```

Here, `unsigned int* dofs` is a pointer to the first element of an array of unsigned integers that will be filled with the local-to-global mapping on the current cell during the function call.

In the third step of each stage of Algorithm 2, we may use the tabulated local-to-global mapping to interpolate (extract) the local values of any of the coefficients in $\{W_h^j\}_{j=1}^n$.

If a coefficient w_j is not given as a linear combination of basis functions for W_h^j, it must at this step be interpolated into W_h^j, using the interpolant defined by the degrees of freedom of W_h^j (for example point evaluation at a set of nodal points). In this case, the coefficient function is passed as an implementation of the `function` interface (a simple functor) to the function `evaluate_dofs`.

```
/// Evaluate linear functionals for all dofs on the function f
void finite_element::evaluate_dofs(double *values,
                                   const function& f,
                                   const cell& c) const
```

In the fourth step, the local element tensor contributions (cell or exterior/interior facet tensors) are computed. This is done by a call to the function `tabulate_tensor`, illustrated below for a cell integral.

```
void cell_integral::tabulate_tensor(double* A,
                                    const double * const * w,
                                    const cell& c) const
```

Similarly, one may evaluate interior and exterior facet contributions using slightly different function signatures.

Finally, at the fifth step, the local element tensor contributions are added to the global tensor, using the local-to-global mappings previously obtained by calls to the `tabulate_dofs` function. This is a simple operation that depends on the linear algebra library in use.

6 Examples

In this section, we demonstrate how UFC is used in practice in DOLFIN, FFC, and SFC. First, we show a part of the assembly algorithm (Algorithm 2) as implemented in DOLFIN. We then show examples of input to the form compilers FFC and SFC as well as part of the corresponding UFC code generated as output. Examples include Poisson's equation and linear convection (see Example 2.2).

6.1 An Example UFC Assembler

To demonstrate how one may implement an assembler based on the UFC interface, we provide here a (somewhat simplified) excerpt from the DOLFIN assembler.[3]

```
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  ufc.update(*cell);

  for (uint i = 0; i < ufc.form.rank(); i++)
    dof_map_set[i].tabulate_dofs(ufc.dofs[i], *cell);

  for (uint i = 0; i < coefficients.size(); i++)
    coefficients[i]->interpolate(ufc.w[i], ufc.cell,
                                 *ufc.coefficient_elements[i],
                                 *cell);

  integral->tabulate_tensor(ufc.A, ufc.w, ufc.cell);

  A.add(ufc.A, ufc.local_dimensions, ufc.dofs);
}
```

The outer loop iterates over all cells in a given mesh. For each cell, a `ufc::cell` is updated and the local-to-global mapping is constructed. We then interpolate all the form coefficients on the cell and compute the element tensor. At the end of the iteration, the local-to-global mapping is used to add the local tensor to the global tensor.

6.2 FFC Examples

The form compiler FFC provides a simple language for specification of variational forms, which may be entered either directly in Python or in text files given to the compiler on the command-line. For each variational form given as input, FFC generates UFC-compliant C++ code for evaluation of the corresponding element tensor(s).

[3]The `ufc` object is here an instance of a simple DOLFIN class that holds pointers to arrays and UFC container classes, such as the array `A` and cell data `ufc::cell`, needed to communicate through the UFC interface.

Poisson's Equation

We consider the following input file to FFC for Poisson's equation.

```
element = FiniteElement("Lagrange", "triangle", 1)

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

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

Here, two forms `a` (bilinear) and `L` (linear) are defined. Both the test and trial spaces are spanned by linear Lagrange elements on triangles in two dimensions. When compiling this code using FFC, a C++ header file is created, containing UFC code that may be used to assemble the global sparse stiffness matrix and load vector. Below, we present the code generated for evaluation of the element stiffness matrix for the bilinear form `a`.

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];

  // Compute determinant of Jacobian
  double detJ = J_00*J_11 - J_01*J_10;

  // Compute inverse of Jacobian
  const double Jinv_00 =  J_11 / detJ;
  const double Jinv_01 = -J_01 / detJ;
  const double Jinv_10 = -J_10 / detJ;
  const double Jinv_11 =  J_00 / detJ;

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensors
  const double G0_0_0 = det*(Jinv_00*Jinv_00 + Jinv_01*Jinv_01);
  const double G0_0_1 = det*(Jinv_00*Jinv_10 + Jinv_01*Jinv_11);
  const double G0_1_0 = det*(Jinv_10*Jinv_00 + Jinv_11*Jinv_01);
  const double G0_1_1 = det*(Jinv_10*Jinv_10 + Jinv_11*Jinv_11);

  // Compute element tensor
  A[0] = 0.5*G0_0_0 + 0.5*G0_0_1 + 0.5*G0_1_0 + 0.5*G0_1_1;
  A[1] = -0.5*G0_0_0 - 0.5*G0_1_0;
  A[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  A[3] = -0.5*G0_0_0 - 0.5*G0_0_1;
  A[4] = 0.5*G0_0_0;
  A[5] = 0.5*G0_0_1;
  A[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
```

```
  A[8] = 0.5*G0_1_1;
}
```

In FFC, an element tensor contribution is computed as a tensor contraction between a geometry tensor varying from cell to cell, and a geometry independent tensor on a reference element, see [20, 21]. For simple forms, like the one under discussion, the main work is then to construct the geometry tensor, related to the geometrical mapping between the reference element and physical element.

Having computed the element tensor, one needs to compute the local-to-global mapping in order to know where to insert the local contributions in the global tensor. This mapping may be obtained by calling the member function `tabulate_dofs` of the class `ufc::dof_map`. FFC uses an implicit ordering scheme, based on the indices of the topological entities in the mesh. This information may be extracted from the `ufc::cell` attribute `entity_indices`.

```
virtual void tabulate_dofs(unsigned int* dofs,
                           const ufc::mesh& m,
                           const ufc::cell& c) const
{
  dofs[0] = c.entity_indices[0][0];
  dofs[1] = c.entity_indices[0][1];
  dofs[2] = c.entity_indices[0][2];
}
```

For Lagrange elements on triangles, each degree of freedom is associated with a global vertex. Hence, FFC constructs the mapping by picking the corresponding global vertex number for each degree of freedom.

Linear Convection

Consider the variational form in Example 2.2. The input file to FFC reads as follows.

```
vector_element = VectorElement("Lagrange", "triangle", 1)
scalar_element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(vector_element)
u = TrialFunction(vector_element)
w = Function(vector_element)
rho = Function(scalar_element)

a = rho*v[i]*w[j]*u[i].dx(j)*dx
```

The code generated for the `tabulate_tensor` function is presented below. Computations involving coefficients are performed by interpolating the functions w and ρ on the cell under consideration. These values are stored in the array `w` below. For clarity, some code has been omitted in this example.

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  // Extract vertex coordinates and compute Jacobian etc
  // as in previous example
  ...
  // Compute coefficients
  const double c1_0_0_0 = w[1][0];
  const double c1_0_0_1 = w[1][1];
  const double c1_0_0_2 = w[1][2];
  const double c0_0_1_0 = w[0][0];
  ...
  const double c0_0_1_5 = w[0][5];
  // Compute geometry tensors
  const double G0_0_0_0_0 = det*c1_0_0_0*c0_0_1_0*Jinv_00;
  const double G0_0_0_1_0 = det*c1_0_0_0*c0_0_1_0*Jinv_10;
  const double G0_0_1_0_0 = det*c1_0_0_0*c0_0_1_1*Jinv_00;
  ...
  // Compute element tensor
  A[0] = -0.05*G0_0_0_0_0 - ...
  A[1] =  0.05*G0_0_0_0_0 + ...
  A[2] =  0.05*G0_0_0_1_0 + ...
  ...
}
```

The local-to-global mapping for the space of piecewise linear vectors is computed by associating two values with each vertex. The code generated for `tabulate_dofs` is presented below.

```
virtual void tabulate_dofs(unsigned int* dofs,
                           const ufc::mesh& m,
                           const ufc::cell& c) const
{
  dofs[0] = c.entity_indices[0][0];
  dofs[1] = c.entity_indices[0][1];
  dofs[2] = c.entity_indices[0][2];
  unsigned int offset = m.num_entities[0];
  dofs[3] = offset + c.entity_indices[0][0];
  dofs[4] = offset + c.entity_indices[0][1];
  dofs[5] = offset + c.entity_indices[0][2];
}
```

FFC generates code for arbitrary multilinear forms and currently supports arbitrary degree continuous Lagrange elements, discontinuous elements, RT elements, BDM elements, BDFM elements and Nedelec elements in two and three space dimensions.

6.3 SFC Examples

SFC is another form compiler producing UFC code, in which the user defines variational forms in Python using a symbolic engine based on GiNaC ([16]). It has a slightly different feature set than FFC, such as using symbolic differentiation to automatically compute the Jacobi matrix of a nonlinear form. The resulting low-level UFC code is very similar.

Power-law Viscosity

Example 2.3 is specified in SFC as follows.

```
vector_element = VectorElement("Lagrange", "triangle", 1)
scalar_element = FiniteElement("DG", "triangle", 0)

v = TestFunction(vector_element)
w = Function(vector_element)
mu = Function(scalar_element)
rho = Function(scalar_element)

def power_law(v, w, mu, rho, itg):
    q = 0.3
    GinvT = itg.GinvT()
    Dw  = grad(w, GinvT)
    Dv  = grad(v, GinvT)
    wDw = dot(w, Dw)

    return rho*dot(wDw, v) + mu*inner(Dw,Dw)**q * inner(Dw,Dv)

F_form = Form(basisfunctions = [v],
              coefficients   = [w, mu, rho])
F_form.add_cell_integral(power_law)
J_form = Jacobi(F_form)
```

The syntax for defining elements and arguments is the same as in FFC, but the integrand is specified in a slightly different syntax[4]. This code also computes the form corresponding to the Jacobian matrix, using symbolic differentiation. The generated code for computing the Jacobian matrix is in this case more complicated but it implements the UFC interface in the same manner as in the previous examples.

In the current implementation, SFC explicitly constructs a local-to-global mapping at run-time. In this case, with Lagrange elements, the global coordinates identify the degrees of freedom. The UFC interface supports constructing the local-to-global mapping through the `init_mesh` and `init_cell` methods of `ufc::dof_map`. Below, we present the code generated for `init_cell` (where we use additional structures of type `Ptv` (point) for representing degrees of freedom and the container `Dof_Ptv dof` for building the local-to-global mapping).

```
void dof_map_2D::init_cell(const ufc::mesh& m, const ufc::cell& c)
{
  // coordinates
  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];
```

[4]The variable `itg` is an integral object containing information about the mapping between physical coordinates and the reference element.

```
  // affine map
  double G00 = x1 - x0;
  double G01 = x2 - x0;

  double G10 = y1 - y0;
  double G11 = y2 - y0;

  unsigned int element = c.entity_indices[2][0];

  double dof0[2] = { x0, y0 };
  Ptv pdof0(2, dof0);
  dof.insert_dof(element, 0, pdof0);

  double dof1[2] = {  G00+x0,  y0+G10 };
  Ptv pdof1(2, dof1);
  dof.insert_dof(element, 1, pdof1);

  double dof2[2] = {  x0+G01,  G11+y0 };
  Ptv pdof2(2, dof2);
  dof.insert_dof(element, 2, pdof2);
}
```

The `dof_map` class is only responsible for the uniqueness of the local-to-global mapping. Possible renumbering strategies may be imposed by the assembler, for example to minimize communication when assembling in parallel.

7 Discussion

We have used (generated) UFC for many applications, including Poisson's equation; convection–diffusion–reaction equations; continuum equations for linear elasticity, hyperelasticity, and plasticity; the incompressible Navier-Stokes equations; and mixed formulations for the Hodge Laplacian. The types of finite elements involved include standard continuous Lagrange elements of arbitrary order, discontinuous Galerkin formulations, BDM elements, Raviart–Thomas elements, Crouzeix–Raviart elements, and Nedelec elements.

The form compilers FFC and SFC are UFC compliant, both generating efficient UFC code from an abstract problem definition. Assemblers have been implemented in DOLFIN and PyCC, using the DOLFIN mesh representation, and together covering linear algebra formats from PETSc, Trilinos (Epetra), uBLAS, and PyCC. Parallel assembly is supported in the current development version of DOLFIN, without requiring any modifications to UFC since it operates on an element level. Altogether, this demonstrates that the UFC interface is flexible both in terms of the applications and finite element formulations it covers, and in terms of its interoperability with existing libraries.

One of the main limitations in the current version of the UFC interface (v1.1) is the assumption of a homogeneous mesh, that is, only one cell shape is allowed throughout the mesh. Thus, although mesh ordering conventions have been defined for the interval, triangle, tetrahedron, quadrilateral, and hexahedron, only one type of shape can be used at any time. Also, higher order (non-affinely mapped) meshes are not supported in the current version of the interface. Another limitation is that only one fixed finite element space can be chosen for each argument of the form, which excludes p-refinement (increasing the element order in a subset of the cells). All these limitations may be removed in future versions of UFC, and we encourage interested developers to make contact to address these limitations.

UFC provides a unified interface for code generated as *output* by form compilers such as FFC and SFC. Similarly, we are currently working on a specification for a unified form language (UFL) to function as a common *input* to form compilers. Currently, both FFC and SFC provide (different) form languages for easy specification of variational forms in a high-level syntax. With a unified form language, a user may specify a variational form in that language and assemble the corresponding discrete operator (tensor), independently of the components being used to generate the UFC code from the UFL, and independently of the components being used to assemble the tensor from the UFC form.

8 Conclusion

We have presented a general framework for assembly of finite element variational forms. Based on this framework, we have then extracted an interface (UFC) that may be used to provide a communication layer between general-purpose and problem-specific code for assembly of finite element variational forms.

The interface makes minimal assumptions on the type of problem being solved and the data structures involved. For example, the discrete variational form may in general be multilinear and hence assemble into a tensor of arbitrary rank. The basic data structures used to pass data through the interface are composed of plain C arrays. The minimal set of assumptions on problem and data structures enables application of the interface to a wide range of variational forms and a large collection of finite element libraries.

We have used the UFC interface in the implementation of the FEniCS suite of finite element tools. In a simple Python script, one may define a variational form and a mesh, and assemble the corresponding global sparse matrix (or vector). When doing so, UFC code is generated by either of the form compilers FFC or SFC, and passed to the UFC-compatible assembler of the general-purpose finite element library DOLFIN.

We encourage developers of finite element software to use the UFC interface in their libraries. By doing so, those libraries may directly take advantage of the form compilers FFC or SFC to specify finite element problems. Moreover, one can think of already existing specifications of complicated finite element problems that via UFC can be combined with other libraries than the specifications were originally written for. We have tried to make minimal assumptions to make this possible.

We believe that UFC itself and the ideas behind it constitute an important step

towards greater flexibility in finite element software. By code generation via tools like FFC and SFC, this flexibility may be retained also in combination with very high performance.

Bibliography

[1] *Cactus. http://www.cactuscode.org/.*

[2] *COMSOL Multiphysics. http://www.comsol.com.*

[3] *FEniCS. http://www.fenics.org.*

[4] *Getfem++. http://home.gna.org/getfem/.*

[5] *Hypre. http://acts.nersc.gov/hypre/.*

[6] *Kaskade.*
http://www.zib.de/Numerik/numsoft/kaskade/.

[7] *Trilinos. http://software.sandia.gov/trilinos.*

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

[9] M. S. Alnæs and K.-A. Mardal, *SyFi: Symbolic finite elements*, 2008. *http://www.fenics.org/syfi/.*

[10] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc*, 2008. *http://www.mcs.anl.gov/petsc/.*

[11] W. Bangerth, R. Hartmann, and G. Kanschat, `deal.II` *Differential Equations Analysis Library*, 2006. *http://www.dealii.org/.*

[12] W. Bangerth, R. Hartmann, and G. Kanschat, *Deal.II — a general-purpose object-oriented finite element library*, ACM Trans. Math. Softw., 33 (2007).

[13] P. Bastian, K. Birken, S. Lang, K. Johannsen, N. Neuss, H. Rentz-Reichert, and C. Wieners, *UG - a flexible software toolbox for solving partial differential equations*, Computing and Visualization in Science, 1 (1997), pp. 27–40.

[14] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. part II: implementation and tests in DUNE*, Computing, 82 (2008), pp. 121–138.

[15] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klfkorn, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework*, Computing, 82 (2008), pp. 103–119.

