Automated Solution of Partial Differential Equations with Discontinuities using the Partition of Unity Method

Automated Solution of Partial Differential Equations with Discontinuities using the Partition of Unity Method

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,
voorzitter van het College van Promoties,
in het openbaar te verdedigen op maandag 17 december 2012 om 15.00 uur

door

Mehdi NIKBAKHT

Master of Science in Structural Engineering, Sharif University of Technology
geboren te Mianeh, Iran

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. L. J. Sluys

Copromotor:
Dr. G. N. Wells

Samenstelling promotiecommissie:

Rector Magnificus	Voorzitter
Prof. dr. ir. L. J. Sluys	Technische Universiteit Delft, promotor
Dr. G. N. Wells	University of Cambridge, copromotor
Prof. dr. ir. C. Vuik	Technische Universiteit Delft
Prof. dr. H. P. Langtangen	Universitetet i Oslo
Prof. dr. M. A. Hicks	Technische Universiteit Delft
Dr. ir. R. B. J. Brinkgreve	Technische Universiteit Delft
Dr. A. Simone	Technische Universiteit Delft
Prof. dr. A. V. Metrikine	Technische Universiteit Delft, reservelid

This research has been supported by the Dutch Technology Foundation (STW) and the Ministry of Public Works and Water Management under grant number 06368.

Keywords: partition of unity method, automatic code generation, partial differential equations, discontinuities, object oriented library, compiler, finite element methods

The thesis cover is designed by M. Sahebi Afzal. The dolphin mesh used in the cover is created by M. Rogners and it is avaibale inside the DOLFIN library.

Printed by Ipskamp Drukkers B.V., Enschede, The Netherlands

ISBN: 978-94-6191-548-1

Acknowledgement

This PhD thesis is a formal closure to more than twenty three years of my life as a student, starting from a city in the north-west of Iran and finishing in a city in the south-west of the Netherlands. These years filled with memories; Memories identified with people; People who played a significant role in my student life. It is my great pleasure to begin my thesis with acknowledgements.

First and foremost, I would like to thank Garth Wells for giving me the opportunity to work with him and under his supervision. Although Garth moved to Cambridge in the second year of my PhD, he continued his supervision beside all his activities in Cambridge. I am very grateful for his invaluable guidance, our discussions and his insightful comments on my manuscripts. Garth is very enthusiastic about his work and he also communicates his enthusiasm to those connected to him. I am sure I would not have been able to finish this thesis without his help and remarkable ideas.

Also, I would like to thank Prof. Bert Sluys for his support during my PhD, reading my thesis as my promoter, and for creating a very flexible environment for the members of his group. I always had Bert's support during different periods of my PhD. Bert is not only a leading scientist but also a true gentleman. The other members of my thesis committee are gratefully acknowledged for reading the thesis, providing useful comments and being present in my defense session. It is my privilege to have Prof. Hans Peter Langtangen, Prof. Kees Vuik, Prof. Michael Hicks, Dr. Ronald Brinkgreve, Dr. Angelo Simone, Prof. Andrei Metrikine in my thesis committee. The financial support from the Dutch Science Foundation and the Ministry of Public Works and Water Management are also acknowledged.

My special thanks go to Kristian Ølgaard who was working in the computational mechanics group on the FEniCS project as I was. After Garth's departure to Cambridge, we were only guys in the group working on the automatic code generation topics. I always had nice and fruitful discussions with Kristian and I liked his organized way of working. I also want to acknowledge support from the FEniCS community all across the world most notably guys from Simula in Oslo.

I really enjoyed my time in the computational mechanics group which was an international group full of nice people. It was always a wonderful opportunity for me to learn about different cultures and countries. I would like to gratefully thank Zahid Shabir, Frans van der Meer, Mojtaba Talebian, Oriol Lloberas Valls, Peter Moonen, Ronnie Pedersen, Frank Radtke, Xuming Shan, Mohammad Mahdi Banatehrani, Amin Karamnejad, Edlira Kondo, Vinh Phu Nguyen, Awais Ahmed, Aliyeh Alipour,

Roberta Bellodi, Prithvi Mandapalli, Adriaan Sillem, Jitang Fan, Nghi Le, Mehdi Musivand, Tien Dung Nguyen, Mirella Villani, Cor Kasbergen, Rafid Al-khoury, Angelo Simone, Frank Everdij, Jaap Weerheijm, Marjon van der Perk and Anneke Meijer for their friendship, support and help during my PhD. Frans also accepted my request to translate the summary and prepositions of my thesis to Dutch. I know how difficult it can be when it comes to a 13th-century Persian poem. I would like to appreciate this favor and his friendship during last few years.

During my PhD, I had numerous visits to the University of Cambridge. I would like to acknowledge the Engineering Department and Jesus College in Cambridge for providing me places to work and live. I would like to thank all my friends and colleagues in Cambridge who helped me to enjoy my stays there. Special thanks goes to Hamed Nili in this regard.

Moving to a foreign country is always challenging. I would also like to highly thank Mohammad Ali Abam and Amir Hossein Ghamarian who helped me to settle smoothly in the Netherlands and they continued their support when I needed any help. I express my best thanks to my Iranian friends all over the globe which surely I will miss some if I want to mention all the names. I would like to thank the members of our bi-weekly gatherings in Delft which I learned a lot with our discussions about different topics. I also appreciate Mahmoud Sahebi Afzal for being a good friend and designing the elegant cover of this thesis.

My professors at Sharif University of Technology taught me basics of structural mechanics and introduced me to the wonderful world of academic research. For that, my best thanks go to Prof. Amir Reza Khoei and Prof. Vahid Khonsari. I also want to appreciate Heydar Zandiyeh and all my teachers in Roshd high school who helped me to discover my abilities and provided an environment to grow.

Last but certainly not least comes my family. I think that now, at the end of my PhD studies, would be the right moment to express my deepest gratitude to my parents and my brother, Meisam for their unconditional support, encouragement, love, faith and prayers to God in me throughout my whole life. I am sure that I could not be here without them and I missed them a lot when I moved to the Netherlands. I also want to thank my parents in-law and sister in-law. Their support, trust and prayers to God during the last few years have been an invaluable asset for me. Finally, I would like to thank my wife, Somayeh. Her support, encouragement, patience and unconditional love were undeniably the bedrock upon which my life has been built. I would like to appreciate her understanding during the busy period of writing my thesis. For sure, she is my greatest achievement from Delft. I dedicate this thesis to my family, with love and gratitude.

Mehdi Nikbakht
Eindhoven, October 2012

Contents

Chapter 1 Introduction

1.1 Background

Numerical solution of partial differential equations with discontinuities is important in a wide range of physical problems. A well-known example in solid mechanics is the modelling of the propagation of cracks. Modelling shear-bands, dislocations and material inclusions are other examples in solid mechanics. Modelling shocks and interfaces in multi-phase flows can also be classified as problems whose solutions are discontinuous in fluid mechanics. In electromagnetism, discontinuous solutions may also happen along the boundaries between materials with different electromagnetic properties.

Using the finite element method (Hughes, 2000; Cook et al., 2002; Zienkiewicz et al., 2005) for the numerical modeling of evolving discontinuities across *a priori* unknown surfaces involves considerable challenges. In early implementations of finite element models for problems with discontinuity surfaces, the main focus was on mesh adaptation to construct meshes that conformed to discontinuity surfaces (Ingraffea and Saouma, 1985; Swenson and Ingraffea, 1988). Not only is generating a mesh compatible with discontinuity surfaces a challenging task in developing the finite element models, but computed solutions of such models may also suffer from inaccuracy and mesh-dependency (Bažant, 1976; de Borst et al., 1993). Moreover, updating the mesh to capture the solution is inevitable for evolving discontinuities (Camacho and Ortiz, 1996). The remeshing becomes cumbersome, time consuming and a computationally demanding task especially for three-dimensional problems (Carter et al., 1997). Furthermore, for nonlinear problems, it may be necessary to transfer data between different meshes many times which is expensive and it can considerably decrease accuracy of solutions (Bittencourt et al., 1992; Tijssens et al., 2000a,b).

To overcome the draw-backs related to modeling problems with discontinuity surfaces using the classical finite element method, a new approach, the so-called continuous/discontinuous finite element methods, has been developed (Ortiz et al., 1987; Belytschko et al., 1988; Belytschko and Black, 1999). In this approach, specific kinematics have been added to the classical finite element approximations to capture discontinuities. This new approach essentially consists of enriching a standard smooth finite element basis, with additional (discontinuous) functions, devised for capturing physical discontinuities. These additional functions are selected to take

advantages of the information that is already known about the expected behavior of discontinuities (e.g. Heaviside function for problems whose solutions exhibit jumps). This approach decouples topology of a mesh from discontinuity surfaces; therefore, no special treatment in the mesh discretization is required for subdomains containing discontinuities.

As for the enriching techniques, two broad families can be distinguished in terms of enrichment strategies. A first family, the so-called embedded finite element methods, contains methods in which elemental enrichments are performed. The enrichment functions are defined on the local enhanced degrees of freedom for each element and these enhanced degrees of freedom are removed by static condensation prior to the global tensor assembly. For this reason, no new global degrees of freedom are introduced to computational domains and the total number of degrees of freedom does not change. In these methods, discontinuity surfaces are embedded in finite elements without considering them in mesh generation stages. The embedded finite element methods, in which enrichment functions are added locally for each element intersected by discontinuities and remain at the element level, were inspired by a work of Ortiz et al. (1987). Their model could capture the behavior of one weak discontinuity line crossing a finite element. Belytschko et al. (1988) proposed another formulation which could capture a softening band between two parallel weak discontinuity lines within an element. The idea was also used in modeling strong discontinuity surfaces (Simo and Oliver, 1994; Dvorkin et al., 1990; Klisinski et al., 1991).

Many instances of the embedded finite element methods exist in literature (Belytschko et al., 1988; Simo et al., 1993; Lotfi and Shing, 1995; Larsson et al., 1996; Oliver, 1996; Sluys and Berends, 1998; Wells and Sluys, 2001b). Jiràsek (2000a) performed a comparative study on these methods. He showed that individual models are different in many aspects, e.g. the type of parent element, the type of discontinuity (weak/strong) and constitutive laws. He divided the embedded finite element methods into three different groups which differ in traction continuity conditions and kinematical descriptions of discontinuity surfaces.

However, a number of problems have been experienced in using the embedded finite element methods to model domains with discontinuities (Jiràsek, 2000b; Jiràsek and Belytschko, 2002). A first problem is using the elemental enrichments that are defined on the internal degrees of freedom corresponding to the jump over discontinuity surfaces. The enrichment functions are discontinuous not only on the surfaces but also at the boundaries of elements intersected by discontinuities. This leads to a non-conforming formulation in which the compatibility of strain fields is not satisfied and it is only enforced in a weak sense.

Another problem is related to the lack of the kinematic decoupling of the embedded finite element formulations. Using the enrichment functions, arbitrary displacement

jumps can be reproduced in the embedded finite element methods. However, the strains on both sides of the discontinuity surface are still coupled (in elements that are crossed by the discontinuity surface).

This limitation has severe implications and it has the consequence that even after complete failure (formation of a stress-free crack), the strain field approximations in the two parts of the element intersected by a discontinuity are not independent. For example, using a constant-strain triangle, the strains in these two parts are approximated by the same constant tensor. Of course, a higher-order formulation with a spatially variable strain approximation can be used to increase the decoupling, but a certain bond always remains that prevents the modeling of the two separated material bodies in full generality.

The uniqueness and numerical robustness of solutions are a third problem for the embedded finite element methods. The additional enrichment degrees of freedom have an internal character; thus, they can be eliminated on the element level by special treatments. Although this elimination has an advantage from the computational point of view, because the number of global degrees of freedom remains the same, it introduces numerical problems for the embedded finite element methods. In order to avoid these numerical problems, special attention must be devoted to element sizes and orientation of elements with respect to discontinuity surfaces.

To overcome these problems, another family of enriching techniques has been introduced for modeling problems with discontinuities. This family covers methods in which the idea of the nodal enrichment using the Partition of Unity (PU) concept (Babuška et al., 1994) is applied. In this approach, the discontinuity surfaces are modeled by enriching the classical polynomials with special functions that are defined on additional degrees of freedom, called enriched degrees of freedom. These enriched degrees of freedom are added globally to the discretized system; therefore, they increase the total number of degrees of freedom. Special attentions must be devoted to handle entries corresponding to the new degrees of freedom in the assembly stage.

Babuška and Melenk (1996, 1997) developed a method based on the partition of unity concept. In their method, they used the global enrichments to improve the finite element approximation properties in the entire domain in comparison to the classical finite element approximations. They showed that a partition of unity formulation can be constructed using finite element basis functions and the quadrature of weak formulations. They utilised the global enrichments to approximate solutions of the Helmholtz equation and the Laplace equation. The enrichment for capturing locally non-smooth phenomena for boundary layers was also briefly discussed in their work.

Later on, local partition of unity enrichment functions have been used to model problems with discontinuity surfaces. The eXtended Finite Element Method (XFEM) (Belytschko and Black, 1999; Moës et al., 1999) and the Generalized Finite

Element Method (GFEM) (Strouboulis et al., 2000b, 2001) are two examples of the local enrichments. In these methods, the local enrichments have been used in subdomains around discontinuities and special numerical integration algorithms have been utilised for cells intersected by discontinuities. Note that the eXtended Finite Element Method and the Generalized Finite Element Method have similar formulations and their different names are mainly because of historical reasons.

1.2 Motivations and objectives

With the introduction of the partition of unity enrichment methods and their applications in modeling physical problems with discontinuous solutions in the last decade, the computational technology for the modelling of these types of problems is now maturing. However, the implementation of these techniques can be tedious, difficult and requires a significant investment of time, especially for coupled nonlinear problems in which different combinations of continuous and discontinuous function spaces might be used. Therefore, the application of the partition of unity enrichment methods is mainly limited to a small group (e.g. computational scientists) who can develop finite element software rather than a broader group (e.g. engineers) that uses the computational technology.

A limited number of finite element libraries which support the partition of unity enrichment methods are available (see for example Bordas et al. (2007), Giner et al. (2009) and Chamrová and Patzák (2010)). These libraries follow the traditional paradigm in which a user is required to program by hand the innermost parts of a finite element solver.

To overcome the cumbersome and time consuming task of translating a partial differential equation to a discretized system of algebraic equations, the automatic generation of code is a possibility (Kirby and Logg, 2006, 2007; Logg et al., 2012a). In the automatic code generation approach, the required code for the innermost assembly loop in the finite element methods is generated automatically using a compiler approach. The compiler approach hides implementation details from users by providing an interface which mimics mathematical formulations.

One of the novel projects which widely uses the compiler approach is the FEniCS project (Logg et al., 2012f). FEniCS relies on the automatic code generation of finite element models and facilitates modelling complex problems by removing the need for a hand-generated code for the discretized systems of finite element formulations. This approach improves the speed and efficiency of implementing different finite element models. FEniCS supports not only conforming Lagrange formulations but also discontinuous Lagrange formulations. Moreover, a wide range of finite elements are also supported within the FEniCS project.

The objective of this work is to design a general, efficient, simple and reliable

framework to model problems whose solutions exhibit jumps over surfaces (strong discontinuities) in an automated way. This automated framework can facilitate the modelling of discontinuities for a wide range of physical problems by the automatic code generation approach for users of computational technology. It uses available tools from FEniCS and extends them to provide required functionalities to support the partition of unity enrichment methods. In summary, the following goals should be achieved through the automated framework:

- a detachment of underlying partial differential equations (PDEs) from the partition of unity implementation details;
- uncoupling surface representations from the rest of the finite element implementation to allow testing various competing representations with a minimum rework;
- a rapid development of different models for the simulation of discontinuity surfaces in a wide range of physical problems using Lagrange/non-Lagrange families of finite element function spaces;
- providing a framework to use different enrichment strategies easily; and
- a fast implementation of different combinations of continuous/discontinuous finite element spaces for modeling discontinuity surfaces in coupled problems.

The implementation of the automated framework for modeling problems with discontinuity surfaces is divided into two components: a form compiler and a solver library. The form compiler is used to generate PDE–specific low-level code using an input representing a variational formulation. The generated code is then used to assemble element tensors and nodal mapping required for the partition of unity enrichment methods inside a solver. The solver library provides information about discontinuity surfaces, meshes, boundary conditions and coefficient functions. It also solves variational problems and post processes results. In the case of evolving discontinuity surfaces, the evolution criteria are also defined inside the solver. Both the compiler and the solver library are licensed as open–source software and they can be downloaded from the FEniCS project website (Logg et al., 2012f).

1.3 Thesis outline

This thesis is organized as follows. Chapter 2 elaborates the partition of unity methods in modelling problems with discontinuity surfaces and provides literature reviews on the application of these methods. The implementation of these methods is challenging and careful attention must be devoted to designing corresponding

software packages. Implementation aspects specific for the partition of unity method are explained at the end of this chapter. An overview of the automation of computational mathematical modelling is given in Chapter 3. In this chapter, the main focus is on the FEniCS project and its key components are explained. At the end of this chapter, a number of examples of solving different partial differential equations are presented to show the versatility and possibility of using FEniCS to model different physical problems. The partition of unity compiler is the first component of the automated framework. The structure and interface of the compiler are explained in Chapter 4. This compiler is built on top of the FEniCS From Compiler (Logg et al., 2012b) and generates the required low-level code to model problems with discontinuity surfaces in the partition of unity framework. The structure of the generated code using the partition of unity compiler is then elaborated. Chapter 5 explains the other component of the proposed framework, which is the partition of unity method library. The solver library provides components which can use the automatically generated code to model discontinuity surfaces. The key components of the library are explained. In Chapter 6, modeling of discontinuities in a wide range of two- and three-dimensional physical problems is presented. Finally, this thesis is closed by conclusions and suggestions for future work in Chapter 7.

Chapter 2 Partition of unity methods

To allow discontinuity surfaces to evolve independently from the mesh topology, the traditional finite element formulations have been extended. This extension is achieved by adding enrichment functions to the standard approximations in the partition of unity context. The enrichment functions are defined on new degrees of freedom and they change the structure of the discretized system of variational equations. Enriching the standard finite element enables the modelling of problems by finite elements with no explicit meshing of discontinuity surfaces. This facilitates the development of finite element models for physical problems with discontinuity surfaces especially in the case of evolving surfaces.

Partition of unity enrichment methods (Belytschko and Black, 1999; Moës et al., 1999; Strouboulis et al., 2000b, 2001) have been widely used for the analysis of static and propagating discontinuities in different physical problems. In these methods, no restriction exists on the type of underlying finite element spaces and the continuity of displacement jumps across element boundaries is satisfied. (in contrast to the embedded finite element methods (Simo et al., 1993; Lotfi and Shing, 1995; Larsson et al., 1996; Oliver, 1996; Wells and Sluys, 2001b)).

This chapter continues by a literature review on the application of the partition of unity enrichment methods. Next, a partition of unity formulation for an elasticity problem is presented in a domain in which the solution exhibits a jump over a discontinuity surface. The structure of the discretized system of algebraic equations and required extensions in comparison with the standard finite element approximation are elaborated. At the end, implementation aspects specific to the partition of unity enrichment methods are discussed.

2.1 Application

Using the partition of unity enrichment can dramatically simplify modelling problems with discontinuity surfaces. The success of using these methods in modelling challenging topics like cracks has motivated their applications for other physical problems containing discontinuity surfaces. To illustrate this, a brief literature overview on the application of the partition of unity methods is given in solid mechanics and fluid mechanics. For a more complete overview, interested readers are referred to review papers by Yazid et al. (2009), Belytschko et al. (2009) and Fries and Belytschko (2010).

2.1.1 Applications in solid mechanics

In solid mechanics, modelling problems characterized by discontinuities, singularities, localized deformations and complex geometries using the partition of unity approaches can be found in literature. In the following, a literature review on the modelling of cracks, frictional contacts and grain boundaries is given to show the diversity of problems which can be tackled using the partition of unity enrichment methods in the field of solid mechanics.

Cracks The local enrichment in the partition of unity concept was applied to fracture mechanics in a paper by Belytschko and Black (1999). They used enrichment functions obtained from the asymptotic solutions at crack tips for the entire crack length. The idea of enriching cracks with the Heaviside function was introduced in Moës et al. (1999), besides using asymptotic enrichment functions just in crack tips. In this work, the method's name was also coined as XFEM. This method was later extended to model branched and intersecting cracks in Daux et al. (2000).

XFEM was extended for the modelling of three-dimensional cracks by Sukumar et al. (2000). This method was also employed in modeling propagating three-dimensional cracks in Areias and Belytschko (2005) and Gasser and Holzapfel (2005). Using a similar approach, referred to as GFEM, Duarte et al. (2001) also simulated three-dimensional dynamic crack propagation.

Cohesive cracks can also be modeled using XFEM. Wells and Sluys (2001a) used the step enrichment function to model cohesive cracks in simulating fracture in quasi-brittle heterogeneous materials. In their implementation, crack tips were limited to element edges. In Moës and Belytschko (2002) and Zi and Belytschko (2003), near crack-tip enrichment functions in addition to the Heaviside function were used to model cohesive cracks. This allowed crack tips to be located anywhere within elements.

Remmers et al. (2003) have proposed a method for cohesive cracks where discontinuities are inserted element-wise. This method eliminates the need for the definition of a crack surface and the topology of the cracks emerges naturally as elements meet the insertion criterion.

Frictional contact Frictional contact plays an important role in many mechanical devices. Numerical modelling of frictional contact in the standard finite element method suffers heavily form instability and the computed solution strongly depends on model variables and solution algorithms (Wriggers, 2006).

Dolbow et al. (2001) showed how nonlinear constitutive laws for contacts on arbitrary surfaces can be enforced using an XFEM formulation. They studied two-dimensional crack growth with three different interfacial constitutive laws for

crack surfaces, including a perfect contact and a unilateral contact with or without friction. They also used an iterative method called LATIN to resolve the non-linear boundary value problem. The penalty approach in combination with the extended finite element method was used to model frictional contact with large sliding in Khoei and Nikbakht (2006, 2007). The extended finite element formulation was also used in Vitali and Benson (2006, 2009) with classical kinetic friction laws in a Multi-Material Arbitrary Lagrangian Eulerian (MMALE) formulation. Recently, Liu and Borja (2010a) used an XFEM formulation to model frictional cracks in elastoplastic solids. They considered mechanisms including the combined opening and frictional sliding in initially straight, curved and S-shaped cracks with or without bulk plasticity. Liu and Borja (2010b) also employed XFEM to address instability issues existing in classical contact formulations.

Grain boundaries in polycrystals In the classical finite element approach, modelling grain boundaries in polycrystals relies on designing a mesh which conforms with the topology of grains (Weyer et al., 2002; Kuprat et al., 2003). This poses challenging demands already at the discretization stage of the polycrystals. The singularities at the grain junctions require relatively refined meshes that must fit the grain boundaries while considering the aspect ratio of the elements within acceptable ranges. These requirements are not always easily achieved. Meshing around the grain junctions can also be difficult and expensive and it may lead to a large number of elements when the angle between the branches is small.

As proposed in Sukumar et al. (2003) and Simone et al. (2006), the enrichment concept defined in the partition of unity approach can also be used to facilitate the modelling of grain boundaries. The proposed approaches do not require a mesh generator to mimic the grains geometry. It is enough to have an arbitrary background mesh in which the grain boundaries are superimposed, as shown in Figure 2.1. For cells intersected by grain boundaries, the classical finite element approximations are enriched by adding discontinuous enrichment functions corresponding to the enriched degrees of freedom. In contrast to the standard finite element method, no limitation exists on grain shapes and the number of grain boundaries meeting at junctions. Decoupling the underlying mesh from the grains structure allows to model irregular polycrystals in an efficient way.

2.1.2 Applications in fluid mechanics

Problems with discontinuous solutions also exist in fluid mechanics. Modelling two-phase flow and fluid-structure interaction are two examples of this type of problem which have a lot of applications in the real engineering world. Modelling two-phase flow appears in oil and gas reservoirs and underground water flows (Aziz

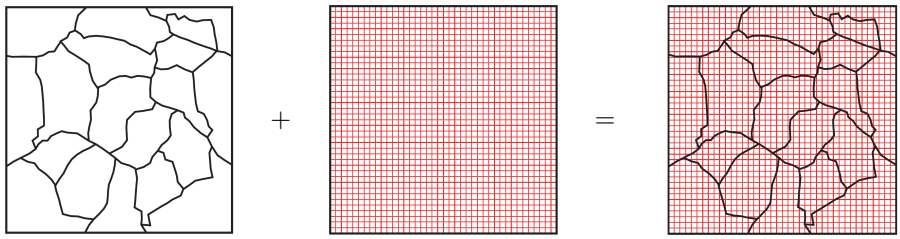


Figure 2.1 The partition of unity approximation for the modelling of polycrystals (adapted from Simone et al. (2006)).

and Settari, 1979; Levy, 1995; Helmig, 1997). Designing aircraft, cars, bridges and dams are examples of engineering problems in which fluid-structure interaction plays an important role (Bungartz and Schäfer, 2006; Wang, 2008). An overview on the application of the partition of unity approaches in the modelling of these two problems is presented in the following.

Two-phase flow Two-phase flow occurs in a system containing gas or liquid with a meniscus separating phases. XFEM is a promising method in the modelling of immiscible two-phase flows, a coupled problem between velocity fields and pressure fields. The velocity and pressure fields can be either weakly or strongly discontinuous in a domain. If a field is strongly discontinuous, then the field and its derivatives are discontinuous in the domain. However, if a field is weakly discontinuous, then the field itself is continuous but its derivatives are discontinuous in the domain. The velocity fields are weakly discontinuous across the interfaces, while the pressure fields may be considered either weakly or strongly discontinuous based on the surface-tension effects over the interfaces. Interested readers are referred to Fries (2008) to see how the two-phase flow can be modeled using a partition of unity formulation.

A fractional step method was used in Chessa and Belytschko (2003a,b) to uncouple the pressure and velocity fields in the XFEM framework. In their works, the velocity fields were assumed the sole enriched fields. Another approach was used in Groß and Reusken (2007) and Reusken (2008) in which the pressure fields were assumed to be enriched instead. However, they have not enriched the velocity field; on the contrary, they only refined the underlying mesh near interfaces.

Fluid-structure interaction Fluid-structure interaction plays an important role in the design of many engineering systems. Failing to consider the effects of oscillatory interactions can be catastrophic, especially in structures comprising materials susceptible to fatigue. A famous example of failure of such structures is

the Tacoma Narrows bridge (Ross, 1984; Billah and Scanlan, 1991).

The XFEM framework has also been used for the modelling of fluid-structure interaction (Legay et al., 2006; Wang et al., 2008; Gerstenberger and Wall, 2008a,b). The level set method has been used to represent fluid and solid interfaces implicitly. Based on the interfacial condition between solid and fluid phases, the tangential components of velocity across the interface can be assumed either weakly or strongly discontinuous.

Recently, a three-dimensional framework, combining the dual mortar contact formulation and the extended finite element method, to model fluid-structure interaction has been proposed in Mayer et al. (2009, 2010). The combined XFEM Fluid-Structure-Contact Interaction method (FSCI) allows to compute contact of arbitrarily moving and deforming structures embedded in an arbitrary flow field.

2.2 A discretized form of an elasticity problem with discontinuities

To clarify issues specific to the partition of unity enrichment formulations, as a canonical example a formulation for an elasticity problem with discontinuities is presented in this section. In this problem, a discontinuity surface in an elastic body is modeled independently of a mesh using a partition of unity formulation.

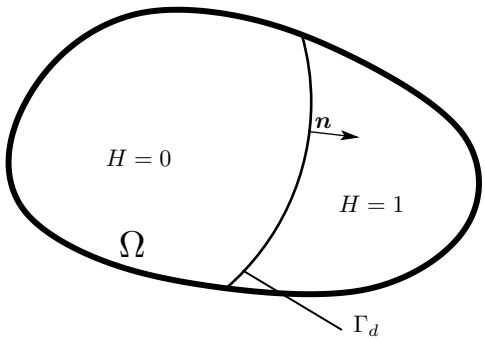


Figure 2.2 A physical domain Ω containing a discontinuity surface Γ_d whose unit normal vector denoted by $\boldsymbol{n}$.

A domain $\Omega \subset \mathbb{R}^d$, where d denotes the geometric dimension, is considered. This domain is intersected by a discontinuity surface Γ_d, as illustrated in Figure 2.2. The governing equations and boundary conditions for the elastic body Ω crossed by the

cohesive discontinuity surface Γ_d read

$$-\nabla \cdot \boldsymbol{\sigma} = \boldsymbol{f} \qquad \text{in } \Omega, \tag{2.1}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{m} = \boldsymbol{h} \qquad \text{on } \Gamma_t, \tag{2.2}$$

$$\boldsymbol{\sigma}^+ \cdot \boldsymbol{n} = \boldsymbol{t} \qquad \text{on } \Gamma_d, \tag{2.3}$$

$$\boldsymbol{u} = \boldsymbol{0} \qquad \text{on } \Gamma_u, \tag{2.4}$$

$$[\![\boldsymbol{\sigma}]\!] \cdot \boldsymbol{n} = \boldsymbol{0} \qquad \text{on } \Gamma_d, \tag{2.5}$$

where $\boldsymbol{u}$ is displacement field, $\boldsymbol{\sigma}$ is stress tensor, $\boldsymbol{f} : \Omega \to \mathbb{R}^d$ is a sufficiently regular body force and $\boldsymbol{h} : \Gamma_t \to \mathbb{R}^d$ is a boundary condition and $\boldsymbol{t} : \Gamma_d \to \mathbb{R}^d$ is a traction force across the discontinuity surface. The traction $\boldsymbol{t}$ may be prescribed or may be determined via a constitutive model. The normal vectors to the discontinuity surface Γ_d and the external boundary $\partial\Omega$ are expressed as $\boldsymbol{n}$ and $\boldsymbol{m}$, respectively. Furthermore, Γ_u and Γ_t are Dirichlet boundary and Neumann boundary domains, respectively. These domains are defined such that $\Gamma_u \cup \Gamma_t = \partial\Omega$ and $\Gamma_u \cap \Gamma_t = \emptyset$. The domains on different sides of the discontinuity surface are denoted as Ω^- and Ω^+. Note that $\Omega^- \cup \Omega^+ \cup \Gamma_d = \Omega$. The jump operator is defined as $[\![(\cdot)]\!] = (\cdot)^+ - (\cdot)^-$ to represent fields restricted to the discontinuity surface that may have different values on the positive and negative sides of the discontinuity surface.

The constitutive law for the elastic domain can be written by

$$\boldsymbol{\sigma} = \mathscr{C} : \boldsymbol{\epsilon}, \tag{2.6}$$

where $\mathscr{C}$ is a fourth-order tensor and $\boldsymbol{\epsilon}$ is a second-order strain tensor that is defined as

$$\boldsymbol{\epsilon} = \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T). \tag{2.7}$$

A variational formulation of this problem reads: find $\boldsymbol{u} \in V$ such that

$$\int_{\Omega \backslash \Gamma_d} \sigma(\boldsymbol{u}) : \nabla \boldsymbol{v} \, d\Omega + \int_{\Gamma_d} \boldsymbol{t}([\![\boldsymbol{u}]\!]) \cdot [\![\boldsymbol{v}]\!] \, d\Gamma = \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{v} \, d\Omega + \int_{\Gamma_t} \boldsymbol{h} \cdot \boldsymbol{v} \, d\Gamma \quad \forall \boldsymbol{v} \in V, \tag{2.8}$$

where the function space V reads

$$V = \left\{ \boldsymbol{v}_h \in \left(L^2(\Omega)\right)^d \cap \left(H^1(\Omega \backslash \Gamma_d)\right)^d, \boldsymbol{u}_h = \boldsymbol{0} \text{ on } \Gamma_u \right\}. \tag{2.9}$$

A finite–dimensional formulation of this problem using the extended finite element

basis functions, which does not require considering discontinuity surfaces in the triangulation of Ω, is now considered. To describe the displacement jump over the discontinuity surfaces, the Heaviside function operating on a smooth and continuous function is used. This leads to decompose the finite element solution $\boldsymbol{u}_h$ in the domain Ω as

$$\boldsymbol{u}_h = \bar{\boldsymbol{u}}_h + H_d \hat{\boldsymbol{u}}_h, \tag{2.10}$$

where $\bar{\boldsymbol{u}}_h$ and $\hat{\boldsymbol{u}}_h$ are the "standard" and "enriched" parts of the displacement approximation whose corresponding function spaces are respectively defined as

$$\bar{V} = \left\{ \bar{\boldsymbol{u}}_h \in (H^1(\Omega))^d, \bar{\boldsymbol{u}}_h|_E \in (P_{k_1}(E))^d \ \forall E : \boldsymbol{u}_h = \boldsymbol{0} \text{ on } \Gamma_u \right\}, \tag{2.11}$$

$$\hat{V} = \left\{ \hat{\boldsymbol{u}}_h \in (H^1(\Omega_d))^d, \hat{\boldsymbol{u}}_h|_E \in (P_{k_2}(E))^d \ \forall E \in \Omega_d : \hat{\boldsymbol{u}}_h = \boldsymbol{0} \text{ on } \Gamma_u \cap \partial\Omega^+ \right\}, \tag{2.12}$$

where $\Omega_d \subset \Omega$ is a "small" region around the discontinuity surface Γ_d. More precisely, Ω_d is the union of the supports of all basis functions whose support is intersected by the discontinuity surface. Moreover, $P_{k_i}(E)$ denotes a space of Lagrange polynomials of degree k_i on element E.

The Heaviside function, H_d, is defined as

$$H_d = \begin{cases} 1 & \boldsymbol{x} \in \Omega^+, \\ 0 & \boldsymbol{x} \in \Omega^-. \end{cases} \tag{2.13}$$

Decomposing the test function $\boldsymbol{v}_h$ similarly, a finite element variational problem is expressed as: find $\bar{\boldsymbol{u}}_h \in \bar{V}$ and $\hat{\boldsymbol{u}}_h \in \hat{V}$ such that

$$\begin{aligned} &\int_{\Omega \setminus \Gamma_d} \bar{\boldsymbol{\epsilon}}_h : \mathscr{C} : \nabla \bar{\boldsymbol{v}}_h \, d\Omega + \int_{\Omega^+} \hat{\boldsymbol{\epsilon}}_h : \mathscr{C} : \nabla \bar{\boldsymbol{v}}_h \, d\Omega + \int_{\Omega^+} \bar{\boldsymbol{\epsilon}}_h : \mathscr{C} : \nabla \hat{\boldsymbol{v}}_h \, d\Omega \\ &+ \int_{\Omega^+} \hat{\boldsymbol{\epsilon}}_h : \mathscr{C} : \nabla \hat{\boldsymbol{v}}_h \, d\Omega + \int_{\Gamma_d} \boldsymbol{t}(\hat{\boldsymbol{u}}_h) \cdot \hat{\boldsymbol{v}}_h \, d\Gamma = \int_{\Omega} \boldsymbol{f} \cdot \bar{\boldsymbol{v}}_h \, d\Omega + \int_{\Omega^+} \boldsymbol{f} \cdot \hat{\boldsymbol{v}}_h \, d\Omega \\ &+ \int_{\Gamma_t} \boldsymbol{h} \cdot \bar{\boldsymbol{v}}_h \, d\Gamma + \int_{\Gamma_t{}^+} \boldsymbol{h} \cdot \hat{\boldsymbol{v}}_h \, d\Gamma \qquad \forall \bar{\boldsymbol{v}}_h \in \bar{V}, \quad \forall \hat{\boldsymbol{v}}_h \in \hat{V}, \end{aligned} \tag{2.14}$$

where $\bar{\boldsymbol{\epsilon}}_h$ and $\hat{\boldsymbol{\epsilon}}_h$ are respectively the "standard" and "enriched" parts of the approximated strain tensor $\boldsymbol{\epsilon}_h$, defined using a similar decomposition presented in Equation (2.10). The partition of unity formulation for the elasticity problem is now complete.

In the definition of the finite element spaces, the use of different order functions for $\bar{\boldsymbol{u}}_h$ and $\hat{\boldsymbol{u}}_h$ ($k_1 \neq k_2$) is deliberately permitted. However, except for a few formulations (see for example Duarte et al. (2007)), the majority of the partition of

unity formulations use the same function spaces for the standard and enriched parts ($k_1 = k_2$). By this assumption, a compact notation for the finite element space can be used as

$$V = \left\{ \boldsymbol{v}_h \in \left(L^2\left(\Omega\right)\right)^d \cap \left(H^1\left(\Omega \backslash \Gamma_d\right)\right)^d, \boldsymbol{v}_h|_E \in \left(P_k\left(E \backslash \Gamma_d\right)\right)^d \; \forall E \right\} \tag{2.15}$$

for a finite element function $\boldsymbol{v}_h \in V$ which is discontinuous across surfaces.

