Algorithms **2009**, *2*, 1008-1030; doi:10.3390/a2031008

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Automated Modelling of Evolving Discontinuities

Mehdi Nikbakht [1] **and Garth N. Wells** [2,⋆]

[1] Faculty of Civil Engineering and Geosciences, Delft University of Technology, Stevinweg 1, 2628 CN Delft, The Netherlands; E-Mail: m.nikbakht@tudelft.nl

[2] Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, UK

⋆ Author to whom correspondence should be addressed; E-Mail: gnw20@cam.ac.uk

Received: 7 May 2009, in revised form: 28 July 2009 / Accepted: 7 August 2009 / Published: 18 August 2009

Abstract: The automated approximation of solutions to differential equations which involve discontinuities across evolving surfaces is addressed. Finite element technology has developed to the point where it is now possible to model evolving discontinuities independently of the underlying mesh, which is particularly useful in simulating failure of solids. However, the approach remains tedious to program, particularly in the case of coupled problems where a variety of finite element bases are employed and where a mixture of continuous and discontinuous fields may be used. We tackle this point by exploring the scope for employing automated code generation techniques for modelling discontinuities. Function spaces and variational forms are defined in a language that resembles mathematical notation, and computer code for modelling discontinuities is automatically generated. Principles underlying the approach are elucidated and a number of two- and three-dimensional examples for different equations are presented.

Keywords: partition of unity; extended finite element method; fracture; automation; form compiler

---

1. Introduction

The computational modelling of evolving discontinuities has witnessed considerable advances in recent times. Using the partition of unity property of finite element shape functions [1], it is possible to simulate discontinuities in a finite element solution across surfaces which are not aligned with the under-

lying mesh. Such methods are known as the extended finite element method [2, 3] and the generalised finite element method [4]. The approach has obvious application to the modelling of failure in solids, but it can also be applied for a variety of other applications.

While the technology for modelling evolving discontinuities is now maturing, the implementation of such techniques can be tedious and requires a significant investment of time. For these reasons, application of extended finite element techniques is still largely the domain of those in a position to develop software rather than of the broader group, that is, users of computational technology. A limited number of libraries which support the extended finite element method are available (see for example [5]), however these libraries follow the traditional paradigm in which a user is required to program by hand the innermost parts of a solver. We present here our efforts towards the automated solution of partial differential equations which involve evolving discontinuities in the solution. The approach relies on automated code generation from high-level input. Progress has been made in automating large parts of the finite element modelling process for conventional formulations [6, 7], as well as discontinuous Galerkin methods [8] and $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming elements [9]. We will extend and generalise some of these ideas to the modelling of discontinuities. Namely, we will use a *form compiler* to generate low-level computer code from a high-level input language. This permits a high degree of mathematical expressiveness, and by generating low-level code, there is the potential to generate more efficient code than that can be reasonably produced by hand [6, 10]. Furthermore, the approach detaches the model (differential equation) of interest from underlying implementation aspects, such as the discontinuity surface representation.

The rest of this work is structured as follows. In Section 2. we introduce briefly aspects of automated computational modelling for continuous problems and present a simple example. This is followed by a concise overview of modelling discontinuities using the extended finite element method. We then present our approach to automating the solution of problems which involve discontinuities, and this is followed by a collection of examples.

The approach described in this work is manifest in computer code which is available under the GNU Public License (GPL) and the GNU Lesser Public License (LGPL). We build on a number of tools which are part of the FEniCS Project [11], and which are freely available at www.fenics.org. The specific extensions and examples presented in this work are archived at [12].

2. Automated Mathematical Modelling

In the context of a finite element solver, it is possible to divide the code into problem-specific and generic components. The approach which we follow is to generate automatically code for those parts which are specific to a given finite element variational form and to make use of, and develop, where necessary, reusable library components for tasks which are independent of the precise finite element variational form. The code which is specific to a particular finite element variational problem is the finite element basis, the degree of freedom mapping and the element matrices and vectors.

For complicated problems, particularly those that involve a number of coupled equations and combinations of different and possibly unusual finite element spaces, developing the code to compute element matrices and vectors is error prone, tedious and time consuming. Furthermore, developing high performance code by hand for complicated problems is not trivial. A number of efforts are currently under way

to address the gulf between performance and generality in scientific software design (several of which are listed in [7]). An approach to reconciling mathematical expressiveness and generality with performance is automated code generation using a compiler for variational forms [6–8]. A domain specific language can be created which mirrors the standard mathematical notation for variational methods, thereby providing a high degree of expressiveness, and a compiler can be used to generate low-level code from the high-level input. The compiler approach permits different strategies for representing element matrices and vectors [6, 10] and various optimisation approaches can be employed. In particular, optimisations can be employed that are not tractable in hand-written code [6, 8]. In this work, we will present generalisations of the FEniCS Form Compiler (FFC) [6, 8, 13] for extended finite element methods. Firstly though we illustrate the compiler approach for a conventional continuous problem.

For the weighted Poisson equation on the domain $\Omega \subset \mathbb{R}^d$, $1 \leq d \leq 3$ with homogeneous Dirichlet boundary conditions, the variational form of the problem reads: given sufficiently regular functions $w : \Omega \rightarrow \mathbb{R}$, and $f : \Omega \rightarrow \mathbb{R}$, find $u \in H_0^1(\Omega)$ such that

$$a(v, u) = L(v) \quad \forall\, v \in H_0^1(\Omega), \tag{1}$$

where

$$a(v, u) = \int_\Omega w \nabla v \cdot \nabla u \, d\Omega, \tag{2}$$

$$L(v) = \int_\Omega v f \, d\Omega. \tag{3}$$

The term $a(v, u)$ is known as the "bilinear form" and $L(v)$ is known as the "linear form". A finite element formulation of this problem follows from replacing the function space $H_0^1(\Omega)$ by a suitable finite element space $V \subset H_0^1(\Omega)$. If we choose a continuous piecewise linear Lagrange finite element basis in three dimensions ($d = 3$) on tetrahedra,

$$V = \left\{ v \in H_0^1(\Omega), v|_{\Omega_e} \in P^1(\Omega_e) \, \forall e \right\}, \tag{4}$$

where Ω_e is a cell of the triangulation of Ω and P^1 is space of standard linear Lagrange shape functions on a tetrahedron. The input to the form compiler FFC for this variational problem reads:

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

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

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

In this example, `element` defines the finite element space, `v` and `u` are the test and trial functions, respectively, and `w` and `f` are supplied functions. It is assumed in the above input that w and f come from the finite element space V or will be interpolated in V. The bilinear form is denoted by `a`, and the

linear form is denoted by `L`. A number of operators, such as the inner product between two vector-valued functions and the gradient are employed. Integration over a cell is indicated by `*dx`. In FFC, integration over external facets is denoted by `*ds` and integration over internal facets is denoted by `*dS`. The form compiler FFC is developed in Python, which makes it easily extensible.

The form code can be entered into a text file, and FFC can be called from the command line to generate C++ code from this input. The generated code conforms to the Unified Form-assembly Code (UFC) specification [14, 15] and serves as input to any assembly library which supports the UFC interface, such as DOLFIN [7, 16]. Alternatively, the above input to the form compiler code can be included in the Python interface of DOLFIN which will then use just-in-time complication to generate and compile C++ code on demand. A detailed explanation and a range of examples can be found in Logg *et al.*[7].