[16] C. Bauer, C. Dams, A. Frink, V. V. Kisil, R. Kreckel, A. Sheplyakov, and J. Vollinga, *GiNaC*, 2006. http://www.ginac.de.

[17] P. G. Ciarlet, *The Finite Element Method for Elliptic Problems*, North-Holland, Amsterdam, New York, Oxford, 1978.

[18] T. J. R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1987.

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

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

[21] ——, *Efficient compilation of a class of variational forms*, ACM Transactions on Mathematical Software, 33 (2007), p. Article no. 17.

[22] M. G. Knepley and D. A. Karpeev, *Mesh algorithms for PDE with Sieve I: Mesh distribution*, Scientific Programming, (2009). To appear, `http://www.mcs.anl.gov/uploads/cels/papers/P1455.pdf`.

[23] H. P. Langtangen, *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*, Texts in Computational Science and Engineering, vol 1, Springer, 2nd ed., 2003.
http://www.diffpack.com.

[24] ——, *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*, Springer-Verlag, Berlin, 2003. 2nd edition, 855 pages.

[25] A. Logg, *Automating the finite element method*, Arch. Comput. Methods Eng., 14 (2007), pp. 93–138.

[26] ——, *Efficient representation of computational meshes*, International Journal of Computational Science and Engineering, (2009). To appear.

[27] A. Logg, K. Oelgaard, M. Rognes, and G. N. Wells, *FFC: FEniCS form compiler. http//www.fenics.org/ffc/*, 2008.

[28] A. Logg, G. Wells, J. Hoffman, J. Jansson, et al., *DOLFIN: A general-purpose finite element library. http://www.fenics.org/dolfin/*.

[29] K. Long, *Sundance*, 2006.
http://software.sandia.gov/sundance/.

[30] K. Meerbergen, *GLAS: Generic interface for Linear Algebra Software. http://glas.sourceforge.net/*, 2008.

[31] J. Ring, H. P. Langtangen, and R. Bredesen, *Easyviz*. Subpackage of SciTools:
http://code.google.com/p/scitools/, 2008.

[32] O. Skavhaug, M. S. Alnæs, K.-A. Mardal, G. Staff, Å. Ødegård, and G. T. Lines, *PyCC*, 2008. Software framework under development. *http://www.simula.no/pycc*.

[33] M. Thuné, E. Mossberg, P. Olsson, J. Rantakokko, K. Åhlander, and K. Otto, *Object-oriented construction of parallel PDE solvers*, in Modern Software Tools for Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser, 1997, pp. 203–226.

[34] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The Finite Element Method — Its Basis and Fundamentals, 6th edition*, Elsevier, Oxford, 2005, first published in 1967.

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

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

M. S. Alnæs [1], **K-A. Mardal** [1,2]

[1] Center for Biomedical Computing, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway
[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

Abstract: 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.

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 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 [4].

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 [2], which is part of the FEniCS project [13]. SyFi stands for symbolic finite elements and is implemented in C++ and Python, building on the symbolic C++ library GiNaC [9, 8] and its Python

interface Swiginac [35]. 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 [5], the Crouzeix-Raviart element [11], the Hermite element, the standard Lagrange elements, the Nedelec elements [29, 30], the Raviart-Thomas element [34], and the robust Darcy-Stokes element [27]. 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 [17] 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 [12] and FreeFEM [31] implement domain specific languages. Other libraries like Diffpack [10, 22], Deal.II [7, 6], Life [32] and Sundance [26] employ object-orientation and/or template metaprogramming. Another approach which is very similar to ours is taken by the FEniCS Form Compiler (FFC) [19, 20, 23, 24]. 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-assemblyCode) [1, 3]. 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 [19, 20].

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

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

where $0 \leq n \leq 2$, u^0 is the test function, u^1 is the trial function and $w^0, \ldots, w^{m-1}$ 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

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

The weak form is then:

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

where $\mathcal{A}(\ldots)$ and $\mathcal{B}(\ldots)$ are obtained by multiplying $\mathcal{L}(\ldots)$ 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 $W_k = \text{span}(\{\phi^k_j\})$[5] and $w^k = \sum_j w^k_j \phi^k_j$, then the Jacobian with respect to coefficient number k is the derivative of $a_\Omega(\phi^0_i; \ldots)$ with respect to $\{w^k_j\}$,

$$J^k_{ij} = \frac{\partial}{\partial w^k_j} a_\Omega(\phi^0_i; w^0, \ldots, w^{m-1}), \qquad \forall i, j$$

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

$$a^J_\Omega(\phi^0_i, \phi^1_j; w^0, \ldots, w^{m-1}) = J^k_{ij},$$

where $\{\phi^1_j\}$ 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

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

[5]The Sobolev space must be separable.

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 ≥ 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[6] functions, represented by a set of degrees of freedom $\{w^k_j\}$. 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 $\{w^k_j\}$ will be defined by the degrees of freedom of the finite element space used to represent the coefficient, with $w^k = \sum_j w^k_j N^{w^k}_j$. See [1, 3] for more details.

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

$$A_{ij} = a(N^0_i, N^1_j) = \int_T N^0_i \, N^1_j \, d\mathbf{x}$$

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

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

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 $A_i = \sum_j A_{ij} w^0_j$ and $A = \sum A_i w^1_i$.

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.

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

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[7]. 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 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,

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

[7]This is not possible for all elements, e.g. the Rannacher-Turek [33] 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 1: For each element order p, the number of quadrature points used in a quadrature rule of order $q = 2p$ or $q = 2p + 1$.

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 [36], 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 1 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.

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

Figure 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)
```

Figure 2: Code for defining and assembling a mass matrix

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 differential operators `grad`, `div`, and `curl`, as well as operators like `dot` and `inner`[8]. 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

[8]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`.

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.

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_{ij}^T is computed by one call to `mass` with the test function $v = \phi_i^0$ and trial function $u = \phi_j^1$, 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[9]. 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 [3] 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 [15, 16] 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

[9]Using the package Instant [28], which caches the compiled modules and compares MD5 sums of the source code to avoid recompilation if the source code doesn't change.

```
/// 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;
}
```

Figure 3: Snippet of generated C++ code for computing the mass matrix

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 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 [32], 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 @ 2.16GHz 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 [20]. Note that FErari [18, 21], 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.

```
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
```

Figure 4: Definition of example forms with SFC

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

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

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

```
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)
```

Figure 5: Code for defining and compiling forms for a convection vector and its Jacobi matrix

points. Figure 3 shows the code generated by SFC for computing this element matrix. The timing results presented in Table 2, Table 3, 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.

4.2 Example: Right Hand Side Vector

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

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