To construct a discretized form of the variational problem, the finite element approximations are inserted to the weak governing equations. This yields the variational formulation, presented in Equation (2.14), to be expressed as a system of linear equations

$$\boldsymbol{K}\boldsymbol{U} = \boldsymbol{f}, \tag{2.16}$$

where $\boldsymbol{K}$ and $\boldsymbol{f}$ are a global stiffness matrix and a global right-hand side vector, respectively. The unknown vector containing both the standard and enriched degrees of freedom is given as $\boldsymbol{U}$. If a linear traction-separation constitutive law across the discontinuity surface is assumed

$$\boldsymbol{t} = \boldsymbol{T} : [\![\boldsymbol{u}]\!], \tag{2.17}$$

where $\boldsymbol{T}$ is a constant second-order tensor. An expanded form of the system of linear equations, presented in Equation (2.16), then reads

$$\begin{bmatrix} \int_\Omega \bar{\boldsymbol{B}}^T \boldsymbol{C} \bar{\boldsymbol{B}} \, d\Omega & \int_{\Omega_d^+} \hat{\boldsymbol{B}}^T \boldsymbol{C} \bar{\boldsymbol{B}} \, d\Omega \\ \int_{\Omega_d^+} \bar{\boldsymbol{B}}^T \boldsymbol{C} \hat{\boldsymbol{B}} \, d\Omega & \int_{\Omega_d^+} \hat{\boldsymbol{B}}^T \boldsymbol{C} \hat{\boldsymbol{B}} \, d\Omega + \int_{\Gamma_d} \hat{\boldsymbol{N}}^T \boldsymbol{T} \hat{\boldsymbol{N}} \, d\Gamma \end{bmatrix} \begin{bmatrix} \bar{\boldsymbol{U}} \\ \hat{\boldsymbol{U}} \end{bmatrix} = \begin{bmatrix} \int_\Omega \bar{\boldsymbol{N}}^T \boldsymbol{f} \, d\Omega + \int_{\Gamma_h} \bar{\boldsymbol{N}}^T \boldsymbol{h} \, d\Gamma \\ \int_{\Omega_d^+} \hat{\boldsymbol{N}}^T \boldsymbol{f} \, d\Omega + \int_{\Gamma_h^+} \hat{\boldsymbol{N}}^T \boldsymbol{h} \, d\Gamma \end{bmatrix}. \tag{2.18}$$

where $\boldsymbol{C}$ is the elasticity matrix. In this equation, $\bar{\boldsymbol{B}} = \boldsymbol{L}\bar{\boldsymbol{N}}$ and $\hat{\boldsymbol{B}} = \boldsymbol{L}\hat{\boldsymbol{N}}$ where $\bar{\boldsymbol{N}}$ and $\hat{\boldsymbol{N}}$ contain the basis functions corresponding to the "standard" part of the degrees of freedom $\bar{\boldsymbol{U}}$ and the "enriched" part of the degrees of freedom $\hat{\boldsymbol{U}}$, respectively. In 3D, the matrix $\boldsymbol{L}$ contains differential operators:

$$\boldsymbol{L} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix}. \tag{2.19}$$

If the same finite element basis for the "standard" and "enriched" components ($k_1 = k_2$ in Equations (2.11) and (2.12)) are used, $\bar{\boldsymbol{B}} = \hat{\boldsymbol{B}}$ and $\bar{\boldsymbol{N}} = \hat{\boldsymbol{N}}$.

As can be observed in the expanded form of the system of linear equations in Equation (2.18), adding the enriched degrees of freedom changes the structure of the discretized system of equations. The entries corresponding to the enriched degrees of freedom in the element matrix and the right-hand side vector require additional works, because the evaluation of these entries on the positive side of the domain cannot be performed using standard quadrature rules. Special attention must also be devoted to the evaluation of terms corresponding to the integration along discontinuity surfaces.

In the elasticity problem presented in this section, all coefficient functions (e.g. f and h) are defined on the continuous function spaces. However, the coefficient functions may be also defined on the enriched function spaces. For example, in nonlinear problems, the solution from the previous converged stage is represented as a coefficient function defined on the enriched function space. For this reason, terms corresponding to the standard degrees of freedom in the element tensor should be also evaluated in the positive side of a domain. This makes obtaining a discretized system of equations for these types of variational formulations more difficult, especially for the coupled nonlinear problems.

2.3 Implementation aspects

There are some issues specific to the partition of unity enrichment methods which make their implementations more complex than the conventional finite element methods. This poses challenges in extending existing finite element packages to model discontinuities in the partition of unity framework. The variable number of degrees of freedom, the integration of the enriched cells and surface representations are amongst the implementation aspects that require attention in designing software packages for the partition of unity methods.

2.3.1 Variable number of degrees of freedom

An issue which requires special treatment is the variable number of degrees of freedom for cells depending on their positions with respect to discontinuity surfaces. The total number of degrees of freedom will change during a simulation of an evolving discontinuity surface. This is an obstacle to adjust current finite element software packages to support the partition of unity framework in modelling discontinuity surfaces.

A simple approach is to assume that all cells are enriched with the maximum possible extra degrees of freedom, depending on the number of intersecting

discontinuity surfaces. In this approach, a local element tensor whose dimension is doubled – if a cell is intersected by one discontinuity surface – as the dimension of the standard element tensor is constructed. However, since the enriched degrees of freedom are limited to the cells intersected by discontinuity surfaces, the number of enriched degrees of freedom is considerably smaller than the number of standard degrees of freedom. Nevertheless, this approach significantly increases the size of the element tensors; therefore, it lacks efficiency.

Another approach is to design a framework such that it can handle different numbers of degrees of freedom for each cell. Because only a subset of the nodes is enriched, each cell falls into one of the following groups. The element is either

- a standard finite element if *none* of the element nodes are enriched; or
- an enriched finite element if *part* or *all* of the element nodes are enriched.

The majority of cells falls in the first category in which element tensors are identical to the element tensors computed using the standard finite element framework. For the cells belonging to the second group, the dimension of the element tensor changes and extra entries are evaluated inside the element tensor. To design a powerful software package for modelling discontinuities in the partition of unity approach, an efficient framework should be designed to evaluate element tensors with variable numbers of degrees of freedom. This framework is also used to assemble the global tensor using local element tensors.

2.3.2 Integration of the intersected cells

In the classical finite element methods, the evaluation of the element stiffness matrix and the right-hand side vector generally requires the quadrature of functions which are polynomials. For polynomial functions, using quadrature rules is adequate to perform the numerical integrations in the standard finite element approximations. However, when the continuous function space is enriched by a singular or a discontinuous function, the quadrature rule is not sufficient for numerical integration. Using the standard quadrature rules may lead to inaccurate results, poor convergence and singular systems. Moreover, because a discontinuity surface may change during simulation, the integration schemes cannot be pre-computed in advance and they should also be evaluated at run-time.

To overcome this issue, different approaches have been proposed in literature. Here, a brief review of three approaches is presented.

- Using a higher order quadrature rule was an approach used to compute the additional entries appearing in the weak form. However, it has been shown that this approach performs poorly and has an adverse effect on the accuracy of the approximated solutions (Strouboulis et al., 2000a).

- The subdomain quadrature is a common approach to perform numerical integration in the partition of unity framework (Belytschko and Black, 1999; Moës et al., 1999). In this approach, cells intersected by a surface are sub-divided into subdomains whose boundaries are aligned with the surface. The fixed order of the Gauss quadrature is used inside each subdomain. Note that no additional degrees of freedom have been introduced and the newly added subdomains are just for numerical integration purposes.

- Another approach has been proposed in Ventura (2006) to avoid the sub-division of intersected cells. In this approach, enrichment functions are mapped to equivalent polynomials which can be computed by numerical integration using standard quadrature rules. These polynomials, constructed on the whole cell domain, are defined using coefficients which are functions of surface locations in cells. A similar approach computing quadrature weights of the given quadrature rule based on the position of the discontinuity surface has been proposed in Holdych et al. (2008). A draw-back of this approach is that the definition of the equivalent polynomials is strongly coupled to enrichments and element types. Therefore, for each element type and enrichment type in given weak forms, a new set of equivalent polynomials must be computed.

2.3.3 Surface representations

The accurate description of surface interfaces is an important topic in the partition of unity enrichment framework. Surface representations are used to determine the enriched degrees of freedom and to compute modified quadrature rules for cells intersected by discontinuity surfaces. The surface representations are also coupled with the integration scheme used to compute the traction–like quantities across the discontinuity surfaces.

Surface interfaces can be represented via either implicit or explicit surface descriptions. As an example of explicit surface representations, cracks in two-dimensional domains are parametrized in real element geometry as presented in Belytschko and Black (1999) and Moës et al. (1999). A similar approach has been also used in Sukumar et al. (2000) and Duarte et al. (2001) to model cracks in three-dimensional domains in which the crack surfaces are described by a set of connected planes. However, determining the intersection of the parameterized lines/surfaces with the mesh is not an easy task

Explicit surface representations are also used to model propagating surfaces in two-dimensional and three-dimensional domains using the partition of unity framework. In a two-dimensional setting, the orientation of an element discontinuity is determined by its reference normal vector N and a single point P to characterize the connection to the next element discontinuity. However, unlike two-dimensional

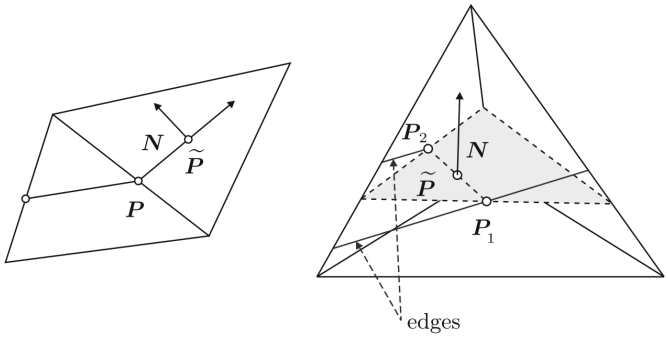


Figure 2.3 Unique connecting point P for the two-dimensional case and averaged connecting points P_i depending on the adjacent cracked elements for the three-dimensional case (Jäger et al., 2008b)

problems, the orientation of element discontinuities in three-dimensional problems is not uniquely defined, as shown in Figure 2.3.

The tracking of a discrete surface can be performed in several conceptually different ways. Areias and Belytschko (2005) used a local tracking algorithm to model three-dimensional propagating surfaces. In their approach, a surface extends from neighboring surface points and proceeds in a direction normal to the maximum principal stress. As this concept would eventually render non-smooth surfaces, Areias and Belytschko (2005) have suggested to adjust the crack plane normal based on neighboring crack intersection points. However, their approach to represent discontinuity surfaces by the connected polygons was not general enough and it was limited to the planer or slightly kinked discontinuity surfaces.

To overcome this issue, another approach was presented in Gasser and Holzapfel (2005). In this approach, a non-local tracking algorithm was used to track three-dimensional propagated surfaces. By averaging surface plane normals over a certain neighborhood, they ensured that the generated failure surface was smooth in an averaged sense. This approach can be used for a wide range of three-dimensional propagating discontinuity surfaces. However, this algorithm is computationally expensive and does not give continuous discontinuity surfaces. A comparative study between different approaches for representing three-dimensional evolving surfaces has been performed in Jäger et al. (2008a).

Discontinuity surfaces may also be represented implicitly using the level set method (Sethian, 1999; Osher and Fedkiw, 2003). This method defines surfaces by means of the zero levels of scalar functions within the domain and it can be used to represent various types of surface interfaces including open surfaces like cracks and shear bands which usually end inside the domain.

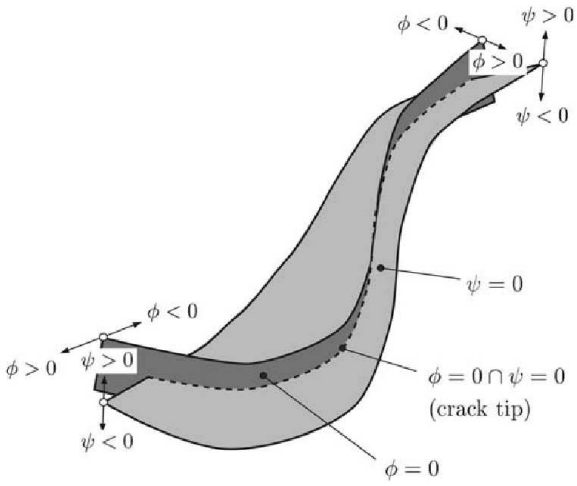


Figure 2.4 The level set description of a three-dimensional crack (Gasser and Holzapfel, 2006).

The level set method has been used to model cracks in the partition of unity framework in Stolarska et al. (2001) and Belytschko et al. (2001). In case of the level set description of a crack, the crack surface is defined using two scalar level set functions: ϕ and ψ. The crack is then represented as

$$\Gamma_d = \{\boldsymbol{x} : \phi(\boldsymbol{x}) = 0 \quad \text{and} \quad \psi(\boldsymbol{x}) \leq 0\}. \tag{2.20}$$

The first function ϕ is a signed-distance function which is zero on the crack surface. The second function ψ is constructed such that it is zero on the crack boundaries (crack tips). For a three-dimensional crack, the level set functions and their definitions are illustrated in Figure 2.4.

For discretized domains, the values of level set functions are interpolated using finite element shape functions inside the domain. Therefore, the approximated level set functions read

$$\phi^h(\boldsymbol{x}) = \sum_j \phi_j N_j(\boldsymbol{x}), \tag{2.21}$$

$$\psi^h(\boldsymbol{x}) = \sum_j \psi_j N_j(\boldsymbol{x}). \tag{2.22}$$

where N_j is a basis function corresponding to node j from the finite element mesh and ϕ_j and ψ_j are the values of level set functions at node j.

To model crack propagation using the level set method, the level set functions ϕ and ψ should be updated in each step. Stolarska et al. (2001) presented an algorithm for these updates in two-dimensional problems. A similar algorithm for the non-planar surface evolution in three dimensional problems was also presented in Moës et al. (2002) and Gravouil et al. (2002). An overview of different level set approaches to represent and update surface geometries is given in Duflot (2007).

Nevertheless, none of these methods can handle the surface evolution in three-dimensional problems in a reliable way. Finding a suitable approach to represent surfaces is still a challenging topic among researchers. Recently, NURBS and the Bezier splines (Piegl and Tiller, 1997; Prautzsch et al., 2002) have been used to represent surfaces in the partition of unity framework. A primary work in this direction has been presented in a recent paper by Moumnassi et al. (2011).

Chapter 3 An overview on the automated computational mathematical modelling

Numerical techniques are widely used to solve mathematical problems expressed by partial differential equations (PDEs). The finite element method (Zienkiewicz et al., 2005; Hughes, 2000; Cook et al., 2002) is one of these numerical techniques. The mathematical base of this method often involves theorems from functional analysis (Oden, 1979; Reddy, 1991). This method discretizes weak forms of a PDE on a mesh and reduces solving PDEs to computing the solutions of algebraic equations. After solving the system of algebraic equations on some discrete points, the solution on the whole domain is obtained by interpolating values on discrete points using the finite element basis functions.

The finite element method has advantages over other numerical analysis methods like the finite difference method (Mitchell and Griffiths, 1979; Smith, 1985) and the finite volume method (Eymard et al., 2000; LeVeque, 2002). Most of the success of the finite element method is because of its generality which allows one to use it for a wide range of physical problems without any restriction on geometrical shapes and boundary conditions. This generality makes the finite element method appealing for real engineering problems where complex geometries with different boundary conditions may exist.

However, implementing finite elements models can be a difficult and error-prone task and requires significant time investment. Automation of the finite element method seems to be a possibility to overcome implementation problems. Today, a number of projects exist that try, at least in part, to automate the finite element method using novel techniques. These projects combine domain specific languages and symbolic computing with finite element methods to achieve the automation goal. Analysa (Bagheri and Scott, 2003), Sundance (Long et al., 2010, 2012), GetDP (Dular and Geuzaine, 2012), FreeFEM++ (Pironneau et al., 2010), Life (Prudhomme, 2007) and COMSOL (2012) are a few examples of such projects.

Recent developments in finite element code generation (Kirby, 2004, 2006) indicate a significant step in the direction of automating the finite element method. The FEniCS project (Logg et al., 2012f), which is somehow similar to the above-mentioned projects, uses a concept of the code generation to automate solutions of finite element models. FEniCS allows one to decouple assembly algorithms from the implementation of variational forms and finite element bases.

This separation allows rapid development of finite element models for a wide range of physical problems.

This chapter continues by the elaboration of the FEniCS project. The key components of this project are explained and it has been shown how these components are combined to provide a framework to automate the developments of finite element models. Then, a number of examples are presented to show the application of FEniCS in automated modelling of different linear/nonlinear physical problems using scalar/mixed finite element function spaces. Finally, the automated framework for solving partial differential equations is summarized.

3.1 FEniCS project

The FEniCS project is a collaborative project toward developing a framework and its required tools to achieve goals of Automated Computational Mathematical Modelling (ACMM), which are efficiency, simplicity, generality and reliability in modelling and simulation. This project provides required tools to facilitate solving partial differential equations in an automated way. All components of the FEniCS project are available under GNU open source licenses and they can be downloaded freely from the FEniCS project homepage (`www.fenicsproject.org`).

FEniCS relies on novel techniques for the automatic code generation which allows one to combine a high level of expressiveness with efficient computation. Finite element variational forms are expressed in near mathematical notations, from which low-level code is automatically generated, compiled and seamlessly integrated with efficient implementations of other general-specific components of finite element models like computational meshes and linear algebra.

3.1.1 Design of the automated framework

To provide a framework to use automatic code generation for modelling variational problems in an efficient way, problem inputs are divided into two sub-sets: (i) input 1, which represents an underlying partial differential equation and (ii) input 2, which contains a computational domain (mesh), boundary conditions, coefficients and material properties. The FEniCS Form Compiler, FFC, (Kirby and Logg, 2006, 2007; Logg et al., 2012b) or SyFi (Alnæs and Mardal, 2012) receives the differential equations (input 1) as high-level code based on UFL (Alnæs, 2012; Alnæs and Logg, 2012) which is close to the mathematical notations and generates C++ low-level code compatible with UFC (Alnæs et al., 2012). The generated code combined with the second part of the problem input (input 2) are then used in DOLFIN (Logg and Wells, 2010; Logg et al., 2012e) to solve the given variational problem. This procedure is presented schematically in Figure 3.1. More information about different

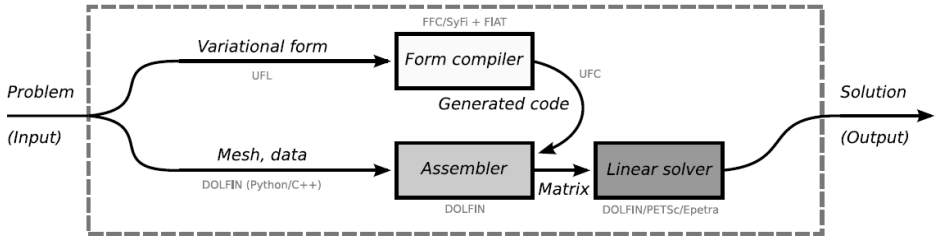


Figure 3.1 The design of the automated system performed using different components of the FEniCS project (Logg et al., 2009).

components of FEniCS is given in the following sections.

3.1.2 Key components

The FEniCS project comprises several components which can facilitate developing finite element models for various physical problems. These components work together to provide an automated framework in computational mathematical modelling. Moreover, each component must not only be compatible with the whole project to achieve generality and simplicity goals, but it must also be an independent entity such that it can be used as a separate package inside other open-source projects. Code reuse improves the generality and thus increases the reliability of code.

Figure 3.2 shows different components of the FEniCS project, their classifications into different layers and their relations within layers. As depicted, the components of FEniCS are divided into four layers: application, interface, core components and external libraries. In the following, more information is given about the core components of the FEniCS project. Amongst the core components, UFL, FFC, UFC and DOLFIN are elaborated in more details in this section. These components have been extended to support the modelling of discontinuities inside FEniCS. A brief explanation will also be given for the other components at the end of this section.

Unified Form Language (UFL)

The Unified Form Language (UFL) is a domain specific language to declare finite element discretizations of variational formulations and functionals. UFL provides a flexible user interface which might be used to represent finite element function spaces and variational weak forms in a notation close to mathematics. The UFL implementation also provides functionalities to simplify the compilation process used inside form compilers.

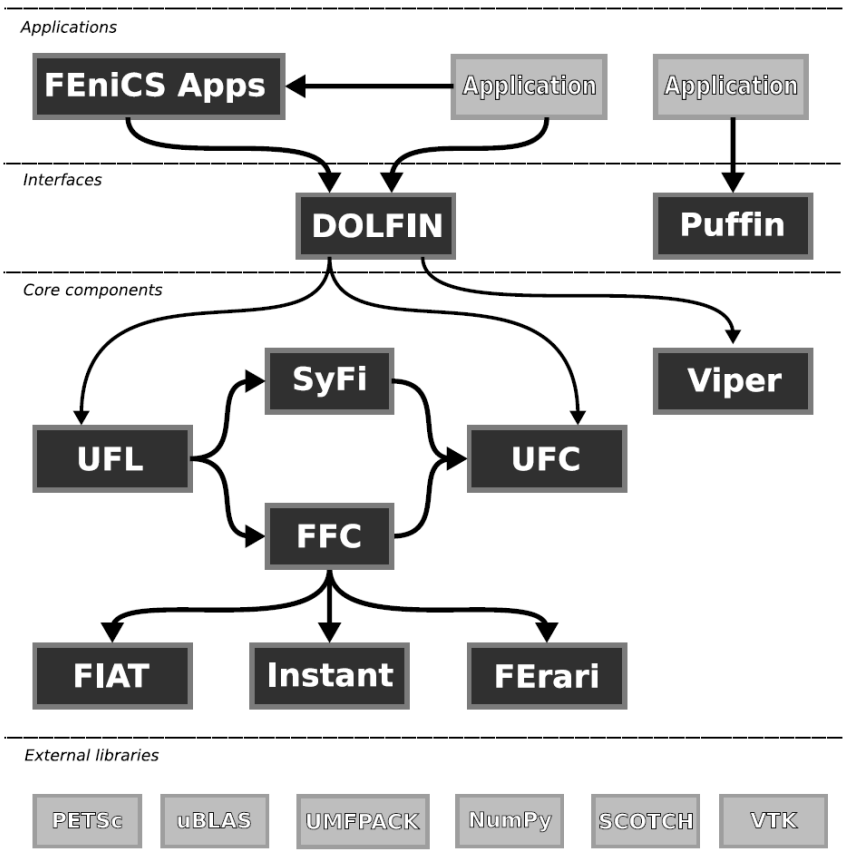


Figure 3.2 FEniCS software map (Logg et al., 2009)

The development of UFL has been motivated by a number of factors (Alnæs, 2012). A first factor was introducing a richer form language than the built-in form language which already existed as part of the FEniCS Form Compiler. This language facilitates expressing mathematical formulations for a wide range of problems. A second factor was the necessity of supporting automatic differentiation which alleviates obtaining the Jacobian of weak forms for nonlinear problems. The last factor was related to the improvement of the form compiler's efficiency to handle more complicated problems.

UFL supports tensor algebra, index notations, several nonlinear operators and functions to facilitate expressing a wide range of the finite element variational formulations in an efficient and simple way. The notation, definitions and operators to

define finite element spaces and variational forms in the UFL notation are explained in following.

Defining finite element spaces UFL provides a syntax for finite element spaces declarations of predefined basic element families. The set of predefined element family names in UFL includes "Lagrange", representing scalar Lagrange finite elements, "Discontinuous Lagrange", representing scalar discontinuous Lagrange finite elements and a range of other families that can be found in the UFL manual (Alnæs and Logg, 2009). Basic scalar elements can be combined to construct vector elements or tensor elements. Furthermore, elements can also be combined in arbitrary mixed element hierarchies. To present a UFL interface declaring the finite element spaces, consider the extract of following code:

```
P = FiniteElement("Lagrange", triangle, 1)
V = VectorElement("Lagrange", triangle, 2)

M = V*P
```

In the first line, a scalar finite element space `P` for a first order Lagrange basis on triangular cells is declared. Then a quadratic vector Lagrange element `V` on triangles is defined. The code proceeds to declare a mixed finite element `M`. This mixed element is created by combining the vector element `V` and the scalar element `P`. This mixed element can be alternatively obtained by using `MixedElement(V, P)`.

Most of UFL deals with how to declare integrand expressions used in variational formulations. The most basic expressions are form arguments, which do not depend on other expressions. Any other expression can be constructed using the form arguments in combination with other expressions called operators. Form arguments include basis test and trial functions and coefficient functions which are represented by `TestFunction`, `TrialFunction` and `Coefficient` classes, respectively.

```
v = TestFunction(V)
u = TrialFunction(V)
f = Coefficient(V)
```

These basic arguments can be used to define linear and bilinear forms of the variational forms.

Defining forms UFL defines different operators that can be used for composing expressions using the basic form arguments. The elementary algebraic operators `+`, `-`, `*`, `/` are used between UFL expressions with a few limitations. Moreover, `dot(a, b)`, `inner(a, b)` and `outer(a, b)` are three often used operators between `a` and `b`, two arbitrary rank tensors. The dot product of `a` and `b` is a summation over the last index of the first tensor and the first index of the second tensor. The inner

product is a summation over all indices of `a` and `b`. The outer product is a tensor product between `a` and `b`. This product results in a matrix if `a` and `b` are both first order tensors (vectors). Other common tensor operators like `transpose(a)` (or `a.T`), `tr(a)`, `det(a)` and `inv(a)`, which respectively define the transpose, trace, determinant and inverse of `a`, are also supported inside UFL.

UFL also implements derivatives with respect to different kinds of variables. The most common one is derivatives with respect to spatial coordinates that can construct compound spatial derivatives like gradient and divergence. Expressions can also be differentiated with respect to arbitrary user defined variables. This type of derivatives is useful for several tasks, from the differentiation of material laws to computing sensitivities. The final type of derivatives is form or functional derivatives with respect to coefficients of a discrete function. This functionality may be used to linearise nonlinear residual equations (linear form) automatically for use in the Newton-Raphson method.

More detail on implementation issues as well as some general information on the application of UFL for representing various PDEs can be found in Alnæs and Logg (2009) and Alnæs (2012). This information is useful for ordinary users as well as advanced users, who may want to develop their own software packages using functionalities provided by UFL.

FEniCS Form Compiler (FFC)

The automatic code generation is a key feature of FEniCS for computing general and efficient solutions of finite element variational problems. The automatic code generation depends on a form compiler for the compilation of code for variational forms. FFC (Kirby and Logg, 2006; Logg et al., 2012b,c) is one of the compilers supported inside FEniCS (the other compiler is SFC (Alnæs and Mardal, 2012) which is not discussed in this thesis).

FFC receives weak variational forms as input and returns as output low-level C++ code for the evaluation of element tensors and degrees of freedom mapping corresponding to the finite element variational formulations. At the beginning, FFC was using a built-in form language as an input interface. However, after the introduction of a new form language UFL in 2009 which supports some appealing functionalities, FFC now uses UFL as the form language to represent variational formulations. The generated code is also compatible with UFC (Alnæs et al., 2012). This compatibility allows one to use any assembler which supports the UFC interface. The structure of UFC will be explained in more detail in the next section.

The form compiler also supports a wide range of finite element spaces. FFC relies on FIAT (Kirby, 2012c) for the evaluation of finite element basis functions and their derivatives. At this moment, the following families of finite elements are supported

inside FFC (Logg and Wells, 2010).

- H^1-conforming finite elements:
  - CG_q, arbitrary degree continuous Lagrange elements.
- $H(\mathrm{div})$-conforming finite elements:
  - RT_q, arbitrary degree Raviart–Thomas elements (Raviart and Thomas, 1977);
  - BDM_q, arbitrary degree Brezzi–Douglas–Marini elements (Brezzi et al., 1985); and
- $H(\mathrm{curl})$-conforming finite elements:
  - NED_q, arbitrary degree Nédélec elements (first kind, Nédélec (1980)).
- L^2-conforming finite elements:
  - DG_q, arbitrary degree discontinuous Lagrange elements; and
  - CR_1, first degree Crouzeix–Raviart elements (Crouzeix and Raviart, 1973).

Note q is an arbitrary integer to represent the order of polynomials defining finite element spaces.

Form representations FFC supports two different methods to compute element tensors: the tensor contraction representation and the standard quadrature representation. The tensor contraction was the first representation supported inside FFC (Kirby and Logg, 2006) and there were a couple of attempts to improve its performance for code generation (Kirby and Logg, 2007, 2008).

The tensor contraction approach is based on the decomposition of an element tensor into two parts: a reference tensor and a geometry tensor. The reference tensor depends on the underlying PDE and the chosen finite element space and can be computed prior to run-time while the geometry tensor depends on the geometry of a cell and it must be computed at run-time. During assembly the geometry tensor is updated for each cell based on its coordinates.

The tensor contraction representation has been shown to be efficient for classes of problems, but it also has some limitations. This approach cannot be extended to the non-affine isoparametric mapping in an efficient way and it is also not suitable for a class of problems in which functions do not come from finite element spaces (like trigonometric and logarithmic functions). Moreover, the tensor contraction approach does not scale well for moderately complicated and complicated forms. Furthermore,

this representation can also not be used for problems in which a reference element is not uniquely defined. For example in modelling discontinuities using the partition of unity enrichment approaches, reference elements are not unique and they depend on the location of their corresponding cells to discontinuity surfaces.

To overcome these limitations, support for the standard quadrature representation was added to the compiler (Ølgaard and Wells, 2010, 2012a). As the name suggests, the evaluation of the local element tensor for this approach involves a loop over integration points and then adding the contribution from each quadrature point to a local element tensor.

The performance of these two representations has been studied in Ølgaard and Wells (2010). To assess performance quantitatively, the code generation time, the number of FLoating-point OPerations (FLOP) in the generated code, the compilation time of the generated code inside the solver and the assembly time for each representation have been considered. The results showed that the performance greatly depends on the type of problem being solved. For less-complicated PDEs even with a high order finite element basis, the tensor contraction is considerably faster. However, the tensor contraction for some other PDEs with a higher order of derivatives and a larger number of coefficients does not perform efficiently and the quadrature representation is more favorable for such PDEs. In general, the more complex the form (in terms of the number of derivatives and the number of function products), the more likely the quadrature representation is to be preferred.

FFC supports the automatic selection for the "best" possible element tensor representation. The best representation is a representation which is believed to give the best run-time performance. The detail of implementing this strategy to select the best representation inside FFC is explained in Ølgaard and Wells (2010).

Unified Form-assembly Code (UFC)

Another key component of the FEniCS project is UFC (Alnæs et al., 2012). UFC is an interface layer between problem-specific and general-purpose components of finite element programs. The UFC interface defines the structure of the generated code using a form compiler which will be included in a solver. UFC can be employed as an interface in a wide range of finite element methods including the standard Galerkin finite element method and the discontinuous Galerkin finite element method. Generated code that supports the UFC interface can be used with different solver libraries which may differ significantly in their designs. For this reason, the UFC interface is independent of any other FEniCS components and it only consists of a small collection of C++ base classes with pure virtual functions. Data are passed through plain C arrays for minimum dependencies.

Abstract C++ classes defined by UFC can represent common components for

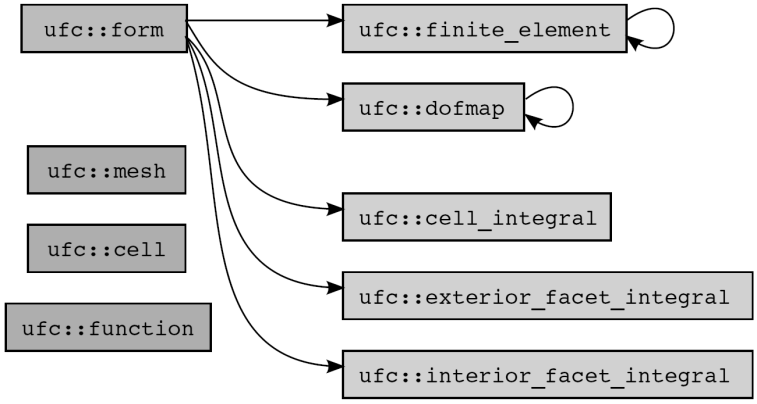


Figure 3.3 A schematic overview of the relation among the UFC classes. Dependencies are shown with arrows. All classes are defined in the `ufc` namespace (Alnæs et al., 2012).

assembling tensors using the finite element method. Moreover, they also provide some components to represent degrees of freedom mapping and finite element spaces.

The communication between a mesh and a coefficient function data as arguments are provided by introducing `ufc::mesh`, `ufc::cell` and `ufc::function` classes. Every argument of a variational form including basis functions (test and trial functions) and coefficient functions is expressed by a `ufc::finite_element` object and a `ufc::dof_map` object. The `ufc::cell_integral`, `ufc::interior_facet_integral`, and `ufc::exterior_facet_integral` classes are used to represent the integrals defined in the variational weak formulations. At the end, a core class called `ufc::form` is defined.

Subclasses of the `ufc::form` class implement member functions which may be called to create `ufc::cell_integral`, `ufc::exterior_facet_integral` and `ufc::interior_facet_integral` objects. These objects in turn know how to compute their respective contributions from a cell or a facet during assembly. The `ufc::form` class also specifies functions for creating `ufc::finite_element` and `ufc::dof_map` objects for the finite element function spaces of the variational form. A schematic overview which shows the relation among different components of UFC is depicted in Figure 3.3.

More information on member functions of the abstract UFC classes and implementations of some member functions of derived classes, generated automatically using the form compiler approach, are presented in Alnæs et al. (2009, 2012).

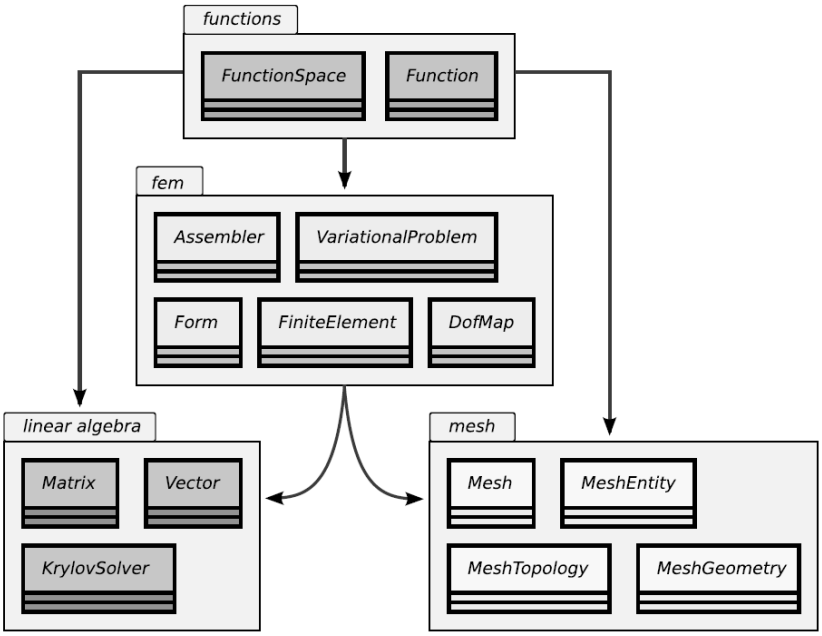


Figure 3.4 A schematic overview of different functionalities inside DOLFIN and their corresponding classes (Logg et al., 2012d)

DOLFIN

DOLFIN is a C++/Python solver library that functions as the main user interface of the FEniCS project. DOLFIN provides a problem solving environment for finite element models based on partial differential equations and implements some core functionalities of FEniCS, including algorithms for manipulating meshes and finite element assembly. At the beginning, DOLFIN was a monolithic and object-oriented C++ finite element library similar to the traditional object-oriented libraries like deal.II (Bangerth et al., 2007) and Diffpack (Langtangen, 2003). Since then, there have been some major improvements inside DOLFIN. It is now designed to rely on code generation for PDE-specific parts of a problem. DOLFIN also uses external libraries to perform some tasks such as sparse linear algebra.

Figure 3.4 presents an overview of the most important classes of the DOLFIN library schematically and classifies these classes in different groups based on their functionalities. DOLFIN contains member classes which provide a range of linear algebra objects and functionality, including vectors, dense and sparse matrices,

various solvers including direct and iterative solvers as well as eigenvalue solvers. For most of this functionality, DOLFIN relies on third-party libraries like PETSc (Balay et al., 2012), Epetra (Herox et al., 2005), uBLAS (Walter et al., 2012) and MTL4 (Gottschling and Lumsdaine, 2011). Nevertheless, a common interface is implemented inside DOLFIN to facilitate communications with these external algebraic libraries.

To manipulate meshes, a mesh library has been implemented inside DOLFIN. The mesh library provides data structures and algorithms for manipulating meshes including the computation of mesh connectivities, mesh refinements, mesh partitioning and mesh intersections. The mesh library includes a collection of classes and it has been optimised to minimize storage requirements and to enable an efficient access to mesh data. More information about the mesh library and its components can be found in Logg (2009).

The interfaces for DOLFIN are provided both in the form of a C++ library and a Python module. These two interfaces are almost identical but in some cases particular features of either C++ or Python cause some minor differences in the interfaces. Except for a few extensions written in Python manually, the bulk of the Python module are generated automatically from the C++ code using SWIG (Beazley et al., 2012).