3. Extended Finite Element Method: Review

The partition-of-unity property of finite element shape functions can be exploited to extend a finite element basis with arbitrary functions, as formulated in the works of Melenk *et al.*[1] and Babuska *et al.* [17]. The partition-of-unity property was utilised by Belytschko *et al.*[18] to extend locally a finite element basis with functions coming from the near-tip solution to linear elastic fracture mechanics problems, and in Moës *et al.*[2] a discontinuous function was introduced to model the jump in the displacement field across a crack surface away from the crack tip in a linear elastic body. Crucially, discontinuous functions which are independent of the underlying finite element mesh structure can be incorporated into the finite element basis, thereby permitting the resolution of discontinuities across paths which are not aligned with the structure of the mesh. While the extended finite element method can be applied to a variety of problems (see for example [3]), we restrict our attention to automated code generation for problems that involve discontinuous solutions across surfaces and for which the flux (traction) is prescribed on the discontinuity surface [19].

3.1. Incorporating Discontinuities into Finite Element Spaces

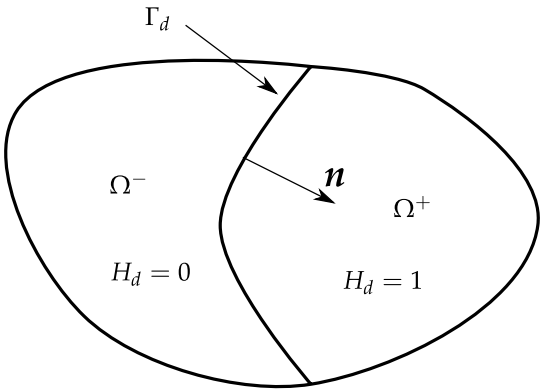
Consider a domain Ω which is crossed by a discontinuity surface Γ_d, as illustrated in Figure 1 (for simplicity we show a body which is completely crossed by a discontinuity surface, but it also possible to consider bodies which are not bisected by a discontinuity surface, and we will do so for the examples presented in Section 5.). The domains on different sides of the discontinuity surface are denoted by Ω^- and Ω^+ such that $\Omega^- \bigcup \Omega^+ \bigcup \Gamma_d = \Omega$. The unit normal vector $\boldsymbol{n}$ on Γ_d is defined such that it points towards Ω^+, as illustrated in Figure 1.

We wish to represent functions which are continuous on $\Omega \backslash \Gamma_d$ and exhibit jumps across the surface Γ_d. Such a function u can be expressed in terms of continuous functions and a jump function by

$$u = \bar{u} + H_d \hat{u}, \tag{5}$$

where $\bar{u}$ and $\hat{u}$ are defined on Ω and are continuous, and H_d is the Heaviside function centred on the discontinuity surface Γ_d and is defined such that $H_d(\boldsymbol{x}) = 1$ if $\boldsymbol{x} \in \Omega^+$ and $H_d(\boldsymbol{x}) = 0$ if $\boldsymbol{x} \in \Omega^-$. The jump in u across the surface Γ_d, denoted by $[\![u]\!] = u^+ - u^-$, and is equal to $\hat{u}$ for $\boldsymbol{x} \in \Gamma_d$. The jump may or may not be required to go to zero on $\partial\Gamma_d$. In the context of a crack, $\partial\Gamma_d$ is the crack tip.

Figure 1. Domain $\Omega \subset \mathbb{R}^2$ crossed by a discontinuity surface Γ_d.



The Heaviside function can be built into a finite element basis by exploiting the partition-of-unity property of the finite element shape functions. A regular finite element function u_h is expressed in terms of the finite element shape functions $\{\phi_i\}$ and the nodal degrees of freedom $\{\bar{u}_i\}$,

$$u_h = \sum_i \phi_i \bar{u}_i. \tag{6}$$

A finite element function which exhibits a jump across the surface Γ_d can be represented by adding the Heaviside function to the finite element basis,

$$u_h = \sum_i \phi_i \bar{u}_i + \sum_i H_d \phi_i \hat{u}_i \tag{7}$$

where $\hat{u}_i$ are "enriched" degrees of freedom and are associated with the field $\hat{u}$ in Equation (5). The expression in (7) will in general lead to a linear dependency if used in a finite element formulation, since away from a discontinuity surface the Heaviside function is a constant over a finite element cell, and the regular finite element basis contains constant functions. This linear dependency can be obviated as follows: enriched degrees of freedom $\hat{u}_i$ at node i are active only if the support of the shape function ϕ_i is intersected by a discontinuity surface. Formally,

$$u_h = \sum_i \phi_i u_i + \sum_i \{\psi\}_i \phi_i \hat{u}_i \tag{8}$$

where $\{\psi\}_i = H_d$ if the support of ϕ_i is intersected by a discontinuity surface, otherwise $\{\psi\}_i = \emptyset$. The extra degrees of freedom are therefore localised to the small region around the discontinuity surface and eliminated elsewhere.

3.2. Example: Poisson Equation

We consider a Poisson problem and deliberately phrase the problem in a mathematically concrete and abstract fashion as we wish our domain specific language, which we will use to define the input to the form compiler, to inherit this syntax. The Poisson problem on Ω, with the boundary $\Gamma = \partial\Omega$ partitioned

such that $\overline{\Gamma_g \cup \Gamma_h} = \Gamma$ and $\Gamma_g \cap \Gamma_h = \emptyset$, involves finding u such that:

$$-\Delta u = f \quad \text{in } \Omega, \tag{9}$$

$$u = 0 \quad \text{on } \Gamma_g, \tag{10}$$

$$\nabla u \cdot \boldsymbol{n} = h \quad \text{on } \Gamma_h, \tag{11}$$

$$\nabla u^+ \cdot \boldsymbol{n} = q \quad \text{on } \Gamma_d, \tag{12}$$

$$[\![\nabla u]\!] \cdot \boldsymbol{n} = 0 \quad \text{on } \Gamma_d, \tag{13}$$

where $f : \Omega \to \mathbb{R}$ a sufficiently regular source term, $g : \Gamma_g \to \mathbb{R}$ and $h : \Gamma_h \to \mathbb{R}$ are the prescribed boundary conditions, q is the flux across the discontinuity surface. The flux may be prescribed or it may be determined via a constitutive model. The jump operator is defined as $[\![\nabla a]\!] = \nabla a^+ - \nabla a^-$. Equation (13) therefore implies continuity of the flux across the surface Γ_d.

Assuming that the flux on the discontinuity surface is given by $q = q\left([\![u]\!]\right)$, the variational formulation of this problem reads: find $u \in V$ such that

$$\int_{\Omega \backslash \Gamma_d} \nabla v \cdot \nabla u \, d\Omega + \int_{\Gamma_d} [\![v]\!] \, q\left([\![u]\!]\right) \, d\Gamma = \int_{\Omega} v f \, d\Omega, + \int_{\Gamma_h} v h \, d\Gamma \quad \forall v \in V, \tag{14}$$

where

$$V = \left\{ v \in L^2(\Omega) \cap H^1(\Omega \backslash \Gamma_d) : v = 0 \text{ on } \Gamma_g \right\}. \tag{15}$$