```
# 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])
```

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

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 quadrilateral 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.

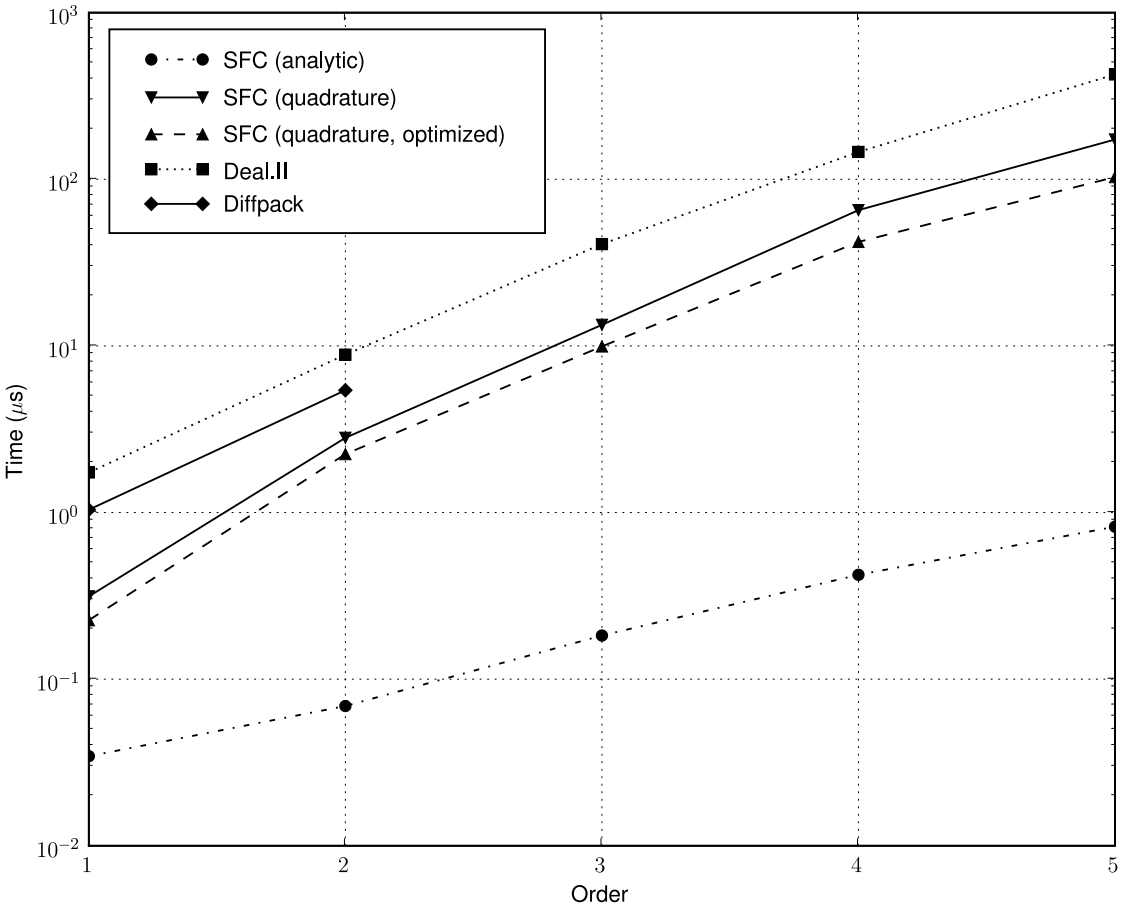

Figure 7: Time to compute the element tensor of the mass form on quadrilateral elements, in μs

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 4 and 5 and Figure 9 shows the timing results. Here we see a growing speed-up with element order similar to the behaviour 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.

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

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

4.4 Example: Nonlinear Convection Vector

The nonlinear convection vector is

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

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

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

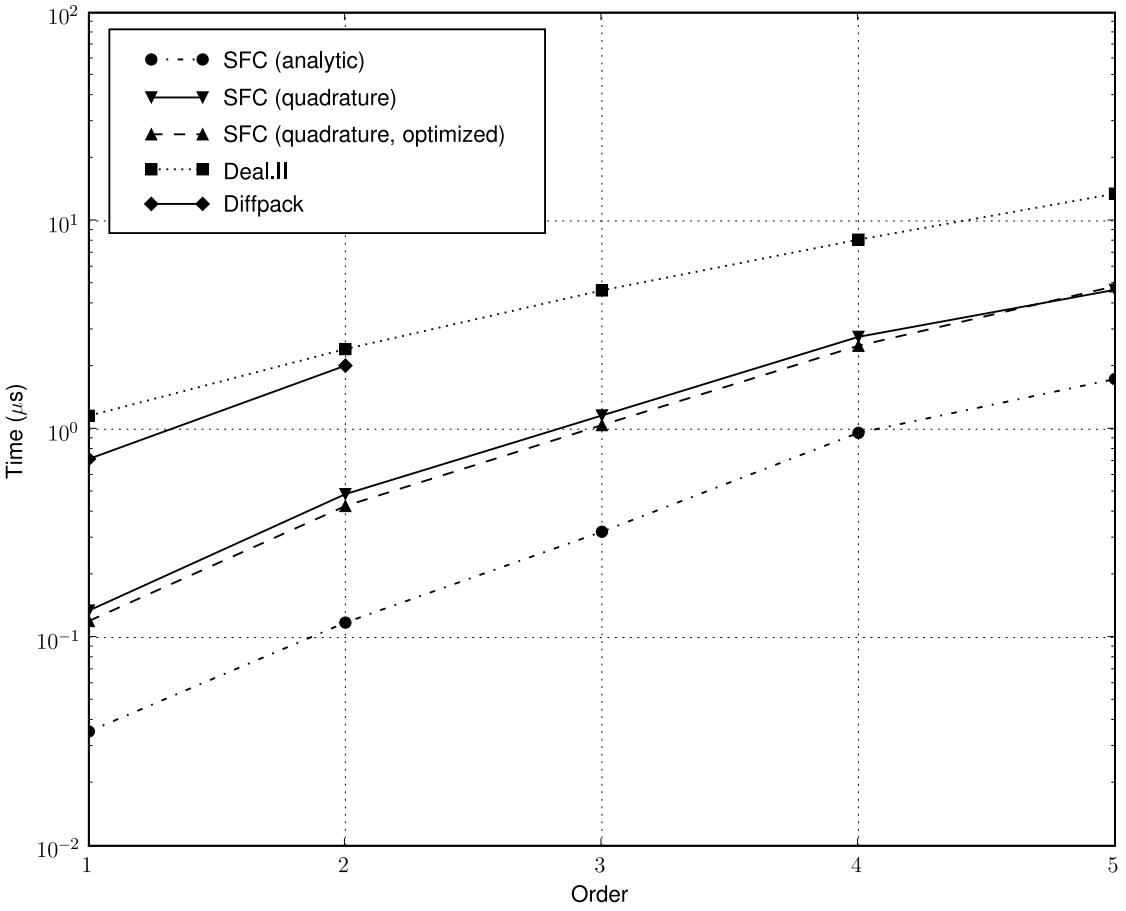

Figure 8: Time to compute the element tensor of the rhs_vector form on quadrilateral elements, in μs

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 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.

	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 4: 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 5: Time to compute the element tensor of the stiffness form for each order respectively on quadrilateral and hexahedron elements.

4.5 Example: Convection Jacobian Matrix

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

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

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.

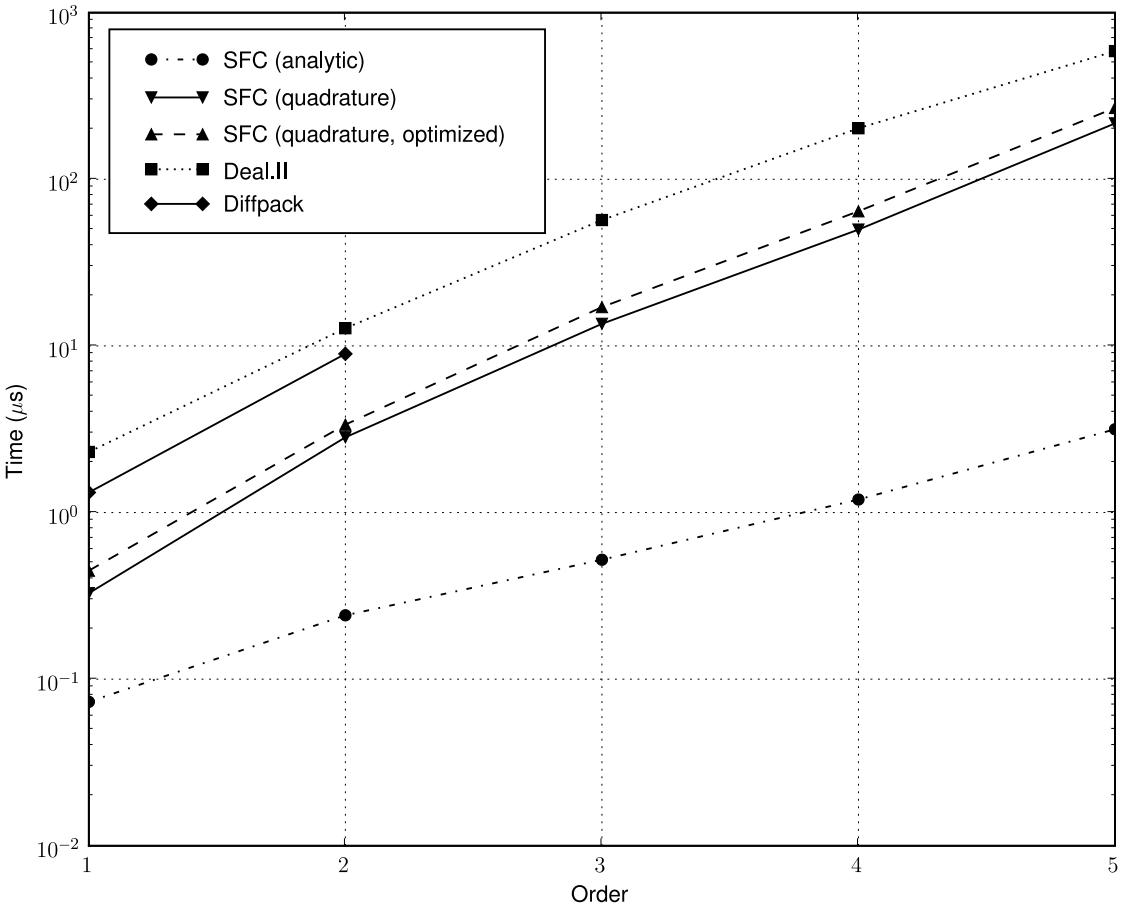

Figure 9: Time to compute the element tensor of the stiffness form on quadrilateral elements, in μs

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.

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

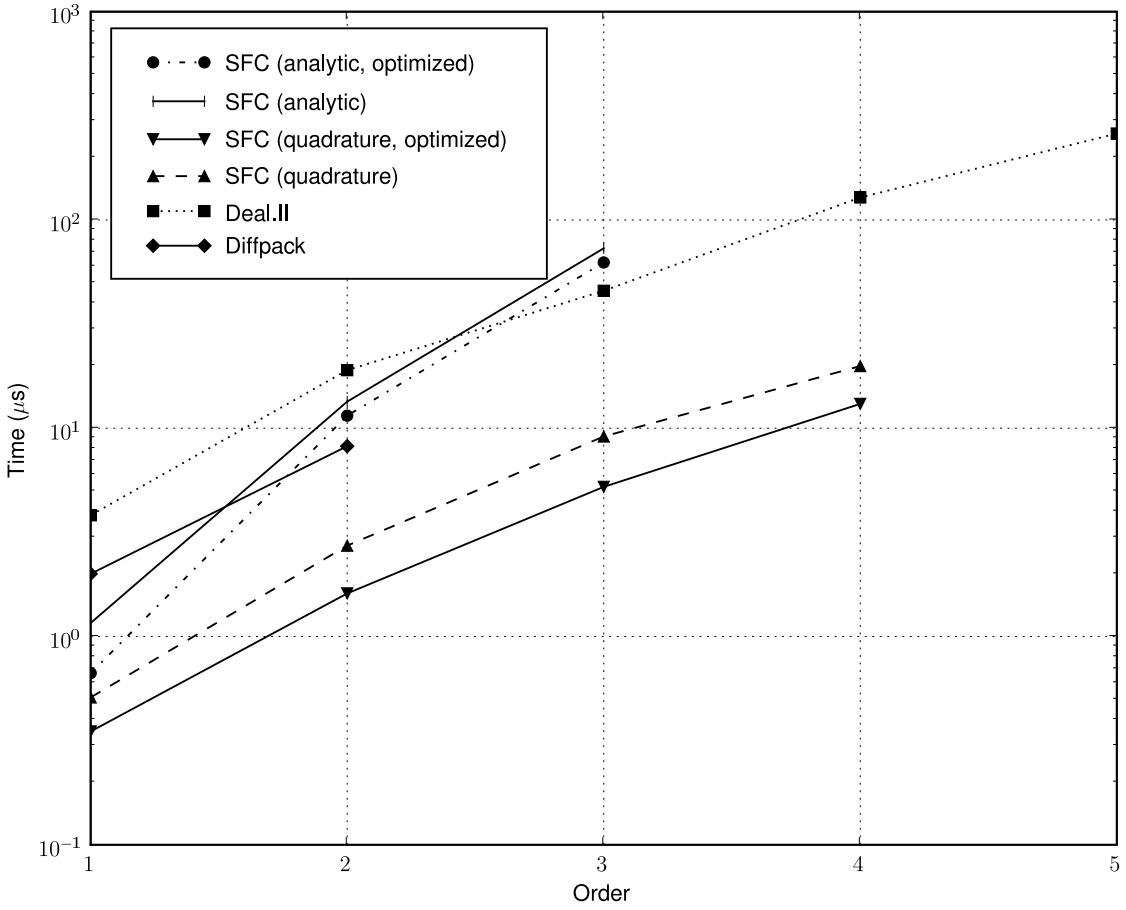

Figure 10: Time to compute the element tensor of the convection_vector form on quadrilateral elements, in μs

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[10].

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

[10]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.

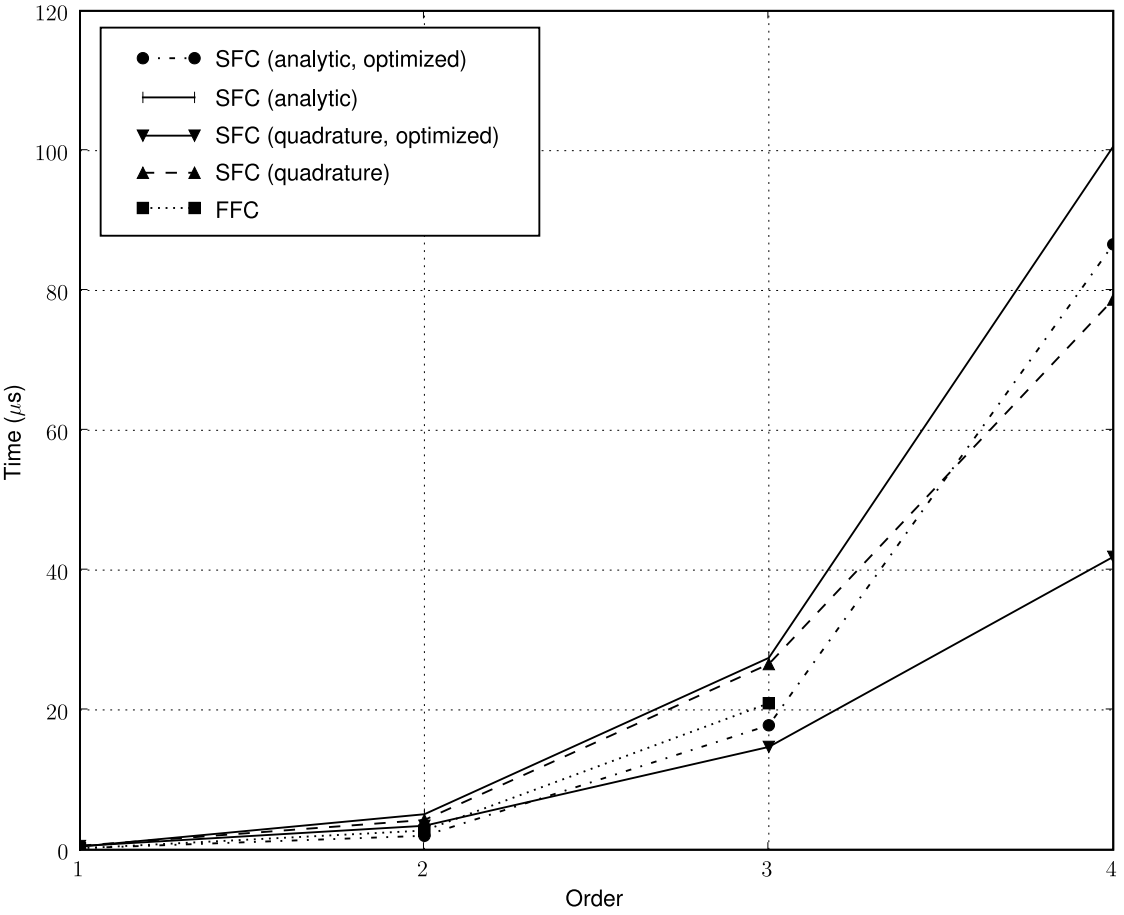

Figure 11: Time to compute the element tensor of the convection_jacobi form on triangle elements, in μs

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.

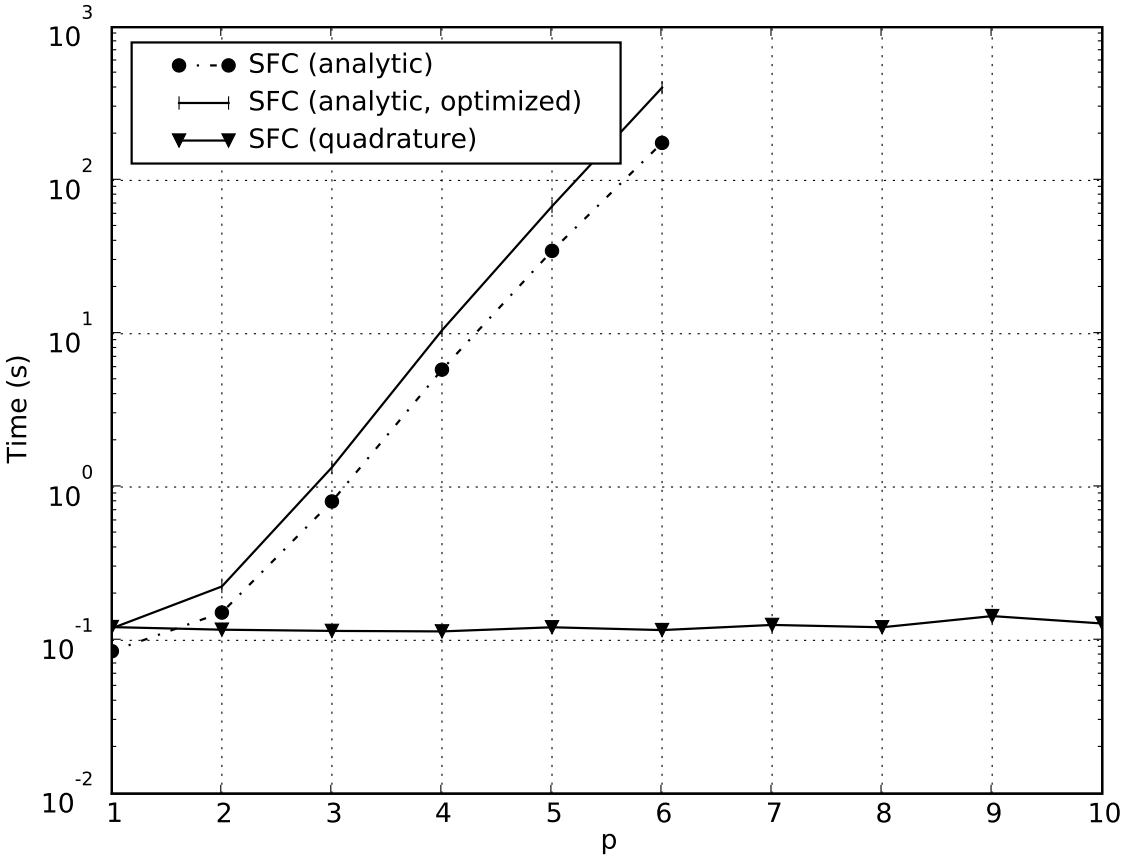

Figure 12: Time to generate code for the power functional on quadratic tetrahedron elements, in s

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 [4], 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 optimalizations. 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

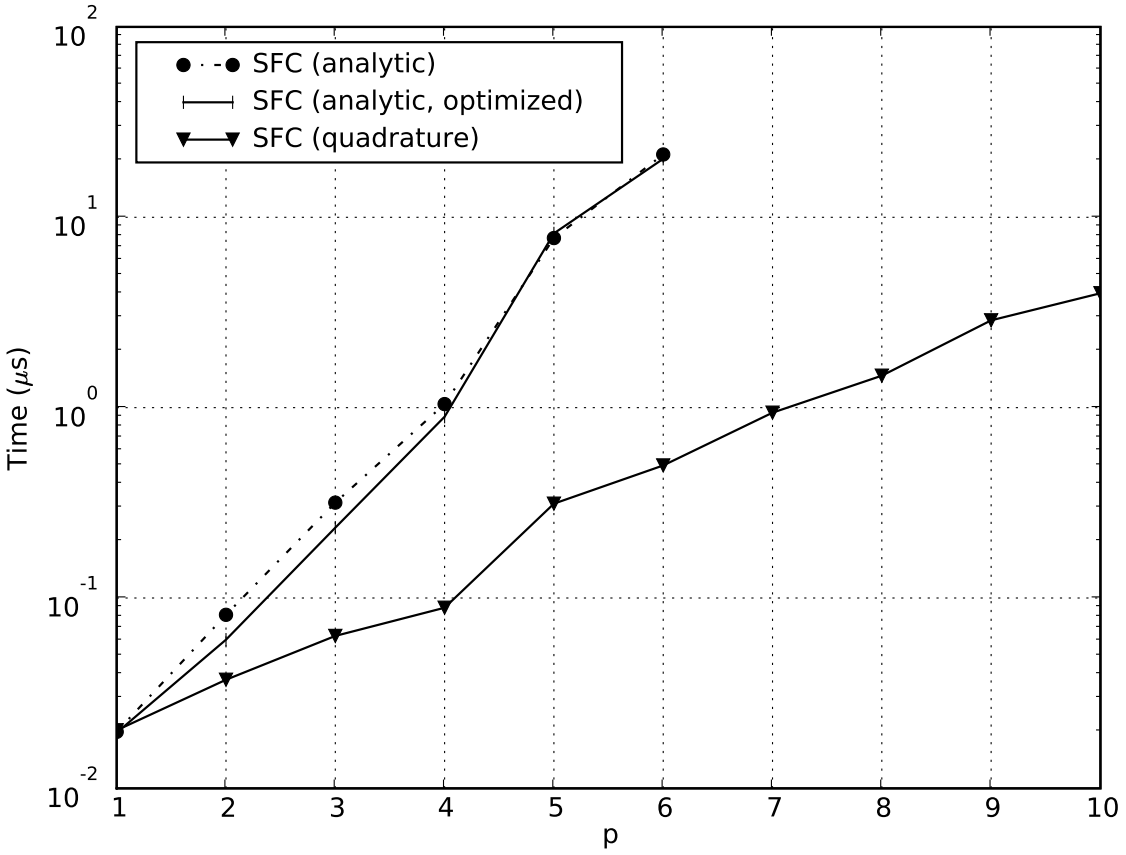

Figure 13: Time to compute the power functional on quadratic tetrahedron elements, in μs

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 3, 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 [4]. 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) [25, 14, 37] 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.

Bibliography

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

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

[3] M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen, *Unified Framework for Finite Element Assembly*, International Journal of Computational Science and Engineering, (2009). Accepted for publication. Preprint: `http://simula.no/research/scientific/publications/Simula.SC.96`.

[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*, Math. Comp., 76 (2007), pp. 1699–1723.

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

[7] ——, `deal.II` *Differential Equations Analysis Library, Technical Reference*, 2007. URL: `http://www.dealii.org`.

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

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

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

[11] 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.

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

[13] FEniCS, *FEniCS project*. URL: `http//www.fenics.org/`, 2008.

[14] A. Griewank, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, 1989, pp. 83–108.

[15] J. Hoffman, J. Jansson, A. Logg, and G. N. Wells, *DOLFIN*, 2008. http://www.fenics.org/dolfin/.

[16] ——, *DOLFIN User Manual*, 2008. URL: http://www.fenics.org/dolfin/.

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

[18] R. C. Kirby, M. G. Knepley, A. Logg, and L. R. Scott, *Optimizing the evaluation of finite element matrices*, SIAM J. Sci. Comput., 27 (2005), pp. 741–758.

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

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

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

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

[23] A. Logg, *FFC user manual*, 2008. URL: http//www.fenics.org/ffc/.

[24] A. Logg et al., *FFC*. URL: http//www.fenics.org/ffc/, 2008.

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

[26] ——, *Sundance*, 2006. URL: http://software.sandia.gov/sundance/.

[27] 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.

[28] K.-A. Mardal and M. Westlie, *Instant*. URL: http//www.fenics.org/instant, 2008.

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

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

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

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

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

[34] 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.

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

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

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

Paper IV: The Unified Form Language

The Unified Form Language

M. S. Alnæs [1,2]

[1] Center for Biomedical Computing, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway
[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

The Unified Form Language – UFL [1, 4] – is a domain specific language for the declaration of finite element discretizations of variational forms and functionals. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation.

The FEniCS project [16, 17, 27] provides a framework for building applications for solving partial differential equations (PDEs). UFL is one of the core components of this framework. It defines the language you *express* your PDEs in. It is the input language and front-end of the form compilers FFC [22, 23, 28, 29, 36, 37] and SFC [7, 8]. The UFL implementation provides algorithms that the form compilers can use to simplify the compilation process. The output from these form compilers is UFC [2, 3, 5] conforming C++ [43] code. This code can be used with the C++ library DOLFIN[11] [30, 31, 32] to efficiently assemble linear systems and compute solution to PDEs.

The combination of domain specific languages and symbolic computing with finite element methods has been pursued from other angles in several other projects. Sundance [33, 34, 35] implements a symbolic engine directly in C++ to define variational forms, and has support for automatic differentiation. The Life [39, 40] project uses a domain specific language embedded in C++, based on expression template techniques to specify variational forms. SfePy [13] uses SymPy as a symbolic engine, extending it with finite element methods. GetDP [14, 15] is another project using a domain specific language for variational forms. The Mathematica package AceGen [24, 25] uses the symbolic capabilities of Mathematica to generate efficient code for finite element methods. All these packages have in common a focus on high level descriptions of partial differential equations to achieve higher human efficiency in the development of simulation software.

UFL almost resembles a library for symbolic computing, but its scope, goals and priorities are different from generic symbolic computing projects such as GiNaC [9, 10], swiginac [42] and SymPy [46]. Intended as a domain specific language and form compiler frontend, UFL is not suitable for large scale symbolic computing.

This chapter is intended both for the FEniCS user who wants to learn how to express her equations, and for other FEniCS developers and technical users who wants to know how UFL works on the inside. Therefore, the sections of this chapter are organized with an increasing amount of technical details. Sections 1-5 give an overview of the language

[11] Note that in PyDOLFIN, some parts of UFL is wrapped to blend in with other software components and make the compilation process hidden from the user. This is not discussed here.

as seen by the end-user and is intended for all audiences. Sections 6-9 explain the design of the implementation and dive into some implementation details. Many details of the language has to be omitted in a text such as this, and we refer to the UFL manual [1] for a more thorough description. Note that this chapter refers to UFL version 0.3, and both the user interface and the implementation may change in future versions.

Starting with a brief overview, we mention the main design goals for UFL and show an example implementation of a non-trivial PDE in Section 1. Next we will look at how to define finite element spaces in Section 2, followed by the overall structure of forms and their declaration in Section 3. The main part of the language is concerned with defining expressions from a set of data types and operators, which are discussed in Section 4. Operators applying to entire forms is the topic of Section 5.

The technical part of the chapter begins with Section 6 which discusses the representation of expressions. Building on the notation and data structures defined there, how to compute derivatives is discussed in Section 7. Some central internal algorithms and key issues in their implementation are discussed in Section 8. Implementation details, some of which are specific to the programming language Python [45], is the topic of Section 9. Finally, Section 10 discusses future prospects of the UFL project.

1 Overview

1.1 Design goals

UFL is a unification, refinement and reimplementation of the form languages used in previous versions of FFC and SFC. The development of this language has been motivated by several factors, the most important being:

- A richer form language, especially for expressing nonlinear PDEs.
- Automatic differentiation of expressions and forms.
- Improving the performance of the form compiler technology to handle more complicated equations efficiently.

UFL fulfils all these requirements, and by this it represents a major step forward in the capabilities of the FEniCS project.

Tensor algebra and index notation support is modeled after the FFC form language and generalized further. Several nonlinear operators and functions which only SFC supported before have been included in the language. Differentiation of expressions and forms has become an integrated part of the language, and is much easier to use than the way these features were implemented in SFC before. In summary, UFL combines the best of FFC and SFC in one unified form language and adds additional capabilities.

The efficiency of code generated by the new generation of form compilers based on UFL has been verified to match previous form compiler benchmarks [6, 37]. The form compilation process is now fast enough to blend into the regular application build process. Complicated forms that previously required too much memory to compile, or took tens of minutes or even hours to compile, now compiles in seconds with both SFC and FFC.

1.2 Motivational example

One major motivating example during the initial development of UFL has been the equations for elasticity with large deformations. In particular, models of biological tissue use complicated hyperelastic constitutive laws with anisotropies and strong nonlinearities. To implement these equations with FEniCS, all three design goals listed above had to be addressed. Below, one version of the hyperelasticity equations and their corresponding UFL implementation is shown. Keep in mind that this is only intended as an illustration of the close correspondence between the form language and the natural formulation of the equations. The meaning of equations is not necessary for the reader to understand. Note that many other examples are distributed together with UFL.

In the formulation of the hyperelasticity equations presented here, the unknown function is the displacement vector field $\mathbf{u}$. The material coefficients c_1 and c_2 are scalar constants. The second Piola-Kirchoff stress tensor $\mathbf{S}$ is computed from the strain energy function $W(\mathbf{C})$. W defines the constitutive law, here a simple Mooney-Rivlin law. The equations relating the displacement and stresses read:

$$\begin{aligned}
\mathbf{F} &= \mathbf{I} + (\nabla \mathbf{u})^T, \\
\mathbf{C} &= \mathbf{F}^T \mathbf{F}, \\
I_C &= \mathrm{tr}(\mathbf{C}), \\
II_C &= \frac{1}{2}(\mathrm{tr}(\mathbf{C})^2 - \mathrm{tr}(\mathbf{C}\mathbf{C})), \\
W &= c_1(I_C - 3) + c_2(II_C - 3), \\
\mathbf{S} &= 2\frac{\partial W}{\partial \mathbf{C}}, \\
\mathbf{P} &= \mathbf{F}\mathbf{S}.
\end{aligned} \tag{1}$$

Approximating the displacement field as $\mathbf{u} = \sum_k u_k \boldsymbol{\phi}_k^1$, the weak forms of the equations are as follows (ignoring boundary conditions):

$$L(\boldsymbol{\phi}^0; \mathbf{u}, c_1, c_2) = \int_\Omega \mathbf{P} : (\nabla \boldsymbol{\phi}^0)^T \,\mathrm{d}x, \tag{2}$$

$$a(\boldsymbol{\phi}^0, \boldsymbol{\phi}_k^1; \mathbf{u}, c_1, c_2) = \frac{\partial L}{\partial u_k}. \tag{3}$$

Figure 1.2 shows an implementation of these equations in UFL. Notice the close relation between the mathematical notation and the UFL source code. In particular, note the automated differentiation of both the constitutive law and the residual equation. This means a new material law can be implemented by simply changing W, the rest is automatic. In the following sections, the notation, definitions and operators used in this implementation are explained.