To use the DOLFIN C++ interface in the solution of partial differential equations, finite element variational problems must be expressed in the UFL form language. This is done by entering the variational forms into separate `.ufl` files and then compiling them using the form compiler FFC. The generated code, which is UFC compatible C++ code, is then included in a DOLFIN-based C++ solver.

The DOLFIN Python interface offers users to employ an intuitive high-level scripting language, similar to the MATLAB language, as well as the strength of an object-oriented language. The Python interface provides some functionalities which are not accessible from the C++ interface. In particular, the UFL form language is consistently embedded inside the Python interface and code generation is automatically handled at run-time. This allows one to obtain the solution of the partial differential equations using a unique file which contains both the variational formulation and the real solver.

More detail on the design considerations and implementations, as well as a number of examples in which DOLFIN has been used as application code, can be found in Logg and Wells (2010) and Logg et al. (2012d).

Other components

A brief introduction is provided for other components of FEniCS in this subsection. FIAT (Kirby, 2012c), which is one of the first FEniCS projects, implements a

mathematical framework to construct a general class of finite elements on reference domains as a Python module. This framework permits one to construct simplicial finite elements with very complicated bases automatically. FIAT provides the basis function back-end for the form compiler and enabling high-order H^1, $H(\mathrm{div})$ and $H(\mathrm{curl})$ elements. FIAT can also tabulate quadrature points and quadrature weights required for the numerical integration. More information about the design of FIAT and its supported functionalities can be found in Kirby (2004, 2006, 2012a).

FErari (Kirby, 2012b) is another component of FEniCS which provides an option within the form compiler to apply optimizations at compile-time. This optimization improves the run-time evaluations of forms represented by the tensor contraction representation. FErari examines the structure of the tensor contraction to check whether it can be performed in a reduced number of arithmetic operations. Interested readers are referred to Kirby et al. (2005), Kirby and Scott (2007) and Kirby and Logg (2012) for more information about FErari.

FEniCS also uses Instant (Westlie et al., 2012) as a Just-In-Time (JIT) compiler. Instant can accept C/C++ code and therefore it can be combined with the code generating tools in DOLFIN and FFC. Instant generates wrapper code needed for making the C/C++ code usable from Python (Wilbers et al., 2012). Moreover, Viper (Skavhaug, 2012) has been introduced as a built-in application for the graphical post processing of functions and meshes.

3.2 Examples

To exhibit the power of the automatic code generation in modelling different physical problems, a number of examples are presented in this section. These examples cover a wide range of mathematical formulations including a Poisson problem, a linear elasticity problem, a nonlinear hyperelasticity problem and a coupled incompressible elasticity problem.

To avoid lengthy and intricate definitions of case-specific function spaces, variational forms of examples in this thesis are presented for the case of homogeneous Dirichlet boundary conditions, despite the computed problems possibly using more elaborate boundary conditions. For each example, a bilinear form a, a linear form L, and a function space V are defined and the discretized fields are indicated by "h" subscripts.

3.2.1 Poisson problem

As a canonical example, a solution of the Poisson equation using the H^1-conforming Galerkin approach is presented. The relevant function space reads

$$V = \left\{ v_h \in H^1(\Omega),\, v_h|_E \in P_k(E) \;\; \forall E \right\}, \tag{3.1}$$

where $\Omega \subset \mathbb{R}^d$ is a domain with a geometrical dimension d and $P_k(E)$ denotes the space of the polynomials of degree k on element E from a finite element mesh. The bilinear and linear forms read

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, dx, \tag{3.2}$$

$$L(v) = \int_\Omega f v \, dx, \tag{3.3}$$

where f is a source term. The aim is to find $u_h \in V$ such that

$$a(u_h, v_h) = L(v_h) \qquad \forall v_h \in V. \tag{3.4}$$

The given variational formulation is used to model a three-dimensional problem with the following definitions of a computational domain, a boundary condition and an input function. The computational domain, a unit cube domain, is defined as

$$\Omega : (1,0,0) \times (0,1,0) \times (0,0,1), \tag{3.5}$$

with a Dirichlet boundary condition

$$u = 0 \quad \text{on} \quad \Gamma_u = (x,y,0) \subset \partial\Omega, \tag{3.6}$$

where $\partial\Omega$ denotes the boundary of the unit cube. A source term

$$f = \sin(x)\sin(y)\sin(z), \tag{3.7}$$

is adopted.

UFL input

The first step is the definition of the variational problem as a UFL input. To define the variational problem, the relevant function space is declared. For this example, a continuous piecewise quadratic Lagrange element defined on tetrahedrons is used:

```
V = FiniteElement("Lagrange", tetrahedron, 2)
```

Next, the test and trial functions are declared on the above space:

```
# Finite element function space
V = FiniteElement("Lagrange", tetrahedron, 2)

# Test and trial functions and source term
u, v = TrialFunction(V), TestFunction(V)
f = Coefficient(V)

# Linear and bilinear forms
a = dot(grad(u), grad(v))*dx
L = f*v*dx
```

Figure 3.5 Complete UFL input for the Poisson problem in three dimensions using quadratic Lagrange elements.

```
u, v = TrialFunction(V), TestFunction(V)
```

A coefficient corresponding to the source term is also declared on the same space:

```
f = Coefficient(V)
```

It has been assumed that the source term f is interpolated in the function space represented by `V`. Finally, the bilinear and linear forms are defined according to the variational weak formulations as given in Equations (3.2) and (3.3):

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

The complete UFL input is presented in Figure 3.5.

The input file is then compiled using FFC to generate a low-level code compatible with UFC. The generated code contains subclasses of the UFC classes which are used to compute integrals, degrees of freedom mappings and finite element spaces. The generated code is then included inside a C++/DOLFIN solver to complete the variational formulations.

C++ Solver

To complete the variational problem, problem-specific data are also required. These data include information about a mesh, coefficient functions, boundary conditions and algebraic solvers. The C++/DOLFIN solver provides these functionalities for the automated framework.

The solver starts with including the automatically generated file and the DOLFIN header file. From the automatically generated header file, `Poisson::FunctionSpace`, `Poisson::BilinearForm` and `Poisson::LinearForm` classes are used. These classes contain data that are specific to the given variational forms.

The Dirichlet boundary, presented in Equation (3.6), is implemented by a subclass of the `SubDomain` class from the DOLFIN library:

```
// Sub domain for Dirichlet boundary condition
class DirichletBoundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return x[2] < DOLFIN_EPS && on_boundary;
  }
};
```

The source term f is supplied with a class from `Expression` subclass:

```
// Source term (right-hand side)
class Source : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0])*sin(x[1])*sin(x[2]);
  }
};
```

Inside the main function, the mesh discretizing the domain is declared by using an instance of the `UnitCube` class:

```
// Create mesh
UnitCube mesh(9, 9, 9);
```

This mesh, that contains $9 \times 9 \times 9 \times 6 = 4374$ tetrahedra, is then used to initialise the function space:

```
// Create Function Space
Poisson::FunctionSpace V(mesh);
```

The Dirichlet boundary condition is created by using the `DirichletBC` class. To create an object of `DirichletBC`, three arguments are required: a function space that the boundary condition applies to, a value of the boundary condition (which is zero in this case) and it is represented by the `Constant` class and a subdomain on which the boundary condition is applied. The definition of the boundary condition is as follows:

```
// Create boundary condition
Constant u0(0.0);
DirichletBoundary boundary;
DirichletBC bc(V, u0, boundary);
```

At the next step, the bilinear and linear forms, which have been generated using FFC, are initialized using the function space `V` and any necessary coefficient functions are then attached:

```
// Define variational problem
```

```
Poisson::BilinearForm a(V, V);
Poisson::LinearForm L(V);
Source f; L.f = f;
```

Now, the linear and bilinear forms have been initialized and the solution of the variational problem is considered. The solution of this problem is represented using an object of `Function` which lives on the function space `V`. To solve this problem, the `solve` function is called with `a == L`, `u` and `bc` as arguments.

```
// Compute solution
Function u(V);
solve(a == L, u, bc);
```

The solution of the variational problem can be manipulated in different ways. It may passed to another variational problem as a coefficient or it can be saved to a file. In this example, the computed solution is saved in the VTK format for visualization purposes using the `File` class.

```
// Save solution in VTK format
File file_solution("poisson.pvd");
file_solution << u;
```

The complete C++ solver interface for the Poisson problem is presented in Figure 3.6.

3.2.2 Discontinuous Galerkin approach to linearised elasticity

The form compiler supports the integration on interior cell facets, which means that code for the discontinuous Galerkin finite element methods can also be generated (Ølgaard et al., 2008). As a simple example, the modelling of an elastic domain $\Omega \subset \mathbb{R}^d$ is assumed in the automated way using a discontinuous Galerkin approach with an interior penalty formulation. The domain Ω is discretized using finite element meshes whose internal facets are denoted by Γ_0. The external boundary of the domain Ω is also denoted by $\partial\Omega$.

The relevant function space V for the L^2-conforming discontinuous Galerkin approach reads:

$$V = \left\{ \boldsymbol{v}_h \in \left(L^2\left(\Omega\right)\right)^d, \boldsymbol{v}_h|_E \in \left(P_k\left(E\right)\right)^d \; \forall E \right\}. \tag{3.8}$$

For a discontinuous interior penalty formulation of the elasticity equation, the bilinear and linear forms with homogeneous Dirichlet boundary conditions read

```
#include <dolfin.h>
#include "Poisson.h"

using namespace dolfin;

// Sub domain for Dirichlet boundary condition
class DirichletBoundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return x[2] < DOLFIN_EPS && on_boundary;
  }
};

// Source term (right-hand side)
class Source : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0])*sin(x[1])*sin(x[2]);
  }
};

int main()
{
  // Create mesh
  UnitCube mesh(9, 9, 9);

  // Create Function Space
  Poisson::FunctionSpace V(mesh);

  // Define boundary condition
  Constant u0(0.0);
  DirichletBoundary boundary;
  DirichletBC bc(V, u0, boundary);

  // Define variational problem
  Poisson::BilinearForm a(V, V);
  Poisson::LinearForm L(V);
  Source f; L.f = f;

  // Compute solution
  Function u(V);
  solve(a == L, u, bc);

  // Save solution in VTK format
  File file_solution("poisson.pvd");
  file_solution << u;
}
```

Figure 3.6 Complete C++ solver for the Poisson example.

(Nguyen, 2008):

$$a(\boldsymbol{u},\boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \nabla \boldsymbol{v}\, dx - \int_{\Gamma_0} [\![\boldsymbol{u}]\!] \cdot \langle \boldsymbol{\sigma}(\boldsymbol{v})\rangle \boldsymbol{n}_+\, ds$$
$$- \int_{\Gamma_0} \langle \boldsymbol{\sigma}(\boldsymbol{u})\rangle \boldsymbol{n}_+ \cdot [\![\boldsymbol{v}]\!]\, ds + \int_{\Gamma_0} \frac{E\alpha}{\langle h\rangle} [\![\boldsymbol{u}]\!] \cdot [\![\boldsymbol{v}]\!]\, ds - \int_{\partial\Omega} \boldsymbol{u} \cdot \boldsymbol{\sigma}(\boldsymbol{v})\boldsymbol{n}\, ds$$
$$- \int_{\partial\Omega} \boldsymbol{\sigma}(\boldsymbol{u})\boldsymbol{n} \cdot \boldsymbol{v}\, ds + \int_{\partial\Omega} \frac{E\alpha}{h} \boldsymbol{u} \cdot \boldsymbol{v}\, ds, \quad (3.9)$$

and

$$L(\boldsymbol{v}) = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v}\, dx, \quad (3.10)$$

where $\boldsymbol{\sigma}(\boldsymbol{u}) = 2\mu(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T) + \lambda \mathrm{tr}(\nabla \boldsymbol{u})\boldsymbol{I}$ is the stress tensor, μ and λ are the Lamé parameters and $\boldsymbol{f}$ is the body force acting on the domain. The penalty parameter for the discontinuous Galerkin formulation is denoted by α. Moreover, h and $\boldsymbol{n}$ are assumed an average cell size and an outward unit normal vector to the interior facets Γ_0, respectively. In the discontinuous Galerkin formulation, the average operator $\langle\cdot\rangle$ and the jump operator $[\![\cdot]\!]$ on the interior facets Γ_0 are respectively defined as $((\cdot)^+ + (\cdot)^-)/2$ and $(\cdot)^+ - (\cdot)^-$ where "+" and "−" superscripts denote the positive and negative sides of the domain intersected by Γ_0, respectively.

The given variational problem is used to model the deformation of a cantilever beam. The domain of the cantilever beam is defined as $\Omega = (0, 0.5) \times (2, 0)$. The beam is fully clamped at the left hand side ($\boldsymbol{u} = \boldsymbol{0}$ at $x = 0$) and subjected to body force $\boldsymbol{f} = (0.0, -10.0)$N. The Young's modulus and the Poisson ratio are $E = 2 \times 10^5$Pa and $\nu = 0.3$, respectively.

The form compiler input for this problem in two dimensions using second order discontinuous Lagrange elements is presented in Figure 3.7. Definitions of the facet normal and the cell size using built-in functions inside UFL in the UFL input and using `avg` and `jump` operators to represent averages and jumps over internal facets are also included. Inside the linear and bilinear forms, the integration over cells, external facets and internal facets are respectively represented by `*dx`, `*ds` and `*dS`.

After providing PDE-specific data as UFL input, a computational domain, boundary conditions and coefficients are also defined inside the solver to complete the variational problem. The complete C++ solver used for modelling the cantilever beam is presented in Figure 3.8. Similar to the previous example, after including the automatically generated code and declaring subclasses for the Dirichlet boundary subdomain and the body force, the mesh as an instance of `Rectangular` from the DOLFIN library is defined. This mesh is used to

```
# Define continuous and discontinuous spaces
element = VectorElement("Discontinuous Lagrange", triangle, 2)

# Create test and trial functions
v, u = TestFunction(element), TrialFunction(element)

# Compute material properties
E, nu  = 2000000.0, 0.3
mu, lmbda = E/(2*(1 + nu)), E*nu/((1 + nu)*(1 - 2*nu))

# Facet normal component, cell size and source term
n, h = element.cell().n, element.cell().circumradius
f = Coefficient(element)

# Penalty parameters
alpha = 4.0

# Stress
def sigma(v):
    return 2.0*mu*sym(grad(v)) \
         + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

# Bilinear and linear forms
a = inner(sigma(u), grad(v))*dx \
  - inner(jump(u), avg(sigma(v))*n('+'))*dS \
  - inner(avg(sigma(u))*n('+'), jump(v))*dS \
  + (E*alpha/avg(h))*inner(jump(u), jump(v))*dS \
  - inner(u, sigma(v)*n)*ds - inner(sigma(u)*n, v)*ds \
  + (E*alpha/h)*inner(u, v)*ds
L = inner(f, v)*dx
```

Figure 3.7 The UFL input for the elasticity equation using the discontinuous Lagrange formulation.

initialise `DG_Elasticity::Functionspace`, which is an automatically generated class from the UFL input. This function space object is then used to declare `DG_Elasticity::BilinearForm` and `DG_Elasticity::LinearForm` objects. After attaching the source term as a coefficient function to the linear form, the variational problem is solved using the bilinear form, the linear form and the Dirichlet boundary condition and its solution is saved as an instance of the `Function` class. Finally, the `Function` object is saved in the `VTK` format for visualization purposes.

3.2.3 Continuous Galerkin formulation for hyperelasticity

FEniCS provides functionalities which can facilitate modelling nonlinear problems. As an example of a nonlinear problem, modeling a hyperstatic domain subjected to static loads is presented. The boundary value problem for the static hyperelastic problem can be expressed as a minimisation problem (Ølgaard and Wells, 2012b).

```
#include <dolfin.h>
#include "DG_Elasticity.h"

using namespace dolfin;

// Sub domain for clamp at the left end
class boundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return std::abs(x[0]) < DOLFIN_EPS && on_boundary;
  }
};

// Body force term
class BodyForce : public Expression
{
public:
  BodyForce() : Expression(2) {}

  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] =  0.0;
    values[1] = -10.0;
  }
};

int main()
{
  // Create mesh and function space
  Rectangle mesh(0, 0, 2.0, 0.5, 44, 11);
  DG_Elasticity::FunctionSpace V(mesh);

  // Define boundary condition
  Constant u0(0.0, 0.0);
  DirichletBoundary boundary;
  DirichletBC bc(V, u0, boundary);

  // Define variational problem
  DG_Elasticity::BilinearForm a(V, V);
  DG_Elasticity::LinearForm L(V);
  BodyForce f; L.f = f;

  // Compute solution
  Function u(V);
  solve(a == L, u, bc);

  // Save solution in VTK format
  File file("dg_elasticity.pvd");
  file << u;
}
```

Figure 3.8 The complete C++ solver for the elasticity equation using the discontinuous Galerkin formulation.

The relevant function space V reads

$$V = \left\{ \boldsymbol{v}_h \in \left(H^1 \left(\Omega \right) \right)^d, \boldsymbol{v}_h|_E \in \left(P_k \left(E \right) \right)^d \ \forall E \right\}. \tag{3.11}$$

The goal is to find $\boldsymbol{u} \in V$ which minimises the total potential energy Π:

$$\min_{\boldsymbol{u} \in V} \Pi, \tag{3.12}$$

where the potential energy Π defined on a reference domain $\Omega \subset \mathbb{R}^d$ reads

$$\Pi = \int_\Omega \psi(\boldsymbol{u}) dx - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} dx - \int_{\partial\Omega} \boldsymbol{h} \cdot \boldsymbol{u} ds, \tag{3.13}$$

where $\psi(\boldsymbol{u})$ is an elastic energy stored in a unit reference domain. Furthermore, $\boldsymbol{f}$ and $\boldsymbol{h}$ are the body force and traction force on the reference domain, respectively. Various stored elastic energy density functions $\psi(\boldsymbol{u})$ can be considered. For this example, a compressible neo-Hookean model is used. To express the neo-Hookean model, the deformation gradient $\boldsymbol{F}$ and the right Cauchy-Green tensor $\boldsymbol{C}$,

$$\boldsymbol{F} = \boldsymbol{I} + \nabla \boldsymbol{u}, \tag{3.14}$$

$$\boldsymbol{C} = \boldsymbol{F}^T \boldsymbol{F}, \tag{3.15}$$

and the Jacobian of the deformation gradient and the first invariant of $\boldsymbol{C}$,

$$J = \det(\boldsymbol{F}), \tag{3.16}$$

$$I_c = \mathrm{tr}(\boldsymbol{C}), \tag{3.17}$$

are used. The stored energy for the neo-Hookean model then reads

$$\psi = \frac{\mu}{2}(I_c - 3) - \mu \ln(J) + \frac{\lambda}{2} \ln(J)^2, \tag{3.18}$$

where μ and λ are material properties.

A minimisation is achieved by computing directional derivatives of the potential energy Π in the direction of $\boldsymbol{v}$ and setting it equal to zero:

$$F(\boldsymbol{u}; \boldsymbol{v}) = D_{\boldsymbol{v}} \Pi = \left. \frac{d\Pi \left(\boldsymbol{u} + \epsilon \boldsymbol{v} \right)}{d\epsilon} \right|_{\epsilon = 0} = 0 \quad \forall \boldsymbol{v} \in V. \tag{3.19}$$

The functional F is linear in $\boldsymbol{v}$ but nonlinear in $\boldsymbol{u}$. The solution for this problem can be obtained using the Newton-Raphson method, which requires the computation of

the Jacobian of F. The Jacobian is computed as:

$$dF(\boldsymbol{u}; d\boldsymbol{u}, \boldsymbol{v}) = D_{d\boldsymbol{u}}F = \left.\frac{dF\left(\boldsymbol{u} + \epsilon d\boldsymbol{u}; \boldsymbol{v}\right)}{d\epsilon}\right|_{\epsilon=0}. \tag{3.20}$$

The Jacobian of F (the stiffness matrix) is used to complete the solution algorithm for the Newton-Raphson method.

The derivation of an analytical expression for the Jacobian can be lengthy and error prone. For this task, the exact automatic differentiation is particularly attractive. Firstly, it eliminates a source of errors. Secondly, it means that if details of the equation of interest are changed, there is no need to re-evaluate the Jacobian by hand. UFL provides a useful functionality for the automatic differentiation and the directional derivative feature can be used to compute the Jacobian from a functional, with the Jacobian filling the role of the bilinear form in the linearised system.

If the C++ interface is chosen for a DOLFIN-based solver, the implementation is divided into two separate files. Figure 3.9 shows the complete code for the implementation of the neo-Hookean model for a hyperelastic domain in UFL using linear Lagrange elements on tetrahedrons. Notice the close relation between the mathematical formulations and the UFL input. In particular, note the automated differentiation of both linear and bilinear forms. This means that a new material law can be implemented by simply changing ψ and the rest of the input file remains unchanged.

After compiling the UFL input file, the generated output is included in the main solver to model a three-dimensional hyperelastic unit cubic domain with given boundary conditions. The extract of C++ solver is presented in Figure 3.10. Note that `FunctionSpace`, `LinearForm`, and `BilinearForm` declared inside `HyperElasticity` namespace are automatically generated classes which contain PDE-specific information. After initializing the function space, the linear and bilinear forms are also initialized and corresponding functions are attached using the automatically generated classes. Then the `solve` function is called to solve the nonlinear variational problem. For nonlinear problems, this function receives `F == 0`, `u`, `bcs` and `J` as input arguments. The solution `u` can be saved in an instance of `File` or it can be visualised directly using the `plot` function.

In the case of a Python-based solver interface, the whole solver is implemented inside a single Python file. This file contains both the variational forms and the solver. An extract of the solver with the Python interface is presented in Figure 3.11. As can be seen, the Python interface is similar to the C++ interface components presented in Figures 3.9 and 3.10. At the first step, the DOLFIN module is imported. Then the mesh is defined which is used to declare the corresponding function space using `VectorFunctionSpace`. The function space is used to define the basis and coefficients

```
# Finite element space
element = VectorElement("Lagrange", "tetrahedron", 1)

# Trial and test functions
v, du = TestFunction(element), TrialFunction(element)

# Displacement from previous iteration, body force per
# unit mass and traction force on the boundary
u     = Coefficient(element)
B, T  = Coefficient(element), Coefficient(element)

# Kinematics
I = Identity(element.cell().d)  # Identity tensor
F = I + grad(u)                 # Deformation gradient
C = F.T*F                       # Right Cauchy-Green tensor

# Invariants of deformation tensors
J  = det(F)
Ic = tr(C)

# Elasticity parameters
mu, lmbda = Constant("tetrahedron"), Constant("tetrahedron")

# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2

# Total potential energy
Pi = psi*dx - inner(B, u)*dx - inner(T, u)*ds

# First variation of Pi (directional derivative
# about u in the direction of v)
F = derivative(Pi, u, v)

# Compute Jacobian of F
dF = derivative(F, u, du)
```

Figure 3.9 The UFL input for the hyperelasticity equation with the neo-Hookean material law using linear Lagrange elements on tetrahedrons.

functions. The automatic differentiation is called to construct the residual and the Jacobian of a nonlinear variational problem in the next step. To obtain the solution of the variational problem, the `solve` function is used. The solution `u` is then saved in a file for visualisation purposes.

3.2.4 Incompressible elasticity

Incompressible elasticity (or Stokes flow) is a coupled problem in which displacement (velocity) and pressure fields are unknowns. Here, an incompressible elasticity example serves as a demonstration of the ease with which multi-physics problems using different function spaces can be dealt with using automated code generation.

```
// Create mesh and function space
dolfin::UnitCube mesh(16, 16, 16);
HyperElasticity::FunctionSpace V(mesh);

// Solution function
dolfin::Function u(V);

// Create linear form
HyperElasticity::LinearForm F(V);
F.mu = mu; F.lmbda = lambda; F.B = B;
F.T = T; F.u = u;

// Create Jacobian dF = F' (for use in nonlinear solver).
HyperElasticity::BilinearForm dF(V, V);
dF.mu = mu; dF.lmbda = lambda; dF.u = u;

// Solve nonlinear variational problem F(u; v) = 0
solve(F == 0, u, bcs, J);

// Save solution in VTK format
dolfin::File file("hyper_elasticity.pvd");
file << u;

// plot solution
plot(u);
```

Figure 3.10 The C++ code extract for the solver of the three-dimensional hyperelasticty problem.

The partial differential equations for the incompressible elasticity on a domain $\Omega \subset \mathbb{R}^d$ consist of a pair of the momentum balance and the incompressibility condition equations:

$$-\mu\Delta\boldsymbol{u} + \nabla p + \boldsymbol{f} = \boldsymbol{0} \qquad \text{in } \Omega, \tag{3.21}$$

$$\nabla \cdot \boldsymbol{u} = 0 \qquad \text{in } \Omega, \tag{3.22}$$

where $\boldsymbol{u}$, p, μ and $\boldsymbol{f}$ are the displacement field, the pressure field, a shear modulus and a source term, respectively. By applying zero Neumann boundary conditions, the standard variational form of the incompressible elasticity equations reads: find $(\boldsymbol{u}, p) \in V \times W$ such that

$$a\left(\boldsymbol{u}; p, \boldsymbol{v}; q\right) = L\left(\boldsymbol{v}; q\right) \quad \forall\ (\boldsymbol{v}, q) \in V \times W, \tag{3.23}$$

where V and W are function spaces for the displacement and the pressure fields,

```
from dolfin import *

# Create mesh and define function space
mesh = UnitCube(16, 16, 16)
V = VectorFunctionSpace(mesh, "Lagrange", 1)

# Define boundary conditions, body force, traction force
# and material properties
[...]

# Define functions
du = TrialFunction(V)      # Incremental displacement
v  = TestFunction(V)       # Test function
u  = Function(V)           # Displacement from previous iteration

# Kinematics
I = Identity(V.cell().d)   # Identity tensor
F = I + grad(u)            # Deformation gradient
C = F.T*F                  # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic = tr(C)
J  = det(F)

# Elasticity parameters
mu, lmbda = Constant(E/(2*(1 + nu))), \
            Constant(E*nu/((1 + nu)*(1 - 2*nu)))

# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) \
    + (lmbda/2)*(ln(J))**2

# Total potential energy
Pi = psi*dx - dot(B, u)*dx - dot(T, u)*ds

# Compute first variation of Pi (directional derivative about
# u in the direction of v)
F = derivative(Pi, u, v)

# Compute Jacobian of F
dF = derivative(F, u, du)

// Solve nonlinear variational problem F(u; v) = 0
solve(F == 0, u, bcs, J);

# Save solution in VTK format
file = File("displacement.pvd");
file << u;
```

Figure 3.11 The code extract for the Python-based solver of the three-dimensional hyperelasticity problem using Linear Lagrange elements.

respectively. The bilinear and linear forms are defined as

$$a\left(\boldsymbol{u};p,\boldsymbol{v};q\right)=\int_{\Omega}\mu\nabla\boldsymbol{u}\cdot\nabla\boldsymbol{v}-p\nabla\cdot\boldsymbol{v}+(\nabla\cdot\boldsymbol{u})q\;dx, \tag{3.24}$$

$$L\left(\boldsymbol{v};q\right)=\int_{\Omega}\boldsymbol{f}\cdot\boldsymbol{v}\;dx. \tag{3.25}$$

In the context of the finite element analysis, stability requirements pose restrictions on the allowable combinations of finite element spaces for the displacement and pressure fields. Developing different finite element spaces for this system is challenging due to the Ladyzhenskaya-Babuška-Brezzi compatibility condition, see Brezzi and Fortin (1991) for details. For example, it is well known that using equal order Lagrange basis functions for the displacement and pressure fields leads to an unstable formulation.

In this example, two families of stable finite element spaces for modelling incompressible elasticity equations are considered. The first family of the stable finite element spaces are the well-known Taylor–Hood elements (Taylor and Hood, 1973). They include a P_k element for the displacement field and P_{k-1} for the pressure field with $k>1$. One instance of the Taylor-Hood elements is using a continuous piecewise quadratic Lagrange basis for the displacement field and continuous piecewise linear Lagrange basis for the pressure field on simplices. The relevant function spaces for the Taylor-Hood elements read

$$V=\left\{\boldsymbol{v}_h\in\left(H^1\left(\Omega\right)\right)^d,\boldsymbol{v}_h|_E\in\left(P_2\left(E\right)\right)^d\;\forall E\right\}, \tag{3.26}$$

$$W=\left\{p_h\in H^1\left(\Omega\right),p_h|_E\in P_1\left(E\right)\;\forall E\right\}. \tag{3.27}$$

The second family of stable elements considered is a first order Crouzeix–Raviart CR_1 element (Crouzeix and Raviart, 1973), a non-conforming element that uses integral moments over the cell facets, as a basis for the displacement field and a discontinuous constant space P_0 for the pressure field. For this case, the relevant function spaces read

$$V=\left\{\boldsymbol{v}_h\in\left(L^2\left(\Omega\right)\right)^d,\boldsymbol{v}_h|_E\in\left(P_1(E)\right)^d\cap\int_F[\![\boldsymbol{v}_h]\!]\,ds=0\;\forall E,\forall F\right\}, \tag{3.28}$$

$$W=\left\{p_h\in\left(L^2\left(\Omega\right)\right),p_h|_E\in P_0(E)\;\forall E\right\}, \tag{3.29}$$

where $[\![(\cdot)]\!]$ denotes a jump across an internal facet F of the triangulation of Ω.

The automated approach allows users to easily switch between formulations corresponding to different function spaces. A UFL input for variational formulations using the Taylor-Hood elements is presented in Figure 3.12. Note the mixed finite

```
# Finite element spaces for displacement and pressure fields
P2 = VectorElement("Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)

# Taylor-Hood element
S = P2 * P1

# Test and Trial functions for displacement and pressure
(v, q) = TestFunctions(S)
(u, p) = TrialFunctions(S)

# Source term and shear modulus
f  = Coefficient(P2)
mu = Constant("triangle")

# Linear and bilinear forms
a = (inner(mu*grad(u), grad(v)) - p*div(v) + div(u)*q)*dx
L = dot(f, v)*dx
```

Figure 3.12 The UFL input for the incompressible elasticity equations using P_2 elements for the displacement field and P_1 elements for the pressure field.

```
# Finite element spaces for displacement and pressure fields
P2 = VectorElement("Crouzeix-Raviart", "triangle", 1)
P1 = FiniteElement("Discontinuous Galerkin", "triangle", 0)

# Mixed element space
S = P2 * P1
```

Figure 3.13 The UFL input for the definition of finite element spaces corresponding to CR_1 elements for the displacement field and DG_0 elements for the pressure field.

element space is constructed by simply multiplying the spaces of the displacement and pressure fields and then the test and trial functions are defined on that space.

To switch to a formulation of the first order Crouzeix–Raviart element for the displacement field and the discontinuous constant space for the pressure field, the function spaces in the input for the form compiler are simply re-defined. The form compiler input for declaring the function spaces with the Crouzeix–Raviart element for the displacement field and the discontinuous constant space for the pressure field is shown in Figure 3.13.

Terrel et al. (2012) have performed a comparative study using different stable finite element spaces including Taylor-Hood elements and CR_1/DG_0 elements in the modelling of incompressible elasticity (Stokes flow) equations. They showed that the compiler approach can speed up the development of finite element codes for different models for incompressible elasticity equations. This is because the development of separate and special-purpose code for each model can be avoided by automatic code generation.

3.3 Summary

The FEniCS project as a framework to automate computational mathematical modelling is presented in this chapter. The design and core components of FEniCS are also studied. Among the core components, UFL, UFC, FFC and DOLFIN are elaborated in more details. At the end of this chapter, examples of using the automated framework are also presented. It is demonstrated that the automated framework facilitate the development of the finite element models and hides the implementation details from users.

In the following chapters, a similar framework will be presented to model discontinuity surfaces in the partition of unity framework. This framework provides a Problem Solving Environment (PSE) on top of FEniCS. It uses components of the FEniCS project for general purposes which are not specific to the partition of unity method. The required specific functionalities for the automated modelling of problems with discontinuities have been implemented inside a compiler and a solver library which will be elaborated in chapter 4 and chapter 5, respectively.

Chapter 4 A form compiler for modeling discontinuities

A form compiler has been developed to facilitate the modelling of domains with discontinuities in the partition of unity framework. The form compiler generates low-level code for modelling discontinuities for a range of physical problems with different underlying partial differential equations and underlying finite element function spaces by using variational forms as an input. This approach is appealing for coupled problems in which different combinations of continuous/discontinuous finite element function spaces are often used. The required code for each combination is obtained by small modifications to the finite element space definitions inside the input for the form compiler.

The partition of unity method compiler, hereafter PUM compiler, will be developed as a black-box for users. As illustrated in Figure 4.1, it receives input as weak forms of partial differential equations based on the partition of unity framework in the UFL syntax (Alnæs and Logg, 2012) and returns low-level C++ code compatible with UFC (Alnæs et al., 2012). The generated low-level code contains components to manipulate degrees of freedom maps, to compute the entries of element tensors, and to evaluate functions defined on the enriched function spaces. The generated code also includes wrapper classes which facilitates using the generated code inside solvers.

The PUM compiler is built on the top of FFC (Logg et al., 2012b). This compiler is licensed as open source software and it can be downloaded from `http://www.launchpad.net/ffc-pum`. Early implementations of the PUM compiler did not use UFL and therefore the compiler was relatively slow in compiling inputs for complicated nonlinear equations (Nikbakht and Wells, 2009). However, with the introduction of UFL which provides a flexible interface for declaring finite element spaces and expressions of weak forms in mathematical notations,

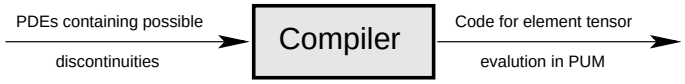


Figure 4.1 Input/output of the PUM compiler. The compiler receives the variational formulations in the UFL syntax as input and generates automatically required C++ code compatible with UFC.

complicated equations can be handled without losing computational efficiency in both code generation and solving stages.

Partition of unity methods can be applied to a variety of problems with non-smooth solutions. At this moment, the automated code generation for the partition of unity methods is restricted for problems that involve discontinuous solutions across surfaces.

This chapter is organized as follows. In the first section, design requirements for the PUM compiler are explained and specific issues related to generating code for the automated modelling of discontinuities are presented. The UFL-based input of the PUM compiler is elaborated in the next section. The UFL syntax provides a powerful and user friendly interface to represent variational formulations defined in the partition of unity framework. The given input file is manipulated inside the PUM compiler to generate low-level C++ code. Later on, the structure of the compiler is explained and different steps of generating C++ code from the UFL input are examined. At the end, the core components of the generated code using the PUM compiler are elaborated.

4.1 Design requirements

To design a form compiler that can generate the required code for modelling discontinuities, the first step is designing an interface to represent the variational formulations. There are three issues that are specific to the variational formulations in the partition of unity framework. The first issue is the definition of function spaces that are discontinuous and restricted to subdomains that contain discontinuity surfaces. These function spaces are created by enriching the standard function spaces with additional function spaces.

Functions defined in domains with discontinuity surfaces may have different values on the positive and negative sides of a discontinuity surface. For functions appearing in surface integrals, it is required to define a restriction syntax which allows the definition of functions on each side of a surface. The second issue is about defining this restriction syntax to represent the discontinuous functions.

The last issue is the computation of surface integrals for problems in which flux-like quantities exist on the discontinuity surfaces. In these problems, a number of terms, appearing in the variational formulations, are integrated along discontinuity surfaces. The form compiler input should be designed such that it can represent the surface integrations.

In the second step, components of the PUM form compiler are designed to manipulate the given variational formulations via a UFL input to generate low-level code. The PUM form compiler is built as an extension of the FEniCS Form Compiler (Logg et al., 2012b); therefore, it follows not only a structure similar to the

structure of FFC, but it also re-uses functionalities and Python modules provided by FFC. Special code is required only for a small number of cells which are close to discontinuity surfaces. Therefore, the FFC functionality can be used for cells away from surfaces without any modification.

For the partition of unity framework, the possible representation for element tensors is limited. This is due to discontinuity surfaces being defined globally in terms of the real coordinates, unlike the shape functions which are usually defined on reference cells. This eliminates the possibility of using the tensor contraction approach (Kirby and Logg, 2007, 2008) which relies on all functions being defined on the reference cell to pre-compute reference element tensors. For this reason, the PUM compiler only supports the conventional quadrature representation (Ølgaard and Wells, 2010, 2012a) for the evaluation of element tensors.

The PUM compiler generates code which is independent of surface representations. This allows one to use the generated code without modification for different surface representations. To permit this decoupling, information related to the surfaces is transferred using objects of an interface layer to the generated code. The objects of the interface layer are responsible to handle extra information (e.g. enriched degrees of freedom and intersected cells) related to the modelling of surfaces in the partition of unity framework. This interface layer is implemented as a base class called `GenericPUM` inside the solver library. The details of the `GenericPUM` class is explained in the next chapter. However, the design of the automated framework is such that the objects of the interface layer are not exposed to users and they are automatically initialized inside the generated code.