We wish to construct a finite-dimensional counterpart of this problem using finite element basis functions without requiring that the triangulation of Ω conform in any way to the discontinuity surface. We can do this by decomposing the finite element solution u_h along the same lines as Equation (5):

$$u_h = \bar{u}_h + H_d \hat{u}_h. \tag{16}$$

Decomposing the weight function v similarly, the variational problem can be expressed equivalently as: find $\bar{u}_h \in \bar{V}$ and $\hat{u}_h \in \hat{V}$ such that

$$\begin{aligned} \int_{\Omega} \nabla \bar{v} \cdot \nabla \bar{u} \, d\Omega + \int_{\Omega^+} \nabla \bar{v} \cdot \nabla \hat{u} \, d\Omega + \int_{\Omega^+} \nabla \hat{v} \cdot \nabla \left(\bar{u} + \hat{u}\right) \, d\Omega + \int_{\Gamma_d} \hat{v} q\left(\hat{u}\right) \, d\Gamma \\ = \int_{\Omega} \bar{v} f \, d\Omega + \int_{\Omega^+} \hat{v} f \, d\Omega + \int_{\Gamma_h} \bar{v} h \, d\Gamma + \int_{\Gamma_h^+} \hat{v} h \, d\Gamma \quad \forall \bar{v} \in \bar{V}, \, \forall \hat{v} \in \hat{V}, \end{aligned} \tag{17}$$

where $\Gamma_h^+ = \Gamma_h \cap \partial\Omega^+$. The task now is to generate a suitable finite element space. Following from the expression for u_h in Equation (8), we define the finite element spaces

$$\bar{V} = \left\{ \bar{u} \in H^1(\Omega), \bar{u}|_{\Omega_e} \in P^{k_1}(\Omega_e) \, \forall e : u = 0 \text{ on } \Gamma_g \right\}, \tag{18}$$

$$\hat{V} = \left\{ \hat{u} \in H_0^1(\Omega_d), \hat{u}|_{\Omega_e} \in P^{k_2}(\Omega_e) \, \forall e \in \Omega_d : \hat{u} = 0 \text{ on } \Gamma_g \cap \partial\Omega^+ \right\}, \tag{19}$$

where $\Omega_d \subset \Omega$ is a "small" region around the discontinuity surface Γ_d. More precisely, Ω_d as the union of the supports of all basis functions whose support is intersected by the discontinuity surface. The extended finite element formulation for the Poisson equation is now complete. In the definition of the finite element spaces, we have deliberately permitted the use of different order functions for $\bar{u}_h$ and $\hat{u}_h$ since it will be straightforward using the form compiler to generate code for $k_1 \neq k_2$. For the examples

presented in Section 5., it is assumed that $k_1 = k_2$ and that the solution goes to zero on the boundary, therefore we will use the compacter notation

$$V = \left\{ v_h \in L^2(\Omega) \cap H_0^1(\Omega \backslash \Gamma_d), v_h|_{\Omega_e} \in P^k(\Omega_e \backslash \Gamma_d) \, \forall e \right\} \tag{20}$$

for a finite element function $v_h \in V$, which is discontinuous across surfaces when defining the examples.

The extended finite element problem can be expressed as a system of linear equations

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f}, \tag{21}$$

which in an expanded format for the case $q = k \, [\![u]\!]$, where k is a positive constant, reads

$$\begin{bmatrix} \int_\Omega \bar{\boldsymbol{B}}^T \bar{\boldsymbol{B}} \, d\Omega & \int_{\Omega_d^+} \bar{\boldsymbol{B}}^T \hat{\boldsymbol{B}} \, d\Omega \\ \int_{\Omega_d^+} \hat{\boldsymbol{B}}^T \bar{\boldsymbol{B}} \, d\Omega & \int_{\Omega_d^+} \hat{\boldsymbol{B}}^T \hat{\boldsymbol{B}} \, d\Omega + \int_{\Gamma_d} \hat{\boldsymbol{N}} k \hat{\boldsymbol{N}} \, d\Gamma \end{bmatrix} \begin{bmatrix} \bar{\boldsymbol{u}} \\ \hat{\boldsymbol{u}} \end{bmatrix} = \begin{bmatrix} \int_\Omega \bar{\boldsymbol{N}}^T f \, d\Omega \\ \int_{\Omega_d^+} \hat{\boldsymbol{N}}^T f \, d\Omega \end{bmatrix}, \tag{22}$$

where $\bar{\boldsymbol{B}}$ and $\hat{\boldsymbol{B}}$ contain derivatives of the shape functions $\left\{\bar{\phi}_i\right\}$ and $\left\{\hat{\phi}_i\right\}$, respectively, $\bar{\boldsymbol{N}}$ and $\hat{\boldsymbol{N}}$ contain shape functions, and $\bar{\boldsymbol{u}}$ and $\hat{\boldsymbol{u}}$ contain the degrees of freedom. Usually, the same finite element basis will be used for the regular and enriched components ($k_1 = k_2$ in Equations (18) and (19)), in which case $\bar{\boldsymbol{B}} = \hat{\boldsymbol{B}}$ and $\bar{\boldsymbol{N}} = \hat{\boldsymbol{N}}$. Details of the matrix formulation and some practical aspects can be found in [19].

4. Automated Modelling of Discontinuities

The development of our automated approach for extended finite element methods can be broken down into three keys parts. Firstly, the domain-specific language for variational forms which we have outlined in Section 2. requires extension. It is necessary that finite element spaces with discontinuities across surfaces can be represented and that terms can be evaluated and integrated on discontinuity surfaces. Secondly, the form compiler must be able to translate the high-level input into low-level computer code that can perform the tasks required at the innermost assembly loop of a finite element program, namely computing the element matrices and vectors. For this, we have extended the form compiler FFC [6, 13]. The third component is the solver. We have constructed a solver on top of the C++/Python library DOLFIN [7, 16]. It is at the solver level that discontinuity surfaces are represented, coefficients of finite element functions that appear in forms (such as w in the case of the weighted Poisson equation) are supplied to the assembler and that the global system of linear equations is assembled and solved. We describe each of the three key software library components in this section. The implementation of the form compiler and the solver components which support the extended finite element method are available in the supporting material [12].

As already alluded, automated code generation offers a number of interesting possibilities. For example, it is possible to employ representations of finite element matrices and vectors which cannot be reasonably coded by hand. As an example, standard element matrices and vectors can be expressed using a "tensor contraction" approach [6] which involves extensive "pre-computation" (prior to runtime), instead of the conventional quadrature-loop representation. Different representations can lead to dramatic differences in performance. The relative performance of different representations depends heavily on the nature of the differential Equation [6, 10]. At first inspection, for the extended finite element method we

are limited in terms of possible representations. This is due to the discontinuity surface being defined globally in terms of the real coordinates, unlike shape functions which are usually defined on reference cells. This eliminates the possibility of using certain approaches, such as the tensor contraction approach, which rely on all functions being defined on a reference element. Deeper inspections may yield interesting possibilities, but for now we limit ourselves to the automated generation of code which uses the conventional quadrature representation. Even for quadrature representations, special optimisations through automation can be applied, for example the automated rearrangement of loops which can have dramatic performance consequences beyond those which can be realised by modern generic compiler optimisations [10]. Automation is particularly amenable to specialised performance optimisations, which is not only because of the possibility of employing special strategies but also the simple regeneration of the code for a particular problem by following improvement to the form compiler.

4.1. Domain-Specific Language Extensions

In the domain-specific language used by the form compiler FFC, we need to include discontinuous finite element function spaces, various operators at discontinuity surfaces and integration over discontinuity surfaces. Inspired by the decomposition of the displacement field in (5), we define "continuous" and "discontinuous" function spaces which will correspond to $\bar{u}_h$ and $\hat{u}_h$, respectively, and a mixed space is then defined as the sum of the two,

```
elem_cont     = FiniteElement(type, shape, order)
elem_discont  = DiscontinuousFiniteElement(type, shape, order)

element       = elem_cont + element_discont
```

where `type` is the finite element type e.g. "Lagrange", "discontinuous Lagrange" (discontinuous across cell facets), or "Raviart-Thomas", among others, `shape` defines the shape of the finite element cell ("interval", "triangle" or "tetrahedron") and `order` is the order of the finite element basis (which is arbitrary). A finite element space which is suitable for modelling problems with discontinuous solutions has been created from the sum of the continuous and discontinuous element spaces. Not all FFC supported finite element spaces are currently compatible with the extended finite element implementation. We will focus on arbitrary order continuous Lagrange elements.