```
# Finite element spaces
cell = tetrahedron
element = VectorElement("CG", cell, 1)

# Form arguments
phi0 = TestFunction(element)
phi1 = TrialFunction(element)
u = Function(element)
c1 = Constant(cell)
c2 = Constant(cell)

# Deformation gradient Fij = dXi/dxj
I = Identity(cell.d)
F = I + grad(u).T

# Right Cauchy-Green strain tensor C with invariants
C = variable(F.T*F)
I_C = tr(C)
II_C = (I_C**2 - tr(C*C))/2

# Mooney-Rivlin constitutive law
W = c1*(I_C-3) + c2*(II_C-3)

# Second Piola-Kirchoff stress tensor
S = 2*diff(W, C)

# Weak forms
L = inner(F*S, grad(phi0).T)*dx
a = derivative(L, u, phi1)
```

Figure 1: UFL implementation of hyperelasticity equations with a Mooney-Rivlin material law.

2 Defining finite element spaces

A polygonal cell is defined by a basic shape and a degree[12], and is declared

```
cell = Cell(shape, degree)
```

UFL defines a set of valid polygonal cell shapes: "interval", "triangle", "tetrahedron", "quadrilateral", and "hexahedron". Linear cells of all basic shapes are predefined and can be used instead by writing

```
cell = tetrahedron
```

In the rest of this chapter, a variable name `cell` will be used where any cell is a valid argument, to make the examples dimension independent wherever possible.

UFL defines syntax for *declaring* finite element spaces, but does not know anything about the actual polynomial basis or degrees of freedom. The polynomial basis is selected implicitly by choosing among predefined basic element families and providing a polynomial degree, but UFL only assumes that there *exists* a basis with a fixed ordering for each finite element space V_h, i.e.

$$V_h = \text{span}\,\{\phi_j\}_{j=1}^{n}\,. \tag{4}$$

Basic scalar elements can be combined to form vector elements or tensor elements, and elements can easily be combined in arbitrary mixed element hierarchies.

The set of predefined[13] element family names in UFL includes "Lagrange" (short name "CG"), representing scalar Lagrange finite elements (continuous piecewise polynomial functions), "Discontinuous Lagrange" (short name "DG"), representing scalar discontinuous Lagrange finite elements (discontinuous piecewise polynomial functions), and a range of other families that can be found in the manual. Each family name has an associated short name for convenience. To print all valid families to screen from Python, call `show_elements()`.

The syntax for declaring elements is best explained with some examples.

```
cell = tetrahedron

P = FiniteElement("Lagrange", cell, 1)
V = VectorElement("Lagrange", cell, 2)
T = TensorElement("DG", cell, 0, symmetry=True)

TH = V + P
ME = MixedElement(T, V, P)
```

[12]Note that at the time of writing, the other components of FEniCS does not yet handle higher degree cells.

[13]Form compilers can register additional element families.

In the first line a polygonal cell is selected from the set of predefined linear cells. Then a scalar linear Lagrange element `P` is declared, as well as a quadratic vector Lagrange element `V`. Next a symmetric rank 2 tensor element `T` is defined, which is also piecewise constant on each cell. The code proceeds to declare a mixed element `TH`, which combines the quadratic vector element `V` and the linear scalar element `P`. This element is known as the Taylor-Hood element. Finally another mixed element with three sub elements is declared. Note that writing `T + V + P` would not result in a mixed element with three direct sub elements, but rather `MixedElement(MixedElement(T + V), P)`.

3 Defining forms

Consider Poisson's equation with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$,

$$a(v, u; w) = \int_\Omega w \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{5}$$

$$L(v; f, g, h) = \int_\Omega f v \,\mathrm{d}x + \int_{\partial\Omega_0} g^2 v \,\mathrm{d}s + \int_{\partial\Omega_1} h v \,\mathrm{d}s. \tag{6}$$

These forms can be expressed in UFL as

```
a = w*dot(grad(u), grad(v))*dx
L = f*v*dx + g**2*v*ds(0) + h*v*ds(1)
```

where multiplication by the measures `dx`, `ds(0)` and `ds(1)` represent the integrals $\int_{\Omega_0}(\cdot)\,\mathrm{d}x$, $\int_{\partial\Omega_0}(\cdot)\,\mathrm{d}s$, and $\int_{\partial\Omega_1}(\cdot)\,\mathrm{d}s$ respectively.

Forms expressed in UFL are intended for finite element discretization followed by compilation to efficient code for computing the element tensor. Considering the above example, the bilinear form a with one coefficient function w is assumed to be evaluated at a later point with a range of basis functions and the coefficient function fixed, that is

$$V_h^1 = \mathrm{span}\left\{\phi_k^1\right\}, \quad V_h^2 = \mathrm{span}\left\{\phi_k^2\right\}, \quad V_h^3 = \mathrm{span}\left\{\phi_k^3\right\}, \tag{7}$$

$$w = \sum_{k=1}^{|V_h^2|} w_k \phi_k^3, \quad \{w_k\} \text{ given}, \tag{8}$$

$$A_{ij} = a(\phi_i^1, \phi_j^2; w), \quad i = 1, \ldots, |V_h^1|, \quad j = 1, \ldots, |V_h^2|. \tag{9}$$

In general, UFL is designed to express forms of the following generalized form:

$$a(\phi^1, \ldots, \phi^r; w^1, \ldots, w^n) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \,\mathrm{d}x + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e \,\mathrm{d}s + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i \,\mathrm{d}S. \tag{10}$$

Most of this chapter deals with ways to define the integrand expressions I_k^c, I_k^e and I_k^i. The rest of the notation will be explained below.

The form arguments are divided in two groups, the basis functions $\phi^1, \ldots, \phi^r$ and the coefficient functions $w^1, \ldots, w^n$. All $\{\phi^k\}$ and $\{w^k\}$ are functions in some discrete

function space with a basis. Note that the actual basis functions $\{\phi_j^k\}$ and the coefficients $\{w_k\}$ are never known to UFL, but we assume that the ordering of the basis for each finite element space is fixed. A fixed ordering only matters when differentiating forms, explained in Section 7.

Each term of a valid form expression must be a scalar-valued expression integrated exactly once, and they must be linear in $\{\phi^k\}$. Any term may have nonlinear dependencies on coefficient functions. A form with one or two basis function arguments ($r = 1, 2$) is called a linear or bilinear form respectively, ignoring its dependency on coefficient functions. These will be assembled to vectors and matrices when used in an application. A form depending only on coefficient functions ($r = 0$) is called a functional, since it will be assembled to a real number.

The entire domain is denoted Ω, the external boundary is denoted $\partial\Omega$, while the set of interior facets of the triangulation is denoted Γ. Sub domains are marked with a suffix, e.g., $\Omega_k \subset \Omega$. As mentioned above, integration is expressed by multiplication with a measure, and UFL defines the measures `dx`, `ds` and `dS`. In summary, there are three kinds of integrals with corresponding UFL representations

- $\int_{\Omega_k}(\cdot)\,\mathrm{d}x \leftrightarrow$ `(·)*dx(k)`, called a *cell integral*,
- $\int_{\partial\Omega_k}(\cdot)\,\mathrm{d}s \leftrightarrow$ `(·)*ds(k)`, called an *exterior facet integral*,
- $\int_{\Gamma_k}(\cdot)\,\mathrm{d}S \leftrightarrow$ `(·)*dS(k)`, called an *interior facet integral*,

Defining a different quadrature order for each term in a form can be achieved by attaching meta data to measure objects, e.g.,

```
dx02 = dx(0, { "integration_order": 2 })
dx14 = dx(1, { "integration_order": 4 })
dx12 = dx(1, { "integration_order": 2 })
L = f*v*dx02 + g*v*dx14 + h*v*dx12
```

Meta data can also be used to override other form compiler specific options separately for each term. For more details on this feature see the manuals of UFL and the form compilers.

4 Defining expressions

Most of UFL deals with how to declare expressions such as the integrand expressions in Equation 10. The most basic expressions are terminal values, which do not depend on other expressions. Other expressions are called operators, which are discussed in sections 4.2-4.5.

Terminal value types in UFL include form arguments (which is the topic of Section 4.1), geometric quantities, and literal constants. Among the literal constants are scalar integer and floating point values, as well as the d by d identity matrix `I = Identity(d)`. To get unit vectors, simply use rows or columns of the identity matrix, e.g., `e0 = I[0,:]`. Similarly, `I[i,j]` represents the Dirac delta function δ_{ij} (see Section 4.2 for details on

index notation). Available geometric values are the spatial coordinates $\mathbf{x} \leftrightarrow$ `cell.x` and the facet normal $\mathbf{n} \leftrightarrow$ `cell.n`. The geometric dimension is available as `cell.d`.

4.1 Form arguments

Basis functions and coefficient functions are represented by instances of `BasisFunction` and `Function` respectively. The ordering of the arguments to a form is decided by the order in which the form arguments were declared in the UFL code. Each basis function argument represents any function in the basis of its finite element space

$$\phi^j \in \{\phi_k^j\}, \quad V_h^j = \operatorname{span}\left\{\phi_k^j\right\}. \tag{11}$$

with the intention that the form is later evaluated for all ϕ_k such as in equation (9). Each coefficient function w represents a discrete function in some finite element space V_h; it is usually a sum of basis functions $\phi_k \in V_h$ with coefficients w_k

$$w = \sum_{k=1}^{|V_h|} w_k \phi_k. \tag{12}$$

The exception is coefficient functions that can only be evaluated pointwise, which are declared with a finite element with family "Quadrature". Basis functions are declared for an arbitrary element as in the following manner:

```
phi = BasisFunction(element)
v = TestFunction(element)
u = TrialFunction(element)
```

By using `TestFunction` and `TrialFunction` in declarations instead of `BasisFunction` you can ignore their relative ordering. The only time `BasisFunction` is needed is for forms of arity $r > 2$.

Coefficient functions are declared similarly for an arbitrary element, and shorthand notation exists for declaring piecewise constant functions:

```
w = Function(element)
c = Constant(cell)
v = VectorConstant(cell)
M = TensorConstant(cell)
```

If a form argument u in a mixed finite element space $V_h = V_h^0 \times V_h^1$ is desired, but the form is more easily expressed using sub functions $u_0 \in V_h^0$ and $u_1 \in V_h^1$, you can split the mixed function or basis function into its sub functions in a generic way using `split`:

```
V = V0 + V1
u = Function(V)
u0, u1 = split(u)
```

The `split` function can handle arbitrary mixed elements. Alternatively, a handy shorthand notation for argument declaration followed by `split` is

```
v0, v1 = TestFunctions(V)
u0, u1 = TrialFunctions(V)
f0, f1 = Functions(V)
```

4.2 Index notation

UFL allows working with tensor expressions of arbitrary rank, using both tensor algebra and index notation. A basic familiarity with tensor algebra and index notation is assumed. The focus here is on how index notation is expressed in UFL.

Assuming a standard orthonormal Euclidean basis $\langle \mathbf{e}_k \rangle_{k=1}^{d}$ for $\mathbb{R}^d$, a vector can be expressed with its scalar components in this basis. Tensors of rank two can be expressed using their scalar components in a dyadic basis $\{\mathbf{e}_i \otimes \mathbf{e}_j\}_{i,j=1}^{d}$. Arbitrary rank tensors can be expressed the same way, as illustrated here.

$$\mathbf{v} = \sum_{k=1}^{d} v_k \mathbf{e}_k, \tag{13}$$

$$\mathbf{A} = \sum_{i=1}^{d} \sum_{j=1}^{d} A_{ij} \mathbf{e}_i \otimes \mathbf{e}_j, \tag{14}$$

$$\mathcal{C} = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k} C_{ijk} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k. \tag{15}$$

Here, $\mathbf{v}$, $\mathbf{A}$ and $\mathcal{C}$ are rank 1, 2 and 3 tensors respectively. Indices are called *free* if they have no assigned value, such as i in v_i, and *fixed* if they have a fixed value such as 1 in v_1. An expression with free indices represents any expression you can get by assigning fixed values to the indices. The expression A_{ij} is scalar valued, and represents any component (i, j) of the tensor $\mathbf{A}$ in the Euclidean basis. When working on paper, it is easy to switch between tensor notation ($\mathbf{A}$) and index notation (A_{ij}) with the knowledge that the tensor and its components are different representations of the same physical quantity. In a programming language, we must express the operations mapping from tensor to scalar components and back explicitly. Mapping from a tensor to its components, for a rank 2 tensor defined as

$$A_{ij} = \mathbf{A} : (\mathbf{e}_i \otimes \mathbf{e}_j), \tag{16}$$

$$\tag{17}$$

is accomplished using indexing with the notation `A[i,j]`. Defining a tensor $\mathbf{A}$ from component values A_{ij} is defined as

$$\mathbf{A} = A_{ij} \mathbf{e}_i \otimes \mathbf{e}_j, \tag{18}$$

and is accomplished using the function `as_vector(Aij, (i,j))`. To illustrate, consider the outer product of two vectors $\mathbf{A} = \mathbf{u} \otimes \mathbf{v} = u_i v_j \mathbf{e}_i \otimes \mathbf{e}_j$, and the corresponding scalar components A_{ij}. One way to implement this is

```
A = outer(u, v)
Aij = A[i, j]
```

Alternatively, the components of $\mathbf{A}$ can be expressed directly using index notation, such as $A_{ij} = u_i v_j$. A_{ij} can then be mapped to $\mathbf{A}$ in the following manner:

```
Aij = v[j]*u[i]
A = as_tensor(Aij, (i, j))
```

These two pairs of lines are mathematically equivalent, and the result of either pair is that the variable `A` represents the tensor $\mathbf{A}$ and the variable `Aij` represents the tensor A_{ij}. Note that free indices have no ordering, so their order of appearance in the expression `v[j]*u[i]` is insignificant. Instead of `as_tensor`, the specialized functions `as_vector` and `as_matrix` can be used. Although a rank two tensor was used for the examples above, the mappings generalize to arbitrary rank tensors.

When indexing expressions, fixed indices can also be used such as in `A[0,1]` which represents a single scalar component. Fixed indices can also be mixed with free indices such as in `A[0,i]`. In addition, slices can be used in place of an index. An example of using slices is `A[0,:]` which is is a vector expression that represents row 0 of `A`. To create new indices, you can either make a single one or make several at once:

```
i = Index()
j, k, l = indices(3)
```

A set of indices `i`, `j`, `k`, `l` and `p`, `q`, `r`, `s` are predefined, and these should suffice for most applications.

If your components are not represented as an expression with free indices, but as separate unrelated scalar expressions, you can build a tensor from them using `as_tensor` and its peers. As an example, lets define a 2D rotation matrix and rotate a vector expression by $\frac{\pi}{2}$:

```
th = pi/2
A = as_matrix([[ cos(th), -sin(th)],
               [ sin(th),  cos(th)]])
u = A*v
```

When indices are repeated in a term, summation over those indices is implied in accordance with the Einstein convention. In particular, indices can be repeated when indexing a tensor of rank two or higher (`A[i,i]`), when differentiating an expression with a free index (`v[i].dx(i)`), or when multiplying two expressions with shared free indices (`u[i]*v[i]`).

$$A_{ii} \equiv \sum_i A_{ii}, \qquad v_i u_i \equiv \sum_i v_i u_i, \qquad v_{i,i} \equiv \sum_i v_{i,i}. \tag{19}$$

An expression `Aij = A[i,j]` is represented internally using the `Indexed` class. `Aij` will reference `A`, keeping the representation of the original tensor expression `A` unchanged. Implicit summation is represented explicitly in the expression tree using the class `IndexSum`. Many algorithms become easier to implement with this explicit representation, since e.g. a `Product` instance can never implicitly represent a sum. More details on representation classes are found in Section 6.

4.3 Algebraic operators and functions

UFL defines a comprehensive set of operators that can be used for composing expressions. The elementary algebraic operators `+`, `-`, `*`, `/` can be used between most UFL expressions with a few limitations. Division requires a scalar expression with no free indices in the denominator. The operands to a sum must have the same shape and set of free indices.

The multiplication operator `*` is valid between two scalars, a scalar and any tensor, a matrix and a vector, and two matrices. Other products could have been defined, but for clarity we use tensor algebra operators and index notation for those rare cases. A product of two expressions with shared free indices implies summation over those indices, see Section 4.2 for more about index notation.

Three often used operators are `dot(a, b)`, `inner(a, b)`, and `outer(a, b)`. The dot product of two tensors of arbitrary rank is the sum over the last index of the first tensor and the first index of the second tensor. Some examples are

$$\mathbf{v} \cdot \mathbf{u} = v_i u_i, \tag{20}$$
$$\mathbf{A} \cdot \mathbf{u} = A_{ij} u_j \mathbf{e}_i, \tag{21}$$
$$\mathbf{A} \cdot \mathbf{B} = A_{ik} B_{kj} \mathbf{e}_i \mathbf{e}_j, \tag{22}$$
$$\mathcal{C} \cdot \mathbf{A} = C_{ijk} A_{kl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_l. \tag{23}$$

The inner product is the sum over all indices, for example

$$\mathbf{v} : \mathbf{u} = v_i u_i, \tag{24}$$
$$\mathbf{A} : \mathbf{B} = A_{ij} B_{ij}, \tag{25}$$
$$\mathcal{C} : \mathcal{D} = C_{ijkl} D_{ijkl}. \tag{26}$$

Some examples of the outer product are

$$\mathbf{v} \otimes \mathbf{u} = v_i u_j \mathbf{e}_i \mathbf{e}_j, \tag{27}$$
$$\mathbf{A} \otimes \mathbf{u} = A_{ij} u_k \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k, \tag{28}$$
$$\mathbf{A} \otimes \mathbf{B} = A_{ij} B_{kl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k \mathbf{e}_l \tag{29}$$

Other common tensor algebra operators are `cross(u,v)`, `transpose(A)` (or `A.T`), `tr(A)`, `det(A)`, `inv(A)`, `cofac(A)`, `dev(A)`, `skew(A)`, and `sym(A)`. Most of these tensor algebra operators expect tensors without free indices. The detailed definitions of these operators are found in the manual.

A set of common elementary functions operating on scalar expressions without free indices are included, in particular `abs(f)`, `pow(f, g)`, `sqrt(f)`, `exp(f)`, `ln(f)`, `sin(f)`, `cos(f)`, and `sign(f)`.

4.4 Differential operators

UFL implements derivatives w.r.t. three different kinds of variables. The most used kind is spatial derivatives. Expressions can also be differentiated w.r.t. arbitrary user defined variables. And the final kind of derivatives are derivatives of a form or functional w.r.t. the coefficients of a `Function`. Form derivatives are explained in Section 5.1.

Note that derivatives are not computed immediately when declared. A discussion of how derivatives are computed is found in Section 7.

Spatial derivatives

Basic spatial derivatives $\frac{\partial f}{\partial x_i}$ can be expressed in two equivalent ways:

```
df = Dx(f, i)
df = f.dx(i)
```

Here, `df` represents the derivative of `f` in the spatial direction x_i. The index `i` can either be an integer, representing differentiation in one fixed spatial direction x_i, or an `Index`, representing differentiation in the direction of a free index. The notation `f.dx(i)` is intended to mirror the index notation $f_{,i}$, which is shorthand for $\frac{\partial f}{\partial x_i}$. Repeated indices imply summation, such that the divergence of a vector can be written $v_{i,i}$, or `v[i].dx(i)`.

Several common compound spatial derivative operators are defined, namely `div`, `grad`, `curl` and `rot` (rot is a synonym for curl). The definition of these operators in UFL follow from the vector of partial derivatives

$$\nabla \equiv \mathbf{e}_k \frac{\partial}{\partial x_k}, \tag{30}$$

and the definition of the dot product, outer product, and cross product. Hence,

$$\text{div}(\mathcal{C}) \equiv \nabla \cdot \mathcal{C}, \tag{31}$$

$$\text{grad}(\mathcal{C}) \equiv \nabla \otimes \mathcal{C}, \tag{32}$$

$$\text{curl}(\mathbf{v}) \equiv \nabla \times \mathbf{v}. \tag{33}$$

Note that there are two common ways to define grad and div. This way of defining these operators correspond to writing the convection term from, e.g., the Navier-Stokes equations as

$$\mathbf{w} \cdot \nabla \mathbf{u} = (\mathbf{w} \cdot \nabla)\mathbf{u} = \mathbf{w} \cdot (\nabla \mathbf{u}) = w_i u_{j,i}, \tag{34}$$

which is expressed in UFL as

```
dot(w, grad(u))
```

Another illustrative example is the anisotropic diffusion term from, e.g., the bidomain equations, which reads

$$(\mathbf{A}\nabla u) \cdot \mathbf{v} = A_{ij} u_{,j} v_i, \tag{35}$$

and is expressed in UFL as

```
dot(A*grad(u), v)
```

In other words, the divergence sums over the *first* index of its operand, and the gradient *prepends* an axis to the tensor shape of its operand. The above definition of curl is only valid for 3D vector expressions. For 2D vector and scalar expressions the definitions are:

$$\text{curl}(\mathbf{u}) \equiv u_{1,0} - u_{0,1}, \tag{36}$$
$$\text{curl}(f) \equiv f_{,1}\mathbf{e}_0 - f_{,0}\mathbf{e}_1. \tag{37}$$

User defined variables

The second kind of differentiation variables are user-defined variables, which can represent arbitrary expressions. Automating derivatives w.r.t. arbitrary quantities is useful for several tasks, from differentiation of material laws to computing sensitivities. An arbitrary expression g can be assigned to a variable v. An expression f defined as a function of v can be differentiated f w.r.t. v:

$$v = g, \tag{38}$$
$$f = f(v), \tag{39}$$
$$h(v) = \frac{\partial f(v)}{\partial v}. \tag{40}$$

Setting $g = sin(x_0)$ and $f = e^{v^2}$, gives $h = 2ve^{v^2} = 2\sin(x_0)e^{\sin^2(x_0)}$, which can be implemented as follows:

```
g = sin(cell.x[0])
v = variable(g)
f = exp(v**2)
h = diff(f, v)
```

Try running this code in a Python session and print the expressions. The result is

```
>>> print v
var0(sin((x)[0]))
>>> print h
d/d[var0(sin((x)[0]))] (exp((var0(sin((x)[0]))) ** 2))
```

Note that the variable has a label 0 ("var0"), and that `h` still represents the abstract derivative. Section 7 explains how derivatives are computed.

4.5 Other operators

A few operators are provided for the implementation of discontinuous Galerkin methods. The basic concept is restricting an expression to the positive or negative side of an

interior facet, which is expressed simply as `v('+')` or `v('-')` respectively. On top of this, the operators `avg` and `jump` are implemented, defined as

$$\text{avg}(v) = \frac{1}{2}(v^+ + v^-), \tag{41}$$

$$\text{jump}(v) = v^+ - v^-. \tag{42}$$

These operators can only be used when integrating over the interior facets (`*dS`).

The only control flow construct included in UFL is conditional expressions. A conditional expression takes on one of two values depending on the result of a boolean logic expression. The syntax for this is

```
f = conditional(condition, true_value, false_value)
```

which is interpreted as

$$f = \begin{cases} t, & \text{if condition is true,} \\ f, & \text{otherwise.} \end{cases} \tag{43}$$

The condition can be one of

- `lt(a, b)` $\leftrightarrow (a < b)$
- `le(a, b)` $\leftrightarrow (a \leq b)$
- `eq(a, b)` $\leftrightarrow (a = b)$
- `gt(a, b)` $\leftrightarrow (a > b)$
- `ge(a, b)` $\leftrightarrow (a \geq b)$
- `ne(a, b)` $\leftrightarrow (a \neq b)$

5 Form operators

Once you have defined some forms, there are several ways to compute related forms from them. While operators in the previous section are used to define expressions, the operators discussed in this section are applied to forms, producing new forms. Form operators can both make form definitions more compact and reduce the chances of bugs since changes in the original form will propagate to forms computed from it automatically. These form operators can be combined arbitrarily; given a semi-linear form only a few lines are needed to compute the action of the adjoint of the Jacobi. Since these computations are done prior to processing by the form compilers, there is no overhead at run-time.

5.1 Differentiating forms

The form operator `derivative` declares the derivative of a form w.r.t. coefficients of a discrete function (`Function`). This functionality can be used for example to linearize your nonlinear residual equation (linear form) automatically for use with the Newton-Raphson method. It can also be applied multiple times, which is useful to derive a linear system from a convex functional, in order to find the function that minimizes

the functional. For non-trivial equations such expressions can be tedious to calculate by hand. Other areas in which this feature can be useful include optimal control and inverse methods, as well as sensitivity analysis.

In its simplest form, the declaration of the derivative of a form `L` w.r.t. the coefficients of a function `w` reads

```
a = derivative(L, w, u)
```

The form `a` depends on an additional basis function argument `u`, which must be in the same finite element space as the function `w`. If the last argument is omitted, a new basis function argument is created.

Let us step through an example of how to apply `derivative` twice to a functional to derive a linear system. In the following, V_h is a finite element space with some basis, w is a function in V_h, and f is a functional we want to minimize. Derived from f is a linear form F, and a bilinear form J.

$$V_h = \operatorname{span}\{\phi_k\}, \tag{44}$$

$$w(x) = \sum_{k=1}^{|V_h|} w_k \phi_k(x), \tag{45}$$

$$f : V_h \to \mathbb{R}, \tag{46}$$

$$F(\phi_i; w) = \frac{\partial}{\partial w_i} f(w), \tag{47}$$

$$J(\phi_i, \phi_j; w) = \frac{\partial}{\partial w_j} F(\phi_i; w). \tag{48}$$

For a concrete functional $f(w) = \int_\Omega \frac{1}{2} w^2 \,\mathrm{d}x$, we can implement this as

```
v = TestFunction(element)
u = TrialFunction(element)
w = Function(element)
f = 0.5 * w**2 * dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

This code declares two forms `F` and `J`. The linear form `F` represents the standard load vector `w*v*dx` and the bilinear form `J` represents the mass matrix `u*v*dx`.