4.2 Form compiler input

The PUM compiler uses UFL as an input interface to represent variational formulations defined in the partition of unity framework. To elaborate the input interface, a variational problem corresponding to the modelling of discontinuities in a Poisson problem using the partition of unity formulation is considered. The variational problem is defined in a domain $\Omega \subset \mathbb{R}^d$ containing a discontinuity surface Γ_d. A flux across the discontinuity surface is $q_d = k \llbracket u \rrbracket$, where $k > 0$ and $\llbracket \cdot \rrbracket$ represents jumps over the surface. The bilinear and linear forms read

$$a(u, v) = \int_{\Omega \setminus \Gamma_d} \nabla u \cdot \nabla v \, dx + \int_{\Gamma_d} k \llbracket u \rrbracket \llbracket v \rrbracket \, ds, \tag{4.1}$$

$$L(v) = \int_{\Omega} f v \, dx - \int_{\Gamma_t} g v \, ds, \tag{4.2}$$

where f is a source term on the domain Ω and g is a flux acting on a Nuemann boundary $\Gamma_t \subset \partial\Omega$. The enriched finite element function space reads

$$V = \left\{ v_h \in H^1\left(\Omega \backslash \Gamma_d\right), v_h|_E \in P_k\left(E \backslash \Gamma_d\right) \ \forall E \right\}. \tag{4.3}$$

An important aspect in the partition of unity formulation is the definition of enriched elements. The UFL interface supports defining these types of elements. More classical examples, in which the enriched space concept in UFL is used, are the enrichment of the Lagrange element with bubble functions for use with the Stokes equations or the Raviart–Thomas element for linear elasticity (Arnold et al., 1984a,b).

A finite element function space whose functions are discontinuous across the discontinuity surface (discontinuous function space) is defined by restricting a continuous function space over subdomains around discontinuity surfaces denoted by `dc`. Inspired by the decomposition of the variable field to continuous/discontinuous parts represented in Equation (2.10), $\boldsymbol{u}_h = \bar{\boldsymbol{u}}_h + H_d \hat{\boldsymbol{u}}_h$, a continuous function space is locally enriched with a space that contains a discontinuity. This is done by adding continuous and discontinuous function spaces to create an "enriched" space `E`:

```
Ec = FiniteElement(family, shape, order)
Ed = RestrictedElement(E0, dc)
E  = Ec + Ed
```

In the above, `Ec` is a continuous scalar finite element space defined by three arguments in which `family` represents a finite element type, `shape` denotes `triangle` or `tetrahedron` for two- or three- dimensional problems, respectively. `order` is an arbitrary positive integer determining the order of the approximation polynomials. `Ed` is an instance of `RestrictedElement` and it is created by restricting a continuous space `E0` to a subdomain `dc` which contains a discontinuity surface. The geometry of the surface across which functions are discontinuous will only be known at runtime, hence details of the restriction can only be determined then. An enriched element `E` is created by summation of `Ec` and `Ed`. Unlike the expression `E = Ec*Ed`, which creates a mixed finite element space, the expression `E = Ec + Ed` results in a scalar finite element space. Although different function spaces can be used for `Ec` and `E0`, the majority of the partition of unity formulations use the same space for the continuous function spaces, i.e. `Ec = E0`.

Once an enriched finite element is declared, basis and coefficient functions can be defined on this function space. For example, enriched test and trial functions, and an enriched coefficient function are defined by

```
v = TestFunction(E)
u = TrialFunction(E)
f = Coefficient(E)
```

In the partition of unity formulations, some terms in the variational forms may be integrated across discontinuity surfaces (see for example the second term in Equation (4.1)). Functions appearing in surface integrals are restricted to discontinuity surfaces. The syntax for the surface restriction appearing in the surface integrals is identical to the UFL restrictions on the interior facets. However, the surface restriction syntax is interpreted differently from the interior facet restriction syntax inside the PUM form compiler and thus results in different generated code. For a function `u` appearing in the surface integrals, `u('+')` and `u('-')` are used to restrict it on the positive and negative sides of discontinuity surfaces, respectively. The restriction may be applied to functions from any finite element space but will only affect expressions that are discontinuous across surfaces. To introduce jump and average of the function `u` on a discontinuity surface, `jump(u)` and `avg(u)` are used. The jump and average operators are defined as

```
jump(u) = u('+') - u('-')
avg(u) = (u('+') + u('-'))/2
```

To support the integration over discontinuity surfaces, a new notation (measure) `*dc` inside UFL is defined. Adding `*dc` enables UFL to represent a wider range of variational forms including those defined in the partition of unity framework. In summary, there are four types of integrals supported in the UFL language:

- a cell integral: $\int_\Omega (\cdot)\, dx \leftrightarrow (\cdot)$`*dx`,
- an exterior facet integral: $\int_{\partial\Omega} (\cdot)\, ds \leftrightarrow (\cdot)$`*ds`,
- an interior facet integral: $\int_{\Gamma_0} (\cdot)\, dS \leftrightarrow (\cdot)$`*dS`,
- a surface integral: $\int_{\Gamma_d} (\cdot)\, dc \leftrightarrow (\cdot)$`*dc`,

where Ω, $\partial\Omega$ and Γ_0 represent the domains of cells, external facets and internal facets of meshes, respectively. Moreover, Γ_d denotes the discontinuity surfaces.

Defining enriched spaces, representing surface integrals and restricting functions to discontinuity surfaces are issues specific for representing the variational forms of the partition of unity framework. For the rest, variational forms inside UFL can be expressed just as they are for conventional problems (Alnæs and Logg, 2009; Alnæs, 2012).

To complete this section, UFL input for the partition of unity formulation of the Poisson equation in a two-dimensional domain is presented. The bilinear and linear forms for this formulation are given at the beginning of this section in Equations (4.1) and (4.2), respectively. The UFL input for this problem using third order Lagrange elements on triangles is shown in Figure 4.2. The definition of the Lagrange enriched element and the new integration syntax for the surface integrals defined in the partition of unity framework in the UFL input is emphasized.

```
# Define continuous and discontinuous spaces
elem_cont = FiniteElement("Lagrange", triangle, 3)
elem_discont = RestrictedElement(elem_cont, dc)

# Define enriched space
element = elem_cont + elem_discont

# Define test and trial functions
v, u = TestFunction(element), TrialFunction(element)

# Source term, traction force and surface stiffness
f, g  = Coefficient(elem_cont), Coefficient(elem_cont)
k  = Constant(triangle)

# Define bilinear and linear forms
a = dot(grad(u), grad(v))*dx + k('+')*jump(u)*jump(v)*dc
L = f*v*dx - g*v*ds
```

Figure 4.2 The UFL input for a variational formulation for the Poisson equation using the partition of unity framework presented in Equations (4.1) and (4.2).

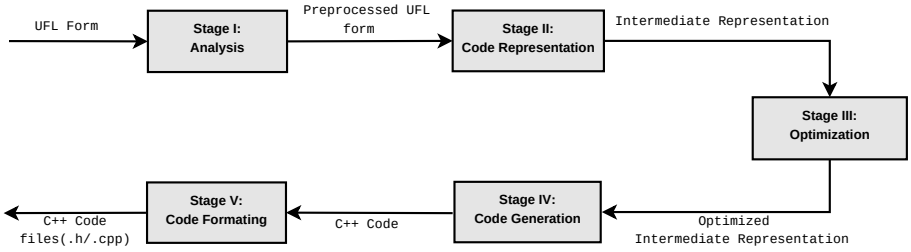


Figure 4.3 The PUM compiler structure and its corresponding data flow.

4.3 Structure of the form compiler

To generate low-level code for modeling discontinuities required for an assembly library, the FEniCS Form Compiler (Logg et al., 2012b) is extended to support new functionalities specific to the partition of unity framework. The extended FEniCS form compiler, PUM compiler, generates the required code to evaluate enriched element tensors corresponding to cell integrals, interior facet integrals, exterior facet integrals and surface integrals. It also generates code to evaluate enriched finite element functions that do not satisfy the interpolation property.

The compilation process is divided into five stages, which are illustrated in Figure 4.3. The initial input is UFL input and the final output is C++ code files and the output generated at each stage serves as input for the following stage. In the following, the functionalities performed in each stage are explained in more detail.

Before starting the compilation, a pre-processing stage is performed. This stage includes interpreting and parsing Python code or a `.ufl` file and storing it as a UFL Abstract Syntax Tree (AST). While Python handles the actual parsing, the operator overloading in UFL implements the transformation of the compiler input to a UFL object.

4.3.1 Analysis of the form language input

This stage involves the analysis of the UFL abstract syntax tree and extracting form metadata (`FormData`). The form metadata includes elements used to define forms, corresponding element maps, coefficients in the forms and integrals for each subdomain. The raw UFL input is preprocessed to obtain a form which can be more easily manipulated by the compiler.

For function spaces defined in the partition of unity framework, `FormData` also includes components to divide the function spaces into standard and enriched parts. The enriched part is used to initialize interface layers (`GenericPUM` objects) which are responsible to transfer information specific to the enriched degrees of freedom data to the generated code.

4.3.2 Intermediate code representation

The PUM compiler receives the preprocessed input in this stage and generates all intermediate representations necessary for code generation. Intermediate representations are computed for finite element spaces and degrees of freedom maps, integrals and forms. These forms contain interfaces to finite elements, degrees of freedom mappings and integrals. While the PUM compiler uses a number of modules from FFC to generate intermediate representations for the standard parts of variational formulations, intermediate representations for the enriched parts are computed using modules from the PUM compiler. The intermediate representations for the enriched parts are used to extract information specific to the partition of unity framework. This information includes the number of enriched spaces corresponding to the finite element function spaces, the standard parts of finite element spaces and a map to determine the configuration of standard and enriched parts inside mixed function spaces.

The intermediate representations are stored as a Python dictionary, mapping names of functions to the data needed for the generation of corresponding code. In some simple cases like `ufc::finite_element::topological_dimension`, this data may be a positive integer (like 2 for a finite element space in a two dimensional domain). In other cases like `ufc::cell_tensor::tabulate_tensor`, the data may be a complex data structure which contains data for both standard and enriched entries of element tensors.

To compute the intermediate representations for integrals, a `Transformer` class provided by UFL is used. This class is a base class for a visitor-like algorithm design pattern (Gamma et al., 1995) to transform an expression tree from one UFL representation to another UFL representation (Alnæs, 2012). The UFL expression tree is a Directed Acyclic Graph (DAG) (Christofides, 1975) which represents the basic arguments (basis functions, coefficient functions and constants) and operators (linear algebra and derivatives) as components of the graph.

Inside the PUM compiler, three sub-classes of the `Transformer` class have been defined to obtain the intermediate representations corresponding to the integrals in the partition of unity framework. A first transformer is `QuadratureTransformer` which is a slightly modified version of the `QuadratureTransformer` class from FFC. This transformer computes the intermediate representations for the standard parts of integrals which will be used to generate code for the standard entries of element tensors. `EnrichedTransformer` is another transformer that is responsible to compute intermediate representations for the enriched parts of integrals.

However, if any coefficient function is defined on enriched function spaces (e.g. a solution from the previous converged stage for a nonlinear problem with discontinuities), the computed intermediate representations are not adequate and extra intermediate representations must also be computed. Since for this type of problem, in addition to element tensor entries corresponding to the enriched degrees of freedom, a number of expressions appearing in the element tensor entries of standard degrees of freedom should also be evaluated using the modified quadrature rules. To obtain the intermediate representations for these entries, a third transformer called `ExpansionTransformer` is defined. An object of the `ExpansionTransformer` class performs the algebraic expansions of the UFL expressions. It receives standard UFL expressions as input and returns expanded expressions as a summation of simple terms. The expanded output is then passed to a function to extract discontinuous terms. These discontinuous terms are then used to generate the intermediate representations for the enriched terms appearing in the standard entries.

4.3.3 Optimisation of the intermediate representations

The automatic code generation provides scopes for employing optimisation that may not be feasible in a hand-generated code. Inside the FEniCS Form Compiler, different strategies for the optimisation are performed based on the chosen representation for the element tensor evaluation (Kirby, 2006, 2012a; Ølgaard and Wells, 2010). As stated before, the supported representation for the PUM compiler is limited to the quadrature representation. For the PUM form compiler, a same optimisation as the symbolic optimisation, used inside FFC for the quadrature representation, has been

employed.

The FFC optimisation modules, as discussed in Ølgaard and Wells (2010), are used to optimise the evaluation of standard entries of element tensors. No other optimisation is performed for the evaluation of the enriched entries of element tensors at this moment.

4.3.4 Code generation from the intermediate representations

The PUM form compiler uses the intermediate representations to generate the actual C++ code for the body of the generated code in this stage. The code is stored as a Python dictionary, mapping names of generated functions to strings containing C++ code for the body of each function. The generated dictionary contains codes for elements, degrees of freedom mappings and integrals.

Most of the code generation for the standard entries is performed using modules from FFC. The code for mapping the standard entries to the local element tensor, as well as computing enriched entries of the local element tensor and evaluating enriched finite element spaces are generated using the PUM compiler.

4.3.5 Code formatting

This stage examines the generated C++ code and formats it according to the UFC interface (Alnæs et al., 2012). At this stage, the actual generation of the C++ files takes place and the generated code inside the Python dictionary from the previous stage is inserted into UFC templates to obtain C++ files.

4.4 Components of the generated code

The generated low-level code, using the PUM compiler, contains components that facilitate modelling discontinuities in the partition of unity framework by removing the need for hand-generated code for the innermost assembly loop. This considerably increases the speed of developing partition of unity models. The generated classes are conformed to the UFC specifications that allow the separation of the problem-specific data from the general-specific data.

4.4.1 The UFC-based classes

In the context of the partition of unity methods, the computation of element matrices and vectors and the degree of freedom maps are affected by the discontinuity surfaces. To utilise the UFC specifications, a standard C++ polymorphic design is followed and sub-classes of the classes defined in the UFC specification are generated. The automatically generated classes are initialised with `GenericPUM` objects (the design

of the GenericPUM interface is described in the next chapter). The GenericPUM objects are used to compute extra information, such as enriched degrees of freedom and intersected cells, which are required to compute element tensors in the partition of unity framework. These objects provide the data which is dependent on the presence of discontinuity surfaces and it is necessary to build element matrices and vectors for the partition of unity models.

The key classes generated by the PUM compiler are derived classes from UFC like ufc::finite_element, ufc::dof_map and ufc::form. Depending on the given mathematical formulation, the initialization of sub-classes of ufc::cell_integrals, ufc::exterior_facet_integrals or ufc::interior_facet_integrals may also be required.

To clarify the structure of the automatically generated code, the core components of generated code for the UFL input of the Poisson equation, presented in Figure 4.2, are considered. The generated code contains classes for the bilinear and linear forms, finite element spaces, degrees of freedom mappings and integrals defined in the variational formulations.

Generated code for the forms

The PUM form compiler generates code to initialise the bilinear and linear forms objects. These objects are sub-classes of the ufc::form class and receive as input pum_objects which is a std::vector containing pointers to the GenericPUM objects.

```
poisson_form_0 a(pum_objects);
poisson_form_1 L(pum_objects);
```

The postfixes “0” and “1” indicate forms with different ranks – “0” for bilinear forms and “1” for linear forms. The 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. These forms are the main interface through which an application developer interacts with the automatically generated code.

Generated code for the finite element spaces

For each finite element space used in a variational statement, the PUM form compiler generates a class derived from the ufc::finite_element class. An object of this class also receives pum_objects as input.

```
poisson_finite_element_2 finite_element(pum_objects);
```

The ufc::finite_element objects provide functionalities, supported in the UFC specification, such as the evaluation of basis functions and their derivatives at the given point, the computation of the space dimension and value rank of the finite

element function space and the interpolation of vertex values from the degrees of freedom values.

To interpolate vertex values of functions defined on enriched finite element spaces, the contribution of enrichment functions to the vertex values must be also considered. This contribution is computed using a member function of the `GenericPUM` class which tabulates basis of enrichment functions at vertices.

Generated code for the degrees of freedom maps

The PUM compiler also generates code to tabulate degrees of freedom. Two different approaches have been examined for the degrees of freedom tabulation in the PUM compiler. At the earlier implementations of the PUM compiler (Nikbakht and Wells, 2009), the required code for both standard and enriched parts of the degrees of freedom was generated inside subclasses of `ufc::dof_map`. This was achieved by passing `pum_objects` to subclasses of `ufc::dof_map`.

However, this approach encountered some problems, most notably in the evaluation of coefficient functions defined on the enriched function spaces (e.g. solutions from the previous converged step in the the nonlinear variational formulations). To avoid these problems, a second approach was developed. In this approach, the PUM form compiler is only used to generate required code to manipulate the standard part of degrees of freedom for each finite element space. The generated code contains a `ufc::dof_map` class. For the Poisson example, an object of degrees of freedom map is initialized as

```
poisson_dof_map_2 dof_map;
```

This object can be used to compute the global dimension and the maximum local dimension of the standard part of degrees of freedom map. It also tabulates the local-to-global map and the local-to-local map from facet degrees of freedom to cell degrees of freedom for the standard part of degrees of freedom.

The enriched degrees of freedom are evaluated using `GenericPUM` objects corresponding to the enriched spaces. For each enriched space, a `GenericPUM` object is constructed to compute required information about the enriched degrees of freedom. To manipulate all degrees of freedom, including the standard and enriched parts, a class called `DofMap` is defined inside the solver library. This class receives the automatically generated sub-classes of `ufc::dof_map` and the `GenericPUM` objects to compute the total degrees of freedom.

Generated code for the integrals

To compute a local element matrix or vector, the form compiler generates subclasses of the integral classes of UFC. To evaluate the local element tensor for integrands

containing the cell integration `dx`, one can create an object for the Poisson example as

```
poisson_cell_integral_0 cell_integral(pum_objects);
```

This object can compute a cell element tensor whose size may vary based on the position of discontinuity surfaces. If a cell is far from the discontinuity surfaces, then the local element tensor is computed without any enriched entries. However, if the support of any node belonging to the cell is intersected with discontinuity surfaces, the enriched entries are also computed in addition to standard entries of the element tensor.

Algorithm 1 presents a framework for the evaluation of the local element tensors for cell integrals. Besides member functions defined in the UFC specifications for the integrals, a private member function called `tabulate_regular_tensor` has been introduced. This member function is responsible for computing the standard entries of the element matrix or vector (the terms which are evaluated on the whole domain). This function contains the standard FFC code with some minor modifications to map the regular entries to the correct positions in the local element tensor.

Inside the `tabulate_tensor` member function, which is the main interface, the `tabulate_regular_tensor` function is called at the first step. Inside this function, values of the basis functions and/or their derivatives are tabulated at pre-defined Gauss quadrature points. The entries of the local element tensor corresponding to the standard part of degrees of freedom are computed by looping over these pre-defined quadrature points computed at compile-time by FFC. These entries are then mapped into suitable positions in the local element tensor.

The number of enriched degrees of freedom is then checked for the current cell. This number is computed using a member function from the `GenericPUM` interface for each discontinuous space. If no enriched degrees of freedom exist, the evaluation of the cell element tensor is completed. Otherwise, the enriched part of the element tensor (the terms that are evaluated on the positive side of the domain) must be computed.

To generate the required code for the enriched entries, relevant `ufc::finite_element` objects are initialised. These objects are used to evaluate values of the basis functions and/or their derivatives at run-time, when the details of discontinuity surfaces are known. After tabulating the standard Gauss quadrature weights and points in the reference cell, a member function of `GenericPUM` receives these tables and computes modified Gauss quadrature weights and points in the physical cell.

To compute the enriched entries of the element tensor, a loop over the modified quadrature points is defined. Before entering to the quadrature loop, a member function from the `GenericPUM` interface is used to tabulate the basis of enrichment

Algorithm 1 The local element tensor evaluation for the cell/surface integrals.

```
1: compute and map the standard entries to the local element tensor for the current
   cell
2: if no enriched dof for the current cell then
3:     End
4: else
5:     if the current cell is intersected then
6:         compute the modified Gauss quadrature rule on the physical domain
7:     else
8:         map the standard Gauss quadrature rule to the physical domain
9:     end if
10:    for all Gauss quadrature points on the current cell do
11:        tabulate values of the basis functions and/or their derivatives
12:        if any coefficient defined on the discontinuous space then
13:            compute the standard entries affected by discontinuous coefficients
14:        end if
15:        compute the enriched entries
16:    end for
17:    if any integration over discontinuity surface (*dc) then
18:        compute the Gauss quadrature rule along surfaces
19:        for all Gauss points on surfaces do
20:            tabulate values of the basis functions and/or their derivatives
21:            compute the enriched entries
22:        end for
23:    end if
24: end if
```

functions at quadrature points. Inside the quadrature loop, the basis functions and/or their derivatives are tabulated for each quadrature point using member functions of `ufc::finite_element`. Parameters for evaluating coefficient functions are also initialised at this step. The tables for the (derivatives of) basis functions and the coefficient parameters in addition to the enrichment functions values (computed outside the quadrature loop) are then used to compute the contributions of the enriched entries of the element tensor at the current quadrature point.

If a surface integral `*dc` exists in the variational formulation, the required code to compute its contribution to the local element tensor is also generated inside sub-classes of `ufc::cell_integral`. The terms appearing in the surface integral are restricted to discontinuity surfaces and they are computed using a quadrature rule defined along the discontinuity surfaces. The required surface quadrature rule is obtained using a member function from the `GenericPUM` interface.

The framework for the evaluation of surface integrals is designed such that it can compute surface integral contributions for functions restricted on the either positive side or negative side of discontinuity surfaces. The restricted terms may have contributions to both standard and enriched parts of the local element tensor.

If there is any coefficient defined on the enriched spaces, the required code is also generated to compute terms inside the standard entries of element tensors which must be evaluated using modified quadrature rules. These terms are obtained using the expansion transformer (which was explained in the previous section). Because the expanded UFL integrands have their own numbering scheme for coefficients, new coefficient parameters are required inside the quadrature loop. These new parameters are then used to evaluate enriched expressions corresponding to the standard degrees of freedom.

To evaluate element tensors for the exterior/exterior facet integrals, the required code may also be generated. The global element tensor corresponding to interior/exterior integrals is obtained by looping over all interior/exterior facets of a corresponding mesh. A similar framework to the previous framework has been designed for this purpose and it is presented in Algorithm 2.

Algorithm 2 The local element tensor evaluation for the interior/exterior integrals.

```
1: compute and map the standard entries to the local element tensor for the current facet
2: if no enriched dof for the current facet then
3:     End
4: else
5:     if the facet is intersected then
6:         compute the modified Gauss quadrature rule on the physical domain
7:     else
8:         map the standard Gauss quadrature rule to the physical domain
9:     end if
10:    for all Gauss quadrature points on the facet do
11:        tabulate values of the basis functions and/or their derivatives
12:        if any coefficient defined on the discontinuous space then
13:            compute the standard entries affected by discontinuous coefficients
14:        end if
15:        compute the enriched entries
16:    end for
17: end if
```

Chapter 5 A Partition of Unity Method library

The design of a partition of unity method library is presented in this chapter. This library provides required functionalities to implement DOLFIN-based solvers to model discontinuities in the partition of unity framework. The DOLFIN-based solvers use the automatically generated code by the PUM form compiler to compute element tensors and degrees of freedom maps. The partition of unity method library, hereafter PUM library, is composed of C++ classes to support the partition of unity framework. It addresses the representation and visualization of surfaces, the management of data related to the enriched degrees of freedom, the definition of enriched function spaces and the evaluation of functions defined on the enriched spaces.

An object oriented design using the polymorphic approach for the PUM library is used. The object oriented design is a design approach organized around "objects" rather than "actions" and "data" rather than "logic". The first step of designing an object oriented library is identifying all required objects for manipulation and their relations to each other. The object oriented programming concepts provide some important benefits for designing mathematical software packages like abstraction for a proper design by using the class concept (Shapira, 2006).

The PUM solver library is built on top of DOLFIN (Logg et al., 2012e) and it is licensed as open source software and it can be downloaded from `http://www.launchpad.net/dolfin-pum`.

This chapter is organized as follows. At the first step, the design requirements of the PUM library are explained. This section is then followed by the implementation details and an overview on the main components of the PUM library. The interfaces of two base classes of the solver library which allow the implementation of various types of surfaces and different enrichment functions are explained in more details. Using the base classes, components have been developed to model problems with non-branching surfaces with discontinuous solutions. These components are also explained in the next sections. In order to use the generated code inside solvers developed using the components of the PUM library and DOLFIN, solver wrapper classes should be generated using the PUM form compiler. The components of the solver wrapper classes are explained at the end of this chapter.

5.1 Design considerations

The PUM library, which is built on top of DOLFIN, has been designed such that a consistent solver interface with the solver interface for DOLFIN-based solvers is obtained. This allows to develop finite element solvers for modelling discontinuities using the well-designed and well-thought interface of the standard problems with some minor modifications.

To use the automatically generated code to model discontinuity surfaces in DOLFIN-based solvers, new functionalities must be introduced inside FEniCS. These functionalities include:

- managing data and tools related to the partition of unity method, which includes evaluating enriched degrees of freedom and modified quadrature rules;
- interacting with the code generated by the form compiler to compute enriched entries of element tensors and local to global mappings to assemble global element tensors;
- representing surfaces such that different approaches can be supported;
- extending surfaces for evolving surface geometry that may happen during simulation; and
- visualizing functions with discontinuities.

Some generic details of how these features are implemented and components of the PUM library are provided in the following sections.

5.2 Core components of the PUM library

The PUM library is designed such that it provides appropriate functionalities specific to the partition of unity framework. The PUM library includes some C++ classes that are defined in the `pum` namespace. These classes together with components of DOLFIN can be used to construct the partition of unity solvers using the PDE-specific code, that is generated by the PUM form compiler.

These member functions, defined in the `pum` namespace, together with components of DOLFIN can be used to construct the partition of unity solvers using the PDE-specific code, that is generated by the PUM form compiler.

The main components of the PUM library can be divided into four groups. The first group contains classes to represent discontinuity surfaces. These classes provide functionalities like defining surfaces and visualizing them. This group includes the following classes:

- `pum::GenericSurface` is an abstract base class to define discontinuity surfaces. Sub-classes of this class implement functionalities like evaluating the position of a point with respect to the given surface, computing normal and tangential vectors on the surface, computing various intersections between different mesh entities and the surface. A wide range of surfaces can be represented using derived classes from this class. The implementation details are hidden from the users and the communication between surface objects and the rest of solver is performed using the interface of the base class.

- `pum::VTKFile` is a class whose objects are used to visualize surfaces using a VTK format. The VTK (Visualization ToolKit) is an open-source C++ library for visualization (Schroeder et al., 2006).

In the second group, a class for handling enriched degrees of freedom is considered. This class acts as a glue layer between the solver and the generated code.

- `pum::GenericPUM` is an abstract base class to handle enriched degrees of freedom for each discontinuous field. Subclasses derived form the base class provide functionalities like tabulating enriched degrees of freedom, tabulating enrichment functions and computing modified quadrature rules for intersected cells or facets.

To define coefficient functions on the enriched function spaces, the following classes have been considered.

- `pum::DofMap` is a derived class from `dolfin::GenericDofMap` to compute degree of freedom mappings including the standard and enriched parts of degrees of freedom. Inside this class, the mapping for the standard part of degrees of freedom are computed using automatically generated `ufc::dof_map` objects and the mapping for the enriched parts of degrees of freedom are computed using the `pum::GenericPUM` objects.

- `pum::FunctionSpace` is a class derived from `dolfin::FunctionSpace` to represent an enriched finite element function space. The enriched finite element function space is created using a `dolfin::Mesh` object, a `dolfin::FiniteElement` object, a `dolfin::GenericDofMap` object and `pum::GenericPUM` objects.

- `pum::SubSpace` is a class derived from `pum::FunctionSpace` to return the sub-components of mixed enriched function spaces.

- `pum::Function` is a derived class from `dolfin::Function` and supports defining functions on enriched function spaces with variable number of

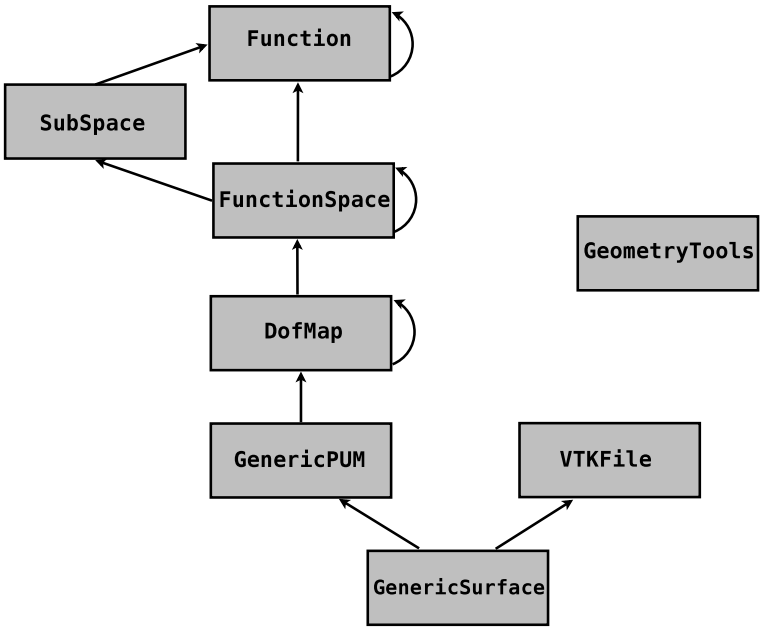


Figure 5.1 The UML diagram of the core components of the PUM library defined in the `pum` namespace.

degrees of freedom. These enriched function spaces are represented by `pum::FunctionSpace` objects.

The last group contains a common class to perform computations on geometry.

- `pum::GeometryTools` is a class with static member functions to perform geometrical calculations. This class covers functionalities like computing volumes or areas of geometrical entities, determining equations of surfaces passing through two, three or four points, computing intersection points between two lines or a line and a plane and computing sub-triangulations for a cell or a facet intersected by a surface.

A UML diagram showing the relation among the core components of the PUM solver library is presented in Figure 5.1. It is worth to re-emphasize that all classes are defined in the `pum` namespace to avoid any name duplication with classes existing in the standard DOLFIN library. These classes can also represent mixed elements, in which case it is possible to obtain `pum::Function`, `pum::FunctionSpace` and `pum::DofMap` objects for each sub-element in a hierarchical manner.

The `pum::GenericPUM` and `pum::GenericSurface` classes are two abstract base

classes defined in the library. In the following, these two classes are elaborated in more detail.

5.2.1 `pum::GenericPUM` base class

The `pum::GenericPUM` class defines an abstract interface through which the generated code can retrieve necessary data from the solver library. Together with the UFC specification, it can define an interface for interactions between the generated code and the solver environment. The objects derived from subclasses of `pum::GenericPUM` provide all required functionality to evaluate extra data related to the enriched degrees of freedom in the generated code.

Design considerations

The `pum::GenericPUM` class should provide interfaces to transfer the required information of the partition of unity framework to the generated code. The `pum::GenericPUM` interface should support the degree of freedom manipulation, and specifically the management of the enriched degrees of freedom. For this purpose, the enriched degrees of freedom and their coordinates for each cell must be tabulated. The total number and the maximum number of enriched degrees of freedom for cells in the computational mesh as well as the number of enriched degrees of freedom for each cell must also be computed.

The `pum::GenericPUM` interface should also support the manipulation of enrichment functions. The values of enrichment functions are required to compute enriched element tensors and to post-process functions defined on discontinuous spaces. The post-processing of discontinuous functions are performed by interpolating discrete values of discontinuous functions defined on nodal points to the cell vertices.

To evaluate enriched entries corresponding to the cells intersected with discontinuity surfaces, the `pum::GenericPUM` interface provides information about modified quadrature rules. The interface layer contains member functions that indicate when a modified quadrature is required on a given cell or facet. There should be also member functions to return tailored quadrature schemes for the intersected cells or facets. Note that most of the computations related to the modified quadrature rules are performed in a class related to representing surfaces and the interface layer just provides access to this information in the generated code.

In the partition of unity framework, local dimensions of element tensors may change during a simulation as a surface evolves. This change is because of the introduction of new enriched degrees of freedom for the evolved surface. Therefore, `pum::GenericPUM` should be able to update data corresponding to the enriched

degrees of freedom to compute new entries of element tensors and map these entries to appropriate positions.

Interface

The interface of the `pum::GenericPUM` class is presented in Figures 5.2 and 5.3. The member functions of `pum::GenericPUM` perform the basic types of functionalities explained in the preceding. For tabulating enriched degree of freedom maps and their coordinates, `tabulate_enriched_dofs`, `tabulate_enriched_local_dofs` and `tabulate_enriched_coordinates` member functions have been introduced. To compute the number of enriched degrees of freedom locally and globally, `enriched_global_dimension`, `enriched_local_dimension`, `enriched_max_local_dimension` member functions have been designed. For numerical integration, `pum::GenericPUM` provides `modified_quadrature`, `facet_quadrature_rule`, `cell_quadrature_rule` and `surface_quadrature_rule`. From the `pum::GenericPUM` interface, `tabulate_enriched_basis` is the member function which computes enriched function values. `update` member function inside the base class is used to modify data related to the enriched degrees of freedom when a surface evolves.

5.2.2 `pum::GenericSurface` base class

Surface representation is an active area of research in the context of the partition of unity framework and the `pum::GenericSurface` interface permits a high degree of flexibility in this respect. The use of the base class `pum::GenericSurface` permits different surface representations to be used interchangeably with the generated code. Moreover, the implementation details of different surface representations are hidden from the users and the communication between surface objects and the rest of the solver is performed using the `pum::GenericSurface` interface. The `pum::GenericSurface` interface also provides various functions for querying a surface object.

Design requirements

To design an abstract class which can support different surface representations, the following functionalities should be supported inside subclasses derived from `pum::GenericSurface`: determining surface geometry, computing intersections and evaluating modified quadrature rules. Determining surface geometry is an important functionality which must be supported inside `pum::GenericSurface`. This functionality includes checking whether a given point is on the surface,

```
using namespace dolfin;
using namespace pum;

class GenericPUM
{
  public:

  /// Return total number of enriched dofs
  virtual unit enriched_global_dimension() const = 0;

  /// Return number of enriched extra dofs for the given cell
  virtual unit enriched_local_dimension(const ufc::cell&
               cell) const = 0;

  /// Return the maximum number of 'enriched' dofs for a cell
  virtual unit enriched_max_local_dimension() const = 0;

  /// Tabulate enriched dofs for the current cell
  virtual void tabulate_enriched_dofs(unit* dofs,
          const ufc::cell& ufc_cell, unit local_offset = 0,
          unit global_offset = 0) const = 0;

  /// Tabulate values of the enriched basis at points in a cell
  virtual void tabulate_enriched_basis(std::vector<double>&
               values, const std::vector<double>& points,
               const ufc::cell& ufc_cell) const = 0;

  /// Tabulate local enriched dofs
  virtual void tabulate_enriched_local_dofs(std::vector<unit>&
               local_dofs, const ufc::cell& ufc_cell) const = 0;

  /// Tabulate coordinates of enriched dofs
  virtual void tabulate_enriched_coordinates(std::vector<double>&
               coordinates, const ufc::cell& ufc_cell) const = 0;

  /// Indicate whether modified quadrature is required for
  /// a given cell
  virtual bool modified_quadrature(const ufc::cell&
                                   ufc_cell) const = 0;

  /// Compute modified quadrature of a cell.
  virtual void cell_quadrature_rule(QuadratureRule& modified,
               ConstQuadratureRule& standard,
               const ufc::cell& ufc_cell) const = 0;
```

Figure 5.2 The `pum::GenericPUM` class interface (part 1)

computing the normal distance of a point from the surface and computing normal and tangential vectors of the surface at a given point.

Obtaining information about intersections of a surface with various mesh entities (e.g. edges, faces and cells) must also be supported for `pum::GenericSurface`.

```
  /// Compute quadrature rule for discontinuity surface
  virtual void surface_quadrature(QuadratureRule& modified,
               ConstQuadratureRule& standard,
               const ufc::cell& ufc_cell) const = 0;

  /// Indicate whether modified quadrature is required on a
  /// given local facet index
  virtual bool modified_quadrature(const ufc::cell& ufc_cell,
                                   unit facet) const = 0;

  /// Compute modified quadrature for a facet of cell.
  virtual void facet_quadrature_rule(QuadratureRule& modified,
               ConstQuadratureRule& standard,
               const ufc::cell& ufc_cell,
               unit facet) const = 0;

  /// Update GenericPUM data for changes in surface
  virtual void update() = 0;
};
```

Figure 5.3 The `pum::GenericPUM` class interface (part 2)

Checking whether a given cell or a given edge (face) of a cell is intersected, checking whether a given cell lies on the boundary of the surface and computing intersection points of a given edge of a cell with the surface are amongst functionalities supported in this category. Note that intersection points between the surface and edges are computed for each cell separately and thus intersection points for an edge might be different within different cells sharing this edge. This allows the support of surfaces that are discontinuous within cells (see for example Gasser and Holzapfel (2005)) using objects of classes derived from the `pum::GenericSurface` class.