Once the finite element spaces have been defined, test, trial and other functions can be defined on the appropriate spaces. For the Poisson equation presented in Section 3.2., we define the test and trial functions, v and u, respectively, and the source function f,

```
v = TestFunctionPUM(element)
u = TrialFunctionPUM(element)
f = Function(elem_cont)
```

The test and trial functions have been defined on the space of functions containing a discontinuity and the source function `f` is defined on a continuous finite element space.

FFC provides a number of operators for computing terms such as inner products, gradients, the divergence and the transpose [6, 7, 13]. It is possible to use compact tensor or index notation. In the context of the extended finite element method, we define some operators which act at discontinuity surfaces. A common operation at a discontinuity surface is the computation of the jump in the field across the surface. For this, we provide the operator `djump(u)` which is equivalent to $u^+ - u^-$, and the operator `ljump(u, n)` which returns the jump in the vector $\boldsymbol{u}^+ - \boldsymbol{u}^-$, but with the components of the jump relative to the local $n - s$ coordinate system in which the n-direction is normal to the discontinuity surface and the s-direction is tangential to the discontinuity surface. Another ingredient is integration over discontinuity surfaces which is denoted by `*dc`. The range of operators at discontinuity surfaces is currently limited. More elaborate operators, akin to those already available on cell facets [8], are under development.

We have now introduced the necessary language elements to represent a finite element variational formulation of the Poisson problem with a discontinuous solution across a surface and a flux acting at the discontinuity surface which is a function of the solution jump, as presented in Equation (14). For a quadratic Lagrange basis, and the constitutive model for the discontinuity surface flux $q = k \llbracket u \rrbracket$ where $k \geq 0$ is a constant, the form compiler input for this problem reads:

```
# Finite element spaces
elem_cont    = FiniteElement("Lagrange", "triangle", 2)
elem_discont = DiscontinuousFiniteElement("Lagrange", "triangle", 2)
element      = elem_cont + elem_discont

# Test and trial functions
v = TestFunctionPUM(element)
u = TrialFunctionPUM(element)

# Source term
f = Function(elem_cont)

# Interface flux parameter
k = Constant("triangle")

# Bilinear and linear forms
a = dot(grad(v), grad(u))*dx + k*djump(v)*djump(u)*dc
L = v*f*dx
```

The bilinear form is denoted by `a` and the linear form is denoted by `L`. The extended FFC can be called from the command line to generate C++ code from this high-level input. The input to the form compiler mirrors the mathematical formulation of the problem, thereby detaching the mathematical problem from the implementation details. This can dramatically reduce the time required to develop and test new models.

4.2. Specialised Generated Code

From the high-level input, low-level code which is specific to the considered equation is automatically generated by the form compiler. FFC generates C++ code which conforms to the Unified Form-assembly Code (UFC) specification [14, 15] (the generated code conforms to the upcoming version 1.2 of UFC). The UFC specification is a C++ interface for the assembly of variational forms and provides a specification against which automated code generators can work. An assembler that supports the UFC specification can therefore assemble the global matrices and vectors from the generated code without modification.

In the context of the extended finite element method, the computation of element matrices and vectors, and the degree of freedom maps are affected by the discontinuity surfaces. To exploit the UFC specification, we follow a standard C++ polymorphic design and generate subclasses of the classes defined in the UFC specification. The automatically generated classes are initialised with a `GenericPUM` object (the design of this object is described later in the section) which provides the data which is dependent on the presence of discontinuity surfaces and is necessary to build element matrices and vectors for the extended finite element method. The member functions of the generated code which are called during assembly are part of the UFC specification, which is why the global system of equations can be assembled by any assembly code which supports the specification. We outline some key elements of the automatically generated code and describe their purpose. A number of other utility classes and functions in addition to those which we will describe are also generated, such as functions for evaluating basis functions and their derivatives at arbitrary points.

The key classes generated by the form compiler address the degree of freedom maps, cell integrals (which include integration over a discontinuity surface within a cell) and forms, which represent mathematical variational forms. At the highest level of abstraction, given an object `pum`, forms are created. For example, for the Poisson equation, classes which represent the bilinear and linear forms are generated,

```
UFC_PoissonBilinearForm a(pum);
UFC_PoissonLinearForm L(pum);
```

Forms are self-aware of various properties, such as their rank, and are able to create the relevant degree of freedom maps, integral objects and finite elements. Forms are the main interface through which an application developer interacts with the automatically generated code. A generated form is a subclass of the UFC class `ufc::form`. Typically, a form is passed together with a matrix/vector and a mesh to an assembler to build the global matrix/vector. A form object is also able to create degree of freedom maps (which are subclasses of the UFC class `ufc::dof_map`). For the Poisson example, the form compiler will generate code for a class from which an object is created by

```
Poisson_dof_map_0 dof_map(pum);
```

This object is able to tabulate the degree of freedom map for a given cell. The degree of freedom map is aware of the discontinuity and the extra "enriched" degrees of freedom via the object `pum`. Degree of

freedom map classes can perform various tasks, as defined in the UFC specification. The numeral in the class name is a convention in the generated code which indicates the particular degree of freedom map. Problems with multiple fields will involve multiple maps. However, the user is not exposed to this as the degree of freedom maps are accessed via the higher-level form object. This naming convention is followed for other generated classes.

Objects which are able to compute an element matrix or vector are generated by the compiler and are subclasses of `ufc::cell_integral` from the UFC specification. A cell integral object for a Poisson problem can be created by

```
Poisson_cell_integral_0 cell_integral(pum);
```

For each form that appears in a problem (usually there is one bilinear form and one linear form), there will typically be a cell integral class. The key member function of the cell integral class is a "tabulate" function which computes the element matrix/vector. The tabulate function is supplied with a finite element cell and any coefficient functions in the variational problem, and the element matrix or element vector is computed. It is in the implementation of cell integrals and degree of freedom maps that aspects of the extended finite element method in the generated code are most evident. The form compiler determines the appropriate order of quadrature for an integral based on the operators and functions appearing in the form. There are also facilities for specifying the quadrature order manually.

In addition to code for computing degree of freedom maps and tabulating element matrices and vectors, tools which are useful for post-processing are also generated. Most prominent among these are tools for interpolating an extended finite element solution in a continuous space for visualisation purposes.

An important feature of the design of the generated code is that it is independent of the method by which discontinuity surfaces are represented. This emphasises the separation of the mathematical definition of the problem and implementation aspects.

At this stage, limited attention has been paid to specialised optimisations which are possible with automated code generation. There may be considerable scope for this, as reported in [6, 10]. Automated optimisations are particularly pertinent to problems which involve multiple fields, possibly from different finite element spaces and which may or may not be discontinuous, as developing optimised code for such problems is difficult and time consuming.

4.3. Solver Environment

The key tasks of the solver environment are to make use of the generated code, provide representations of discontinuity surfaces, and manage the local extension of the finite element function space and the resulting changes to the number of degrees of freedom. The solver is also responsible for a range of standard tasks, such as linear algebra data structures, linear solvers, application of Dirichlet boundary conditions, mesh management and input/output. The necessary solver components are developed as extensions to the library DOLFIN [7, 16], which is designed to interact with automated code generators. The solver environment facilitates the use of generated code, therefore we provide an overview only.

Surface representation

Surfaces across which finite element functions are discontinuous are represented by the `Surface` class. The interface to `Surface` is designed such that other parts of the library and the generated code are unaffected by the method with which a surface is represented internally. We presently work with simple representations of surfaces, with the initial discontinuity surface represented by a mathematical function. This is not the most general approach as surfaces cannot "double back", but it does permit a wide range of quite elaborate initial discontinuity paths to be considered. In two dimensions, a discontinuity surface (a line) is represented by a user-defined function and the start and end points. In three dimensions, the surface is represented using two level set functions, ϕ and ψ, where for a point on the surface $\phi = 0$ and $\psi < 0$. In both two and three dimensions, the functions used to describe the surface make use of the `Function` abstraction in DOLFIN. The interface to the class `Surface` is designed such that it is dimension-independent.