Derivatives can also be defined w.r.t. coefficients of a function in a mixed finite element space. Consider the Harmonic map equations derived from the functional

$$f(\mathbf{x}, \lambda) = \int_\Omega \nabla \mathbf{x} : \nabla \mathbf{x} + \lambda \mathbf{x} \cdot \mathbf{x} \,\mathrm{d}x, \tag{49}$$

where $\mathbf{x}$ is a function in a vector finite element space V_h^d and λ is a function in a scalar finite element space V_h. The linear and bilinear forms derived from the functional in Equation 49 have basis function arguments in the mixed space $V_h^d + V_h$. The implementation of these forms with automatic linearization reads

```
Vx = VectorElement("CG", triangle, 1)
Vy = FiniteElement("CG", triangle, 1)
u = Function(Vx + Vy)
x, y = split(u)
f = inner(grad(x), grad(x))*dx + y*dot(x,x)*dx
F = derivative(f, u)
J = derivative(F, u)
```

Note that the functional is expressed in terms of the subfunctions `x` and `y`, while the argument to `derivative` must be the single mixed function `u`. In this example the basis function arguments to `derivative` are omitted and thus provided automatically in the right function spaces.

Note that in computing derivatives of forms, we have assumed that

$$\frac{\partial}{\partial w_k} \int_\Omega I \,\mathrm{d}x = \int_\Omega \frac{\partial}{\partial w_k} I \,\mathrm{d}x, \tag{50}$$

or in particular that the domain Ω is independent of w. Furthermore, note that there is no restriction on the choice of element in this framework, in particular arbitrary mixed elements are supported.

5.2 Adjoint

Another form operator is the adjoint a^* of a bilinear form a, defined as $a^*(u, v) = a(v, u)$, which is similar to taking the transpose of the assembled sparse matrix. In UFL this is implemented simply by swapping the test and trial functions, and can be written:

```
a = inner(M*grad(u), grad(v))*dx
ad = adjoint(a)
```

which corresponds to

$$a(M; v, u) = \int_\Omega (\mathbf{M}\nabla\mathbf{u}) : \nabla\mathbf{v} \,\mathrm{d}x = \int_\Omega M_{ik} u_{j,k} v_{j,i} \,\mathrm{d}x, \tag{51}$$

$$a^*(M; v, u) = a(M; u, v) = \int_\Omega (\mathbf{M}\nabla\mathbf{v}) : \nabla\mathbf{u} \,\mathrm{d}x. \tag{52}$$

This automatic transformation is particularly useful if we need the adjoint of nonsymmetric bilinear forms computed using `derivative`, since the explicit expressions for a are not at hand. Several of the form operators below are most useful when used in conjunction with `derivative`.

5.3 Replacing functions

Evaluating a form with new definitions of form arguments can be done by replacing terminal objects with other values. Lets say you have defined a form `L` that depends on

some functions `f` and `g`. You can then specialize the form by replacing these functions with other functions or fixed values, such as

$$L(v; f, g) = \int_\Omega (f^2/(2g))v \, \mathrm{d}x, \tag{53}$$

$$L_2(v; f, g) = L(v; g, 3) = \int_\Omega (g^2/6)v \, \mathrm{d}x. \tag{54}$$

This feature is implemented with `replace`, as illustrated in this case:

```
L = f**2 / (2*g) * v * dx
L2 = replace(L, { f: g, g: 3})
L3 = g**2 / 6 * v * dx
```

Here `L2` and `L3` represents exactly the same form. Since they depend only on `g`, the code generated for these forms can be more efficient.

5.4 Action

Sparse matrix-vector multiplication is an important operation in PDE solver applications. In some cases the matrix is not needed explicitly, only the action of the matrix on a vector, the result of the matrix-vector multiplication. You can assemble the action of the matrix on a vector directly by defining a linear form for the action of a bilinear form on a function, simply writing `L = action(a, w)` or `L = a*w`, with `a` any bilinear form and `w` being any `Function` defined on the same finite element as the trial function in `a`.

5.5 Splitting a system

If you prefer to write your PDEs with all terms on one side such as

$$a(v, u) - L(v) = 0, \tag{55}$$

you can declare forms with both linear and bilinear terms and split the equations afterwards:

```
pde = u*v*dx - f*v*dx
a, L = system(pde)
```

Here `system` is used to split the PDE into its bilinear and linear parts. Alternatively, `lhs` and `rhs` can be used to obtain the two parts separately.

5.6 Computing the sensitivity of a function

If you have found the solution u to Equation (55), and u depends on some constant scalar value c, you can compute the sensitivity of u w.r.t. changes in c. If u is represented

by a coefficient vector x that is the solution to the algebraic linear system $Ax = b$, the coefficients of $\frac{\partial u}{\partial c}$ are $\frac{\partial x}{\partial c}$. Applying $\frac{\partial}{\partial c}$ to $Ax = b$ and using the chain rule, we can write

$$A\frac{\partial x}{\partial c} = \frac{\partial b}{\partial c} - \frac{\partial A}{\partial c}x, \tag{56}$$

and thus $\frac{\partial x}{\partial c}$ can be found by solving the same algebraic linear system used to compute x, only with a different right hand side. The linear form corresponding to the right hand side of Equation (56) can be written

```
u = Function(element)
sL = diff(L, c) - action(diff(a, c), u)
```

or you can use the equivalent form transformation

```
sL = sensitivity_rhs(a, u, L, c)
```

Note that the solution `u` must be represented by a `Function`, while u in $a(v, u)$ is represented by a `BasisFunction`.

6 Expression representation

6.1 The structure of an expression

Most of the UFL implementation is concerned with expressing, representing, and manipulating expressions. To explain and reason about expression representations and algorithms operating on them, we need an abstract notation for the structure of an expression. UFL expressions are representations of programs, and the notation should allow us to see this connection without the burden of implementation details.

The most basic UFL expressions are expressions with no dependencies on other expressions, called *terminals*. Other expressions are the result of applying some *operator* to one or more existing expressions. All expressions are immutable; once constructed an expression will never change. Manipulating an expression always results in a new expression being created.

Consider an arbitrary (non-terminal) expression z. This expression depends on a set of terminal values $\{t_i\}$, and is computed using a set of operators $\{f_i\}$. If each subexpression of z is labeled with an integer, an abstract program can be written to compute z by computing a sequence of subexpressions $\langle y_i \rangle_{i=1}^{n}$ and setting $z = y_n$. Algorithm 3 shows such a program.

Each terminal expression $y_i = t_i$ is a literal constant or input arguments to the program. A non-terminal subexpression y_i is the result of applying an operator f_i to a sequence of previously computed expressions $\langle y_j \rangle_{j \in \mathfrak{I}_i}$, where $\mathfrak{I}_i$ is a set of expression labels. Note that the order in which subexpressions are computed can be arbitrarily chosen, except that we require $j < i \, \forall j \in \mathfrak{I}_i$, such that all dependencies of a subexpression y_i has been computed before y_i. In particular, all terminals are numbered first in this algorithm for notational convenience only.

Algorithm 3 Program to compute an expression z

for $i = 1, \ldots, m$:
    $y_i = t_i =$ terminal expression
for $i = m + 1, \ldots, n$:
    $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
$z = y_n$

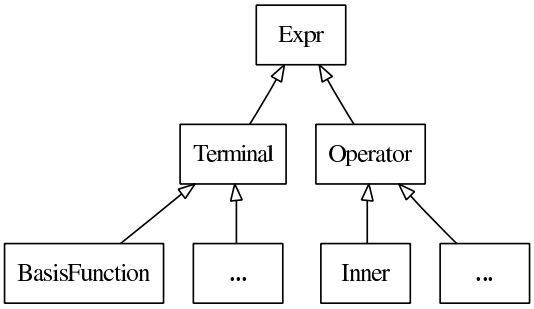

Figure 2: Expression class hierarchy.

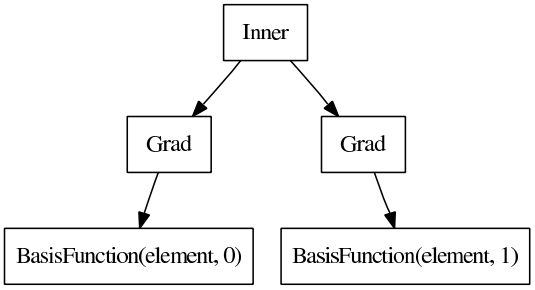

Figure 3: Expression tree for $\nabla \mathbf{u} : \nabla \mathbf{v}$.

The program can be represented as a graph, where each expression y_i corresponds to a graph vertex and each direct dependency between two expressions is a graph edge. More formally,

$$G = (V, E), \tag{57}$$
$$V = \langle v_i \rangle_{i=1}^{n} = \langle y_i \rangle_{i=1}^{n}, \tag{58}$$
$$E = \{e_i\} = \bigcup_{i=1}^{n} \{(i, j) \forall j \in \mathfrak{I}_i\}. \tag{59}$$

This graph is clearly directed, since dependencies have a direction. It is acyclic, since an expression can only be constructed from existing expressions and never be modified. Thus we can say that an UFL expression represents a program, and can be represented using a directed acyclic graph (DAG). There are two ways this DAG can be represented in UFL, a linked representation called the expression tree, and a linearized representation called the computational graph.

6.2 Tree representation

An expression is usually represented as an expression tree. Each subexpression is represented by a tree node, which is the root of a tree of its own. The leaves of the tree are terminal expressions, and operators have their operands as children. An expression tree for the stiffness term $\nabla \mathbf{u} : \nabla \mathbf{v}$ is illustrated in Figure 3. The terminals $\mathbf{u}$ and $\mathbf{v}$ have no children, and the term $\nabla \mathbf{u}$ is itself represented by a tree with two nodes. The names in this figure, `Grad`, `Inner` and `BasisFunction`, reflect the names of the classes used in UFL to represent the expression nodes. Taking the gradient of an expression with `grad(u)` gives an expression representation `Grad(u)`, and `inner(a, b)` gives an expression representation `Inner(a, b)`. In general, each expression node is an instance

of some subclass of `Expr`. The class `Expr` is the superclass of a hierarchy containing all terminal types and operator types UFL supports. `Expr` has two direct subclasses, `Terminal` and `Operator`, as illustrated in Figure 2.

Each expression node represents a single vertex v_i in the DAG. Recall from Algorithm 3 that non-terminals are expressions $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$. The operator f_i is represented by the class of the expression node, while the expression y_i is represented by the instance of this class. The edges of the DAG is not stored explicitly in the tree representation. However, from an expression node representing the vertex v_i, a tuple with the vertices $\langle y_j \rangle_{j \in \mathfrak{I}_i}$ can be obtained by calling `yi.operands()`. These expression nodes represent the graph vertices that have edges pointing to them from y_i. Note that this generalizes to terminals where there are no outgoing edges and `t.operands()` returns an empty tuple.

6.3 Expression node properties

Any expression node `e` (an `Expr` instance) has certain generic properties, and the most important ones will be explained here. Above it was mentioned that `e.operands()` returns a tuple with the child nodes. Any expression node can be reconstructed with modified operands using `e.reconstruct(operands)`, where `operands` is a tuple of expression nodes. The invariant `e.reconstruct(e.operands()) == e` should always hold. This function is required because expression nodes are immutable, they should never be modified. The immutable property ensures that expression nodes can be reused and shared between expressions without side effects in other parts of a program.

In Section 4.2 the tensor algebra and index notation capabilities of UFL was discussed. Expressions can be scalar or tensor-valued, with arbitrary rank and shape. Therefore, each expression node has a value shape `e.shape()`, which is a tuple of integers with the dimensions in each tensor axis. Scalar expressions have shape `()`. Another important property is the set of free indices in an expression, obtained as a tuple using `e.free_indices()`. Although the free indices have no ordering, they are represented with a tuple of `Index` instances for simplicity. Thus the ordering within the tuple carries no meaning.

UFL expressions are referentially transparent with some exceptions. Referential transparency means that a subexpression can be replaced by another representation of its value without changing the meaning of the expression. A key point here is that the value of an expression in this context includes the tensor shape and set of free indices. Another important point is that the derivative of a function $f(v)$ in a point, $f'(v)|_{v=g}$, depends on function values in the vicinity of $v = g$. The effect of this dependency is that operator types matter when differentiating, not only the current value of the differentiation variable. In particular, a `Variable` cannot be replaced by the expression it represents, because `diff` depends on the `Variable` instance and not the expression it has the value of. Similarly, replacing a `Function` with some value will change the meaning of an expression that contains derivatives w.r.t. function coefficients.

The following example illustrate this issue.

```
e = 0
v = variable(e)
```

```
f = sin(v)
g = diff(f, v)
```

Here `v` is a variable that takes on the value 0, but `sin(v)` cannot be simplified to 0 since the derivative of `f` then would be 0. The correct result here is `g = cos(v)`.

6.4 Linearized graph representation

A linearized representation of the DAG is useful for several internal algorithms, either to achieve a more convenient formulation of an algorithm or for improved performance. UFL includes tools to build a linearized representation of the DAG, the *computational graph*, from any expression tree. The computational graph $G = V, E$ is a data structure based on flat arrays, directly mirroring the definition of the graph in equations (57)-(59). This simple data structure makes some algorithms easier to implement or more efficient than the recursive tree representation. One array (Python list) `V` is used to store the vertices $\langle v_i \rangle_{i=1}^n$ of the DAG. For each vertex v_i an expression node y_i is stored to represent it. Thus the expression tree for each vertex is also directly available, since each expression node is the root of its own expression tree. The edges are stored in an array `E` with integer tuples `(i,j)` representing an edge from v_i to v_j, i.e. that v_j is an operand of v_i. The graph is built using a post-order traversal, which guarantees that the vertices are ordered such that $j < i \forall j \in \mathfrak{I}_i$.

From the edges E, related arrays can be computed efficiently; in particular the vertex indices of dependencies of a vertex v_i in both directions are useful:

$$\begin{aligned} V_{out} &= \langle \mathfrak{I}_i \rangle_{i=1}^n , \\ V_{in} &= \langle \{j | i \in \mathfrak{I}_j\} \rangle_{i=1}^n \end{aligned} \tag{60}$$

These data structures can be easily constructed for any expression:

```
G = Graph(expression)
V, E = G
Vin = G.Vin()
Vout = G.Vout()
```

A nice property of the computational graph built by UFL is that no two vertices will represent the same identical expression. During graph building, subexpressions are inserted in a hash map (Python dict) to achieve this.

Free indices in expression nodes can complicate the interpretation of the linearized graph when implementing some algorithms. One solution to that can be to apply `expand_indices` before constructing the graph. Note however that free indices cannot be regained after expansion.

6.5 Partitioning

UFL is intended as a front-end for form compilers. Since the end goal is generation of code from expressions, some utilities are provided for the code generation process. In

principle, correct code can be generated for an expression from its computational graph simply by iterating over the vertices and generating code for each operation separately, basically mirroring Algorithm 3. However, a good form compiler should be able to produce better code. UFL provides utilities for partitioning the computational graph into subgraphs (partitions) based on dependencies of subexpressions, which enables quadrature based form compilers to easily place subexpressions inside the right sets of loops. The function `partition` implements this feature. Each partition is represented by a simple array of vertex indices.

7 Computing derivatives

When a derivative expression is declared by the end-user of the form language, an expression node is constructed to represent it, but nothing is computed. The type of this expression node is a subclass of `Derivative`. Differential operators cannot be expressed natively in a language such as C++. Before code can be generated from the derivative expression, some kind of algorithm to evaluate derivatives must be applied. Computing exact derivatives is important, which rules out approximations by divided differences. Several alternative algorithms exist for computing exact derivatives. All relevant algorithms are based on the chain rule combined with differentiation rules for each expression node type. The main differences between the algorithms are in the extent of which subexpressions are reused, and in the way subexpressions are accumulated.

Below, the differences and similarities between some of the simplest algorithms are discussed. After the algorithm currently implemented in UFL has been explained, extensions to tensor and index notation and higher order derivatives are discussed. Finally, the section is closed with some remarks about the differentiation rules for terminal expressions.

7.1 Relations to form compiler approaches

Before discussing the choice of algorithm for computing derivatives, let us concider the context in which the results will be used. Although UFL does not generate code, some form compiler issues are relevant to this context.

Mixing derivative computation into the code generation strategy of each form compiler would lead to a significant duplication of implementation effort. To separate concerns and keep the code manageable, differentiation is implemented as part of UFL in such a way that the form compilers are independent of the chosen differentiation strategy. Before expressions are interpreted by a form compiler, differential operators should be evaluated such that the only operators left are non-differential operators[14]. Therefore, it is advantageous to use the same representation for the evaluated derivative expressions and other expressions.