For the cells intersected with surfaces, `pum::GenericSurface` must also compute modified quadrature rules to perform numerical integrations. Computing modified quadrature schemes to sub-cells or sub-facets of intersected cells, computing quadrature schemes for intersection surfaces, computing sub-volumes of intersected cells are functionalities that are performed for numerical integrations inside `pum::GenericSurface`. Member functions of `pum::GenericSurface` should receive standard quadrature rules on reference elements and return modified quadrature rules on physical elements.

Interface

Figures 5.4 and 5.5 present the interface of the `pum::GenericSurface` class. The member functions `on_surface`, `f0_eval`, `normal` and `tangent` are used to implement functionalities for surface geometry. For functionalities related to the intersection between surfaces and meshes, `intersects`, `intersects_boundary`, `intersections`

```
using namespace dolfin;
using namespace pum;

class GenericSurface
{
public:

  typedef std::pair<const std::vector<double>,
          const std::vector<double> > ConstQuadratureRule;
  typedef std::pair<std::vector<double>,
          std::vector<double> > QuadratureRule;

  /// Check whether a point is on the surface
  virtual bool on_surface(const Point& p) const = 0;

  /// Check whether a point is  on the surface
  virtual bool on_surface(const Point& p,
                          const Cell& cell) const = 0;

  /// Evaluate the signed function f0 = 0 on surface, f0 < 0
  /// on one side, f0 > 0 on the other side
  virtual double f0_eval(const Point& p,
                         const Cell& cell) const = 0;

  /// Normal vector to the surface at a point
  virtual void normal(std::vector<double>& n,
                      const Point& p) const = 0;

  /// Tangent vector to the surface at a point
  virtual void tangent(std::vector<double>& t,
                       const Point& p) const = 0;

  /// Normal vector to the surface at a point in a cell
  virtual void normal(std::vector<double>& n, const Point& p,
                      const Cell& cell) const = 0;

  /// Tangent vector to the surface at a point in a cell
  virtual void tangent(std::vector<double>& t, const Point& p,
                       const Cell& cell) const = 0;

  /// Determine whether cell and discontinuity intersect
  virtual bool intersects(const Cell& cell) const = 0;
```

Figure 5.4 The `pum::GenericSurface` class interface (part 1)

and `volumes` member functions are introduced. For the numerical integrations, `surface_quadrature`, `facet_quadrature` and `cell_quadrature` member functions are defined.

```
  /// Check whether a facet is intersected by discontinuity
  /// surface
  virtual bool intersects(const Facet& facet,
                          uint cell_index) const = 0;

  /// Determine whether cell contains discontinuity boundary
  virtual bool intersects_boundary(const Cell& cell) const = 0;

  /// Determine whether edge and discontinuity intersect
  virtual bool intersects(const Edge& edge,
                          uint cell_index) const = 0;

  /// Intersection point with an edge
  virtual dolfin::Point intersection(const Edge& edge,
                                     uint cell_index) const = 0;

  /// Compute volume of cell on either side of the surface
  virtual std::pair<double, double>
                            volumes(const Cell& cell) const = 0;

  /// Apply quadrature scheme to the intersection surface
  virtual void surface_quadrature(QuadratureRule& output_rule,
                          ConstQuadratureRule& input_rule,
                          const Cell& cell) const = 0;

  /// Apply quadrature scheme to sub-cells on either side
  /// of a surface
  virtual void cell_quadrature(QuadratureRule& output_rule,
                          ConstQuadratureRule& input_rule,
                          const Cell& cell) const = 0;

  /// Apply quadrature scheme to sub-facets on either side
  /// of a surface
  virtual void facet_quadrature(QuadratureRule& output_rule,
                          ConstQuadratureRule& input_rule,
                          const Facet& facet,
                          uint cell_index) const = 0;
};
```

Figure 5.5 The `pum::GenericSurface` class interface (part 2)

5.3 Enriched degrees of freedom manipulation

As an example of enriched degrees of freedom manipulation, the design of an interface layer between the generated code and the solver for modeling problems whose solutions exhibit jumps on surfaces is presented in this section. For these problems, the Heaviside function is used as the enrichment function to enhance the classical finite element approximations.

```
using namespace dolfin;
using namespace pum;

class PUM : public GenericPUM
{
public:

  /// Constructor
  PUM(const std::vector<const GenericSurface*>& surfaces,
      const Mesh& mesh, const dolfin::DofMap& standard_dof_map,
      const std::string& support_type = "vertex");

  /// Destructor
  ~PUM();

  /// GenericPUM implementation
  [...]

private:

  /// surfaces, mesh, standard dof map and support type
  const std::vector<const GenericSurface*> surfaces;
  const Mesh& mesh;
  const dolfin::DofMap& standard_dof_map;
  const std::string support_type;
  [...]
};
```

Figure 5.6 A code extract from the `pum::PUM` class interface.

5.3.1 Implementation

To implement the interface layer that is required to communicate between the generated code and the solver library for the problems with discontinuous solutions over surfaces, a `pum::PUM` class is introduced. This class is derived from the abstract base `pum::GenericPUM` class. An extract of the `pum::PUM` class interface is presented in Figure 5.6. Considering the interface of the `pum::PUM` class, an instance of this class for each discontinuous field is created by

```
pum::PUM pum_object(surfaces, mesh, standard_dof_map,
                    support_type);
```

where `surfaces` is a standard template library vector (`std::vector`) for pointers of `pum::GenericSurface` objects, `mesh` is a mesh, `standard_dof_map` is an object of the `dolfin::DofMap` class containing information of the nodal mapping for the standard degrees of freedom and finally `support_type` is a string that denotes the support type for the underlying finite element space. For continuous Lagrange elements, discontinuous Lagrange elements and $H(\mathrm{div})/H(\mathrm{curl})$ elements, corresponding

supports are defined vertex-wise, cell-wise and facet-wise, respectively. To consider different support types, `"vertex"`, `"cell"` or `"facet"` can be passed as an input argument for `support_type`. The default argument for `support_type` is `"vertex"` (which is the support type for the Lagrange elements).

For a problem with multiple discontinuous fields, such as a three-dimensional incompressible elasticity problem, a `pum::PUM` object is associated with each field (one for each displacement component plus one for the pressure field),

```
std::vector<const pum::GenericPUM*> pum_objects;

pum_objects.push_back(&pum_object_u);
pum_objects.push_back(&pum_object_u);
pum_objects.push_back(&pum_object_u);
pum_objects.push_back(&pum_object_p);
```

where `pum_object_u` and `pum_object_p` are instances of the `pum::PUM` class corresponding to each component of the displacement fields and the pressure field. The container of `pum::GenericPUM` objects is passed to the generated code to compute enriched entries of the element tensors and the degrees of freedom corresponding to the incompressible elasticity problem.

As already alluded in 5.5, with the current design of the PUM compiler, details of initializing the `pum::PUM` objects are hidden from users. These objects are automatically initialized in the wrapper classes inside the generated code. All required information to create the `pum::PUM` objects including the `dolfin::DofMap` objects and the support type are automatically determined for each object at compile time.

5.4 Non-branching continuous surface representation

An approach to represent non-branching discontinuity surfaces using simple mathematical functions is discussed in this section. This is not the most general approach to represent all possible geometrical configurations but it does permit a wide range of initial surface paths to be considered. The non-branching surfaces are expressed in the physical coordinates. Intersections between various mesh entities and the surfaces are computed explicitly. The surface evolution is also achieved by adding new sub-surfaces to the initial configuration of the non-branching surfaces. These sub-surfaces are defined on cells in the neighborhood of the discontinuity surface boundaries

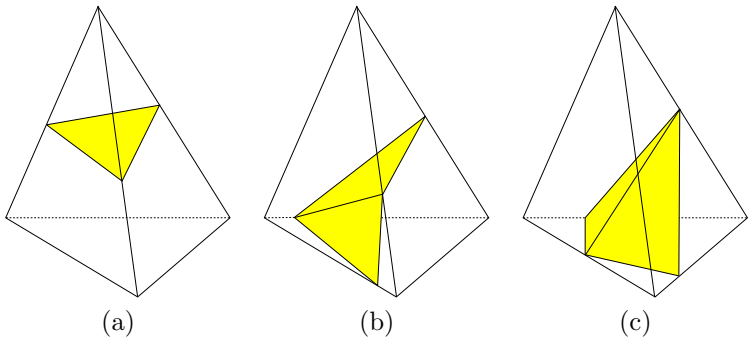


Figure 5.7 Different configurations for the intersections between a discontinuity surface and a tetrahedron cell: (a) with three edge intersection points (b) with four edge intersection points dividing the cell into two parts such that each part has two vertices (c) with four edge intersection points dividing the cell into two parts such that one part with one vertex and the other part with three vertices.

5.4.1 Surface representation

Discontinuity surfaces are approximated by continuous sub-surfaces within computational cells. For triangular cells, discontinuity surfaces are approximated using straight line segments within cells. For tetrahedron cells, a discontinuity surface for each cell (sub-surface) is approximated using a tri-linear equation:

$$ax + by + cz + dxyz + f = 0, \tag{5.1}$$

where a, b, c, d and f are constants and x, y and z are Cartesian coordinates. Note that this equation can also be used to represent plane surfaces if $d = 0$.

For tetrahedron cells, three different configurations for sub-surfaces exist based on the number of edge intersection points and the arrangement of intersected edges. If a surface intersects edges of a tetrahedron cell in three points, the generated sub-surface is a plane as presented in Figure 5.7(a). However, If the surface intersects four edges of the tetrahedron cell, sub-surfaces can not always be represented by a single plane but they can be approximated by two intersecting planes. Two different configurations for this case with four intersection points exist based on the arrangement of the intersected edges. Figure 5.7(b) shows a configuration in which the cell is divided into two parts such that each part has two vertices. Figure 5.7(c) represents another configuration where the cell is divided into two parts such that one part has one vertex and the other part has three vertices.

Surface evolution

A discontinuity surface may evolve during simulation. An approach to represent discontinuity surfaces must be designed such that it can update the initial configurations to consider the possible evolution. For two-dimensional problems, discontinuity surface evolutions have been well-studied and they have been used in a wide range of problems (see for example Wells and Sluys (2001a)). However, the evolution of discontinuity surfaces in three-dimensional problems is not easy. Using a similar approach to two-dimensional problems eventually yields a non-smooth surface in three-dimensional settings. Different algorithms exist for the evolution of three-dimensional surfaces, see for example Areias and Belytschko (2005) and Gasser and Holzapfel (2005).

In this work, an evolution algorithm called extended local evolution is developed. Using this algorithm helps to represent the evolved surfaces by continuous sub-surfaces, defined by the tri-linear equations. This algorithm is somehow an extension of the local evolution algorithm, presented in Areias and Belytschko (2005). But unlike the local evolution algorithm, which was limited to coplanar or slightly kinked surfaces, the proposed algorithm can be used to evolve a larger group of surfaces including relatively kinked surfaces which have a large gradient of surface normal vectors.

Figure 5.8 illustrates different configurations that may happen during a surface evolution in a tetrahedron cell. A configuration presented in Figure 5.8(a) may happen at the first step of the surface evolution when there is no initial discontinuity surface. In this case, the evolved surface is a plane and it is determined using a given normal $\boldsymbol{n}$. The given normal $\boldsymbol{n}$ may be determined by external conditions (for example the stress fields in the crack propagation problems). If one face is already intersected, the evolved surface can be either triangular or quadrilateral depending on the direction of the evolution, as presented in Figure 5.8(b). In this case, the evolved surface is determined using two pre-existing edge intersection points and a modified normal $\boldsymbol{n}_m$. The modified normal $\boldsymbol{n}_m$ is computed by adjusting the given normal $\boldsymbol{n}$ to satisfy the continuity restriction (Areias and Belytschko, 2005):

$$\boldsymbol{n}_m = \boldsymbol{n} - \frac{\boldsymbol{n} \cdot (\boldsymbol{A} - \boldsymbol{B})}{\|(\boldsymbol{A} - \boldsymbol{B})\|^2}(\boldsymbol{A} - \boldsymbol{B}), \tag{5.2}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are two pre-existing edge intersection points. In the remaining cases, evolved surfaces are independent of the normal $\boldsymbol{n}$ and they are fully determined by geometrical restrictions imposed by the pre-existing surfaces. If two faces are already intersected, the evolved surfaces can be either a triangle or a quadrilateral based on the arrangement of the existing intersected edges. If the intersected edges meet each other at one vertex, then the evolved surface is triangular as depicted in Figure 5.8(c).

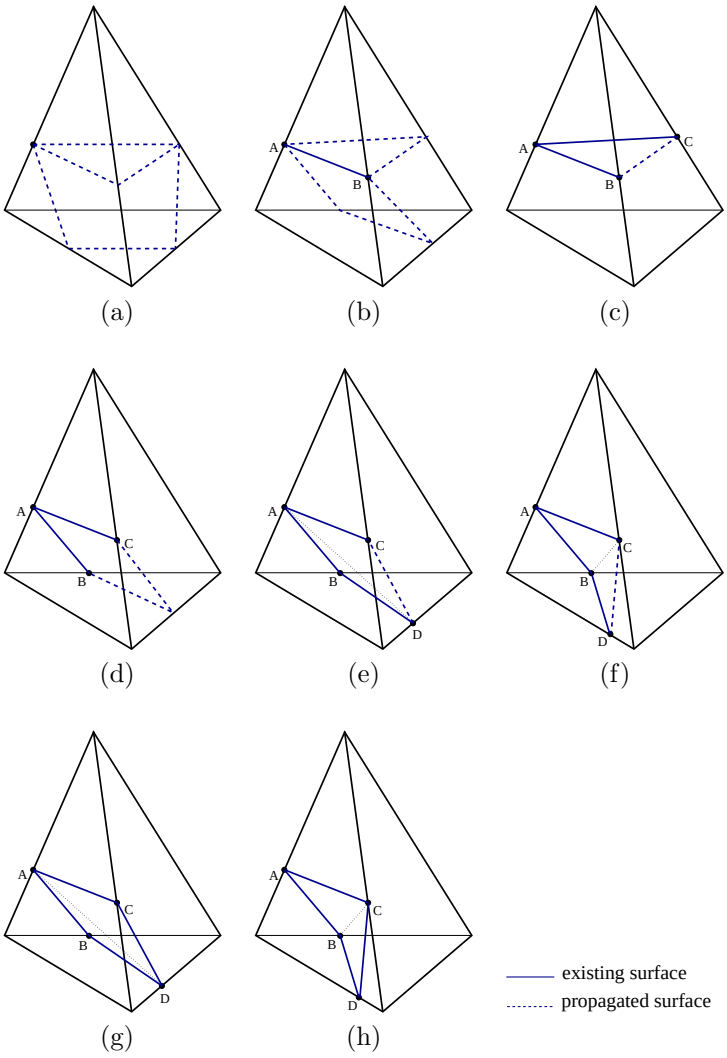


Figure 5.8 Different configurations for a surface evolution inside a tetrahedron cell.

Otherwise, the evolved surface is quadrilateral as presented in Figure 5.8(d).

When three faces are already intersected, evolved surfaces cannot always be represented by a single plane. These evolved surfaces are approximated by two intersected planes as illustrated in Figure 5.8(e) and Figure 5.8(f). In some cases for the three-dimensional surface evolutions, it is also possible to have all four faces

already intersected. In these cases when the surface evolves, no newly intersected face is introduced as presented in Figure 5.8(g) and Figure 5.8(h). The tri-linear equations given in Equation (5.1) are used to represent these non-planar evolved surfaces.

5.4.2 Implementation

The non-branching discontinuity surface has been implemented inside the `pum::NonBranchingSurface` class. This class inherits from the `pum::GenericSurface` class and implements the functionalities defined inside the base class. To evolve surfaces that are represented by `pum::NonBranchingSurface` objects, another class called `pum::SurfaceExtender` with static member functions has been implemented. The interface of `pum::NonBranchingSurface` is designed such that other parts of the PUM solver library and the generated code are unaffected by the approach with which the surface is represented internally.

An extract of the interface for the `pum::NonBranchingSurface` class is presented in Figure 5.9. In two dimensions, a discontinuity surface (a line) can be constructed in two ways. A curved discontinuity surface is defined by a mesh, end points and a user-defined function.

```
pum::NonBranchingSurface d(mesh, end_points, shape);
```

where `mesh` is a `dolfin::Mesh` object, `end_points` is a pair containing end points and `shape` is a user-defined function representing shape which is given as an object of `dolfin::Expression`. Careful attention must be devoted to the functions representing the shapes since the surface path cannot be "double back". For the special case of a straight line, a `pum::NonBranchingSurface` object can be constructed with only a mesh and end points.

```
pum::NonBranchingSurface d(mesh, end_points);
```

In three dimensions, a surface location is determined using two level set functions ϕ and ψ which are scalar functions of x, y and z as Cartesian coordinates. A point on the surface is characterized by $\phi < 0 \cap \psi = 0$ and a point on the boundary of the surface (crack–tip if the surface represents a crack) is characterized by $\phi = 0 \cap \psi = 0$. For three-dimensional problems, a surface instance is created by

```
pum::NonBranchingSurface d(mesh, shape, boundary);
```

where `mesh` is a mesh, `shape` and `boundary` are user defined functions representing scalar level set functions. In both two and three dimensions, the functions used to describe the surface make use of the `dolfin::Expression` abstraction in DOLFIN. The interface to the `pum::NonBranchingSurface` class is designed such that it is dimension-independent.

```
using namespace dolfin;
using namespace pum;

class NonBranchingSurface : public GenericSurface
{
public:

  /// Constructor for a surface (line) in a 2D space.
  /// A point (x, y) on the surface satisfied f0(x,y) = 0.
  /// The boundary of the surface is defined by the end points.
  NonBranchingSurface(const Mesh& mesh,
                      const std::pair<Point,Point>& end_points,
                      const GenericFunction& f0);

  /// Constructor for a surface (straight line) in a 2D space.
  /// The boundary of the surface is defined by the end points.
  NonBranchingSurface(const Mesh& mesh,
                      const std::pair<Point,Point>& end_points);

  /// Constructor for a surface in a 3D space. A point (x,y,z)
  /// on the surface satisfies f0(x,y,z) = 0 and f1(x,y,z) <= 0.
  NonBranchingSurface(const Mesh& mesh, const GenericFunction&f0,
                      const GenericFunction& f1);

  /// Destructor
  ~NonBranchingSurface();

  /// GenericSurface implementation
  [...]

private:

  /// Underlying mesh
  const Mesh& mesh;

  /// Start and end points for a surface (line) in 2D
  const std::pair<Point, Point>* end_points;

  /// Signed distance function (f0(x)=0 -> possibly on surface)
  const GenericFunction* f0;

  /// Signed distance function (for x such that f0(x)=0,
  /// f1(x)<=0 -> on surface)
  const GenericFunction* f1;

  [...]
};
```

Figure 5.9 A code extract from the `pum::NonBranchingSurface` class interface.

To compute modified quadrature rules for surfaces represented by the `pum::NonBranchingSurface` class, a separate class called `pum::SurfaceQuadrature` with static member functions has been defined. This class provides the following functionalities:

- computing the modified quadrature schemes for sub-cells or sub-facets on either side of a surface; and
- computing quadrature schemes for the intersection interfaces between surfaces and cells.

For surface evolutions, another class called `pum::SurfaceExtender` with static member function is defined. This class is a friend class of the `pum::NonBranchingSurface` class; thus, it can modify private data of the `pum::NonBranchingSurface` class. Inside the `pum::SurfaceExtender` class, the extension of a surface is handled by updating the corresponding level set functions.

Discontinuity surfaces can be extended within either one cell or a pre-determined neighborhood in front of the surface boundary. If a surface evolution within just one cell is desired, one can call a member function of `pum::SurfaceExtender`, `extend`, with the following arguments

```
pum::SurfaceExtender::extend(surface, mesh, normal, cell_index);
```

where `surface` is an evolving `pum::NonBranchingSurface` instance, `mesh` is a mesh, `normal` is a normal vector determining the direction of the evolved surface and `cell_index` is the index of a cell containing the boundary of surface in which the evolution happens. The discontinuity surfaces can also be extended in a pre-determined neighborhood in front of the surface boundary by

```
pum::SurfaceExtender::extend(surface, mesh, normal, distance);
```

where `surface`, `mesh` and `normal` are defined as before and `distance` determines a neighborhood in which extension happens.

To illustrate this approach for a surface evolution, an example of a three-dimensional surface evolution inside a unit cube is presented in Figure 5.10. The initial surface, represented by level set functions $\psi = z - 0.34 = 0$ and $\phi = (x - 0.5)^2 + (y - 0.5)^2 - 0.08 \leq 0$, evolves freely with an initial normal equal to $(0.00, 0.392, 0.920)$ until it reaches to the external faces of the domain. As can be observed, the evolved surface is not flat and it is continuous in the whole domain. Furthermore, the normal of the evolved surface is not constant and changes considerably during evolution. The normal vectors are mostly determined by the constraints imposed from surfaces within the neighboring cells.

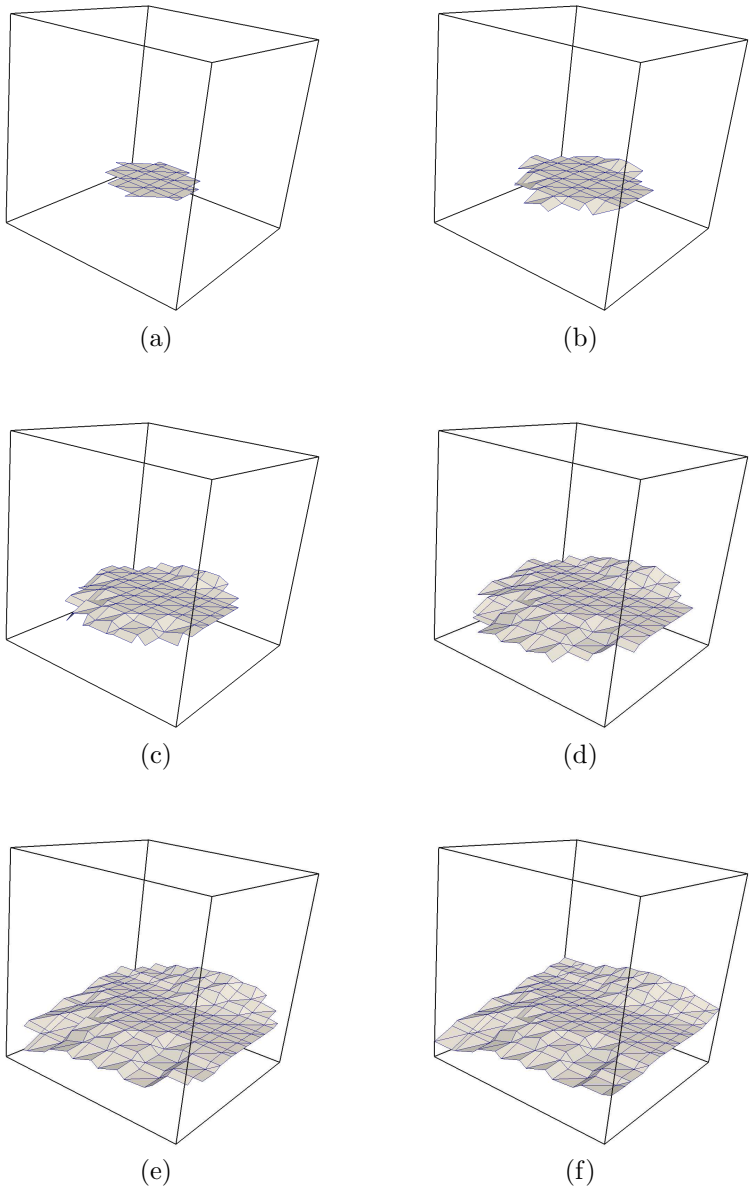


Figure 5.10 Different stages of an evolution of a surface inside a unit cube.

5.5 The solver wrapper classes

To develop the finite element solvers using the automatically generated code by the PUM form compiler, wrapper classes can be generated. The wrapper classes facilitate the communication between the UFC-based generated code and the solver library. Inside these wrapper classes, some objects of classes from both the solver library and the DOLFIN library (e.g. `pum::GenericPUM`, `dolfin::FiniteElement`, `pum::DofMap`, `dolfin::DofMap`, `pum::FunctionSpace` and `dolfin::FunctionSpace`) are initialized. These intialisations are performed using automatically information from the generated UFC-based classes. To generate these wrappers, a suitable flag is added to the command line when compiling the input file with the PUM compiler. For example to generate wrapper classes for the DOLFIN-based solver, `-l dolfin` is added to the command line.

```
ffcpum -l dolfin foo.ufl
```

Components of the solver wrappers are divided into two main groups. A first group contains classes which are used to declare the function spaces of coefficients and basis functions. These wrapper classes are derived from the `pum::FunctionSpace` or `dolfin::FunctionSpace` classes. Inside a wrapper class, a function space object is initialised using a mesh, a finite element defined by an object of the `ufc::finite_element` class and a local to global mapping of the degrees of freedom defined by an object of the `ufc::dof_map` class. For the enriched function spaces, the `pum::GenericPUM` objects are also initialized internally to compute data related to the enriched degrees of freedom. Because of the internal initialisation, the `pum::GenericPUM` objects are not exposed to the users and they are automatically generated inside the wrapper classes (unlike earlier implementations of wrapper classes in Nikbakht and Wells (2009)).

A second group of the solver wrapper classes are those representing forms defined inside the variational formulations. These wrapper classes are subclasses of the `dolfin::Form` class that is a base class for the UFC-based generated code for the DOLFIN-based solver. The form wrapper classes are generated for the forms with different ranks. `BilinearForm`, `LinearForm` and `Functional` classes are generated for the forms with rank two, one and zero, respectively. From these forms, the objects of the `ufc::finite_element`, `ufc::dof_map`, `ufc::cell_integrals`, `ufc::exterior_facet_integrals` and `ufc::interior_facet_integrals` classes are accessible using specific member functions. These forms are passed to the assembler inside a solver library to compute global system matrices, the global system vectors or functional values.

Chapter 6 Applications in modelling different physical problems

A PUM form compiler and a PUM library on top of FEniCS components have been introduced in Chapters 4 and 5. These two components allow one to develop DOLFIN-based solvers to model physical problems in domains with stationary and propagating discontinuity surfaces in an automated framework. This automatic framework relies on automated code generation to develop finite element models based on the partition of unity methods.

In order to use the automated approach to model problems with discontinuous solutions, the PDE-specific input data is passed as weak forms of variational formulations to the PUM form compiler at the first stage. The generated output from the compiler is then included in a C++ solver which provides required functionalities to define problem-specific data. The problem-specific input data includes a mesh, boundary conditions, coefficient functions and discontinuity surfaces. The criteria for the evolution of surfaces can also be defined in the C++ solver interface. The C++ solver uses components of DOLFIN and the PUM library to solve variational formulations defined in the partition of unity framework.

Examples are presented in this chapter to demonstrate the use of the automated code generation for modeling physical problems with discontinuity surfaces. Through these examples, generality, simplicity, efficiency and reliability of the proposed framework have been illustrated. The proposed framework is not only limited to the traditional conforming Galerkin formulations, but it can also be used for the discontinuous Galerkin formulations. To show the generality of the proposed framework in using different families of elements, the examples are chosen such that they cover a wide range of basis functions including continuous Lagrange, discontinuous Lagrange, $H(\text{div})$ and/or $H(\text{curl})$ families of finite elements. More examples can also be found in Nikbakht and Wells (2009, 2012a).

6.1 H^1-conforming primal approach to the weighted Poisson equation

As a canonical example, the weighted Poisson equation is presented in which the solution u defined on $\Omega \subset \mathbb{R}^d$ is discontinuous across a surface Γ_d. The flux across the surface is equal to $k\,(u^+ - u^-)$, where k is a parameter and u^+ and u^- are the values

of u on the positive and negative sides of the discontinuity surface, respectively. The relevant function space reads

$$V = \left\{ v_h \in H^1\left(\Omega \backslash \Gamma_d\right),\, v_h|_E \in P_k\left(E \backslash \Gamma_d\right)\ \forall E \right\}, \tag{6.1}$$

and the bilinear and linear forms read

$$a\left(u,v\right) = \int_{\Omega\backslash\Gamma_d} w\nabla u \cdot \nabla v\, dx + \int_{\Gamma_d} k\left[\!\left[u\right]\!\right]\left[\!\left[v\right]\!\right] ds, \tag{6.2}$$

$$L\left(v\right) = \int_{\Omega} fv\, dx, \tag{6.3}$$

where f and w are a source term and a weight coefficient.

A unit cube domain $\Omega = (1,0,0)\times(0,1,0)\times(0,0,1)$ with two disjoint discontinuity surfaces is considered as the computational domain. The first discontinuity surface is defined using $\psi = z - 0.14 = 0$ and $\phi = (x-0.5)^2 + (y-0.5)^2 - 0.09 \leq 0$ as level set functions. For the second discontinuity surface, $\psi = x^2 + z^2 - 0.6 = 0$ and $\phi = x^2 + 0.1y^2 + 0.2z^2 - 0.5 \leq 0$ are used. For this example, $f = \sin(x)\sin(y)\sin(z)$, $w = 1 + e^{x^2}$ and $k = 1$. A zero Dirichlet boundary condition ($u = 0$) is applied at $z = 0$. The remaining boundaries are flux-free.

The complete UFL input representing the variational formulation using the partition of unity framework is presented in Figure 6.1. The relevant enriched finite elements are created by adding the linear Lagrange elements on tetrahedral cells to the restricted linear Lagrange elements by the discontinuity surfaces. While the source term and the weight coefficient functions are defined using linear Lagrange elements, the test and trial functions are constructed on the enriched finite elements. At the end of the UFL input, the linear and bilinear forms are defined using coefficient and basis functions. Note the notation `*dc` has been used to represent the surface integral in the bilinear form.

The automatically generated code is then included inside a C++ solver. The C++ solver is designed following the DOLFIN style of mirroring mathematical abstractions and keeping the code compact. The code for the `Poisson::FunctionSpace`, `Poisson::BilinearForm` and `Poisson::LinearForm` objects is PDE-specific and has been generated by the PUM form compiler, whereas the other elements in the C++ solver are standard DOLFIN objects, unless prefaced with the `pum` namespace.

At the first step, subclasses of `dolfin::Expression` are implemented to represent the source term and the weight function:

```
// Define source term
class Source : public Expression
{ void eval(Array<double>& values, const Array<double>& x) const
```

```
# Enriched function space
elem_cont = FiniteElement("Lagrange", tetrahedron, 1)
elem_discont = RestrictedElement(elem_cont, dc)
element = elem_cont + elem_discont

# Test and Trial functions
u, v = TrialFunction(element), TestFunction(element)

# Source term, weight coefficient and surface interface
f, w = Coefficient(elem_cont), Coefficient(elem_cont)
k  = Constant(tetrahedron)

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

Figure 6.1 The UFL input to model the discontinuity surfaces in a three-dimensional weighted Poisson problem.

```
  { values[0] = sin(x[0])*sin(x[1])*sin(x[2]); }
};

// Define weight coefficient
class Weight : public Expression
{ void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = 1.0 + std::exp((x[0]*x[0])); }
};
```

Then the Dirichlet boundary is defined by providing a subclass of the `dolfin::SubDomain` class from the DOLFIN library:

```
// Sub domain for Dirichlet boundary condition
class Bottom : public SubDomain
{ bool inside(const Array<double>& x, bool on_boundary) const
  { return x[2] < DOLFIN_EPS && on_boundary; }
};
```

In the next step, the level set functions of the discontinuity surfaces are represented as subclasses of the `dolfin::Expression` class. As mentioned before, $\psi = z - 0.14 = 0$ and $\phi = (x-0.5)^2+(y-0.5)^2-0.09 \leq 0$ have been assumed as the level set functions for the first discontinuity surface. These level set functions are implemented as

```
// Define the surface and boundary of discontinuity #0
class Surface0 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = x[2] - 0.14; }
};

class Boundary0 : public Expression
```

```
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = pow(x[0] - 0.5,2) + pow(x[1] - 0.5,2) - 0.09; }
};
```

For the second discontinuity, the level set functions ($\psi = x^2 + z^2 - 0.6 = 0$ and $\phi = x^2 + 0.1y^2 + 0.2z^2 - 0.5 \leq 0$) are implemented as:

```
// Define the surface and boundary of discontinuity #0
class Surface1 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = pow(x[0], 2) + pow(x[2], 2) - 0.6; }
};

class Boundary1 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = x[0]*x[0] + x[1]*x[1]/10 + x[2]*x[2]/5 - 0.5; }
};
```

Inside the main solver, after defining the mesh, the objects of these classes are used to initialise instances of the `pum::NonBranchingSurface` class. The pointers to the `pum::NonBranchingSurface` objects are then collected inside `surfaces`, a container of `pum::GenericSurface` objects.

```
// Define discontinuity surfaces and add them to a vector
Surface0 s0; Boundary0 b0;
Surface1 s1; Boundary1 b1;
pum::NonBranchingSurface d0(mesh, s0, b0);
pum::NonBranchingSurface d1(mesh, s1, b1);
std::vector<const pum::GenericSurface*>
            surfaces = boost::assign::list_of(&d0)(&d1);
```

Note that the function space in the code extract is initialized with `surfaces`. This is a convenience wrapper for the UFC function space, with the `pum::PUM` (derived from `GenericPUM`) objects being created internally from the surfaces and then used to initialize the UFC objects. The function space is then used to initialize Dirichlet boundary conditions.

```
// Create function space and boundary condition
Poisson::FunctionSpace V(mesh, surfaces);
Constant u0(0.0); Bottom bot;
DirichletBC bc(V, u0, bot);
```

The linear and bilinear forms are created using the function space and their corresponding coefficients are attached.

```
// Create bilinear and linear forms
```

```
Poisson::BilinearForm a(V, V);
Poisson::LinearForm L(V);
Weight w; Source f; Constant k(1.0);
a.k = k; a.w = w; L.f = f;
```

To solve the variational problem, an object of `pum::Function` is defined. `pum::Function` is a subclass of `dolfin::Function` and implements primarily restrictions of discontinuous coefficient functions for use in forms and the interpolation of functions to cell vertices for use in the post-processing.

```
// Solve pde
pum::Function u(V);
solve(a == L, u, bc);
```

To visualise the discontinuity surfaces, a `std::pair` containing the pointers to the `pum::GenericSurface` objects and the mesh object is created. This pair is then passed to an object of `pum::VTKFile` to visualise surfaces.

```
// Save solution and surfaces to files for visualisation
File file("poisson.pvd");
pum::VTKFile file_surface("surface.pvd");
std::pair<std::vector<const pum::GenericSurface*>,
          const Mesh*> out_surfaces(surfaces, &mesh);

file  << u;
file_surface << out_surfaces;
```

The complete C++ solver for this problem is illustrated in Figures 6.2 and 6.3. Figure 6.2 shows the implementation of the subclasses representing coefficients, the Dirichlet boundary and level set functions. The main solver, which includes defining a mesh and using the automatically generated linear and bilinear forms to solve the variational problem, is shown in Figure 6.3.

A mesh on the unit cube and computed solution contours for this problem, with the superimposed discontinuity surfaces, are shown in Figure 6.4. The impact of the discontinuities on the computed solution contours can be clearly seen in the solution contours.

6.2 L^2-conforming discontinuous Galerkin approach to the elasticity equation

As shown in the example presented in Section 3.2.2, the FEniCS components can also be used to model problems with discontinuous Galerkin formulations. By extending FEniCS in the context of the partition of unity framework, it is now also possible to include discontinuities across arbitrary surfaces in the discontinuous Galerkin methods in an automated way.

```
#include <dolfin.h>
#include <PartitionOfUnity.h>
#include "Poisson.h"
using namespace dolfin;

// Define source term
class Source : public Expression
{ void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = sin(x[0])*sin(x[1])*sin(x[2]); }
};

// Define weight coefficient
class Weight : public Expression
{ void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = 1.0 + std::exp((x[0]*x[0])); }
};

// Sub domain for Dirichlet boundary condition at the bottom
class Bottom : public SubDomain
{ bool inside(const Array<double>& x, bool on_boundary) const
  { return x[2] < DOLFIN_EPS && on_boundary; }
};

// Define the surface and boundary of discontinuity #0
class Surface0 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = x[2] - 0.14; }
};

class Boundary0 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2) - 0.09; }
};

// Define the surface and boundary of discontinuity #1
class Surface1 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = pow(x[0], 2) + pow(x[2], 2) - 0.6; }
};

class Boundary1 : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = x[0]*x[0] + x[1]*x[1]/10 + x[2]*x[2]/5 - 0.5; }
};
```

Figure 6.2 The C++ code for the solver of the weighted Poisson problem with discontinuities in the solution (the class definitions).