Once a `Surface` has been created, it can perform a range of geometric tasks, such as determining on which side of the surface a given point lies, finding the intersection point of a straight line and the surface, determining whether a cell is intersected by the surface and computing the ratio between the volumes of an intersected cell on either side of the surface. The design can be extended as extra functionally is required. Multiple surfaces can be stored in standard STL containers, which is useful when dealing with multiple discontinuities. When a crack grows, the surface object which represents the crack surface simply needs to be updated.

The `Surface` interface is not exposed in the generated code, which enforces the desirable separation between the generated code and the surface representation. Rather, only the `GenericPUM` interface, which is described below, is exposed in the generated code. Details of the surface representation can be changed without disruption to other parts of the code. A user can define their own surface representation without needing to modify the form compiler.

Interface to the generated code and local extension of the finite element basis management

The local extension of the finite element basis is controlled by an object derived from the abstract base class `GenericPUM`. Its task is to manage aspects related to the partition of unity approach. The base class `GenericPUM` provides an interface for interacting with the generate code. Key tasks of a `GenericPUM`-based object include the evaluation of the Heaviside function at arbitrary points, computing quadrature schemes for intersected elements based on an input reference scheme which is provided by the generated code and managing which nodes have enriched functions attached. Each discontinuous field in a problem has an associated `GenricPUM` object. For example, for a three-dimensional elasticity problem, there will be a `GenricPUM` object associated with each of the three fields If `GenricPUM` objects are the same for each field, then they can be shared. Partition of unity related data which is required by the generated code is accessed exclusively through the `GenricPUM` public interface.

For modelling discontinuities, we define an object `PUM`, which is a subclass of `GenericPUM` and is initialised with the discontinuity surfaces, a mesh and the standard degree of freedom map (a map for the standard degrees of freedom),

```
std::vector<const Surface*> surfaces;
PUM pum(surfaces, mesh, standard_dof_map);
```

When a discontinuity surface evolves, the `PUM` object simply needs to be updated to reflect the change.

An important feature of `GenericPUM`-based objects is that the public interface is designed such that it only involves arguments and return variables that are standard C++ types or part of the UFC specification. This is important as it allows a `PUM` object to act as the "glue" between the generated code and different finite element libraries. A user of any library which supports UFC can use the generated code by just providing their own sub-class of `GenericPUM`.

To ensure that the jump in a solution goes to zero at the boundary of a discontinuity surface which does not intersect the boundary of the domain, the enriched degrees of freedom are carefully selected close to the boundary of a discontinuity surface, as described in [19].

Function spaces and variational forms

We are now in a position to define function spaces and variational forms which are represented by the DOLFIN classes `FunctionSpace` and `Form`, respectively. A `FunctionSpace` represents the mathematical function space within which a solution is sought, such as those defined in Equation (20) which describe the type of finite element functions. A `FunctionSpace` is also aware of the domain Ω and the degree of freedom map (which in part defines the regularity of the function space). A `Form` is defined by the variational problem and the relevant function space(s). A `Form` can be assembled into a global matrix (in the case of a bilinear form) or a global vector (in the case of a linear form). These concepts and abstractions are explained in detail in [7] for conventional finite element methods. For the Poisson problem presented in Section 3.2., a function space, and bilinear and linear forms are created by

```
Poisson::FunctionSpace V(mesh, pum_objects);
Poisson::BilinearForm a(V, V);
Poisson::LinearForm L(V);
```

with the form objects being initialised with function spaces. The code for the above function space and form classes is generated by the form compiler. The form classes are essentially wrappers for the `UFC_PoissonBilinearForm` and `UFC_PoissonLinearForm` objects and are created to simply the interaction with the DOLFIN environment. For a problem with multiple fields, such as three-dimensional incompressible elasticity, a `PUM` object is associated with each field (one for each displacement component plus one for the pressure field), therefore to create a function space we would have

```
std::vector<const GenericPUM*> pum_objects;
pum_objects.push_back(&pum_u);
pum_objects.push_back(&pum_u);
pum_objects.push_back(&pum_u);
pum_objects.push_back(&pum_p);

Incompressible3D::FunctionSpace V(mesh, pum_objects);
```

Since the displacement components are all discontinuous across the same surface and they share the same finite element type, they can share the same `PUM` object. Usually the finite element field used for the pressure field differs from that used for the displacement components, therefore it requires its own `PUM` object. An incompressible elasticity formulation is elaborated in Section 5..

With the exception of Dirichlet boundary conditions, the finite element variational problem is completely defined by the `Form` objects and the mesh. They can be passed to the assembler to compute the global matrix and vector.

```
Matrix A;
Vector b;
Assembler::assemble(A, a);
Assembler::assemble(b, L);
```

Once the global matrix and vector have been assembled, Dirichlet boundary conditions can be applied to the linear system if necessary, and the system of linear equations can be solved as usual.

5. Examples

A number of examples are presented in this section to demonstrate various aspects of the automated modelling approach for simulating discontinuities. To avoid lengthy and intricate definitions of case-specific function spaces, the variational form of each example is presented for the case of homogeneous Dirichlet boundary conditions, despite the computed problems using more elaborate boundary conditions. For each example we define the forms a and L, and the function space V. The form compiler input is listed in this section for each example and the complete code for each example is available in the supporting material [12]. All parameters which we use are non-dimensional.

The presented examples are linear, although the code generation approach can be used equally for nonlinear problems (see [7]).

5.1. Weighted Poisson Equation

We consider first the weighted Poisson with discontinuities in the solution u across the surfaces Γ_d and fluxes acting on the discontinuity surfaces. The bilinear and linear forms associated with this problem

read:

$$a\left(u_h, v_h\right) = \int_{\Omega \backslash \Gamma_d} w_h \nabla v_h \cdot \nabla u_h \, d\Omega + \int_{\Gamma_d} [\![v_h]\!] \, k \, [\![u_h]\!] \, d\Gamma, \tag{23}$$

$$L\left(v_h\right) = \int_{\Omega} v_h f \, d\Omega, \tag{24}$$

where f is the source term and w_h is the interpolant of the supplied weighting function w in a finite element space. The flux across the discontinuity surface is $q_d = k \, [\![u]\!]$, where $k > 0$. The relevant finite element function space is given in Equation (20). The form compiler input for an extended finite element formulation of this problem is shown in Figure 2 for the case of quadratic basis functions on triangular cells.

Figure 2. Form compiler input for the weighted Poisson equation with discontinuous u.