The properties of each differentiation algorithm is strongly related to the structure of

[14] An exception is made for spatial derivatives of terminals which are unknown to UFL because they are provided by the form compilers.

the expression representation. However, UFL has no control over the final expression representation used by the form compilers. The main difference between the current form compilers is the way in which expressions are integrated. For large classes of equations, symbolic integration or a specialized tensor representation have proven highly efficient ways to evaluate element tensors [6, 22, 23]. However, when applied to more complex equations, the run-time performance of both these approaches is beaten by code generated with quadrature loops [6, 37]. To apply symbolic differentiation, polynomials are expanded which destroys the structure of the expressions, gives potential exponential growth of expression sizes, and hides opportunities for subexpression reuse. Similarly, the tensor representation demands a canonical representation of the integral expressions.

In summary, both current non-quadrature form compiler approaches change the structure of the expressions they get from UFL. This change makes the interaction between the differentiation algorithm and the form compiler approach hard to control. However, this will only become a problem for complex equations, in which case quadrature loop based code is more suitable. Code generation using quadrature loops can more easily mirror the inherent structure of UFL expressions.

7.2 Approaches to computing derivatives

Algorithms for computing derivatives are designed with different end goals in mind. Symbolic Differentiation (SD) takes as input a single symbolic expression and produces a new symbolic expression for the derivative of the input. Automatic Differentiation (AD) takes as input a program to compute a function and produces a new program to compute the derivative of the function. Several variants of AD algorithms exist, the two most common being Forward Mode AD and Reverse Mode AD [20]. More advanced algorithms exist, and is an active research topic.A UFL expression is a symbolic expression, represented by an expression tree. But the expression tree is a directed acyclic graph that represents a program to evaluate said expression. Thus it seems the line between SD and AD becomes less distinct in this context.

Naively applied, SD can result in huge expressions, which can both require a lot of memory during the computation and be highly inefficient if written to code directly. However, some illustrations of the inefficiency of symbolic differentiation, such as in [20], are based on computing closed form expressions of derivatives in some stand-alone computer algebra system (CAS). Copying the resulting large expressions directly into a computer code can lead to very inefficient code. The compiler may not be able to detect common subexpressions, in particular if simplification and rewriting rules in the CAS has changed the structure of subexpressions with a potential for reuse.

In general, AD is capable of handling algorithms that SD can not. A tool for applying AD to a generic source code must handle many complications such as subroutines, global variables, arbitrary loops and branches [11, 12, 19]. Since the support for program flow constructs in UFL is very limited, the AD implementation in UFL will not run into such complications. In Section 7.3 the similarity between SD and forward mode AD in the context of UFL is explained in more detail.

7.3 Forward mode Automatic Differentiation

Recall Algorithm 3, which represents a program for computing an expression z from a set of terminal values $\{t_i\}$ and a set of elementary operations $\{f_i\}$. Assume for a moment that there are no differential operators among $\{f_i\}$. The algorithm can then be extended to compute the derivative $\frac{dz}{dv}$, where v represents a differentiation variable of any kind. This extension gives Algorithm 4.

Algorithm 4 Forward mode AD on Algorithm 3

for $i = 1, \ldots, m$:
  $y_i = t_i$
  $\frac{dy_i}{dv} = \frac{dt_i}{dv}$
for $i = m+1, \ldots, n$:
  $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
  $\frac{dy_i}{dv} = \sum_{k \in \mathfrak{I}_i} \frac{\partial f_i}{\partial y_k} \frac{dy_k}{dv}$
$z = y_n$
$\frac{dz}{dv} = \frac{dy_n}{dv}$

This way of extending a program to simultaneously compute the expression z and its derivative $\frac{dz}{dv}$ is called forward mode automatic differentiation (AD). By renaming y_i and $\frac{dy_i}{dv}$ to a new sequence of values $\langle \hat{y}_j \rangle_{j=1}^{\hat{n}}$, Algorithm 4 can be rewritten as shown in Algorithm 5, which is isomorphic to Algorithm 3 (they have exactly the same structure).

Algorithm 5 Program to compute $\frac{dz}{dv}$ produced by forward mode AD

for $i = 1, \ldots, \hat{m}$:
  $\hat{y}_i = \hat{t}_i$
for $i = \hat{m}+1, \ldots, \hat{n}$:
  $\hat{y}_i = \hat{f}_i(\langle \hat{y}_j \rangle_{j \in \hat{\mathfrak{I}}_i})$
$\frac{dz}{dv} = \hat{y}_{\hat{n}}$

Since the program in Algorithm 3 can be represented as a DAG, and Algorithm 5 is isomorphic to Algorithm 3, the program in Algorithm 5 can also be represented as a DAG. Thus a program to compute $\frac{dz}{dv}$ can be represented by an expression tree built from terminal values and non-differential operators.

The currently implemented algorithm for computing derivatives in UFL follows forward mode AD closely. Since the result is a new expression tree, the algorithm can also be called symbolic differentiation. In this context, the differences between the two are implementation details. To ensure that we can reuse expressions properly, simplification rules in UFL avoids modifying the operands of an operator. Naturally repeated patterns in the expression can therefore be detected easily by the form compilers. Efficient common subexpression elimination can then be implemented by placing subexpressions in a hash map. However, there are simplifications such as $0 * f \to 0$ and $1 * f \to f$ which simplify the result of the differentiation algorithm automatically as it is being

constructed. These simplifications are crucial for the memory use during derivative computations, and the performance of the resulting program.

7.4 Extensions to tensors and indexed expressions

So far we have not considered derivatives of non-scalar expression and expressions with free indices. This issue does not affect the overall algorithms, but it does affect the local derivative rules for each expression type.

Consider the expression `diff(A, B)` with `A` and `B` matrix expressions. The meaning of derivatives of tensors w.r.t. to tensors is easily defined via index notation, which is heavily used within the differentiation rules:

$$\frac{d\mathbf{A}}{d\mathbf{B}} = \frac{dA_{ij}}{dB_{kl}}\mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l \tag{61}$$

Derivatives of subexpressions are frequently evaluated to literal constants. For indexed expressions, it is important that free indices are propagated correctly with the derivatives. Therefore, differentiated expressions will some times include literal constants annotated with free indices.

There is one rare and tricky corner case when an index sum binds an index i such as in $(v_i v_i)$ and the derivative w.r.t. x_i is attempted. The simplest example of this is the expression $(v_i v_i)_{,j}$, which has one free index j. If j is replaced by i, the expression can still be well defined, but you would never write $(v_i v_i)_{,i}$ manually. If the expression in the parenthesis is defined in a variable `e = v[i]*v[i]`, the expression `e.dx(i)` looks innocent. However, this will cause problems as derivatives (including the index i) are propagated up to terminals. If this case is encountered it will be detected and an error message will be triggered. To avoid it, simply use different index instances. In the future, this case may be handled by relabeling indices to change this expression into $(v_j v_j)_{,i} u_i$.

7.5 Higher order derivatives

A simple forward mode AD implementation such as Algorithm 4 only considers one differentiation variable. Higher order or nested differential operators must also be supported, with any combination of differentiation variables. A simple example illustrating such an expression can be

$$a = \frac{d}{dx}\left(\frac{d}{dx}f(x) + 2\frac{d}{dy}g(x,y)\right). \tag{62}$$

Considerations for implementations of nested derivatives in a functional[15] framework have been explored in several papers [21, 38, 41].

In the current UFL implementation this is solved in a different fashion. Considering Equation (62), the approach is simply to compute the innermost derivatives $\frac{d}{dx}f(x)$ and $\frac{d}{dy}g(x,y)$ first, and then computing the outer derivatives. This approach is possible

[15]Functional as in functional languages.

because the result of a derivative computation is represented as an expression tree just as any other expression. Mainly this approach was chosen because it is simple to implement and easy to verify. Whether other approaches are faster has not been investigated. Furthermore, alternative AD algorithms such as reverse mode can be experimented with in the future without concern for nested derivatives in the first implementations.

An outer controller function `apply_ad` handles the application of a single variable AD routine to an expression with possibly nested derivatives. The AD routine is a function accepting a derivative expression node and returning an expression where the single variable derivative has been computed. This routine can be an implementation of Algorithm 5. The result of `apply_ad` is mathematically equivalent to the input, but with no derivative expression nodes left[16].

The function `apply_ad` works by traversing the tree recursively in post-order, discovering subtrees where the root represents a derivative, and applying the provided AD routine to the derivative subtree. Since the children of the derivative node has already been visited by `apply_ad`, they are guaranteed to be free of derivative expression nodes and the AD routine only needs to handle the case discussed above with algorithms 4 and 5.

The complexity of the `ad_routine` should be $O(n)$, with n being the size of the expression tree. The size of the derivative expression is proportional to the original expression. If there are d derivative expression nodes in the expression tree, the complexity of this algorithm is $O(dn)$, since `ad_routine` is applied to subexpressions d times. As a result the worst case complexity of `apply_ad` is $O(n^2)$, but in practice $d \ll n$. A recursive implementation of this algorithm is shown in Figure 4.

```
def apply_ad(e, ad_routine):
    if isinstance(e, Terminal):
        return e
    ops = [apply_ad(o, ad_routine) for o in e.operands()]
    e = e.reconstruct(*ops)
    if isinstance(e, Derivative):
        e = ad_routine(e)
    return e
```

Figure 4: Simple implementation of recursive `apply_ad` procedure.

7.6 Basic differentiation rules

To implement the algorithm descriptions above, we must implement differentiation rules for all expression node types. Derivatives of operators can be implemented as generic rules independent of the differentiation variable, and these are well known and not mentioned here. Derivatives of terminals depend on the differentiation variable type.

[16]Except direct spatial derivatives of form arguments, but that is an implementation detail.

Derivatives of literal constants are of course always zero, and only spatial derivatives of geometric quantities are non-zero. Since form arguments are unknown to UFL (they are provided externally by the form compilers), their spatial derivatives ($\frac{\partial \phi^k}{\partial x_i}$ and $\frac{\partial w^k}{\partial x_i}$) are considered input arguments as well. In all derivative computations, the assumption is made that form coefficients have no dependencies on the differentiation variable. Two more cases needs explaining, the user defined variables and derivatives w.r.t. the coefficients of a `Function`.

If v is a `Variable`, then we define $\frac{dt}{dv} \equiv 0$ for any terminal t. If v is scalar valued then $\frac{dv}{dv} \equiv 1$. Furthermore, if $\mathbf{V}$ is a tensor valued `Variable`, its derivative w.r.t. itself is

$$\frac{d\mathbf{V}}{d\mathbf{V}} = \frac{dV_{ij}}{dV_{kl}} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l = \delta_{ik}\delta_{jl} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l. \tag{63}$$

In addition, the derivative of a variable w.r.t. something else than itself equals the derivative of the expression it represents:

$$v = g, \tag{64}$$

$$\frac{dv}{dz} = \frac{dg}{dz}. \tag{65}$$

Finally, we consider the operator `derivative`, which represents differentiation w.r.t. all coefficients $\{w_k\}$ of a function w. Consider an object `element` which represents a finite element space V_h with a basis $\{\phi_k\}$. Next consider form arguments defined in this space:

```
v = BasisFunction(element)
w = Function(element)
```

The BasisFunction instance `v` represents any $v \in \{\phi_k\}$, while the `Function` instance `w` represents the sum

$$w = \sum_k w_k \phi_k(x). \tag{66}$$

The derivative of `w` w.r.t. any w_k is the corresponding basis function in V_h,

$$\frac{\partial w}{\partial w_k} = \phi_k, \qquad k = 1, \ldots, |V_h|, \tag{67}$$

$$\tag{68}$$

which can be represented by `v`, since

$$v \in \langle \phi_k \rangle_{k=1}^{|V_h|} = \left\langle \frac{\partial w}{\partial w_k} \right\rangle_{k=1}^{|V_h|}. \tag{69}$$

Note that `v` should be a basis function instance that has not already been used in the form.

8 Algorithms

In this section, some central algorithms and key implementation issues are discussed, much of which relates to the Python programming language. Thus, this section is mainly intended for developers and others who need to relate to UFL on a technical level.

8.1 Effective tree traversal in Python

Applying some action to all nodes in a tree is naturally expressed using recursion:

```
def walk(expression, pre_action, post_action):
    pre_action(expression)
    for o in expression.operands():
        walk(o)
    post_action(expression)
```

This implementation simultaneously covers pre-order traversal, where each node is visited before its children, and post-order traversal, where each node is visited after its children.

A more "pythonic" way to implement iteration over a collection of nodes is using generators. A minimal implementation of this could be

```
def post_traversal(root):
    for o in root.operands():
        yield post_traversal(o)
    yield root
```

which then enables the natural Python syntax for iteration over expression nodes:

```
for e in post_traversal(expression):
    post_action(e)
```

For efficiency, the actual implementation of `post_traversal` in UFL is not using recursion. Function calls are very expensive in Python, which makes the non-recursive implementation an order of magnitude faster than the above.

8.2 Type based function dispatch in Python

A common task in both symbolic computing and compiler implementation is the selection of some operation based on the type of an expression node. For a selected few operations, this is done using overloading of functions in the subclasses of `Expr`, but this is not suitable for all operations.

In many cases type-specific operations must be implemented together in the algorithm instead of distributed across class definitions. One way to implement type based operation selection is to use a type switch, or a sequence of if-tests such as this:

```
class ExampleFunction(MultiFunction):
    def __init__(self):
        MultiFunction.__init__(self)

    def terminal(self, expression):
        return "Got a Terminal subtype %s." % type(expression)

    def operator(self, expression):
        return "Got an Operator subtype %s." % type(expression)

    def basis_function(self, expression):
        return "Got a BasisFunction."

    def sum(self, expression):
        return "Got a Sum."

m = ExampleFunction()

cell = triangle
element = FiniteElement("CG", cell, 1)
x = cell.x
print m(BasisFunction(element))
print m(x)
print m(x[0] + x[1])
print m(x[0] * x[1])
```

Figure 5: Example declaration and use of a multifunction

```
if isinstance(expression, IntValue):
    result = int_operation(expression)
elif isinstance(expression, Sum):
    result = sum_operation(expression)
# etc.
```

There are several problems with this approach, one of which is efficiency when there are many types to check. A type based function dispatch mechanism with efficiency independent of the number of types is implemented as an alternative through the class `MultiFunction`. The underlying mechanism is a dict lookup (which is $O(1)$) based on the type of the input argument, followed by a call to the function found in the dict. The lookup table is built in the `MultiFunction` constructor. Functions to insert in the table are discovered automatically using the introspection capabilities of Python.

A multifunction is declared as a subclass of `MultiFunction`. For each type that should be handled particularly, a member function is declared in the subclass. The

`Expr` classes use the `CamelCaps` naming convention, which is automatically converted to `underscore_notation` for corresponding function names, such as `BasisFunction` and `basis_function`. If a handler function is not declared for a type, the closest superclass handler function is used instead. Note that the `MultiFunction` implementation is specialized to types in the `Expr` class hierarchy. The declaration and use of a multifunction is illustrated in Figure 5. Note that `basis_function` and `sum` will handle instances of the exact types `BasisFunction` and `Sum`, while `terminal` and `operator` will handle the types `SpatialCoordinate` and `Product` since they have no specific handlers.

8.3 Implementing expression transformations

Many transformations of expressions can be implemented recursively with some type-specific operation applied to each expression node. Examples of operations are converting an expression node to a string representation, an expression representation using an symbolic external library, or an UFL representation with some different properties. A simple variant of this pattern can be implemented using a multifunction to represent the type-specific operation:

```
def apply(e, multifunction):
    ops = [apply(o, multifunction) for o in e.operands()]
    return multifunction(e, *ops)
```

The basic idea is as follows. Given an expression node `e`, begin with applying the transformation to each child node. Then return the result of some operation specialized according to the type of `e`, using the already transformed children as input.

The `Transformer` class implements this pattern. Defining a new algorithm using this pattern involves declaring a `Transformer` subclass, and implementing the type specific operations as member functions of this class just as with `MultiFunction`. The difference is that member functions take one additional argument for each operand of the expression node. The transformed child nodes are supplied as these additional arguments. The following code replaces terminal objects with objects found in a dict `mapping`, and reconstructs operators with the transformed expression trees. The algorithm is applied to an expression by calling the function `visit`, named after the similar Visitor pattern.

```
class Replacer(Transformer):
    def __init__(self, mapping):
        Transformer.__init__(self)
        self.mapping = mapping

    def operator(self, e, *ops):
        return e.reconstruct(*ops)

    def terminal(self, e):
        return self.mapping.get(e, e)
```

```
f = Constant(triangle)
r = Replacer({f: f**2})
g = r.visit(2*f)
```

After running this code the result is $g = 2f^2$. The actual implementation of the `replace` function is similar to this code.