```
int main()
{
  // Create mesh
  dolfin::UnitCube mesh(50, 50, 50);

  // Define discontinuity surfaces and add them to a vector
  Surface0 s0; Boundary0 b0;
  Surface1 s1; Boundary1 b1;
  pum::NonBranchingSurface d0(mesh, s0, b0);
  pum::NonBranchingSurface d1(mesh, s1, b1);
  std::vector<const pum::GenericSurface*>
              surfaces = boost::assign::list_of(&d0)(&d1);

  // Create function space and boundary condition
  Poisson::FunctionSpace V(mesh, surfaces);
  Constant u0(0.0); Bottom bot;
  DirichletBC bc(V, u0, bot);

  // Create bilinear and linear Forms
  Poisson::BilinearForm a(V, V); Poisson::LinearForm L(V);
  Weight w; Source f; Constant k(1.0);
  a.k = k; a.w = w; L.f = f;

  // solve pde
  pum::Function u(V);
  solve(a == L, u, bc);

  // Save solution and surfaces to files for visualisation
  File file("poisson.pvd");
  pum::VTKFile file_surface("surface.pvd");
  std::pair<std::vector<const pum::GenericSurface*>,
            const Mesh*> out_surfaces(surfaces, &mesh);

  file << u;
  file_surface << out_surfaces;
}
```

Figure 6.3 The C++ code for the solver of the weighted Poisson problem with discontinuities in the solution (the main solver). The notation resembles closely DOLFIN code for conventional problems.

The modeling of an elastic domain Ω with discontinuity surfaces Γ_d using a discontinuous Galerkin approach with an interior penalty formulation is assumed. If the discontinuity surfaces are in opening states and they are assumed traction-free, the same bilinear and linear forms as those for the discontinuous Galerkin formulation of the elasticity equation, presented in Equations (3.9) and (3.10), can be used. To include discontinuities, it is just enough to redefine the function space corresponding to the weak forms. The relevant function space V is redefined as

$$V = \left\{ \boldsymbol{v}_h \in \left(L^2\left(\Omega\right)\right)^n, \boldsymbol{v}_h|_E \in \left(P_k\left(E \backslash \Gamma_d\right)\right)^n \ \forall E \right\}. \tag{6.4}$$

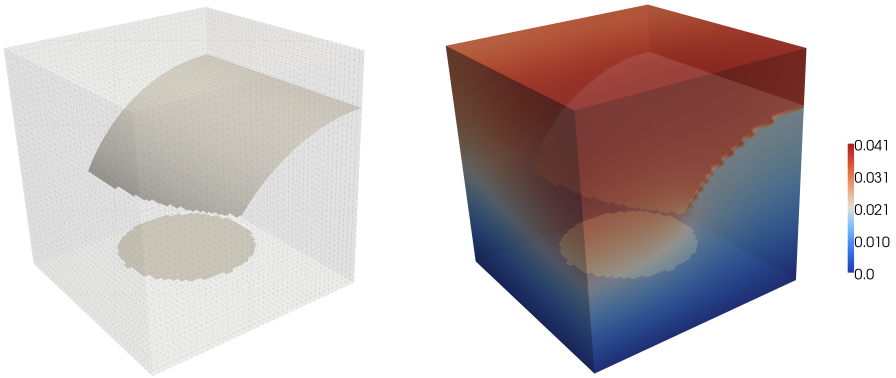


Figure 6.4 The Poisson problem in three dimensions with discontinuity surfaces: (a) the surface mesh and (b) the solution contour.

To implement the relevant function space inside the form compiler input, it is enough to redefine the corresponding function space in Figure 3.7 as

```
elem_cont = VectorElement("Discontinuous Lagrange", triangle, 2)
elem_discont = RestrictedElement(elem_cont, dc)
element = elem_cont + elem_discont
```

For the rest, the remaining parts including the definition of the bilinear and linear forms remain unchanged.

After compiling the UFL input, the generated code is used to model the deformation of a cantilever beam with discontinuity surfaces as shown in Figure 6.5. This beam has the same dimensions, material properties and boundary conditions as the beam presented in Section 3.2.2. Two vertical traction-free discontinuity surfaces are embedded inside this beam. While the first surface Γ_{d1} extends from $(0.5, 0.1)$ to $(0.5, 0.5)$, the second surface Γ_{d2} has $(0.8, 0.2)$ and $(0.8, 0.5)$ as end points.

The complete C++ solver interface is presented in Figures 6.6 and 6.7. Note the definition of the discontinuity surfaces using the mesh and the end points.

The discontinuity surfaces embedded in the computational mesh and the computed displacement contour on the magnified deformed mesh are presented in Figure 6.8. The influence of discontinuity surfaces on the global stiffness of the beam can be easily observed from the computed contours.

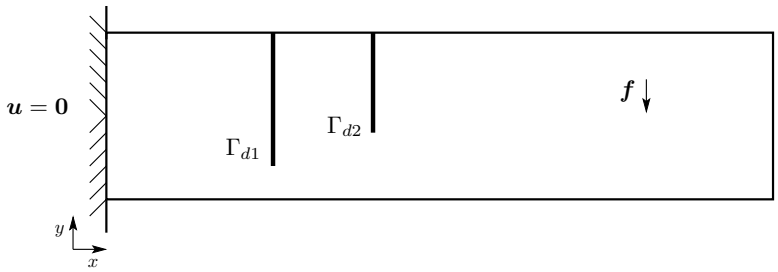


Figure 6.5 A beam with two traction-free discontinuity surface that is restricted at the left edge and subjected to a vertical body force.

```
#include <dolfin.h>
#include <ParitionofUnity.h>
#include "DG_Elasticity.h"

using namespace dolfin;

// Sub domain for clamp at the left end
class boundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  { return std::abs(x[0]) < DOLFIN_EPS && on_boundary;}
};

// Body force term
class BodyForce : public Expression
{
public:
  BodyForce() : Expression(2) {}

  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] =  0.0; values[1] = -10.0;}
};
```

Figure 6.6 The complete C++ solver of the elasticity equation using the discontinuous Galerkin formulation in combination with the partition of unity formulation (the class definitions).

6.3 Continuous/discontinuous interior penalty formulation for the biharmonic equation

The biharmonic equation is a fourth-order elliptic equation. The strong form reads

$$\nabla^4 u = f \qquad \text{in } \Omega, \tag{6.5}$$

```
int main()
{
  // Create mesh
  Rectangle mesh(0, 0, 2.0, 0.5, 44, 11);

  // Define Discontinuity Surface #0
  const Point p0_0(0.5, 0.1), p0_1(0.5, 0.5);
  std::pair<Point, Point> end_points0(p0_0, p0_1);
  pum::NonBranchingSurface d0(mesh, end_points0);

  // Define Discontinuity Surface #1
  const Point p1_0(0.8, 0.2), p1_1(0.8, 0.5);
  std::pair<Point, Point> end_points1(p1_0, p1_1);
  pum::NonBranchingSurface d1(mesh, end_points1);

  // Create vector of discontinuity surfaces
  std::vector<const pum::GenericSurface*>
              discontinuities = boost::assign::list_of(&d0)(&d1);

  // Define function space
  DG_Elasticity::FunctionSpace V(mesh, discontinuities);

  // Define boundary condition
  Constant u0(0.0, 0.0);
  DirichletBoundary boundary;
  DirichletBC bc(V, u0, boundary);

  // Define variational problem
  DG_Elasticity::BilinearForm a(V, V);
  DG_Elasticity::LinearForm L(V);
  BodyForce f; L.f = f;

  // Compute solution
  pum::Function u(V);
  solve(a == L, bc, u);

  // Save solution in VTK format
  File file("dg_elasticity.pvd");
  file << u;
}
```

Figure 6.7 The complete C++ solver of the elasticity equation using the discontinuous Galerkin formulation in combination with the partition of unity formulation (the main solver).

where f is a source term in domain Ω and ∇^4 is the biharmonic operator defined as $\nabla^2\nabla^2$. The boundary conditions for this problem read

$$u = 0 \qquad \text{on } \partial\Omega, \tag{6.6}$$

$$\nabla^2 u = 0 \qquad \text{on } \partial\Omega, \tag{6.7}$$

$$\nabla^2 u = 0 \qquad \text{on } \Gamma_d, \tag{6.8}$$

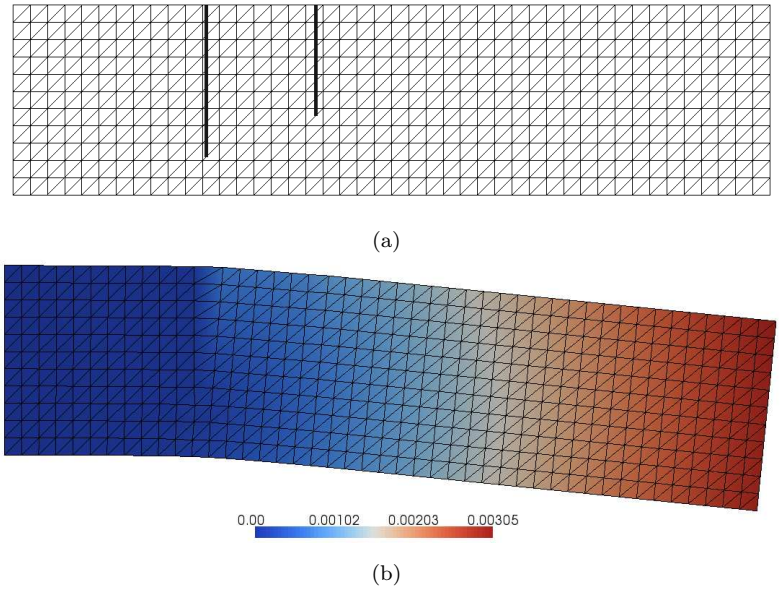


Figure 6.8 (a) The mesh with discontinuity surfaces (b) the displacement contour (m) on the magnified deformed mesh (50 times) for the beam restricted at the left edge and subjected to a vertical body force.

where Γ_d denotes discontinuity surfaces and $\partial\Omega$ represents the external boundaries of the domain Ω.

To obtain weak forms, the multiplication of the strong form by a test function, integration over domain and application of twice integration by parts leads to a problem with second-order derivatives. This problem would conventionally require H^2-conforming basis functions. However, such functions are difficult to construct in the finite element context. If a discontinuous Galerkin formulation is selected, the use of the expensive H^2-conforming basis functions can be avoided. The discontinuous Galerkin formulation allows the use of H^1-conforming elements by imposing a weak continuity of normal derivatives between finite element cells.

A so-called continuous/discontinuous Galerkin method for the biharmonic equation was developed by Engel et al. (2002). The function space used for this problem is the same as the function space presented in Equation (6.1). The bilinear

and linear forms read

$$a(u,v) = \int_{\Omega\setminus\Gamma_d} \nabla^2 u \nabla^2 v \, dx - \int_{\Gamma_0} \langle \nabla^2 u \rangle \cdot [\![\nabla v]\!] \, ds - \int_{\Gamma_0} [\![\nabla u]\!] \cdot \langle \nabla^2 v \rangle \, ds + \alpha \int_{\Gamma_0} \frac{1}{h} [\![u]\!] \cdot [\![v]\!] \, ds, \quad (6.9)$$

and

$$L(v) = \int_{\Omega} f v \, dx, \quad (6.10)$$

where α is a penalty parameter, h is an average size of finite element cells and f is a source term. The jump and average operators over the interior facets Γ_0 are defined as $[\![\nabla u]\!] = \nabla u^+ \cdot \boldsymbol{n}^+ + \nabla u^- \cdot \boldsymbol{n}^-$ and $\langle \nabla^2 u \rangle = (\nabla^2 u^+ + \nabla^2 u^-)/2$ where $\boldsymbol{n}$ is an outward normal vector on Γ_0. In the weak formulation, the boundary condition $\nabla^2 u = 0$ is weakly enforced on the discontinuity surfaces Γ_d and $\partial\Omega$. The interior facet integrals provide the weak continuity of the normal derivative across interior facets of the computational mesh, denoted by Γ_0.

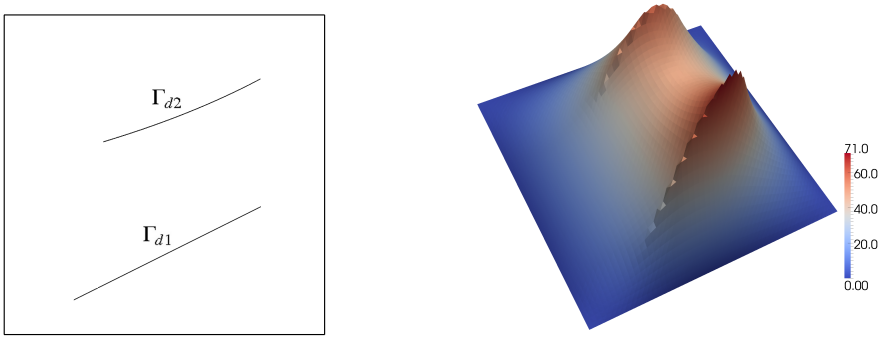
The given variational form is used to model a unit square domain $\Omega = (0,1)\times(1,0)$ subjected to a source term $f = 4\pi^4 \sin(\pi x)\sin(\pi y)$ with homogeneous Dirichlet boundary conditions, defined in Equation (6.6). Two discrete discontinuity surfaces are considered in the unit square domain. The first surface Γ_{d1} is a straight line which is defined by $(0.2, 0.1)$ and $(0.8, 0.4)$ as end points. The second surface Γ_{d2} is a quadratic line represented by $(0.3, 0.6)$ and $(0.8, 0.8)$ as end points and $\phi = y + \frac{1.2}{x-2.3} = 0$ as a level set function. The discontinuity surfaces within the unit square are presented in Figure 6.10(a).

The UFL input for the PUM compiler required to generate low-level code for the biharmonic problem is given in Figure 6.9. This input can be used to model discontinuity surfaces in the biharmonic problem using the partition of unity framework. If this input is compared to the UFL input for modeling a biharmonic problem in a domain without any discontinuity surface that is presented in Ølgaard et al. (2008), a few differences are noticed. The important difference refers to the definition of the finite element function space. In the UFL input presented in Figure 6.9, the relevant finite element space is achieved by enriching scalar quadratic Lagrange finite elements with discontinuous finite elements. In both UFL inputs, the facet normal $\boldsymbol{n}$ and the cell size h are internally computed using UFL built-in functions.

The generated output is used to model the unit square domain with the discontinuities in a C++/DOLFIN solver. The solver provides required data including discontinuity surfaces, the computational domain and the source term

```
# Define continuous and discontinuous spaces
elem_cont    = FiniteElement("Lagrange", "triangle", 2)
elem_discont = RestrictedElement(elem_cont, dc)
element      = elem_cont + elem_discont

# Create test and trial functions and source term
v, u = TestFunction(element), TrialFunction(element)
f = Coefficient(elem_cont)

# Facet normal component and cell size
n, h = element.cell().n, element.cell().circumradius

# Parameter
alpha = Constant(triangle)

# Bilinear and linear forms
a = inner(div(grad(u)), div(grad(v)))*dx \
  - inner(avg(div(grad(u))), jump(grad(v),n))*dS \
  - inner(jump(grad(u),n), avg(div(grad(v))))*dS \
  + alpha('+')/avg(h)*inner(jump(grad(u),n), jump(grad(v),n))*dS
L = f*v*dx
```

Figure 6.9 The UFL input for the partition of unity formulation of the biharmonic equation.



(a) (b)

Figure 6.10 The biharmonic problem in two dimensions: (a) a unit square and Γ_{d1} and Γ_{d2} discontinuity surfaces, and (b) the solution u contour on the wrapped mesh.

to solve the variational problem. The result contour is depicted in Figure 6.10(b). The results shows a discontinuity in the first derivative of the solution due to weakly applied boundary conditions for the Laplacian of u across the discontinuity surfaces. This example shows the possibility of using the automated framework to apply boundary conditions on the surfaces which do not conform to the underlying mesh.

6.4 Mixed formulation for the Poisson equation

The PUM form compiler also supports code generation for the variational formulations defined using non-Lagrange elements in combination with Lagrange finite element bases. To demonstrate this ability, modelling discontinuities using the partition of unity framework in a mixed Poisson problem is presented. The mixed Poisson problem is a coupled problem between a scalar field u and a vector flux field $\boldsymbol{\sigma}$ (Rognes et al., 2009).

The governing partial differential equations read:

$$\boldsymbol{\sigma} - \nabla u = \mathbf{0} \qquad \text{in } \Omega, \tag{6.11}$$

$$\nabla \cdot \boldsymbol{\sigma} = -f \qquad \text{in } \Omega, \tag{6.12}$$

with boundary conditions

$$u = 0 \qquad \text{on } \Gamma_u, \tag{6.13}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = 0 \qquad \text{on } \Gamma_d, \tag{6.14}$$

where f is a source term in the domain Ω and $\boldsymbol{n}$ is an outward vector normal to discontinuity surfaces Γ_d. The Dirichlet boundary of domain Ω is denoted by Γ_u.

After multiplying the strong forms presented in Equations (6.12) and (6.12) with test functions $\boldsymbol{\tau}$ and ω, integrating over the domain Ω and then integrating by parts, one can obtain the following variational formulation: find $\boldsymbol{\sigma} \in V$ and $u \in W$ such that

$$a(\boldsymbol{\sigma}, u; \boldsymbol{\tau}, \omega) = L(\boldsymbol{\tau}; \omega) \qquad \forall \boldsymbol{\tau} \in V, \quad \omega \in W \tag{6.15}$$

where the bilinear form a and the linear form L read

$$a(\boldsymbol{\sigma}, u; \boldsymbol{\tau}, \omega) = \int_{\Omega \setminus \Gamma_d} \boldsymbol{\sigma} \cdot \boldsymbol{\tau} - u(\nabla \cdot \boldsymbol{\tau}) + (\nabla \cdot \boldsymbol{\sigma})\omega \, dx + \int_{\Gamma_d} u_{\pm} \boldsymbol{\tau}_{\pm} \cdot \boldsymbol{n}_{\pm} \, ds, \tag{6.16}$$

$$L(\boldsymbol{\tau}; \omega) = \int_{\Omega} f\omega \, dx. \tag{6.17}$$

The last term in Equation 6.16, which arises from integration by parts, requires the evaluation of restricted functions on the positive and negative sides of the surface. Furthermore, this term can also be used to weakly apply Dirichlet boundary conditions on the discontinuity surface.

To obtain a stable discretized form of weak formulations, choosing suitable finite element function spaces is important. One family of stable finite elements is

$H(\mathrm{div})$-conforming BDM elements (Brezzi et al., 1985) for the flux and discontinuous Lagrange elements for the scalar field. Moreover, different combinations of continuous/discontinuous finite elements also exist. If both scalar and vector fields are assumed discontinuous across the surfaces, the function spaces for the flux and the scalar field respectively read

$$V = \left\{ \boldsymbol{\tau}_h \in H\left(\mathrm{div}, \Omega \backslash \Gamma_d\right), \boldsymbol{\tau}_h|_E \in \left(P_k\left(E \backslash \Gamma_d\right)\right)^d \ \forall E \right\}, \tag{6.18}$$

$$W = \left\{ \omega_h \in L^2\left(\Omega\right), \omega_h|_E \in P_{k-1}\left(E \backslash \Gamma_d\right) \ \forall E \right\}, \tag{6.19}$$

where $k > 1$.

This formulation is used to model the mixed Poisson equation on a unit square $\Omega = (1,0) \times (0,1)$ containing a discontinuity surface. The discontinuity surface is a curved line passing through the end points $(0.5, 0.2)$ and $(0.95, 0.5)$ with a level set function defined as $\phi = y + \frac{4}{3}x^2 - 2.6x + 0.767 = 0$. The source term is also given as $f = 500.0e^{-((x-0.5)^2+(y-0.5)^2)/0.02}$.

The form compiler input for the mixed Poisson formulation is presented in Figure 6.11. First, a mixed finite element space is defined. This mixed finite element function space assumes discontinuities in both the scalar field and the flux. Note the difference between UFL operators used for mixed elements and enriched elements (a multiplication versus a summation). The mixed finite element space is created using a UFL ‘‘`*`’’ operator which combines two enriched elements, that are created by UFL ‘‘`+`’’ operators. Next, test functions and trial functions are defined on the mixed finite element space. Further, two coefficients are defined to represent the source term and the normal into the discontinuity surface. At the end, the linear and bilinear forms are created that contain integrations over cells and discontinuity surfaces.

The UFL input is compiled using the PUM compiler to generate low-level code. The generated code will be included in a DOLFIN-based solver which defines the mesh, discontinuity surfaces and coefficients. The extract of the DOLFIN-based solver is presented in Figure 6.12. The discontinuity surface and a subclass of `dolfin::Expression` to define the level set function ϕ are defined inside the solver. Furthermore, the sub-functions for the flux and the scalar field u from the solution of the variational problem are extracted and then used for post-processing purposes.

The solution contours for the scalar field u and the flux σ are presented in Figure 6.13. As can be seen, both u and $\boldsymbol{\sigma}$ demonstrate jumps over the discontinuity surface.

```
# Define continuous spaces
BDM_c = FiniteElement("Brezzi-Douglas-Marini", "triangle", 2)
DG_c  = FiniteElement("Discontinuous Lagrange", "triangle", 1)

# Define discontinuous spaces
BDM_d = RestrictedElement(BDM_c, dc)
DG_d  = RestrictedElement(DG_c, dc)

# Create enriched spaces
BDM, DG = BDM_c + BDM_d, DG_c + DG_d

# Create mixed element
mixed_element = BDM * DG

# Trial and test functions
sigma, u = TrialFunctions(mixed_element)
tau, w   = TestFunctions(mixed_element)

# Source term
f = Coefficient(DG_c)

# Discontinuity surface normal
element = VectorElement("Discontinuous Lagrange", triangle, 0)
n  = Coefficient(element)

# Bilinear form and linear forms
a = dot(sigma, tau)*dx - u*div(tau)*dx + div(sigma)*w*dx
  + (u*inner(tau, n))('+')*dc + (u*inner(tau, n))('-')*dc
L = f*w*dx
```

Figure 6.11 The UFL input for the mixed Poisson in a domain with discontinuities.

6.5 $H(\mathrm{curl})$-conforming elements for an electromagnetic problem

To show the applicability of the automated approach for modelling discontinuity surfaces using $H(\mathrm{curl})$ elements, modelling magnetic fields is performed in a domain Ω with a discontinuity surface Γ_d in which the solution exhibits jumps.

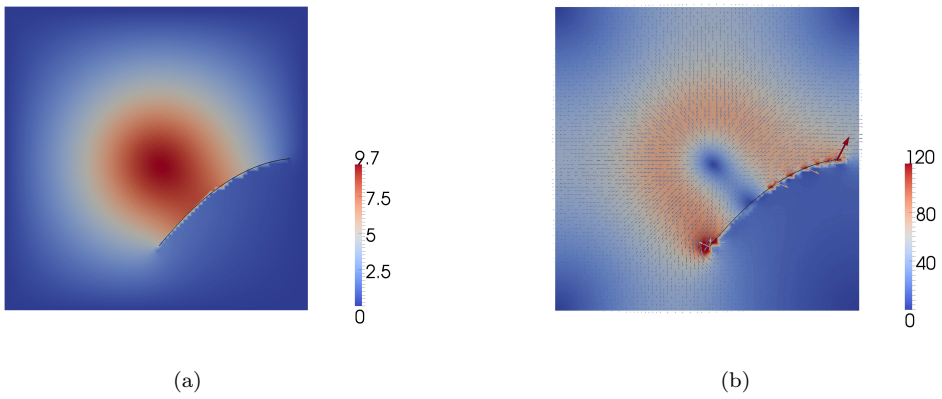
It is well known that using the Lagrange family of elements to represent electric and magnetic fields causes several serious problems (Andersen and Solodukhov, 1978; Rahman and Davies, 1984). The first problem is the occurrence of non-physical or so-called spurious solutions. This is mainly because of the lack of enforcement of the divergence condition. The second problem arises while imposing boundary conditions at material interfaces as well as conducting surfaces. The third problem is the difficulty of modelling conducting and dielectric edges and corners due to the field singularities associated with these structures. Using $H(\mathrm{curl})$-conforming elements overcomes these problems and provides a better approximation of field

```
// Level set function
class Shape : public dolfin::Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  { values[0] = x[1] + (4.0/3.0)*x[0]*x[0] - 2.6*x[0] + 0.767;}
};

// Define a curved line with end points and a function
const Point p0_0(0.5, 0.2); const Point p0_1(0.95, 0.5);
std::pair<Point, Point> end_points0(p0_0, p0_1);
const Shape shape;
Surface d0(mesh, end_points0, shape);

// Create discontinuity surface vector
std::vector<const GenericSurface*> surfaces;
surfaces.push_back(d0);

// Define function Spaces
MixedPoisson::FunctionSpace V(mesh, surfaces);

// Normal of surface
DiscontinuityNormal n(surfaces, mesh);

// Define bilinear and linear forms
MixedPoisson::BilinearForm a(V, V);
MixedPoisson::LinearForm L(V);
a.n = n; L.f = f;

// Compute solution
pum::Function w(V);
solve(a == L, w);

// Extract components
pum::Function& sigma = w[0];
pum::Function& u = w[1];

// Save components for visualization
dolfin::File f3("sigma.pvd");
dolfin::File f4("u.pvd");
f3 << sigma;
f4 << u;
```

Figure 6.12 The extract of C++ solver code for the mixed Poisson problem for a unit square domain subjected to homogeneous boundary conditions.

quantities in electromagnetic problems (Bossavit, 1989).

In electrodynamic problems, the behavior of electric and magnetic fields are described by Maxwell's equations (Jin, 2002; Smith, 1997). These partial differential equations are used to solve different types of boundary value problems. For time-harmonic fields, the vector Helmholtz wave equation derived form Maxwell's equations is used in terms of either an electric field $\boldsymbol{E}$ or a magnetic field $\boldsymbol{H}$.



(a) (b)

Figure 6.13 The mixed Poisson problem in two dimensions: (a) the contour of u and (b) the contour of flux σ.

In this example, the vector Helmholtz wave equation in terms of the magnetic field $\boldsymbol{H}$ is chosen. The governing partial differential equation reads

$$\nabla \times \frac{1}{\mu} \nabla \times \boldsymbol{H} - {k_0}^2 \epsilon \boldsymbol{H} - \nabla \times \frac{1}{\epsilon} \boldsymbol{J} = \boldsymbol{0} \qquad \text{in } \Omega, \tag{6.20}$$

where μ, ϵ and k_0 are electric permitivity, magnetic permeability and wave number in the free space, respectively. The magnetic field is induced by a given electric current density, $\boldsymbol{J}$. The discontinuity surfaces Γ_d and the boundaries of the domain $\partial\Omega$ are assumed to be magnetically conducting surfaces, i.e.

$$\boldsymbol{n} \times \boldsymbol{H} = \boldsymbol{0} \qquad \text{on } \partial\Omega, \tag{6.21}$$

$$\boldsymbol{n} \times \boldsymbol{H} = \boldsymbol{0} \qquad \text{on } \Gamma_d, \tag{6.22}$$

where $\boldsymbol{n}$ is an outward vector perpendicular to Γ_d and $\partial\Omega$. For the given strong form, one can obtain the corresponding weak form on suitable function spaces. The linear and bilinear forms read

$$a(\boldsymbol{H}, \boldsymbol{T}) = \int_{\Omega \setminus \Gamma_d} \frac{1}{\mu} (\nabla \times \boldsymbol{H}) \cdot (\nabla \times \boldsymbol{T}) \, d\Omega - \int_{\Omega \setminus \Gamma_d} \boldsymbol{H} \cdot \boldsymbol{T} \, d\Omega \tag{6.23}$$

$$L(\boldsymbol{T}) = \int_{\Omega} \boldsymbol{T} \cdot (\nabla \times \frac{1}{\epsilon} \boldsymbol{J}) \, d\Omega \tag{6.24}$$

Note that because of the boundary condition presented in Equation (6.22) on the discontinuity surface, no surface integral appears in the bilinear form. If a

H(curl)-conforming element is selected, the relevant function space reads

$$V = \left\{ \boldsymbol{\tau}_h \in H\left(\text{curl}, \Omega \backslash \Gamma_d\right), \boldsymbol{\tau}_h|_E \in \left(P_k\left(E \backslash \Gamma_d\right)\right)^d \ \forall E \right\}. \tag{6.25}$$

This variational formulation is used to model the electromagnetic problem on a unit square domain with a linear discontinuity surface defined by $(0.3, 0.4)$ and $(0.7, 0.4)$ as end points. This domain is subjected to an electric field defined as $\nabla \times \boldsymbol{J} = (1.0, 0.0)\,\text{Vm}^{-1}$. In this case, $k_0 = 1.0$, $\mu = 1.0\,\text{Fm}^{-1}$ and $\epsilon = 1.0\,\text{Hm}^{-1}$.

The FEniCS Form Compiler supports the code generation for problems with H(curl)-conforming formulations (Rognes et al., 2009). Similarly, the PUM form compiler can also be used to generate low-level code for modeling discontinuity surfaces with the H(curl)-conforming formulations. Figure 6.14 shows the PUM form compiler input using a first type of Nédélec elements (Nédélec, 1980) on triangles for the partition of unity formulation of the vector Helmholtz wave equation. Like previous examples, an enriched finite element is defined. To obtain the enriched finite element, a continuous linear first type of Nédélec element is enriched with a discontinuous element that is defined by restricting the continuous Nédélec element to the discontinuity surface. The test and trial functions are defined on this enriched space. A linear Lagrange finite element space is also introduced on which the coefficient representing curl(J) is defined. After defining coefficient functions and constant values, linear and bilinear forms are defined at the end to complete the variational formulation.

The generated code is used to model the discontinuity surface in the unit square domain. The computed solution for the magnetic field is presented in Figure 6.15. The influence of the discontinuity surface in the magnetic field can be easily observed. Note that the discontinuity surface is a magnetically conducting surface.

6.6 H^1-conforming primal approach to the hyperelasticity problem

Recall the modelling of the hyperelasticity problem presented in Section 3.2.3 using the components of FEniCS. The automated framework for the modelling of discontinuities can be used to model pre-existing discontinuity surfaces for a hyperelastic domain. A relevant function space in which the displacement field exhibits jumps over discontinuity surfaces is defined as

$$V = \left\{ \boldsymbol{v}_h \in \left(L^2\left(\Omega\right)\right)^d \cap \left(H^1\left(\Omega \backslash \Gamma_d\right)\right)^d, \boldsymbol{v}_h|_E \in \left(P_k\left(E \backslash \Gamma_d\right)\right)^d \ \forall E \right\}. \tag{6.26}$$

If the discontinuity surfaces are not in closing states and they are traction-free, the corresponding bilinear and linear forms are the same as those given in

```
# Continuous/discontinuous H(curl) finite element space
EN_c = FiniteElement("Nedelec 1st kind H(curl)", "triangle", 1)
EN_d = RestrictedElement(EN_c, dc)
EN = EN_c + EN_d

# Finite element space for curl_J
EL = VectorElement("Lagrange", "triangle", 1)

# Test and trial functions
T = TestFunction(EN)
H = TrialFunction(EN)

# A coefficient for curl_J and electromagnetic constants
curl_J = Coefficient(EL)
k0, epsilon = constant("triangle"), constant("triangle")
mu = Constant("triangle"),

# Bilinear and linear forms
a = (1/mu)*inner(curl(H), curl(T))*dx \
  + (k0**2)*epsilon*inner(H, T)*dx
L = (1/mu)*inner(curl_J, T)*dx
```

Figure 6.14 The UFL input for the vector wave equation with discontinuous magnetic fields.

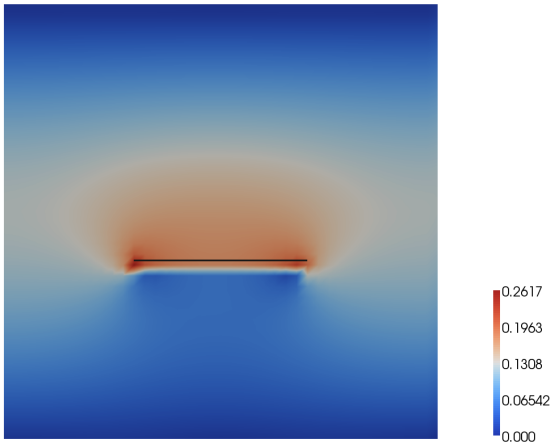


Figure 6.15 The magnetic field (Am^{-1}) in a two-dimensional domain subjected to a constant divergence of electric current density in the horizontal direction.

Equations (3.19) and (3.20) for the hyperelasticity problem without any discontinuity surface. For this example, a neo-Hookean constitutive law for the stored potential energy, similar to the one presented in Equation (3.18), is also selected. For the UFL input, it is enough to redefine code lines corresponding to the definition of the

function spaces inside the UFL input illustrated in Figure 3.9. To do so, the function space that is defined as

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

is changed to

```
elem_c = VectorElement("Lagrange", "tetrahedron", 1)
elem_d = RestrictedElement(elem_cont, dc)
element = elem_c + elem_d
```

inside the UFL input and the rest of the input remains unchanged.

The compiled UFL input is used to model a unit cube $\Omega = (0,0,1) \times (0,1,0) \times (1,0,0)$ containing a discontinuity surface. The discontinuity surface is defined using the scalar level set functions as $\psi = z - 0.6 = 0$ and $\phi = x^2 + y^2 - 0.36 \leq 0$. The mesh on the faces of the unit cube along with the discontinuity surface is presented in Figure 6.17(a).

The displacement field at a subdomain $\Gamma_b = 0 \times (1,0,0) \times (0,1,0)$ is set to zero and relevant Dirichlet boundary conditions are applied on a subdomain $\Gamma_t : 1 \times (1,0,1) \times (0,1,1)$ such that they can produce a 20-degree clock-wise rotation. This rotation is around an axis parallel to the Cartesian z axis and passing through the center of the cube. For this example, $\boldsymbol{B} = (0,0,-0.5)\,\mathrm{Nm}^{-3}$ and $\boldsymbol{T} = (0.1,0,0)\,\mathrm{Nm}^{-2}$. The Young modulus and the Poisson ratio are $10\,Pa$ and 0.3, respectively.

An extract of the DOLFIN/C++ code for solving this nonlinear problem is presented in Figure 6.16. If the C++ code extract is compared with the one for the standard problem presented in Figure 3.10, a number of differences can be merely noticed including the definition of surfaces in which solutions are discontinuous and the use of the `pum::Function` object to manipulate solutions defined on the enriched function space.

The displacement contours are illustrated in Figure 6.17(b). Notice that the contours are presented on the deformed configuration without any magnification.

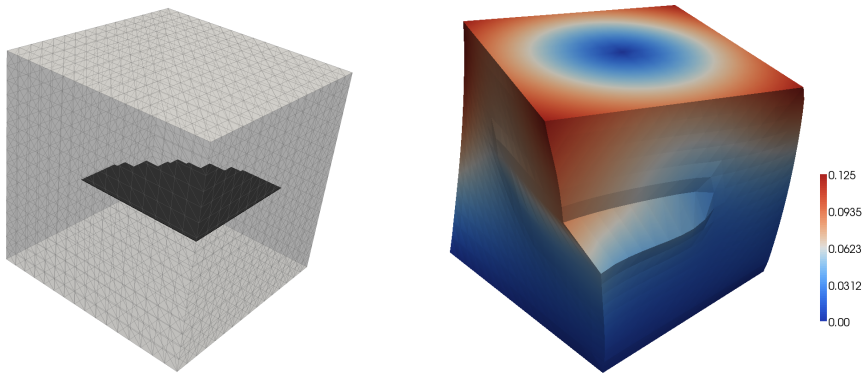
6.7 Cohesive crack propagation

The automated framework may be used to model evolving surfaces whose geometry changes during simulation. Similar to the modelling of static surfaces, the PDE-specific data is passed to the form compiler to generate low-level code which is used to model evolving surfaces. For modelling the evolving surfaces, the solver interface provides required functionalities such as criteria for initiation of surface evolutions and geometrical updates for the evolved surfaces.

In this example, an automated modelling of cohesive cracks in an elastic domain using the partition of unity framework is studied. The problem is defined on a domain

```
// Define discontinuity surface
Surface surface;
Boundary boundary;
Surface d(mesh, surface, boundary);

// Create list of surfaces
std::vector<const GenericSurface*> surfaces;
surfaces.push_back(&d);

// Create function space
HyperElasticity::FunctionSpace V(mesh, surfaces);

// Create linear form
HyperElasticity::LinearForm F(V);
F.mu = mu; F.lmbda = lambda;
F.B = B; F.T = T; F.u = u;

// Create Jacobian dF = F' (for use in nonlinear solver).
HyperElasticity::BilinearForm dF(V, V);
dF.mu = mu; dF.lmbda = lambda; dF.u = u;

// Compute solution
solve(F == 0, bcs, u, dF);