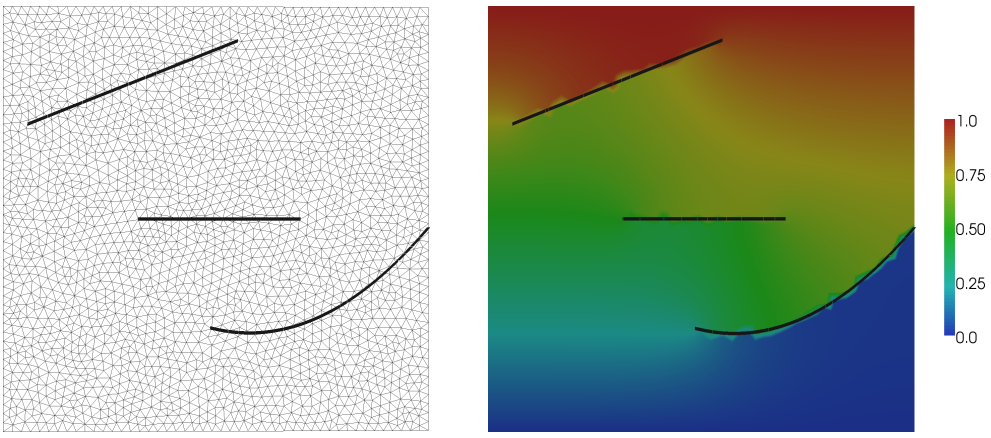
```
elem_cont    = FiniteElement("Lagrange", "triangle", 2)
elem_discont = DiscontinuousFiniteElement("Lagrange", "triangle", 2)
element      = elem_cont + elem_discont

v = TestFunctionPUM(element)
u = TrialFunctionPUM(element)

f = Function(elem_cont)
w = Function(elem_cont)
k = Constant("triangle")

a = w*dot(grad(v), grad(u))*dx + k*djump(v)*djump(u)*dc
L = v*f*dx
```

We present computed results using the form compiler input in Figure 2 for the case $\Omega = (0,1) \times (0,1)$, $f = 1$, $w = 1 + e^{x^2}$ and $k = 10$. Dirichlet boundary conditions are applied along the bottom and top edges ($u_h = 0$ for $y = 0$ and $u_h = 1$ for $y = 1$) of the domain. Along the sides of the domain ($x = 0$ and $x = 1$) a zero flux condition is applied. Three disjoint discontinuity surfaces are included in the domain. Precise details of the crack paths can be found in the supporting material [12]. Having compiled the input file with the form compiler, the generated C++ code serves as input for the solver, which is where the discontinuity surface is defined. For this problem, an extract of the C++ program is shown in Figure 3. The code for the objects `Poisson::BilinearForm`, `Poisson::LinearForm` and `Poisson::PostProcess` has been generated by the form compiler. The other elements of the code which appear in Figure 3 are reusable components which are independent of the considered equation. The computed results for this problem are presented in Figure 4 which shows the discontinuity surfaces, which are not aligned with the mesh, and contours of the solution field in which the jumps in the solution across the discontinuity surfaces can be observed.

Figure 3. C++ code extract for the two-dimensional weighted Poisson problem with discontinuities in the solution.

```
// Create function space
Poisson::FunctionSpace V(mesh, pum_objects);

// Create bilinear and linear Forms
Poisson::BilinearForm a(V, V);
a.k = k; a.w = w;
Poisson::LinearForm L(V);
L.f = f;

// Post processing
Poisson::PostProcess post_process(mesh, pum_objects);

// Create a linear PDE variational problem
dolfin::VariationalProblem pde(a, L, bcs);

// Solve pde
dolfin::Function U(V);
pde.solve(U);

// Interpolate solution for post processing
dolfin::Function U_interpolated(V0);
post_process.interpolate(U.vector(), U_interpolated.vector());

// Save solution and discontinuity surfaces to file for visualisation
pum::VTKFile filep("surface.pvd");
dolfin::File file("poisson.pvd");
file  << U_interpolated;
filep << discontinuities;
```

Figure 4. Weighted Poisson problem in two dimensions: (a) mesh and discontinuity surfaces and (b) solution contours.



(a) (b)