In some cases, child nodes should not be visited before their parent node. This distinction is easily expressed using `Transformer`, simply by omitting the member function arguments for the transformed operands. See the source code for many examples of algorithms using this pattern.

8.4 Important transformations

There are many ways in which expression representations can be manipulated. Here, we describe a few particularly important transformations. Note that each of these algorithms removes some abstractions, and hence may remove some opportunities for analysis or optimization.

Some operators in UFL are termed "compound" operators, meaning they can be represented by other elementary operators. Try defining an expression `e = inner(grad(u), grad(v))`, and print `repr(e)`. As you will see, the representation of `e` is `Inner(Grad(u), Grad(v))` (with some more details for `u` and `v`). This way the input expressions are easier to recognize in the representation, and rendering of expressions to for example LaTeX format can show the original compound operators as written by the end-user.

However, since many algorithms must implement actions for each operator type, the function `expand_compounds` is used to replace all expression nodes of "compound" types with equivalent expressions using basic types. When this operation is applied to the input forms from the user, algorithms in both UFL and the form compilers can still be written purely in terms of basic operators.

Another important transformation is `expand_derivatives`, which applies automatic differentiation to expressions, recursively and for all kinds of derivatives. The end result is that most derivatives are evaluated, and the only derivative operator types left in the expression tree applies to terminals. The precondition for this algorithm is that `expand_compounds` has been applied.

Index notation and the `IndexSum` expression node type complicate interpretation of an expression tree in some contexts, since free indices in its summand expression will take on multiple values. In some cases, the transformation `expand_indices` comes in handy, the end result of which is that there are no free indices left in the expression. The precondition for this algorithm is that `expand_compounds` and `expand_derivatives` have been applied.

8.5 Evaluating expressions

Even though UFL expressions are intended to be compiled by form compilers, it can be useful to evaluate them to floating point values directly. In particular, this makes testing and debugging of UFL much easier, and is used extensively in the unit tests. To

evaluate an UFL expression, values of form arguments and geometric quantities must be specified. Expressions depending only on spatial coordinates can be evaluated by passing a tuple with the coordinates to the call operator. The following code from an interactive Python session shows the syntax:

```
>>> cell = triangle
>>> x = cell.x
>>> e = x[0]+x[1]
>>> print e((0.5,0.7))
1.2
```

Other terminals can be specified using a dictionary that maps from terminal instances to values. This code extends the above code with a mapping:

```
c = Constant(cell)
e = c*(x[0]+x[1])
print e((0.5,0.7), { c: 10 })
```

If functions and basis functions depend on the spatial coordinates, the mapping can specify a Python callable instead of a literal constant. The callable must take the spatial coordinates as input and return a floating point value. If the function being mapped is a vector function, the callable must return a tuple of values instead. These extensions can be seen in the following code:

```
element = VectorElement("CG", cell, 1)
f = Function(element)
e = c*(f[0] + f[1])
def fh(x):
    return (x[0], x[1])
print e((0.5,0.7), { c: 10, f: fh })
```

To use expression evaluation for validating that the derivative computations are correct, spatial derivatives of form arguments can also be specified. The callable must then take a second argument which is called with a tuple of integers specifying the spatial directions in which to differentiate. A final example code computing $g^2 + g_{,0}^2 + g_{,1}^2$ for $g = x_0 x_1$ is shown below.

```
element = FiniteElement("CG", cell, 1)
g = Function(element)
e = g**2 + g.dx(0)**2 + g.dx(1)**2
def gh(x, der=()):
    if der == ():   return x[0]*x[1]
    if der == (0,): return x[1]
    if der == (1,): return x[0]
print e((2, 3), { g: gh })
```

8.6 Viewing expressions

Expressions can be formatted in various ways for inspection, which is particularly useful while debugging. The Python built in string conversion operator `str(e)` provides a compact human readable string. If you type `print e` in an interactive Python session, `str(e)` is shown. Another Python built in string operator is `repr(e)`. UFL implements `repr` correctly such that `e == eval(repr(e))` for any expression `e`. The string `repr(e)` reflects all the exact representation types used in an expression, and can therefore be useful for debugging. Another formatting function is `tree_format(e)`, which produces an indented multi-line string that shows the tree structure of an expression clearly, as opposed to `repr` which can return quite long and hard to read strings. Information about formatting of expressions as LaTeX and the dot graph visualization format can be found in the manual.

9 Implementation issues

9.1 Python as a basis for a domain specific language

Many of the implementation details detailed in this section are influenced by the initial choice of implementing UFL as an embedded language in Python. Therefore some words about why Python is suitable for this, and why not, are appropriate here.

Python provides a simple syntax that is often said to be close to pseudo-code. This is a good starting point for a domain specific language. Object orientation and operator overloading is well supported, and this is fundamental to the design of UFL. The functional programming features of Python (such as generator expressions) are useful in the implementation of algorithms and form compilers. The built-in data structures `list`, `dict` and `set` play a central role in fast implementations of scalable algorithms.

There is one problem with operator overloading in Python, and that is the comparison operators. The problem stems from the fact that `__eq__` or `__cmp__` are used by the built-in data structures dict and set to compare keys, meaning that `a == b` must return a boolean value for `Expr` to be used as keys. The result is that `__eq__` can not be overloaded to return some `Expr` type representation such as `Equals(a, b)` for later processing by form compilers. The other problem is that `and` and `or` cannot be overloaded, and therefore cannot be used in `conditional` expressions. There are good reasons for these design choices in Python. This conflict is the reason for the somewhat non-intuitive design of the comparison operators in UFL.

9.2 Ensuring unique form signatures

The form compilers need to compute a unique signature of each form for use in a cache system to avoid recompilations. A convenient way to define a signature is using `repr(form)`, since the definition of this in Python is `eval(repr(form)) == form`. Therefore `__repr__` is implemented for all `Expr` subclasses.

Some forms are equivalent even though their representation is not exactly the same. UFL does not use a truly canonical form for its expressions, but takes some measures

to ensure that trivially equivalent forms are recognized as such.

Some of the types in the `Expr` class hierarchy (subclasses of `Counted`), has a global counter to identify the order in which they were created. This counter is used by form arguments (both `BasisFunction` and `Function`) to identify their relative ordering in the argument list of the form. Other counted types are `Index` and `Label`, which only use the counter as a unique identifier. Algorithms are implemented for renumbering of all `Counted` types such that all counts start from 0.

In addition, some operator types such as `Sum` and `Product` maintains a sorted list of operands such that `a+b` and `b+a` are both represented as `Sum(a, b)`. The numbering of indices does not affect this ordering because a renumbering of the indices would lead to a new ordering which would lead to a different index renumbering if applied again. The operand sorting and renumbering combined ensure that the signature of equal forms will stay the same. To get the signature with renumbering applied, use `repr(form.form_data().form)`. Note that the representation, and thus the signature, of a form may change with versions of UFL.

9.3 Efficiency considerations

By writing UFL in Python, we clearly do not put peak performance as a first priority. If the form compilation process can blend into the application build process, the performance is sufficient. We do, however, care about scaling performance to handle complicated equations efficiently, and therefore about the asymptotic complexity of the algorithms we use.

To write clear and efficient algorithms in Python, it is important to use the built in data structures correctly. These data structures include in particular `list`, `dict` and `set`. CPython [45], the reference implementation of Python, implements the data structure `list` as an array, which means append, and pop, and random read or write access are all O(1) operations. Random insertion, however, is O(n). Both `dict` and `set` are implemented as hash maps, the latter simply with no value associated with the keys. In a hash map, random read, write, insertion and deletion of items are all $O(1)$ operations, as long as the key types implement `__hash__` and `__eq__` efficiently. Thus to enjoy efficient use of these containers, all `Expr` subclasses must implement these two special functions efficiently. The dict data structure is used extensively by the Python language, and therefore particular attention has been given to make it efficient [26].

10 Future directions

Many additional features can be introduced to UFL. Which features are added will depend on the needs of FEniCS users and developers. Some features can be implemented in UFL alone, while other features will require updates to other parts of the FEniCS project.

Improvements to finite element declarations is likely easy to do in UFL. The added complexity will mostly be in the form compilers. Among the current suggestions are space-time elements and related time derivatives, and enrichment of finite element spaces. Additional geometry mappings and finite element spaces with non-uniform cell

types are also possible extensions.

Additional operators can be added to make the language more expressive. Some operators are easy to add because their implementation only affects a small part of the code. More compound operators that can be expressed using elementary operations is easy to add. Additional special functions are easy to add as well, as long as their derivatives are known. Other features may require more thorough design considerations, such as support for complex numbers which may affect many parts of the code.

User friendly notation and support for rapid development are core values in the design of UFL. Having a notation close to the mathematical abstractions allows expression of particular ideas more easily, which can reduce the probability of bugs in user code. However, the notion of metaprogramming and code generation adds another layer of abstraction which can make understanding the framework more difficult for end-users. Good error checks everywhere are therefore very important, to detect user errors as close as possible to the user input. The error messages, documentation, and unit test suite should be improved to help avoid frequently repeated errors and misunderstandings among new users.

Several algorithms in UFL can probably be optimized if bottlenecks are found as more complicated applications are attempted. The focus in the development has not been on achieving peak performance, which is not important in a tool like UFL.

To support form compiler improvements, algorithms and utilities for generating better code more efficiently can be implemented in UFL. In this area, more work on alternative automatic differentiation algorithms [18, 44] can be useful. Another possibility for code improvement is operation scheduling, or reordering of the vertices of a graph partition to improve the efficiency of the generated code by better use of hardware cache and registers. Since modern C++ compilers are quite good at optimizing low level code, the focus should be on high level optimizations when considering potential code improvement in UFL and the form compilers. At the time of writing, operation scheduling is not implemented in UFL, and the value of implementing such an operation is an open question. However, results from [18] indicates that a high level scheduling algorithm could improve the efficiency of the generated code.

To summarize, UFL brings important improvements to the FEniCS framework: a richer form language, automatic differentiation and improved form compiler efficiency. These are useful features in rapid development of applications for efficiently solving partial differential equations. UFL improves upon the Automation of Discretization that has been the core feature of this framework, and adds Automation of Linearization. In conclusion, UFL brings FEniCS one step closer to its overall goal Automation of Computational Mathematical Modeling.

11 Acknowledgements

This work has been supported by the Norwegian Research Council (grant 162730) and Simula Research Laboratory. I wish to thank everyone who has helped improving UFL with suggestions and testing, in particular Anders Logg, Kristian Ølgaard, Garth Wells, and Harish Narayanan. Both Kent-André Mardal and Marie Rognes performed critical reviews which greatly improved this manuscript.

Bibliography

[1] M. S. Alnæs and A.Logg, *UFL Specification and User Manual*, 2009. http://www.fenics.org/ufl/.

[2] M. S. Alnæs, H.-P. Langtangen, A.Logg, K.-A. Mardal, and O. Skavhaug, *UFC*. http://www.fenics.org/ufc/, 2009.

[3] ——, *UFC Specification and User Manual*, 2009. http://www.fenics.org/ufc/.

[4] M. S. Alnæs and A. Logg, *UFL*. http://www.fenics.org/ufl/, 2009.

[5] M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen, *Unified Framework for Finite Element Assembly*, International Journal of Computational Science and Engineering, (2009). Accepted for publication. Preprint: http://simula.no/research/scientific/publications/Simula.SC.96.

[6] M. S. Alnæs and K.-A. Mardal, *On the efficiency of symbolic computations combined with code generation for finite element methods*, (2009). Submitted.

[7] M. S. Alnæs and K.-A. Mardal, *SyFi*. http://www.fenics.org/syfi/, 2009.

[8] ——, *SyFi User Manual*, 2009. http://www.fenics.org/syfi/.

[9] C. Bauer, C. Dams, A. Frink, V. V. Kisil, R. Kreckel, A. Sheplyakov, and J. Vollinga, *GiNaC*, 2009. http://www.ginac.de.

[10] C. Bauer, A. Frink, and R. Kreckel, *Introduction to the GiNaC framework for symbolic computation within the C++ programming language*, cs/0004015, (2000). J. Symbolic Computation (2002) 33, 1-12.

[11] C. Bischof, A. Carle, G. Corliss, and A. Griewank, *ADIFOR: automatic differentiation in a source translator environment*, in Papers from the international symposium on Symbolic and algebraic computation, Berkeley, California, United States, 1992, ACM, pp. 294–302.

[12] C. H. Bischof, P. D. Hovland, and B. Norris, *Implementation of automatic differentiation tools*, in Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, Portland, Oregon, 2002, ACM, pp. 98–107.

[13] R. Cimrman et al., *SfePy: Simple Finite Elements in Python*, 2008.

[14] P. Dular and C. Geuzaine, *GetDP Reference Manual*, 2005.

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

[16] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and L. R. Scott, *The FEniCS project*, Tech. Rep. 2003–21, Chalmers Finite Element Center Preprint Series, 2003.

[17] FEniCS, *FEniCS project*. http://www.fenics.org/.

[18] S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid, *Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding*, ACM Trans. Math. Softw., 30 (2004), pp. 266–299.

[19] R. Giering and T. Kaminski, *Recipes for adjoint code construction*, ACM Trans. Math. Softw., 24 (1998), pp. 437–474.

[20] A. Griewank, *On automatic differentiation*, In Mathematical Programming: Recent Developments and Applications, (1989), pp. 83—108.

[21] J. Karczmarczuk, *Functional differentiation of computer programs*, Higher-Order and Symbolic Computation, 14 (2001), pp. 35–57.

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

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

[24] J. Korelc, *Automatic generation of finite-element code by simultaneous optimization of expressions*, Theoretical Computer Science, 187 (1997), pp. 231–248.

[25] J. Korelc, *Multi-language and multi-environment generation of nonlinear finite element codes*, Engineering with Computers, 18 (2002), pp. 312–327.

[26] A. Kuchling, *Pythons dictionary implementation: Being all things to all people*, in Beautiful code, A. Oram and G. Wilson, eds., O'Reilly, 2007, ch. 18, pp. 293–302.

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

[28] ——, *FFC User Manual*, 2009. http://www.fenics.org/ffc/.

[29] A. Logg et al., *FFC*. http://www.fenics.org/ffc/, 2009.

[30] A. Logg and G. N. Wells, *DOLFIN: Automated finite element computing*, Submitted, (2009).

[31] A. Logg, G. N. Wells, et al., *DOLFIN*. http://www.fenics.org/dolfin/.

[32] ——, *DOLFIN User Manual*, 2009. http://www.fenics.org/dolfin/.

[33] K. Long, *Sundance, a rapid prototyping tool for parallel PDE-constrained optimization*, in Large-Scale PDE-Constrained Optimization, Lecture notes in computational science and engineering, Springer-Verlag, 2003.

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

[35] ——, *Sundance*, 2006. http://software.sandia.gov/sundance/.

[36] K. B. Ølgaard, A. Logg, and G. N. Wells, *Automated code generation for discontinuous galerkin methods*, SIAM Journal on Scientific Computing, 31 (2008), pp. 849–864.

[37] K. B. Ølgaard and G. N. Wells, *Representations of finite element tensors via automated code generation*, (2009). Submitted, http://www.dspace.cam.ac.uk/handle/1810/214789.

[38] B. A. Pearlmutter and J. M. Siskind, *Lazy multivariate higher-order forward-mode AD*, in Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Nice, France, 2007, ACM, pp. 155–160.

[39] C. Prud'homme, *A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations*, Sci. Program., 14 (2006), pp. 81–110.

[40] ——, *Life: Overview of a unified c++ implementation of the finite and spectral element methods in 1d, 2d and 3d*, in Workshop On State-Of-The-Art In Scientific And Parallel Computing, Lecture Notes in Computer Science, Springer-Verlag, dec 2006, p. 10.

[41] J. M. Siskind and B. A. Pearlmutter, *Nesting forward-mode AD in a functional framework*, Higher Order Symbol. Comput., 21 (2008), pp. 361–376.

[42] O. Skavhaug and O. Čertík, *Swiginac*, 2009. http://swiginac.berlios.de/.

[43] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, July 1997.

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

[45] G. van Rossum et al., *Python*. http://www.python.org/.

[46] O. Čertík et al., *SymPy*, 2009. http://docs.sympy.org.