// Save solution in VTK format
dolfin::File file("hyper_elasticity.pvd");
file << u;
```

Figure 6.16 The C++ code extract for the modelling discontinuities in a hyperelastic domain.



(a) (b)

Figure 6.17 The hyperelasticity problem in three dimensions (a) the mesh on the faces of the unit cube and the discontinuity surface and (b) the contour of the displacement magnitude (m) on the deformed body.

$\Omega \subset \mathbb{R}^d$ and it can be phrased in a variational format as: find $\boldsymbol{u} \in V$ such that

$$F(\boldsymbol{u};\boldsymbol{v}) \equiv \int_{\Omega\setminus\Gamma_d} \boldsymbol{\sigma}(\boldsymbol{u}) : \nabla \boldsymbol{v}\, dx - \int_{\Omega\setminus\Gamma_d} \boldsymbol{f}\cdot\boldsymbol{v}\, dx + \int_{\Gamma_t} \boldsymbol{g}\cdot\boldsymbol{v}\, ds + \int_{\Gamma_d} \boldsymbol{t}\cdot[\![\boldsymbol{v}]\!]\, ds = 0 \quad \forall\ \boldsymbol{v}\in V, \tag{6.27}$$

where $\boldsymbol{f}$ and $\boldsymbol{g}$ are a source term on the whole domain Ω and a traction force on the Neumann boundary Γ_t, respectively. Furthermore, $\boldsymbol{t}$ is a traction force across a discontinuity surface Γ_d. The same function space as the one presented in Equation (6.26) is used. Various traction–separation laws can be defined to obtain traction forces on the discontinuity surfaces. For this example, nonlinear traction–separation laws are assumed across the surfaces. An exponential traction–separation law is utilized for the normal component, while a quadratic constitutive law is used for the tangential component of the traction force. The traction–separation law reads

$$\boldsymbol{t} = \begin{pmatrix} t_n \\ t_s \end{pmatrix} = \begin{pmatrix} k_n e^{-c[\![\boldsymbol{u}]\!]_n} \\ k_s {[\![\boldsymbol{u}]\!]_s}^2 \end{pmatrix}, \tag{6.28}$$

where subscripts n and s denote the normal and tangential components on discontinuity surfaces, respectively. Moreover, k_n, c and k_s are parameters of the constitutive laws. Note that this law cannot capture the closure of cracks and thus the loading for this example is selected such that the crack is always in an opening state.

The functional F, presented in Equation (6.27), is linear in $\boldsymbol{v}$ but nonlinear in $\boldsymbol{u}$. A nonlinear problem posed in this format can be solved using the Newton-Raphson method, in which F is driven to zero by solving a series of linear systems until a prescribed tolerance is reached. The functional F, evaluated at the most recent approximation of $\boldsymbol{u}$, serves as the linear form and the Jacobian of F is used as the bilinear form. The Jacobian is computed as Equation (3.20).

The traction-separation laws across the discontinuity surfaces can be implemented using two approaches. The first approach is the implementation of the traction-separation laws directly inside the UFL input. Figure 6.18 gives a UFL input to generate required code to model cohesive cracks in an elastic domain. As can be seen, the traction-separation law is defined inside the UFL input. Notice also the definition of a zero-order discontinuous Lagrange vector element in the UFL input. Since normal and tangent vectors of the discontinuity surfaces are assumed constant for each cell, coefficients for the normal and tangent vectors are defined on this element. Consider also the automatic differentiation to compute the Jacobian

```
# Continuous and discontinuous spaces
elem_cont = VectorElement("Lagrange", "triangle", 1)
elem_discont = RestrictedElement(elem_cont, dc)

# Enriched space
element = elem_cont + elem_discont

# Test and trial functions
v, du = TestFunction(element), TrialFunction(element)

# Function space and coefficients for normal and tangent
# of discontinuity surface
elem = VectorElement("Discontinuous Lagrange", triangle, 0)
normal, tangent  = Coefficient(elem), Coefficient(elem)

# Source term, traction and solution from previous step
f, g = Coefficient(elem_cont), Coefficient(elem_cont)
u = Coefficient(element)

# Material properties
mu, lmbda = Constant("triangle"), Constant("triangle")

# Parameters for traction-separation laws of surfaces
Kt, Kn = Constant("triangle"), Constant("triangle")
c = Constant("triangle")

# Stress
def sigma(v):
   return 2.0*mu*sym(grad(v)) \
        + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

# Jump over local coordinates of surfaces
def ljump(u):
  return as_vector([inner(jump(u), normal('+')), \
                    inner(jump(u), tangent('+'))])

# Traction-separation laws
def traction(u):
  return as_vector([Kn('+')*exp(c('+')*ljump(u)[0]), \
                    Kt('+')*ljump(u)[1]**2])

# Linear and bilinear forms
F = inner(sigma(u), grad(v))*dx - inner(f, v)*dx \
  + inner(g, v)*ds + inner(traction(u), ljump(v))*dc
dF  = derivative(F, u, du)
```

Figure 6.18 A UFL input for an elastic domain with tractions across the surfaces computed inside the UFL input.

of F for the bilinear form.

The second approach is the implementation of the traction-separation laws inside the C++ solver and using them as coefficients for UFL inputs. This approach is

more general and can be used to implement a wide range of traction-separation laws. Figure 6.19 shows the UFL input that depends on the traction-separation laws defined inside the C++ solver. Note that the automatic differentiation is not used to compute the Jacobian for this case. The tractions on the discontinuity surfaces as well as the derivatives of the tractions are computed inside the C++ solver and they are defined as coefficient functions on a zero-order discontinuous Lagrange vector element. The derivatives of the tractions are used to compute the Jacobian. The rest of the UFL input is the same as the UFL input presented in Figure 6.18, including the definition of the function spaces, the stress and the local jump.

The UFL input (presented in Figure 6.18 or Figure 6.19) can be used to model the failure of a rectangular specimen shown in Figure 6.20. The specimen is restricted at the bottom and is subjected to a displacement controlled tension loading on the top. The height and width of the specimen are 20 cm and 10 cm, respectively. An initial imperfection is placed at the left edge 15 cm above the bottom edge. This imperfection helps the initiation of a crack from this point. The material properties are taken as: Young's modulus $E = 11.93 \times 10^3$ MPa, Poisson's ratio $\nu = 0.49$ and the yield stress $\bar{\sigma} = 100$MPa. The parameters used for the constitutive relations for discontinuity surfaces are as follows: $k_s = 1 \times 10^4\,\mathrm{Nm}^{-2}$, $k_n = 1 \times 10^4\,\mathrm{N}$ and $c = 1000\,\mathrm{m}^{-1}$.

By increasing the tensile loading at the top, the crack starts to propagate until it reaches the other side of the specimen. The crack propagation direction is fixed to $-45°$ with respect to the x-axis. The crack propagates if the Von Misses stress in front of the crack tip reaches the yield stress. Since a linear approximation is used for the displacement field, the Von Misses stress is constant in cells that are not intersected by the crack. Therefore, the Von Misses stress can be easily computed by the evaluation of a functional that is constructed using another UFL input. This UFL input receives the computed displacement field as an input coefficient and computes the Von Misses stress.

The computed displacement contours in magnified deformed meshes during the failure simulation are presented in Figure 6.21. As shown, the crack propagates until it reaches the other side of the specimen. Even after this moment, the specimen can still be loaded because of the tractions existing on the crack surfaces.

6.8 Partially saturated porous media problem

Modelling coupled nonlinear problems, in which different fields exist and interact with each other, is a challenging topic in the finite element framework. The solution algorithms can be designed considering either weak or strong couplings. In order to obtain stable solutions, careful considerations must be devoted to selecting proper finite element spaces. The complexity of variational formulations and thus

```
# Continuous and discontinuous spaces
elem_cont = VectorElement("Lagrange", "triangle", 1)
elem_discont = RestrictedElement(elem_cont, dc)

# Enriched space
element = elem_cont + elem_discont

# Test and trial functions
v, du = TestFunction(element), TrialFunction(element)

# Function space and coefficients for normal and tangent
# of discontinuity surface
elem = VectorElement("Discontinuous Lagrange", triangle, 0)
normal, tangent  = Coefficient(elem), Coefficient(elem)

# Source term, traction and solution from previous step
f, g = Coefficient(elem_cont), Coefficient(elem_cont)
u = Coefficient(element)

# Material properties
mu, lmbda = Constant("triangle"), Constant("triangle")

# Stress
def sigma(v):
   return 2.0*mu*sym(grad(v)) \
        + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

# Jump over local coordinates of surfaces
def ljump(u):
  return as_vector([inner(jump(u), normal('+')), \
                    inner(jump(u), tangent('+'))])

# Traction on the discontinuity Surface
Traction     = Coefficient(elem)

# The derivative of the traction on the discontinuity surfaces
# (Obtained after linearisation)
dTraction_du = Coefficient(elem)

# Linear and bilinear forms
F = inner(sigma(u), grad(v))*dx - inner(f, v)*dx \
   + inner(g, v)*ds + inner(traction, ljump(v))*dc

dF = inner(sigma(du), grad(v))*dx \
   + ljump(du)[i]*dtraction_du[i]*ljump(v)[i]*dc
```

Figure 6.19 A UFL input for the elastic domain with tractions across the surfaces computed offline in the C++ solver.

obtaining element tensors increases significantly in the partition of unity enrichment framework, where different combinations of continuous/discontinuous spaces may be used.

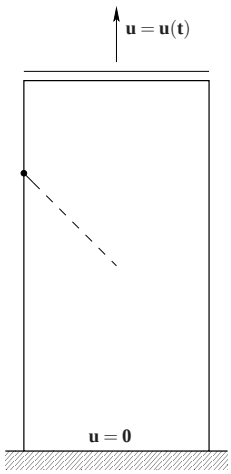


Figure 6.20 The problem configuration for the cohesive crack propagation.

To show the applicability of the compiler approach in coupled nonlinear problems, the modelling of discontinuities in a partially saturated porous medium in a domain Ω with discontinuity surfaces Γ_d is presented. For the partially saturated porous medium, the pore pressure and displacement fields are unknowns. These fields are strongly coupled via a linear momentum balance equation and a mass balance equation. The linear momentum balance equation reads

$$\nabla \cdot \boldsymbol{\sigma} - \alpha \nabla p_w + \rho \boldsymbol{g} = \boldsymbol{0} \qquad \text{in } \Omega, \tag{6.29}$$

where $\boldsymbol{\sigma}$, α, p_w, $\boldsymbol{g}$ are stress tensor, Biot coefficient, pore pressure and gravity vector, respectively. The density of the solid, denoted by ρ, is defined as

$$\rho = (1 - n)\rho_s + nS_w\rho_w, \tag{6.30}$$

where n, ρ_s, S_w and ρ_w are solid porosity, density of solid, saturation level and density of water, respectively. For the linear elastic material, the stress tensor reads $\boldsymbol{\sigma}(\boldsymbol{u}) = 2\mu\boldsymbol{\epsilon}(\boldsymbol{u}) + \lambda tr(\boldsymbol{\epsilon}(\boldsymbol{u}))\boldsymbol{I}$ where μ and λ are material properties of the solid skeleton and $\boldsymbol{\epsilon}$ is the linearised strain tensor.

The mass balance equation for water in a partially saturated medium, neglecting

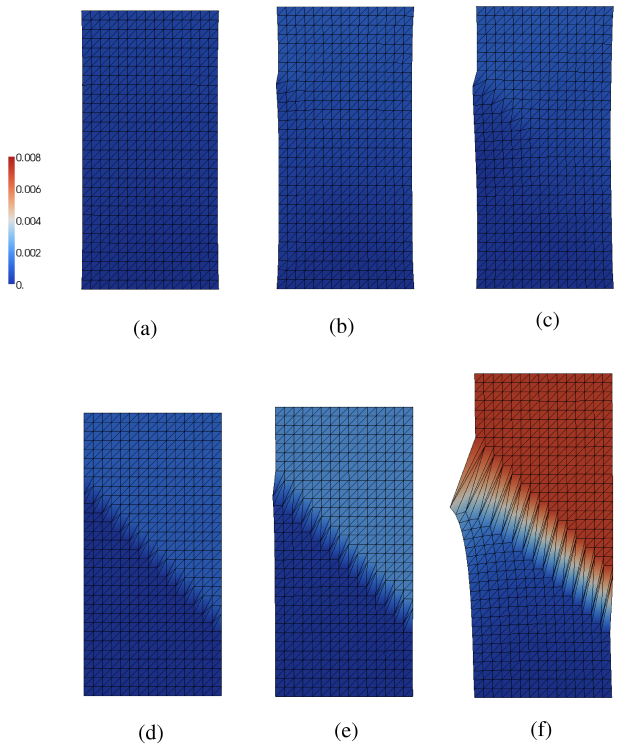


Figure 6.21 The evolution of displacement contours on the magnified deformed meshes. The specimen can continue carrying the load even if the crack is fully developed.

the mass rate evaporation and the gradient of the water density ($\nabla \rho_w = \mathbf{0}$), reads

$$\left(\frac{\alpha - n}{K_s} {S_w}^2 + \frac{n}{K_w} S_w\right) \frac{\partial p_w}{\partial t} + \alpha S_w \nabla \cdot \left(\frac{\partial \boldsymbol{u}}{\partial t}\right) + \left(\frac{\alpha - n}{K_s} p_w S_w + n\right) \frac{\partial S_w}{\partial t} + \nabla \cdot \left[\frac{\boldsymbol{k} k_{rw}}{\mu_w}(-\nabla p_w + \rho_w \boldsymbol{g})\right] = 0, \qquad \text{in } \Omega, \tag{6.31}$$

where K_s is bulk modulus of the solid, K_w denotes bulk modulus of the fluid phase, t is a time variable, μ_w is dynamic viscosity of water, $\boldsymbol{k}$ is a second-order permeability tensor of the solid phase and k_{rw} is relative permeability of water. Unlike the fully saturated porous media problems, S_w and k_{rw} are not constant and may change during simulation.

The following boundary conditions are assumed on the boundaries of domain Ω

and the discontinuity surfaces Γ_d:

$$\boldsymbol{u} = \boldsymbol{0} \qquad \text{on } \Gamma_u, \tag{6.32}$$

$$p = 0 \qquad \text{on } \Gamma_p, \tag{6.33}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{t} \qquad \text{on } \Gamma_t, \tag{6.34}$$

$$\frac{\boldsymbol{k} k_{rw}}{\mu_w}(-\nabla p_w + \rho_w g) \cdot \boldsymbol{n} = f \qquad \text{on } \Gamma_q, \tag{6.35}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = \bar{\boldsymbol{t}} \qquad \text{on } \Gamma_d, \tag{6.36}$$

where $\Gamma_u \cup \Gamma_t = \Gamma_p \cup \Gamma_q = \partial\Omega$ and $\Gamma_u \cap \Gamma_t = \Gamma_p \cap \Gamma_q = \emptyset$. The outward vector normal to the boundaries is defined as $\boldsymbol{n}$. The traction over Γ_t and the flux over Γ_q are respectively denoted by $\boldsymbol{t}$ and f. Moreover, $\bar{\boldsymbol{t}}$ defines a traction over the discontinuity surfaces Γ_d.

To obtain weak formulations, the strong forms are multiplied by test functions $\boldsymbol{w}$ and q, integrated over domain Ω and then integrated by parts. The weak formulations for the linear momentum balance equation and the mass balance equation read:

$$\int_{\Gamma_t} \boldsymbol{t} \cdot \boldsymbol{w}\, ds - \int_\Omega \nabla\boldsymbol{\sigma} : \nabla\boldsymbol{w}\, dx + \int_\Omega \alpha S_w \nabla p \cdot \boldsymbol{w}\, dx - \int_\Omega \rho \boldsymbol{g} \cdot \boldsymbol{w}\, dx + \int_{\Gamma_d} \boldsymbol{t} \cdot [\![\boldsymbol{w}]\!]\, ds = 0, \tag{6.37}$$

$$\int_\Omega (\frac{\alpha - n}{K_s} S_w + \frac{n}{K_w}) S_w \frac{\partial p}{\partial t} q\, dx + \int_\Omega \alpha S_w \nabla \cdot \left(\frac{\partial \boldsymbol{u}}{\partial t}\right) q\, dx + \int_\Omega (\frac{\alpha - n}{K_s} p S_w + n) \frac{\partial S_w}{\partial t} q\, dx + \int_\Omega \frac{\boldsymbol{k}}{\mu_w} k_{rw} \nabla p \cdot \nabla q\, dx - \int_\Omega \frac{\boldsymbol{k}}{\mu_w} \rho_w k_{rw} \boldsymbol{g} \cdot \nabla q\, dx + \int_{\Gamma_q} f q\, ds = 0. \tag{6.38}$$

To treat time derivatives, a θ-family formulation has been used for the time discretization. The discretized forms of the weak variational formulations in the time domain read

$$\int_{\Gamma_t} \boldsymbol{t}^{i+1} \cdot \boldsymbol{w}\, ds - \int_\Omega \nabla\boldsymbol{\sigma}^{i+1} : \nabla\boldsymbol{w}\, dx + \int_\Omega \alpha {S_w}^{i+1} \nabla p^{i+1} \cdot \boldsymbol{w}\, dx - \int_\Omega \rho^{i+1} \boldsymbol{g} \cdot \boldsymbol{w}\, dx + \int_{\Gamma_d} \boldsymbol{t}^{i+1} \cdot [\![\boldsymbol{w}]\!]\, ds = 0, \tag{6.39}$$

$$\int_\Omega (\frac{\alpha - n}{K_s} S_w^{i+\theta} + \frac{n}{K_w}) S_w^{i+\theta} \frac{p^{i+1} - p^i}{\mathrm{d}t} q \, dx + \int_\Omega \alpha S_w^{i+\theta} \frac{\nabla \cdot (\boldsymbol{u}^{i+1} - \boldsymbol{u}^i)}{\mathrm{d}t} q \, dx$$
$$+ \int_\Omega (\frac{\alpha - n}{K_s} p^{i+\theta} S_w^{i+\theta} + n) \frac{S_w^{i+1} - S_w^i}{\mathrm{d}t} q \, dx + \int_\Omega \frac{\boldsymbol{k}}{\mu_w} {k_{rw}}^{i+\theta} S_w^{i+\theta} \nabla p^{i+\theta} \cdot \nabla q \, dx$$
$$- \int_\Omega \frac{\boldsymbol{k}}{\mu_w} \rho_w k_{rw}^{i+\theta} \boldsymbol{g} \cdot \nabla q \, dx + \int_{\Gamma_q} f^{i+\theta} q \, ds = 0, \quad (6.40)$$

where $\mathrm{d}t$ is time step and $(\cdot)^{i+\theta}$ is a quantity evaluated at step $i+\theta$ that is defined as $(1-\theta)(\cdot)^i + \theta(\cdot)^{i+1}$, where θ denotes a time discretization parameter, $(\cdot)^i$ and $(\cdot)^{i+1}$ are these quantities evaluated at time step i and $i+1$. Notice that the first equation is only evaluated at step $i+1$. Equations (6.39) and (6.40) construct a nonlinear system F and the goal is to find $(\boldsymbol{u}^{i+1}, p^{i+1}) \in V \times Q$ such that

$$F(\boldsymbol{u}^{i+1}, p^{i+1}; \boldsymbol{w}, q) = 0 \qquad \forall (\boldsymbol{w}, q) \in V \times Q, \quad (6.41)$$

where V and Q are suitable function spaces corresponding to the displacement and pressure fields, respectively. To solve this system, linearisation of F is performed which leads to the following equation:

$$J(\mathrm{d}\boldsymbol{u}^{i+1}, \mathrm{d}p^{i+1}) = -F({\boldsymbol{u}_0}^{i+1}, {p_0}^{i+1}; \boldsymbol{w}, q) \quad (6.42)$$

where the vector $(\mathrm{d}\boldsymbol{u}^{i+1}, \mathrm{d}p^{i+1})$ contains the unknowns and represents the increment of the solutions at step $i+1$. Furthermore, the vector $({\boldsymbol{u}_0}^{i+1}; {p_0}^{i+1})$ is the solution (at step $i+1$) from the previous time step and J is the Jacobian of F computed as Equation (3.20).

For this example, a simple linear constitutive law has been assumed to compute the traction over discontinuity surfaces from displacement jumps. The constitutive law reads

$$\boldsymbol{t} = \begin{bmatrix} t_n \\ t_s \end{bmatrix} = \boldsymbol{T} [\![\boldsymbol{u}]\!] = \begin{bmatrix} k_n & 0 \\ 0 & k_s \end{bmatrix} \begin{bmatrix} [\![\boldsymbol{u}]\!]_n \\ [\![\boldsymbol{u}]\!]_s \end{bmatrix}, \quad (6.43)$$

where $\boldsymbol{T}$ represents a second order tensor and k_n and k_s denote corresponding surface stiffnesses in the normal and tangential directions of the discontinuity surfaces, respectively.

The constitutive laws for the saturation and the relative permeability as functions of the pressure p_w are also defined as

$$S_w(p_w) = 1 - \frac{c_0}{c_1 + c_2 p_w}, \quad (6.44)$$
$$k_{rw}(p_w) = 1 - c_3 e^{c_4 p_w}, \quad (6.45)$$

where c_0, c_1, c_2, c_3 and c_4 are constant parameters.

The Taylor-Hood elements (Taylor and Hood, 1973) are used for the mixed space of the displacement and the pressure fields. Two different formulations for modelling discontinuities for this example are assumed, 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 the surfaces. For the case in which both the pressure and displacement fields are discontinuous across the surfaces, the relevant function spaces for the Taylor-Hood element are defined. The function spaces for the displacement field and the pressure field are same as function spaces presented in Equations (6.26) and (6.1) and they are again redefined as

$$V = \left\{ \boldsymbol{v}_h \in \left(L^2\left(\Omega\right)\right)^d \cap \left(H^1\left(\Omega \backslash \Gamma_d\right)\right)^d, \boldsymbol{v}_h|_E \in \left(P_k\left(\Omega\right) \backslash \Gamma_d\right)^d \ \forall E \right\}, \tag{6.46}$$

$$Q = \left\{ p_h \in L^2\left(\Omega\right) \cap H^1\left(\Omega \backslash \Gamma_d\right), p_h|_E \in P_{k-1}\left(\Omega \backslash \Gamma_d\right) \ \forall E \right\}, \tag{6.47}$$

where $k > 1$. For the discontinuous displacement field and the continuous pressure field, the relevant function space for the pressure filed, Q, requires re-definitions as

$$Q = \left\{ p_h \in H^1\left(\Omega\right), p_h|_E \in P_{k-1}\left(\Omega\right) \ \forall E \right\}. \tag{6.48}$$

Each combination of continuous/discontinuous function spaces leads to a new variational formulation which results in different element tensors with different dimensions. For example, compare the formulation presented in Armero and Callari (1999) for discontinuous displacement field and continuous pressure field to the formulation for discontinuous pressure and displacement fields in Larsson and Larsson (2000).

Using the compiler approach, one can switch trivially between two formulations with the Taylor-Hood element. For the case of both discontinuous displacement and pressure fields, an extract of the UFL input that closely resembles the formulation explained here is shown in Figures 6.22 and 6.23. In the UFL input, an enriched quadratic Lagrange vector finite element space and an enriched linear Lagrange finite element space are defined. The two finite element spaces are combined to create a Taylor-Hood element on which test functions and trial functions for the displacement and pore pressure fields are defined. After definitions of coefficient functions representing solutions from current and previous time steps, the saturation and permeability, the linear form is defined. At the end, automatic differentiation is used to construct the Jacobian (bilinear form) from the linear form.

Note that in order to use other constitutive models for the saturation and the relative permeability as well as traction-separation laws on discontinuity surfaces, it is enough to define these models inside the UFL input or to compute them inside

```
# Define continuous and discontinuous spaces
u_elem_c = VectorElement("Lagrange", "triangle", 2)
p_elem_c = FiniteElement("Lagrange", "triangle", 1)
u_elem_d = RestrictedElement(u_elem_c, dc)
p_elem_d = RestrictedElement(p_elem_c, dc)

# Define enriched spaces
u_elem, p_elem = u_elem_c + u_elem_d, p_elem_c + p_elem_d
Element = u_elem * p_elem
```

Figure 6.22 The UFL input for the definition of finite element spaces for discontinuous $\boldsymbol{u}$ and p.

the C++ solver and return results as coefficients to the UFL input, presented in Figure 6.23. The rest of the input including the definition of bilinear and linear forms are unchanged.

To switch to a formulation with continuous pressure, only the function spaces in the PUM form compiler input, presented in Figure 6.22, need to be changed. Figure 6.24 presents the UFL input for defining the finite element function spaces for discontinuous $\boldsymbol{u}$ but continuous p.

Using both formulations, the behavior of a specimen, containing discontinuity surfaces as slip planes, subjected to self-weight is studied. The specimen is a $3\,m \times 1\,m$ rectangular sample with three disjoint linear slip planes restricted at the bottom for the displacement field ($\boldsymbol{u} = \boldsymbol{0}$ at $y = 0$) and subjected to zero pressures at the top ($p = 0$ at $y = 3$). The end points for slip planes are defined as follows: $(0.0, 0.8)$ and $(0.5, 1.1)$ for the first slip plane Γ_{d1}, $(1.0, 1.9)$ and $(0.7, 2.1)$ for the second slip plane Γ_{d2} and $(0.3, 0.4)$ and $(0.7, 0.4)$ for the last slip plane Γ_{d3}.

The problem configuration as well as a mesh with embedded slip planes are illustrated in Figure 6.25. The slip planes are assumed to be impervious to the water flow (zero flow flux across the slip planes). The sliding is also allowed along the slip planes by assuming $k_s = 0.0\,\mathrm{Nm}^{-1}$ and $k_n = 1 \times 10^7\,\mathrm{Nm}^{-1}$ for the constitutive law presented in Equation (6.43). Some of the parameters related to the fluid and solid phases as well as the time discretization are shown in Table 6.1. The parameters for the constitutive laws for saturation and relative permeability are assumed as $c_0 = 0.1$, $c_1 = 1 \times 10^5$, $c_2 = 0.01\,\mathrm{Pa}^{-1}$, $c_3 = 1.0$ and $c_4 = -1 \times 10^{-5}\,\mathrm{Pa}^{-1}$.

The extract of the C++ solver is presented in Figure 6.26. The implementation details of slip planes are given in the extracted code. Note the approach used to solve the transient nonlinear problem. Inside each time step, the variational problem is solved using the updated coefficients.

The computed solutions for the pressure and vertical displacement fields considering discontinuities in both fields at different time steps are presented in Figures 6.27 and 6.28, respectively. Although the pressure field is allowed to be

```
# Test and trial functions, solutions from the previous
# and current time step
w, q  = TestFunctions(Element)
dU    = TrialFunction(Element)
U0, U = Coefficient(Element), Coefficient(Element)

# Extract solution for displacement and pressure for previous
# and current time step
[u0, p0], [u, p] = split(U0), split(U)

# Traction and flux
T, flux = Coefficient(u_elem_c), Coefficient(p_elem_c)

# Define constant variables related to fluid and solid phases,
# time-stepping and surface and coefficients for normal and
# tangent of surface
[...]

# Define Saturation
def Sw(p):
  return 1 - c0/(c1 + c2*p)

# Define permeability
def perm(p):
  return 1.0 - c3*exp(-c4*p)

# u_(n+theta), p_(n+theta), Sw_(n+theta) and krw_(n+theta)
u_mid   = as_vector([(1-theta)*u0[i] \
        + theta*u[i] for i in range(len(u))])
p_mid   = (1-theta)*p0 + theta*p
Sw_mid  = (1-theta)*Sw(p0) + theta*Sw(p)
krw_mid = (1-theta)*prem(p0) + theta*prem(p)

# Compute local jump and traction over discontinuity surface
def ljump(v):
  return as_vector([inner(jump(v), normal('+')), \
                    inner(jump(v), tangent('+'))])
def traction(v):
  return as_vector([kn('+')*ljump(v)[0], kt('+')*ljump(v)[1]])

# Linear form
Fsolid = inner(T, w)*ds - (inner(sigma(u), grad(w))
       + alpha*Sw*inner(grad(p), w) - Rhu*inner(G(g), w))*dx
       + inner(traction(u), ljump(w))*dc
Ffluid = ((alpha-n)/Ks*Sw_mid+n/Kw)*Sw_mid*(p-p0)/dt*q*dx
       + alpha*Sw_mid*(div(u)-div(u0))/dt*q*dx
       + (((alpha-n)/Ks)*p_mid*Sw_mid+n)*(Sw(p)-Sw(p0))/dt*q*dx
       + k/mu_w*krw_mid*inner(grad(p_mid), grad(q))*dx
       - k/mu_w*Rhuw*Krw_mid*inner(G(g), grad(q))*dx + flux*q*ds
F = Fsolid + Ffluid

# Bilinear form
J = derivative(F, U, dU)
```

Figure 6.23 The extract of the UFL input for the equations of partially saturated porous media with discontinuities

```
# Define continuous/discontinuous spaces for displacement
u_elem_c = VectorElement("Lagrange", "triangle", 2)
u_elem_d = RestrictedElement(u_elem_c, dc)
u_elem = u_elem_c + u_elem_d

# Define continuous space for pressure
p_elem = FiniteElement("Lagrange", "triangle", 1)

# Mixed space
Element = u_elem * p_elem
```

Figure 6.24 The UFL input for the definition of finite element spaces for discontinuous $\boldsymbol{u}$ and continuous p.

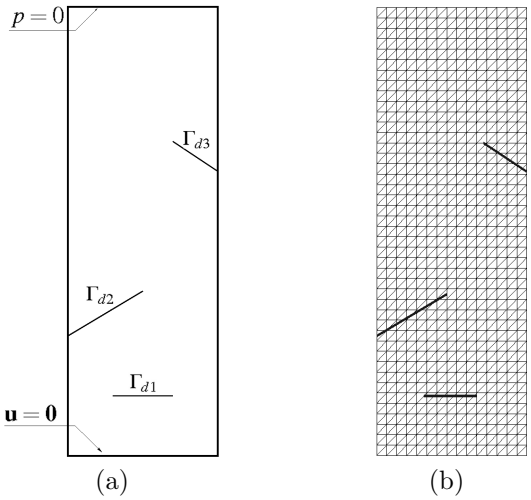


Figure 6.25 The partially saturated porous media example: (a) the problem configuration and assumed boundary conditions (b) a mesh with embedded slip planes.

Solid-skeleton Young modulus	E_s	1.3×10^7	Pa
Solid-skeleton Poisson ratio	ν	0.4	
Density of solid skeleton	ρ_s	2×10^3	kgm^{-3}
Density of water	ρ_w	1×10^3	kgm^{-3}
Initial porosity	n	0.2	
Dynamic viscosity of water	μ_w	1×10^{-3}	Nsm^{-2}
Solid phase bulk modulus	K_s	1.86×10^7	Pa
Water bulk modulus	K_w	2.2×10^9	Pa
Biot coefficient	α	1.0	
Time step	$\mathrm{d}t$	0.01	sec
Time parameter	θ	0.8	

Table 6.1 Parameters considered in the porous media example.

```
// Create slip plane #0
const Point p0_0(0.0, 0.8);
const Point p0_1(0.5, 1.1);
std::pair<Point, Point> end_points0(p0_0, p0_1);
Surface surface0(mesh, end_points0);

// Create slip plane #1
const Point p1_0(1.0, 1.9);
const Point p1_1(0.7, 2.1);
std::pair<Point, Point> end_points1(p1_0, p1_1);
Surface surface1(mesh, end_points1);

// Create slip plane #2
const Point p2_0(0.3, 0.4);
const Point p2_1(0.7, 0.4);
std::pair<Point, Point> end_points2(p2_0, p2_1);
Surface surface2(mesh, end_points2);

// Create vector of discontinuity surfaces
std::vector<const GenericSurface*> surfaces =
     boost::assign::list_of(&surface0)(&surface1)(&surface2);

// Create FunctionSpace
porousPUM::FunctionSpace V(mesh, surfaces);

// solution functions
pum::Function U0(V);
pum::Function U(V);

// Define bilinear and linear forms and attach coefficients
porousPUM::BilinearForm dF(V, V);
porousPUM::LinearForm F(V);
a.U0 = U0; a.U = U; L.U0 = U0; L.U = U;
[...]

// Output files
File file_u("disp.pvd");
File file_p("pressure.pvd");

while (t < T)
{
  // Update the displacement field from the previous time space
  U0.vector() = U.vector();

  // Solve nonlinear variational problem
  solve(F == 0, bcs, U, dF);

  // Save solution in VTK format
  file_u << U[0];
  file_p << U[1];