Figure 5. Form compiler input for a three-dimensional linear elasticity problem with discontinuous $\boldsymbol{u}$.

```
elem_cont    = VectorElement("Lagrange", "tetrahedron", 1)
elem_discont = DiscontinuousVectorElement("Lagrange", "tetrahedron", 1)
element = elem_cont + elem_discont

v = TestFunctionPUM(element)
u = TrialFunctionPUM(element)
f = Function(elem_cont)

# Lame parameters
mu    = Constant("tetrahedron")
lmbda = Constant("tetrahedron")

def epsilon(w):
    return 0.5*(grad(w) + transp(grad(w)))

def sigma(w):
    return 2*mult(mu, epsilon(w)) + mult(lmbda, mult(trace(epsilon(w)), Identity(len(w))))

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

5.2. Three-dimensional elasticity

For a linear elasticity problem with homogeneous Dirichlet boundary conditions and traction-free discontinuity surfaces, the bilinear and linear forms read:

$$a\left(\boldsymbol{v}_h, \boldsymbol{u}_h\right) = \int_{\Omega \backslash \Gamma_d} \nabla \boldsymbol{v}_h : C : \nabla^s \boldsymbol{u}_h \, d\Omega, \tag{25}$$

$$L\left(\boldsymbol{v}_h\right) = \int_{\Omega} \boldsymbol{v}_h \cdot \boldsymbol{f} \, d\Omega, \tag{26}$$

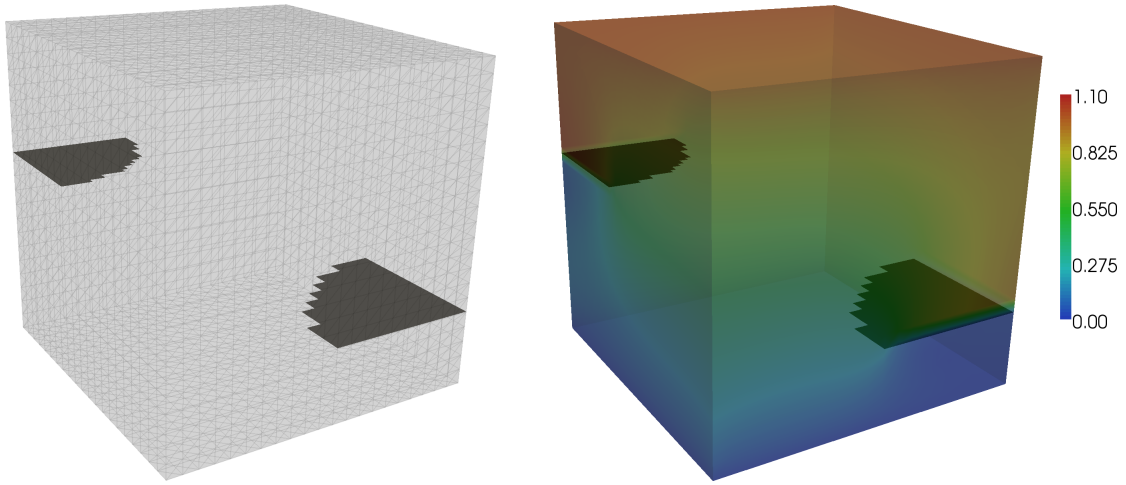
where C is the elasticity tensor, $\nabla^s\left(\cdot\right) = \left(1/2\right)\left(\nabla\left(\cdot\right) + \left(\nabla\left(\cdot\right)\right)^T\right)$ is the symmetric gradient and $\boldsymbol{f}$ is the body force vector. For this problem, the relevant finite element space reads

$$V = \left\{ \boldsymbol{v} \in \left(L^2\left(\Omega\right)\right)^d \cap \left(H_0^1\left(\Omega \backslash \Gamma_d\right)\right)^d, \boldsymbol{v}|_{\Omega_e} \in \left(P^k\left(\Omega_e \backslash \Gamma_d\right)\right)^d \right\}. \tag{27}$$

The form compiler input for this problem in three dimensions using linear tetrahedral elements is shown in Figure 5. The input is dimension independent; "tetrahedron" can simply be changed to "triangle" to change from three dimensions to two dimensions.

A computation has been performed for a unit cube $\Omega = (0,1) \times (0,1) \times (0,1)$, with two discontinuity surfaces. The displacement is prescribed on the top and bottom faces of the cube ($\boldsymbol{u}_h = \boldsymbol{0}$ for $z = 0$ and $\boldsymbol{u}_h = (0,0,1)$ for $z = 1$) and all other faces are traction-free, no body force is applied ($\boldsymbol{f} = \boldsymbol{0}$), Young's modulus $E = 2 \times 10^4$ and Poisson's ratio $\nu = 0.2$. As with the previous example, descriptions of the discontinuity surfaces can be found in the supporting material [12]. The surfaces and the computed displacement contours are illustrated in Figure 6. As expected, the presence of the discontinuity surfaces is reflected in the contours of the magnitude of the displacement field across the discontinuity surfaces.

Figure 6. Three-dimensional elasticity problem. The mesh on the surface of the cube is shown in (a) and the displacement magnitude contours are shown in (b). The discontinuity surfaces are visible in both figures.



(a) (b)

5.3. Incompressible elasticity with cohesive discontinuity surfaces

Incompressible elasticity (or Stokes flow) is usually posed as a mixed problem with the displacement and pressure fields as unknowns. In the context of finite element analysis, stability requirements pose restrictions on the allowable combination of finite element spaces for the displacement and pressure fields. For example, it is well known that using equal order Lagrange basis functions for the displacement and pressure fields leads to an unstable formulation. The Taylor-Hood family of elements are examples of stable elements for incompressible elasticity, and involve a continuous piecewise quadratic basis for the displacement field and continuous piecewise linear basis for the pressure field on simplices. A common criticism of Taylor-Hood elements is that they are difficult to implement due to the different order basis functions. If this objection is accepted, it is compounded when considering discontinuous solutions. One has to deal with not only different basis functions, but also the enriched degrees of freedom for the discontinuous fields. Furthermore, one may wish to use, and perhaps switch between, combinations of continuous and discontinuous fields. At the very least, these issues require careful software design. The complexity argument is completely obviated by automated code generation as one can use arbitrary combinations of basis functions trivially, as well as switch between combinations of continuous and discontinuous functions with minimal effort, as we will illustrate. Incompressible elasticity serves as a demonstration of the ease with which multi-physics problems can be dealt with using automated code generation.

We consider two incompressible elasticity formulations, both involving a displacement field which is discontinuous across surfaces. The first formulation involves a pressure field which is permitted to be discontinuous across surfaces, and the second formulation involves a pressure field which is continuous

across surfaces. In both cases, the problem is expressed abstractly as: find $\boldsymbol{u}_h \in V$ and $p_h \in Q$ such that

$$a\left(\boldsymbol{v}_h; q_h, \boldsymbol{u}_h; p_h\right) = L\left(\boldsymbol{v}_h; q_h\right) \quad \forall \boldsymbol{v}_h \in V,\ q_h \in Q. \tag{28}$$

For a problem with homogeneous Dirichlet boundary conditions on the displacement, the bilinear and linear forms read:

$$a\left(\boldsymbol{v}_h; q_h, \boldsymbol{u}_h; p_h\right) = \int_{\Omega \backslash \Gamma_d} \nabla \boldsymbol{v}_h \cdot 2\mu \nabla^s \boldsymbol{u}_h - \left(\nabla \cdot \boldsymbol{v}_h\right) p_h + q_h \nabla \cdot \boldsymbol{u}_h \, d\Omega + \int_{\Gamma_d} [\![\boldsymbol{v}_h]\!] \cdot \boldsymbol{t} \, d\Gamma, \tag{29}$$

$$L\left(\boldsymbol{v}_h; q_h\right) = \int_{\Omega} \boldsymbol{v}_h \cdot \boldsymbol{f} \, d\Omega, \tag{30}$$

where μ is the shear modulus, $\boldsymbol{t}$ is the traction acting on the discontinuity surface and $\boldsymbol{f}$ is a source term.

For the case in which both the pressure and displacement fields are discontinuous across a discontinuity surface, the relevant functions spaces for the Taylor-Hood element read

$$V = \left\{ \boldsymbol{v} \in \left(L^2\left(\Omega\right)\right)^d \cap \left(H_0^1\left(\Omega \backslash \Gamma_d\right)\right)^d, \boldsymbol{v}|_{\Omega_e} \in \left(P^2\left(\Omega\right) \backslash \Gamma_d\right)^d \forall e \right\}, \tag{31}$$

$$Q = \left\{ p \in L^2\left(\Omega\right) \cap H^1\left(\Omega \backslash \Gamma_d\right), p|_{\Omega_e} \in P^1\left(\Omega \backslash \Gamma_d\right) \forall e \right\}. \tag{32}$$

For the case in which the displacement field is discontinuous and the pressure field is continuous, the pressure space Q requires re-definition:

$$Q = \left\{ p \in H^1\left(\Omega\right), p|_{\Omega_e} \in P^1\left(\Omega\right) \forall e \right\}. \tag{33}$$

The difference between the two formulations is subtle. Inspection of the Euler-Lagrange equations associated with the variational forms shows that in both cases

$$\left(2\mu \nabla^s \boldsymbol{u}^+ - p^+ \boldsymbol{I}\right) \boldsymbol{n}^+ = \boldsymbol{t}, \tag{34}$$

$$\left(2\mu \nabla^s \boldsymbol{u}^- - p^- \boldsymbol{I}\right) \boldsymbol{n}^- = \boldsymbol{t}, \tag{35}$$

which implies that weak continuity of the traction across discontinuity surfaces is enforced. For the continuous pressure case, this can be rephrased as enforcing

$$[\![2\mu \nabla^s \boldsymbol{u}]\!] \, \boldsymbol{n}^+ = \boldsymbol{0} \tag{36}$$

in a weak sense, which is the viscous part of the traction (continuity of pressure contribution is enforced point-wise by construction). For the discontinuous pressure case we have

$$[\![2\mu \nabla^s \boldsymbol{u} - p\boldsymbol{I}]\!] \, \boldsymbol{n}^+ = \boldsymbol{0}, \tag{37}$$

which implies weak continuity of the total traction across the discontinuity surface. There is also a subtle difference between the two formulations in terms of how the incompressibility constraint is enforced, but since the Taylor-Hood element does not enforce $\nabla \cdot \boldsymbol{u} = 0$ element-wise, this is unlikely to be of consequence.

We now show how one can switch trivially between discontinuous and continuous pressure formulations with a Taylor-Hood element using the form compiler. For a problem in which the traction acting on the discontinuity surface is made a function of the displacement jump across the surface according to

$$\boldsymbol{t} = \begin{bmatrix} t_n \\ t_s \end{bmatrix} = \begin{bmatrix} K_{nn} & 0 \\ 0 & K_{ss} \end{bmatrix} \begin{bmatrix} [\![u_h]\!]_n \\ [\![u_h]\!]_s \end{bmatrix}, \tag{38}$$

Figure 7. Form compiler input for incompressible elasticity with discontinuous $\boldsymbol{u}$ and p.

```
P2_cont = VectorElement("Lagrange", "triangle", 2)
P2_dis  = DiscontinuousVectorElement("Lagrange", "triangle", 2)

P1_cont = FiniteElement("Lagrange", "triangle", 1)
P1_dis  = DiscontinuousFiniteElement("Lagrange", "triangle", 1)

P2 = P2_cont + P2_dis
P1 = P1_cont + P1_dis
TH = P2 + P1

(v, q) = TestFunctionsPUM(TH)
(u, p) = TrialFunctionsPUM(TH)

f  = Function(P2_cont)                      # source term
mu = Constant("triangle")                   # shear modulus

n  = DiscontinuityNormal("triangle")        # discontinuity normal
Ks = Constant("triangle")                   # tangential stiffness
Kn = Constant("triangle")                   # normal stiffness

traction = [Kn*ljump(u, n)[0], Ks*ljump(u, n)[1]]
stress = mult(mu, (grad(u) + transp(grad(u))))

a = (dot(grad(v), stress) - div(v)*p + q*div(u))*dx \
   + dot(ljump(v, n), traction)*dc
L = dot(v, f)*dx
```

where the subscripts "n" and "s" denote normal and tangential components relative to the discontinuity surface, the form compiler input for the problem with discontinuous pressure is shown in Figure 7.

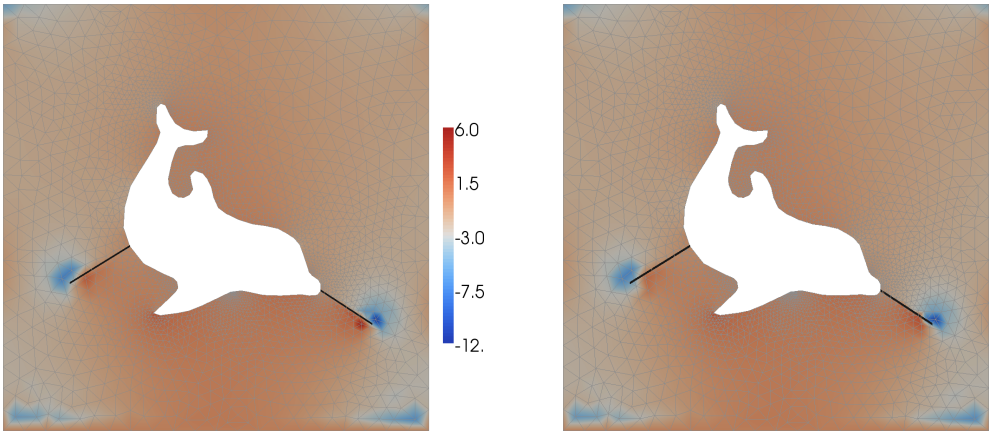
A quadratic Lagrange vector function space and a linear Lagrange function space, both of which include discontinuities, are defined and these are used for the displacement and pressure fields, respectively. The two spaces are combined to create a Taylor-Hood element and the definition of the variational forms follows directly from Equations (29) and (30). To switch to a formulation with continuous pressure, only the function spaces in the input to the form compiler, shown in Figure 7, need to be changed. The form compiler declarations of the function spaces for a Taylor-Hood formulation with a discontinuous displacement field and a continuous pressure field are shown in Figure 8.

Using both formulations, a two-dimensional domain with sides of unit length and an internal cut-out section has been modelled. Two discontinuity surfaces are present. The displacement is prescribed on the bottom edge ($\boldsymbol{u}_h = \boldsymbol{0}$ for $y = 0$) and the top edge ($\boldsymbol{u}_h = (0, 1)$ for $y = 1$). All other faces are traction-free. The shear modulus $\mu = 1$ and for the interface traction $K_{nn} = 2$ and $K_{ss} = 0$. The computed pressure fields are shown in Figure 9. It appears from inspection of the results that differences between the two simulations are small, as expected based on the examination of the Euler-Lagrange equations for the two problems.

Figure 8. Function space definitions for the form compiler input for incompressible elasticity with discontinuous $\boldsymbol{u}$ and continuous p. The definition of the functions and forms is identical to that in Figure 7.

```
P2_cont = VectorElement("Lagrange", "triangle", 2)
P2_dis  = DiscontinuousVectorElement("Lagrange", "triangle", 2)

P1      = FiniteElement("Lagrange", "triangle", 1)

P2 = P2_cont + P2_dis
TH = P2 + P1
```

Figure 9. Pressure fields for (a) the continuous pressure case and (b) the discontinuous pressure case.



(a) (b)

6. Conclusions

We have demonstrated an automated code generation approach for the extended finite element method. A domain-specific language has been extended so that the equation of interest, in a variational format, can be expressed in a high-level language, and low-level C++ code is automatically generated through the form compiler. The approach facilitates the rapid development of new models that involve discontinuities and provides scope for special optimisations not tractable by hand. Furthermore, expressing the equation of interest in an abstract form provides separation between the considered model (the differential equation and the function spaces) and aspects of the implementation. In practice, the approach reduces the implementation burden for developers. A number of concrete examples have been presented in which problems that involve multiple fields with different finite element bases are addressed. The input language is extensible, so a wider range of discontinuity-specific operators can be developed to deal with a wider range of problems.

Acknowledgements

MN acknowledges the support of the Netherlands Technology Foundation STW, the Netherlands Organisation for Scientific Research and the Ministry of Public Works and Water Management.

References and Notes

1. Melenk, J. M.; Babuska, I. The partition of unity finite element method: Basic theory and applications. *Comput. Method. Appl. Mech. Eng.* **1996**, *139*, 289–314.
2. Moës, N.; Dolbow, J.; Belytschko, T. A finite element method for crack growth without remeshing. *Int. J. Numer. Method. Eng.* **1999**, *46*, 231–150.
3. Belytschko, T.; N. Moës, S. U.; Parimik, C. Arbitrary discontinuities in finite elements. *Int. J. Numer. Method. Eng.* **2001**, *50*, 993–1013.
4. Strouboulis, T.; Copps, K.; Babuška, I. The generalized finite element method. *Comput. Method. Appl. Mech. Eng.* **2001**, *190*, 4081 – 4193.
5. Bordas, S.; Nguyen, P. V.; Dunant, C.; Guidoum, A.; Nguyen-Dang, H. An extended finite element library. *Int. J. Numer. Method. Eng.* **2007**, *71*, 703–732.
6. Kirby, R. C.; Logg, A. A compiler for variational forms. *ACM Trans. Math. Software* **2006**, *32*, 417–444.
7. Logg, A.; Wells, G. N. DOLFIN: Automated finite element modelling, 2009. Available online: http://www.dspace.cam.ac.uk/handle/1810/214787.
8. Ølgaard, K. B.; Logg, A.; Wells, G. N. Automated code generation for discontinuous Galerkin methods. *SIAM J. Sci. Comput.* **2008**, *31*, 849–864.
9. Rognes, M. E.; Kirby, R. C.; Logg, A. Efficient assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming finite elements **2009**. Submitted.
10. Ølgaard, K. B.; Wells, G. N. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Software* **2010**, *37*. Available online: http://www.dspace.cam.ac.uk/handle/1810/218613.
11. FEniCS. FEniCS Project 2009. Available online: http://www.fenics.org/.
12. Nikbakht, M.; Wells, G. N. Supporting material 2009. Available online: http://www.dspace.cam.ac.uk/handle/1810/218650.
13. Logg, A.; others. FEniCS Form Compiler 2009. Available online: http://www.fenics.org/ffc.
14. Alnæs, M. S.; Logg, A.; Mardal, K.-A.; Skavhaug, O.; Langtangen, H. P. *UFC Specification User Manual*, **2009**. Available online: http://www.fenics.org/ufc/.
15. Alnæs, M. S.; Logg, A.; Mardal, K.-A.; Skavhaug, O.; Langtangen, H. P. Unified framework for finite element assembly. *Int. J. Computat. Sci. Eng.* **2009**. Available online: http://simula.no/research/scientific/publications/Simula.SC.96/simula_pdf_file.
16. Logg, A.; Wells, G. N.; others. DOLFIN 2009. Available online: http://www.fenics.org/dolfin.
17. Babuška, I.; Melenk, J. M. The Partition of Unity Method. *Int. J. Numer. Method. Eng.* **1997**, *40*, 727–758.
18. Belytschko, T.; Black, T. Elastic crack growth in finite elements with minimal remeshing. *Int. J. Numer. Method. Eng.* **1999**, *45*, 601–620.

19. Wells, G. N.; Sluys, L. J. A new method for modelling cohesive cracks using finite elements. *Int. J. Numer. Method. Eng.* **2001**, *50*, 2667–2682.