  // Move to next interval
  t += dt;
}
```

Figure 6.26 The C++ code extract for the modelling of slip planes in the partially saturated domain.

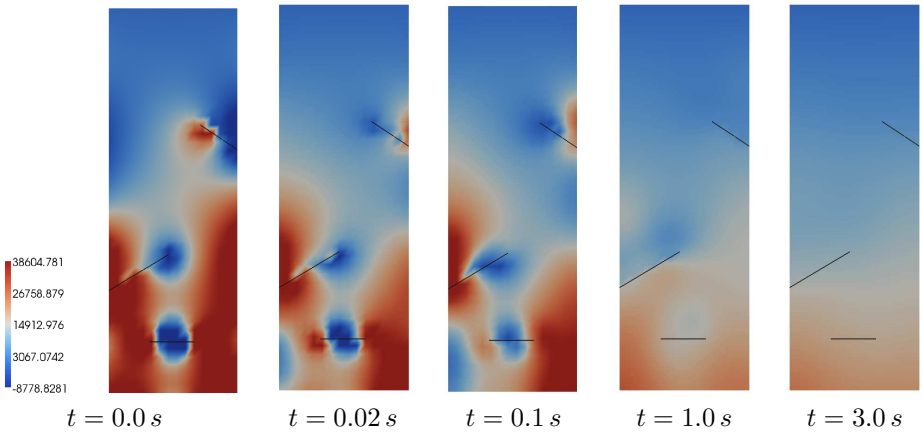


Figure 6.27 The pressure contours (Pa) for the case in which pressure and displacement fields are discontinuous in different time steps.

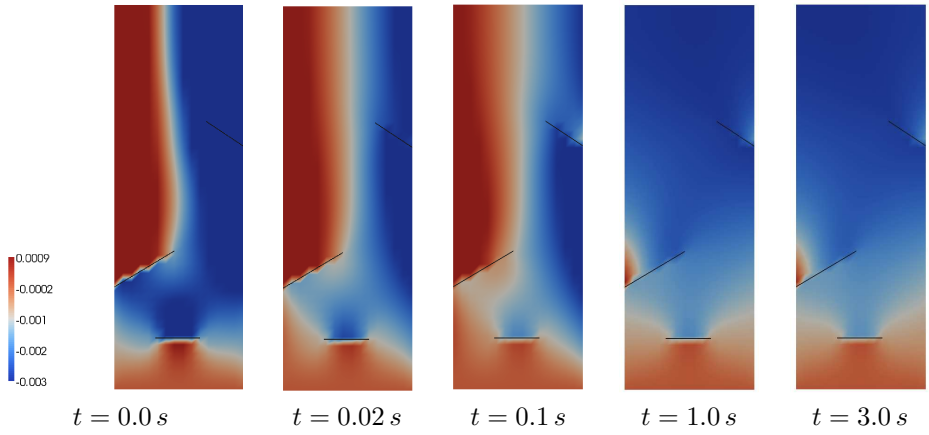


Figure 6.28 The vertical displacement contours (m) for the case in which pressure and displacement fields are discontinuous in different time steps.

discontinuous, it does not show any significant discontinuities in the computed solutions. As it can be observed, the extra pore pressure disappears in time and a hydrostatic pore pressure distribution is obtained as the specimen is drained.

If modelling with just discontinuous displacement is desired, it is just enough to replace the automatically generated header file corresponding to this case in the C++ solver. Except for this small change, the C++ solver is exactly same as the C++ solver used for the formulation in which both the displacement and pressure fields are

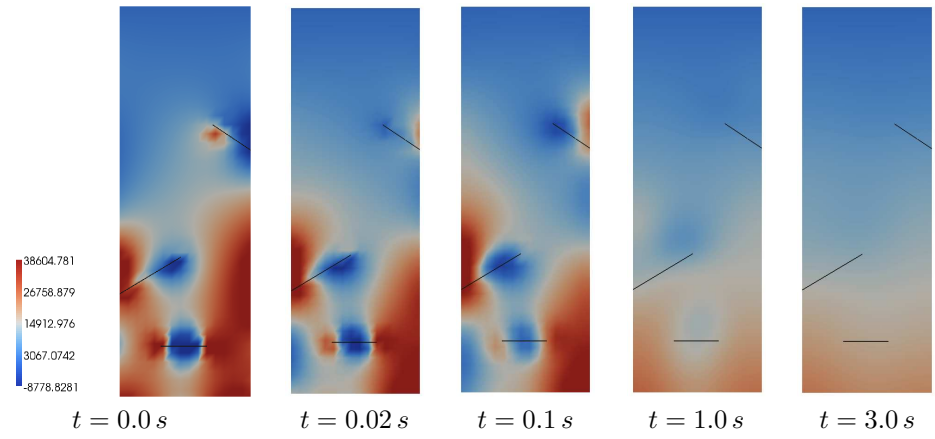


Figure 6.29 The pressure contours (Pa) for the case in which only displacement field is discontinuous in different time steps.

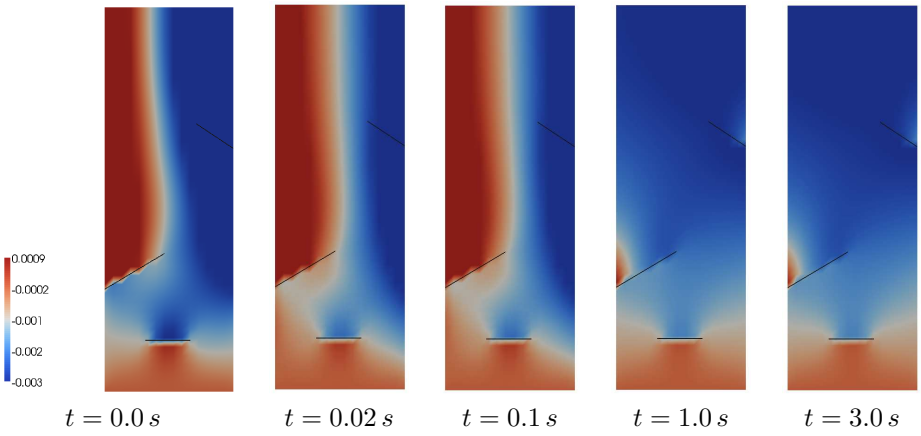


Figure 6.30 The vertical displacement contours (m) for the case in which only displacement field is discontinuous in different time steps.

discontinuous. Figures 6.29 and 6.30 respectively present the pressure and vertical displacement contours for this case in different time steps. Notice differences among the pressure and displacement contours and the contours presented in Figures 6.27 and 6.28 especially in subdomains close to the slip planes. These differences around the slip planes can cause different patterns for the evolved slip planes.

To compare two different formulations in more detail around the slip planes, the evolutions of the pressure and displacement fields at a point in the domain

are computed. The pressure and displacement fields in different steps have been evaluated at a point close to the end point of the slip plane Γ_{d2}, which is located on $1.1m$ from the bottom line and $0.45m$ from the left edge. The pressure and vertical displacement evolutions for these two different formulations are depicted in Figure 6.31. As can be seen, the displacement and pressure fields at the point close to Γ_{d2} have converged to unique values for two different formulations as time passed during simulation. Before reaching to the unique value, the pressure values showed rapid decreases at the beginning before increasing again to reach to a constant value.

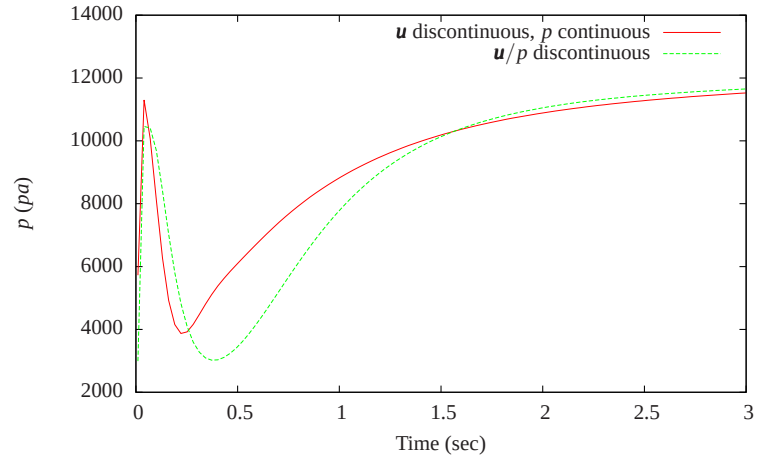
This example showed that the PUM form compiler can also be used to generate low-level code for transient coupled nonlinear problems. Different conditions for the continuity of underlying fields were considered which resulted in different variational formulations in the partition of unity framework. As it shown, switching between different formulations is trivial using the compiler approach and it is done by modifying the UFL inputs that define finite element spaces. Using the compiler approach enables users a fast development of partition of unity models for complex coupled problems and moves focus from the implementation details to the mathematical models.

6.9 Circular slip plane problem

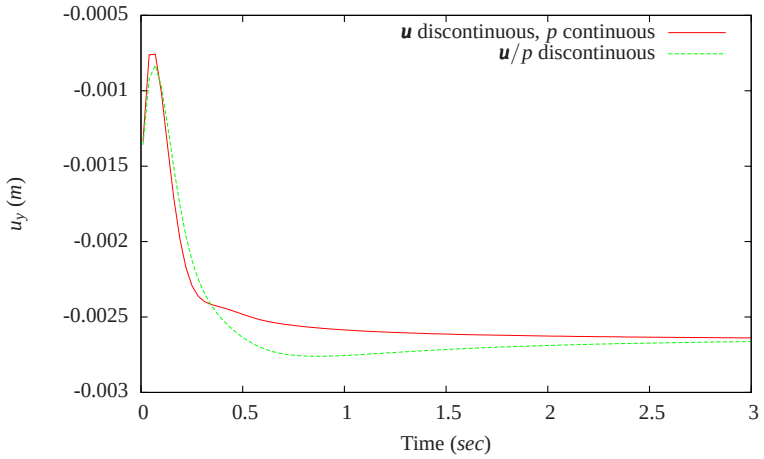
As discussed in Section 2.3.3, surface representations play an important role in the partition of unity framework. Different approaches exist to represent discontinuity surfaces and dependent on a problem one specific surface representation may give better results. Therefore, it is desirable to have a framework in which different surface representations can be examined with minimum reworks.

The proposed automated framework for modeling discontinuities provides possibilities of implementing different surface representations for finite element models. Different surface representations can be implemented as subclasses of the `pum::GenericSurface` base class. The UFL inputs are independent of the surface representations and therefore the same automatically generated code for variational formulations can be used with different surface representations. So, in order to examine a specific surface representation for a problem, it is enough to implement a subclass of `pum::GenericSurface` and no other change in the UFL input or the PUM solver library is required.

To show the possibility of using different surface representations, the automated framework is used to model a circular slip plane problem. For the circular slip plane



(a)



(b)

Figure 6.31 The evolution of (a) the pore pressure (Pa) (b) the vertical displacement (m) with time at a point close to the end point of the slip plane Γ_{d2} for different assumptions on the continuity of spaces.

problem, the bilinear and linear forms read

$$a(\boldsymbol{u}, \boldsymbol{v}) = \int_{\Omega \setminus \Gamma_d} \boldsymbol{\sigma}(\boldsymbol{u}) : \nabla \boldsymbol{v} \, dx + \int_{\Gamma_d} \boldsymbol{t} \cdot [\![\boldsymbol{v}]\!] \, ds, \tag{6.49}$$

$$L(\boldsymbol{v}) = \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{v} \, dx, \tag{6.50}$$

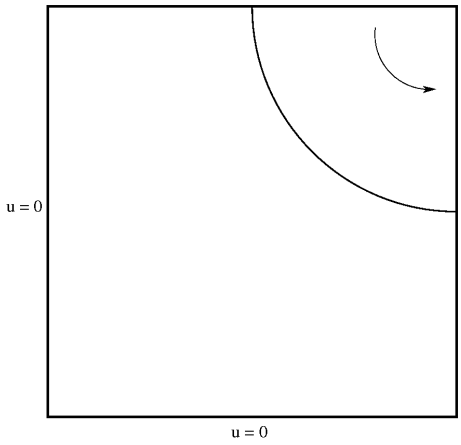


Figure 6.32 The problem configuration for the circular slip plane subjected to a torsional loading.

where $\boldsymbol{\sigma}$ is elastic stress tensor defined as Equation (2.6), $\boldsymbol{t}$ is traction force on a slip plane Γ_d, and $\boldsymbol{f}$ is body force. For this problem, an equivalent function space to the function space presented in Equation (6.26) is used for test and trail functions. The linear traction-separation law is used for this example, as introduced in Equation (6.43).

The computational domain, given in Figure 6.32, is a unit square $\Omega : (1,0) \times (0,1)$ with a circular slip plane Γ_d whose center is located at $(1,1)$ with a radius $R = 0.5$. The Young's modulus and the Poisson ratio are $2000\,Pa$ and 0.2, respectively. To allow sliding along the slip plane, $k_s = 0.0\,\mathrm{Nm}^{-1}$ and $k_n = 1 \times 10^6\,\mathrm{Nm}^{-1}$ are selected as the tangential and normal stiffnesses of the slip plane for the constitutive law presented in Equation (6.43).

The following boundary conditions for this problem are assumed:

$$\boldsymbol{u} = \boldsymbol{0} \quad \text{on} \quad 0 \times (0,1), \tag{6.51}$$

$$\boldsymbol{u} = \boldsymbol{0} \quad \text{on} \quad (0,1) \times 0, \tag{6.52}$$

$$\boldsymbol{u} = \boldsymbol{0} \quad \text{on} \quad (1,1), \tag{6.53}$$

$$u_y = -0.1 \quad \text{on} \quad (0.9,1), \tag{6.54}$$

$$u_x = 0.1 \quad \text{on} \quad (1,0.9), \tag{6.55}$$

$$t_s = 0 \quad \text{on} \quad \Gamma_d, \tag{6.56}$$

where t_s is the tangential traction on the slip plane. Note that all parameters have SI units. Considering these boundary conditions a pure rotation of the circular slip

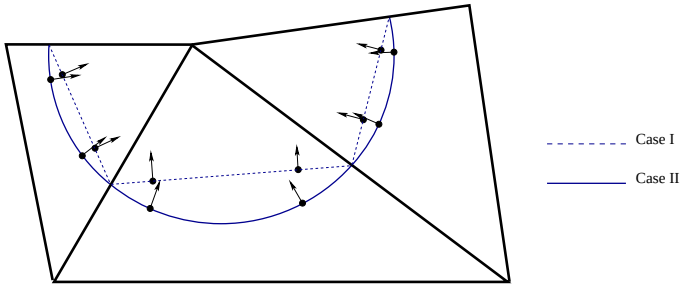


Figure 6.33 Two different surface representations: case I and case II that represent the approximated and exact representations, respectively. The quadrature points for each representation and their corresponding surface normals are also indicated.

plane around a center point $(1, 1)$ occurs.

An exact solution for the displacement field can be computed using this setting for the boundary conditions. The exact displacement field reads

$$\boldsymbol{u}(x, y) = \begin{cases} (1.0 - y)\boldsymbol{i} - (1.0 - x)\boldsymbol{j} & \text{if } (x - 1.0)^2 + (y - 1.0)^2 - R^2 \leq 0, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{6.57}$$

where $\boldsymbol{i}$ and $\boldsymbol{j}$ are the unit vectors in the x and y directions. The maximum magnitude of the displacement field that occurs on the boundaries of the slip plane is $0.5\,m$ and the L_2 norm of the displacement field is computed as $0.0245437\,m$.

Two different surface representations for the circular slip plane, as shown in Figure 6.33, are considered:

- Case I: a surface within each cell is approximated by a straight line passing through edge intersection points. By this assumption, the normal of the surface is constant within each cell and it is perpendicular to the straight line connecting the edge intersection points. The quadrature points used for the evaluation of surface integrals are also located on these straight lines.

- Case II: a surface within each cell is defined exactly by a circular arc passing through edge intersection points. The normal of the surface is not constant within cells and it is computed at quadrature points located on the circular arcs using the equation of the circular slip plane. The radius of each arc is equal to the radius of the slip plane.

As can be seen in Figure 6.33, the position of quadrature points and the normals may change considerably with the approach chosen for the surface representation. These differences are more apparent as the underlying mesh becomes coarser.

If the circular slip plane is approximated by straight lines (case I), spurious tangential traction forces on the slip plane may appear in the finite element models. These spurious traction forces come from the mismatch between the normal to the discretized slip plane (linear segments) and the radial vector normal to the circular slip plane. This mismatch will decrease with mesh refinement. However, since the discretization is not perfect and the normal of the discretized slip plane is never equal to the radial vector and therefore spurious traction forces always remain.

To implement a solver for this problem, the variational problem is represented in a UFL input at the first step. The compiler input file using linear Lagrange elements on triangles to model the circular slip plane is presented in Figure 6.34. To model the displacement field in the partition of unity framework a mixed enriched finite element is created at the first step. Notice the definition of this finite element by first enriching scalar elements and then creating a mixed element from two enriched scalar elements. This syntax enables access to the sub-components of the displacement field using functions defined in the solver library. These sub-components can be used to apply Dirichlet boundary conditions or to post-process for the x or y component of the displacement field, separately.

The generated code is included inside a DOLFIN-based solver. This solver models the unit square domain with a zero tangential stiffness on the circular slip plane with two different representations (case I and case II). Case I is represented using an object of the `pum::NonBranchingSurface` class (which is already defined in the PUM library). Case II is represented by an object of a new class overloading some member functions of the `pum::NonBranchingSurface` class including those related to computing surface quadrature rules and surface normal and tangent vectors. In this case, the member functions for the cell quadrature rules are not overloaded and the same rules as those for case I are used.

To compute the spurious tangential traction forces over the approximated slip plane (case I), two coordinate frames across the slip plane have been defined. The first coordinate system is aligned with the discretized slip plane. This coordinate system is called the (n, s) coordinate system where $k_s = 0.0\,\mathrm{Nm}^{-1}$ is enforced. The other coordinate system is aligned with the real circumference of the circular slip plane and is called the (r, t) coordinate system. In this coordinate system, the actual traction forces (t_r, t_t) can be computed as

$$\begin{bmatrix} t_r \\ t_t \end{bmatrix} = \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} t_n \\ t_s \end{bmatrix}, \tag{6.58}$$

where ϕ is the angle between r and n. The normal and tangential traction forces across the approximated slip plane are given as t_n and t_s. The total spurious tangential traction is then computed as $\int_{\Gamma_d} t_t \, ds$. However, if the exact surface

```
# Continuous and discontinuous function space
elem_cont = FiniteElement("Lagrange", "triangle", 1)
elem_discont = RestrictedElement(elem_cont, dc)

# Enriched function space
element = (elem_cont + elem_discont)*(elem_cont + elem_discont)

# Function space for source term, normal and tangent of surfaces
elem = VectorElement("Discontinuous Lagrange", triangle, 0)

# Define test and trial functions
v, u = TestFunction(element), TrialFunction(element)

# Material properties
mu, lmbda  = Constant("triangle"), Constant("triangle")

# Tangential and normal stiffnesses of slip surface
kt, kn  = Constant("triangle"), Constant("triangle")

# Strain tensor
def epsilon(v):
    return 0.5*(grad(v) + grad(v).T)

# Stress tensor
def sigma(v):
    return 2.0*mu*epsilon(v) \
         + lmbda*tr(epsilon(v))*Identity(v.cell().d)

# Slip surface normal, slip surface normal and source term
normal, tangent = Coefficient(elem), Coefficient(elem)
f  = Coefficient(elem)

# Local jump
def ljump(v):
    return as_vector([inner(jump(v), normal('+')), \
                      inner(jump(v), tangent('+'))])

# Traction force
traction = as_vector([kn('+')*ljump(u)[0], \
                      kt('+')*ljump(u)[1]])

# Bilinear and linear forms
a  = inner(sigma(u), epsilon(v))*dx \
   + inner(traction, ljump(v))*dc
L =  inner(f, v)*dx
```

Figure 6.34 The UFL input to model the circular slip plane using linear Lagrange elements in a two–dimensional problem.

representation for the slip plane (case II) is used, since the exact normal and tangential surface vectors are used, there is no spurious tangential traction force on the slip plane.

```
# Continuous and discontinuous function space
elem_cont = VectorElement("Lagrange", "triangle", 1)
elem_discont = RestrictedElement(elem_cont, dc)
element = elem_cont + elem_discont

# Approximate coefficients
v_pum = Coefficient(element)

# L2 norm
M = inner(v_pum, v_pum)*dx
```

Figure 6.35 The UFL input to compute the L_2 norm using the computed solution.

The computed displacement fields using these two different surface representations are compared with the exact solution presented in Equation (6.57). To obtain a measure to compare the computed displacement fields, a relative error e is defined as

$$e = \frac{|L_2^{\text{exact}} - L_2^{\text{pum}}|}{L_2^{\text{exact}}}, \tag{6.59}$$

where L_2^{exact} is the L_2 norm of the exact displacement field given by Equation (6.57) and L_2^{pum} is the L_2 norm of the approximated displacement field. The L_2^{pum} norm can be evaluated using a functional which is given by a UFL input as presented in Figure 6.35.

Four different mesh discretizations, as shown in Figure 6.36, are used for the comparison between different models.

The relative errors of the computed solutions and the maximum magnitudes for the approximated displacement field are presented in Table 6.2. Note that these results are evaluated for two surface representations with different mesh discretizations. For the approximated slip plane (case I), spurious tangential traction forces have been also computed for different mesh discretizations.

The following remarks can be made about the results presented in this table.

- The slip plane example shows the importance of surface representations in the partition of unity based enrichment methods. From this table, it is clear that surface representation plays an important role in the accuracy of computed solutions. A better approximation can be achieved if a suitable surface representation for the slip plane is chosen even with a coarse mesh.

- The automated framework allows for decoupling the approaches representing surfaces from the generated code and the rest of the library. Therefore, different surface representations can be used without any change in the automatically generated code.

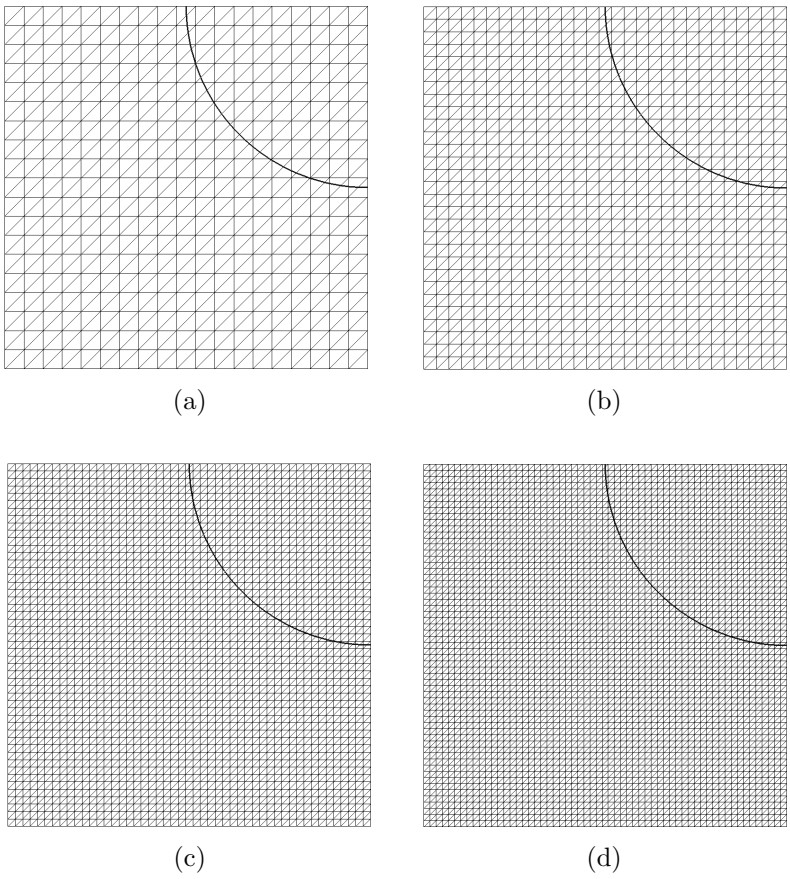


Figure 6.36 Different mesh discretization for the circular slip plane example. A mesh with (a) $2 \times 19 \times 19$ (b) $2 \times 29 \times 29$ (c) $2 \times 49 \times 49$ (d) $2 \times 59 \times 59$ elements.

# dofs	mesh	Case I			Case II	
		traction (N)	e	Δ_{max}(m)	e	Δ_{max}(m)
880	$2 \times 19 \times 19$	954.6	0.3985	0.34494	0.0665	0.47434
1920	$2 \times 29 \times 29$	755.3	0.2073	0.40967	0.0693	0.48074
5200	$2 \times 49 \times 49$	485.2	0.0868	0.45212	0.0563	0.48826
7400	$2 \times 59 \times 59$	428.2	0.0600	0.46240	0.0500	0.49046

Table 6.2 A table for the comparison of the maximum displacement magnitude in m and the relative error for the circular slip plane using the approximated approach (case I) and the exact approach (case II) for the surface representations. The spurious tangential traction forces in N on the slip plane for case I are also presented.

- If a surface is approximated by straight lines (case I), the surface approximation improves significantly with the mesh refinement. For this case the finer the mesh, the better the surface approximation. Although the computed displacement field improves by mesh refinement and the tangential traction forces decrease, the spurious tangential traction forces can not be completely eliminated even with a very fine mesh. Therefore, some unwanted errors always remain in the finite element models.

- However, if the exact surface representation is used (case II), there are no spurious traction forces on the slip plane. Therefore, the computed results are less sensitive with respect to mesh discretization and acceptable approximations are achieved even with a coarse mesh. For case II, the relative error remains almost constant with the mesh refinement.

- The circular slip plane example can be considered as a motivational example to show the importance of using proper surface representations. For the problems with curved surfaces, using better approximations like NURBS or isoparametric surfaces may improve the computed results significantly.

Chapter 7 Conclusions and future works

7.1 Conclusions

An automated framework has been designed to model discontinuities in physical problems independent of computational meshes in the context of partition of unity enrichments. The discontinuity surfaces can be either pre-existing in computational domains as initial geometrical boundaries or developed and evolved during simulations (i.e. crack propagation). The automated framework relies on a form compiler to generate low-level code for modeling discontinuity surfaces and facilitating the use of partition of unity methods for users of computational mechanics technology including engineers and researchers.

In this thesis, the following issues were addressed:

- The partition of unity framework was explained in Chapter 2. A literature review on the application of the partition of unity methods for modeling problems with discontinuities was given. To clarify the implementation aspects of partition of unity methods, a partition of unity formulation for modeling an elastic domain with a discontinuity surface was elaborated. The implementation issues specific to the partition of unity framework were also studied at the end of this chapter.

- The main idea behind the automated modeling of mathematical modeling was addressed in Chapter 3. The design of the FEniCS project and its key components as a framework to automate solving partial differential equations using a compiler approach were explained. This chapter was closed by presenting the modeling of different partial differential equations using components of the FEniCS project.

- The tools and functionalities provided inside the FEniCS project were extended to support automated modelling problems with discontinuities. For the automation of partition of unity models, three key components from the FEniCS project were used.

  - Firstly, the Unified Form Language (UFL) is used to express variational statements (Chapter 4). Particular use is made of the concept of "enriched" spaces and the UFL concept of a "restriction". The latter is

the restriction of functions to a particular entity sub-domain. A notation to represent surface integrals is also defined.

  - To generate code for a finite element assembler and additional helper functions, the FEniCS Form Compiler (Logg et al., 2012b) is extended. The extended compiler, called the PUM compiler (Nikbakht and Wells, 2012c), is explained in Chapter 4. The PUM compiler received high-level inputs resembling the mathematical notations of function spaces and variational formulations defined in the partition of unity framework. The compiler then return low-level C++ codes to evaluate enriched element tensors, to compute local to global mapping, and to manipulate functions defined on the enriched function spaces. The required information related to the discontinuity surfaces is passed via an interface layer defined inside a solver library.
  - Finally, re-usable components for the implementations of the partition of unity methods, including an interface layer to transfer the enriched degrees of freedom data to the generated code, function spaces and surface abstractions, are constructed upon DOLFIN (Logg et al., 2012e). These functionalities are collected inside the PUM solver library (Nikbakht and Wells, 2012b) as addressed in Chapter 5.

- The automated framework has separated discretization of partial differential equations from the partition of unity method implementation details. This separation was achieved by designing the PUM form compiler to generate low-level code from UFL input representing the PDEs and the interface layer which transfered information related to the enriched degrees of freedom to the automatically generated code. This interface layer was implemented inside the `GenericPUM` base class as discussed in Section 5.2.1.

- Surface representations were uncoupled from the rest of the solver library and the automatically generated code. This was performed by introducing `GenericSurface` as a base class to allow easy implementations of different representations. The design of `GenericSurface` was explained in Section 5.2.2.

- The `GenericSurface` and `GenericPUM` base classes can be used to implement different surfaces and various enrichment strategies related to the enriched degrees of freedom. In this thesis, the subclasses derived form base classes were used to model problems whose solutions were discontinuous across surfaces. A simple surface representation which approximates surfaces within cells by minimum order polynomials passing through edge intersection points was implemented as a subclass of `GenericSurface` and it is presented in Section 5.4. The Heaviside function was also used as an enrichment function for

the cells crossed by discontinuities. This enrichment strategy was implemented inside a class derived from `GenericPUM` as discussed in Section 5.3

- Through different examples, presented in Chapter 6, it was shown that the automated framework supports the rapid development of different models for the simulation of discontinuity surfaces in a wide range of linear and non-linear physical problems with various finite element spaces.
  - It has been shown that the automated framework was not limited to conforming Galerkin formulations but could be also used for discontinuous Galerkin formulations as well. This allowed the modelling of problems whose solutions were discontinuous not only on surfaces but also on the internal facets of meshes. In Sections 6.2 and 6.3, examples of automated modeling of discontinuities in the partition of unity method in combination with the discontinuous Galerkin method were presented.
  - A range of Lagrange/non-Lagrange finite element spaces were also supported inside the automated framework for modeling problems with discontinuities. The automated framework was not only limited to the Lagrange family of finite elements, but it also supported $H(\text{div})$ and $H(\text{curl})$ elements like BDM elements and Nédélec elements. Examples of the application of such elements were shown in Sections 6.4 and 6.5.
  - To facilitate the modeling of nonlinear problems with discontinuities, automatic differentiations were used to obtain the Jacobian of functionals (bilinear forms). Examples of such applications were presented in Sections 6.6, 6.7 and 6.8.

7.2 Recommendations for future

- In this thesis, the main focus was on the non-branching discontinuities represented by continuous segments and the Heaviside function used as the enrichment functions. As a future work, the framework can be relatively easily extended to other types of discontinuities and enrichment functions. For example to support branching and multiple discontinuity surfaces, it is just required to add new derived classes from the `GenericSurface` and `GenericPUM` classes, to represent these surfaces and manipulate the enriched degrees of freedom.

- The end points of the discontinuity surfaces, implemented in the PUM solver library, should lie on cell boundaries. Therefore, only the Heaviside function is adequate as an enrichment function in the partition of unity framework. As a

future work, discontinuity surfaces can be generalized such that the end points can lie anywhere in the cells. For these surfaces, other enrichment functions like asymptotic near-tip enrichment functions in addition to the Heaviside function should be considered.

- The automated framework has great potential in developing the partition of unity models for 2D/3D discontinuities in coupled complex multi-physics systems in which different combinations of the continuous/discontinuous spaces can exist. Modelling failure in porous media, flow in fractured reservoirs and hydraulic fracturing are amongst the problems where using the automated framework facilitates modelling considerably.

- Since enriched degrees of freedom are active only for a small subset, the evaluation of the entries corresponding to the enriched degrees of freedom in element tensors are not expensive in comparison to the evaluation of the rest of entries of the element tensors. Nevertheless, the generated code for the enriched entries can also be optimised to improve efficiency and to decrease run-time and compile-time. One approach to optimise the generated code for the problems with discontinuous solutions is flipping the sign for the Heaviside function. This causes only element tensors for the cells that are intersected by discontinuity surfaces to have enriched entries.

- To improve the performance of the automated framework in modeling complex problems, it is required to be able to parallelise the partition of unity models. As a future work, the PUM library should be adapted to handle computing enriched degrees of freedom in parallel for different subdomains.

References

M. S. Alnæs. UFL: a finite element form langauge. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 17. Springer-Verlag, 2012. 22, 24, 26, 53, 56

M. S. Alnæs and A. Logg. *UFL Specification and User Manual*, 2009. URL: `http://launchpad.net/ufl/`. 25, 26, 53

M. S. Alnæs and A. Logg. Unified form language: UFL. `http://launchpad.net/ufl/`, 2012. 22, 49

M. S. Alnæs and K. A. Mardal. SyFi and SFC: symbolic finite elements and form compilation. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 17. Springer-Verlag, 2012. 26

M. S. Alnæs and K. A. Mardal. SyFi. URL: `http://launchpad.net/syfi/`, 2012. 22

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*, 4(4):231–244, 2009. 29

M. S. Alnæs, H. P. Langtangen, A. Logg, K. A. Mardal, and O. Skavhaug. Unified Form-assembly Code: UFC. URL: `http://launchpad.net/ufc/`, 2012. 22, 26, 28

M. S. Alnæs, A. Logg, and K. A. Mardal. UFC: a finite element code generation interface. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 16. Springer-Verlag, 2012. 29, 49, 57, 148

J. Andersen and V. Solodukhov. Field behavior near a dielectric wedge. *Antennas and Propagation, IEEE Transactions*, 26(4):598–602, 1978. 98

P. Areias and T. Belytschko. Analysis of three-dimensional crack initiation and propagation using the extended finite element method. *International Journal for Numerical Methods in Engineering*, 63(5):760–788, 2005. 8, 18, 76

F. Armero and C. Callari. An analysis of strong discontinuities in a saturated poro-plastic solid. *International Journal for Numerical Methods in Engineering*, 46(10):1673–1698, 1999. 113

D. N. Arnold, F. Brezzi, and J. Douglas, Jr. PEERS: a new mixed finite element for plane elasticity. *Japan J. Appl. Math.*, 1(2):347–367, 1984a. 52

D. N. Arnold, F. Brezzi, and M. Fortin. A stable finite element for the Stokes equations. *Calcolo*, 21(4):337–344, 1984b. 52

K. Aziz and A. Settari. *Petroleum Reservoir Simulation*. Applied Science Publishers Ltd., London, 1979. 9

I. Babuška and J. M. Melenk. The partition of unity finite element method: Basic theory and applications. *Computer Methods in Applied Mechanics and Engineering*, 139(1-4):289–314, 1996. 3

I. Babuška and J. M. Melenk. The partition of unity method. *International Journal for Numerical Methods in Engineering*, 40(4):727–758, 1997. 3

I. Babuška, G. Caloz, and J. E. Osborn. Special finite element methods for a class of second order elliptic problems with rough coefficients. *SIAM J. Numer. Anal.*, 31(4):945–981, 1994. 3

B. Bagheri and L. R. Scott. Analysa. URL: `http://people.cs.uchicago.edu/~ridg/al/aa.html`, 2003. 21

S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. Mcinnes, B. F. Smith, and H. Zhang. PETSc web page. `http://www.mcs.anl.gov/petsc/`, 2012. 31

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24–50, 2007. 30

Z. P. Bažant. Instability, ductility, and size effects in strain-softening concrete. *ASCE Journal of Engineering Mechanics*, 102(2):331–344, 1976. 1

D. Beazley et al. Simplified wrapper and interface generator (SWIG). `http://www.swig.org/`, 2012. 31

T. Belytschko and T. Black. Elastic crack growth in finite elements with minimal remeshing. *International Journal for Numerical Methods in Engineering*, 45(5): 601–620, 1999. 1, 3, 7, 8, 17

T. Belytschko, J. Fish, and B. E. Engelmann. A finite element with embedded localization zones. *Computer Methods in Applied Mechanics and Engineering*, 70 (1):59 – 89, 1988. 1, 2

T. Belytschko, N. Moës, S. Usui, and C. Parimi. Arbitrary discontinuities in finite elements. *International Journal for Numerical Methods in Engineering*, 50(4): 993–1013, 2001. 19

T. Belytschko, R. Gracie, and G. Ventura. A review of extended/generalized finite element methods for material modeling. *Modelling and Simulation in Materials Science and Engineering*, 17(4), 2009. 7

K. Y. Billah and R. H. Scanlan. Resonance, tacoma narrows bridge failure, and undergraduate physics textbooks. *American Journal of Physics*, 59(2):118–124, 1991. 11

T. Bittencourt, A. R. Ingraffea, and J. Llorca. Simulation of arbitrary, cohesive crack propagation. *Fracture Mechanics of Concrete Structures*, pages 339–350, 1992. 1

S. Bordas, P. V. Nguyen, C. Dunant, A. Guidoum, and H. Nguyen-Dang. An extended finite element library. *International Journal for Numerical Methods in Engineering*, 71(6):703–732, 2007. 4

A. Bossavit. Simplicial finite elements for scattering problems in electromagnetism. *Comput. Methods Appl. Mech. and Eng.*, 76:299–316, 1989. 99

F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*. Springer-Verlag, 1991. 46

F. Brezzi, J. Douglas, Jr., and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numer. Math.*, 47(2):217–235, 1985. 27, 97

H. J. Bungartz and M. Schäfer, editors. *Fluid-Structure Interaction: Modelling, Simulation, Optimisation*. Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin, 2006. 10

G. T. Camacho and M. Ortiz. Computational modelling of impact damage in brittle materials. *International Journal of Solids and Structures*, 33(20-22):2899–2938, 1996. 1

B. J. Carter, C. S. Chen, A. R. Ingraffea, and P. A. Wawrzynek. A topology-based system for modeling 3d crack growth in solid and shell structures. In *Proceedings of the Ninth International Congress on Fracture ICF9*, pages 1923–1934. Elsevier Science Publishers, Sydney, Australia, 1997. 1

R. Chamrová and B. Patzák. Object-oriented programming and the extended finite-element method. *Proceedings of the Institution of Civil Engineers: Engineering and Computational Mechanics*, 163(4):271–278, 2010. 4

J. Chessa and T. Belytschko. An enriched finite element method and level sets for axisymmetric two-phase flow with surface tension. *International Journal for Numerical Methods in Engineering*, 58(13):2041–2064, 2003a. 10

J. Chessa and T. Belytschko. An extended finite element method for two-phase fluids. *Journal of Applied Mechanics, Transactions ASME*, 70(1):10–17, 2003b. 10

N. Christofides. *Graph theory: an algorithmic approach*. Academic Press, 1975. 56

COMSOL. COMSOL multiphysics, 2012. URL `http://www.comsol.com`. 21

R. D. Cook, D. S. Malkus, M. E. Plesha, and R. J. Witt. *Concepts and Applications of Finite Element Analysis*. John Wiley and Sons, New York, 2002. 1, 21

M. Crouzeix and P. A. Raviart. Conforming and non-conforming finite element methods for solving the stationary Stokes equations. *R. A. I. R. O. Anal. Numer.*, 7:33–76, 1973. 27, 46

C. Daux, N. Moës, J. Dolbow, N. Sukumar, and T. Belytschko. Arbitrary branched and intersecting cracks with the extended finite element method. *International Journal for Numerical Methods in Engineering*, 48(12):1741–1760, 2000. 8

R. de Borst, L. J. Sluys, H. B. Mühlhaus, and J. Pamin. Fundamental issues in finite element analyses of localisation of deformation. *Engineering Computations*, 10(2):99–121, 1993. 1

J. Dolbow, N. Moës, and T. Belytschko. An extended finite element method for modeling crack growth with frictional contact. *Computer Methods in Applied Mechanics and Engineering*, 190(51-52):6825–6846, 2001. 8

C. A. Duarte, O. N. Hamzeh, T. J. Liszka, and W. W. Tworzydlo. A generalized finite element method for the simulation of three-dimensional dynamic crack propagation. *Computer Methods in Applied Mechanics and Engineering*, 190: 2227–2262, 2001. 8, 17

C. A. Duarte, L. G. Reno, and A. Simone. A high-order generalized fem for through-the-thickness branched cracks. *International Journal for Numerical Methods in Engineering*, 72(3):325–351, 2007. 13

M. Duflot. A study of the representation of cracks with level sets. *International Journal for Numerical Methods in Engineering*, 70(11):1261–1302, 2007. 20

P. Dular and C. Geuzaine. GetDP: a general environment for the treatment of discrete problems. URL: `http://www.geuz.org/getdp/`, 2012. 21

E. N. Dvorkin, A. M. Cuitino, and G. Gioia. Finite elements with displacement interpolated embedded localization lines insensitive to mesh size and distortions. *International Journal for Numerical Methods in Engineering*, 30(3):541–564, 1990. ISSN 1097-0207. 2

G. Engel, K. Garikipati, T. J. R. Hughes, M. G. Larson, and R. L. Taylor. Continuous/discontinuous finite element approximations of fourth-order elliptic problems in structural and continuum mechanics with applications to thin beams and plates, and strain gradient elasticity. *Computer Methods in Applied Mechanics and Engineering*, 191(34):3669–3750, 2002. 93

R. Eymard, T. R. Gallouët, and R. Herbin. The finite volume method. In P.G. Ciarlet and J.L. Lions, editors, *Handbook of Numerical Analysis*, volume 7, pages 713–1018. Elsevier, 2000. 21

T. P. Fries. The intrinsic xfem for two-fluid flows. *International Journal for Numerical Methods in Fluids*, 60:437–471, 2008. 10

T. P. Fries and T. Belytschko. The extended/generalized finite element method: An overview of the method and its application. *International Journal for Numerical Methods in Engineering*, 2010. Article in press. 7

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995. 56

T. C. Gasser and G. A. Holzapfel. Modeling 3D crack propagation in unreinforced concrete using pufem. *Computer Methods in Applied Mechanics and Engineering*, 194(25-26):2859–2896, 2005. 8, 18, 70, 76

T.C. Gasser and G.A. Holzapfel. 3d crack propagation in unreinforced concrete. a two-step algorithm for tracking 3d crack paths. *Computer Methods in Applied Mechanics and Engineering*, 195:5198–5219, 2006. 19, 148

A. Gerstenberger and W. A. Wall. Enhancement of fixed-grid methods towards complex fluid-structure interaction applications. *International Journal for Numerical Methods in Fluids*, 57(9):1227–1248, 2008a. 11

A. Gerstenberger and W. A. Wall. An extended finite element method/lagrange multiplier based approach for fluid-structure interaction. *Computer Methods in Applied Mechanics and Engineering*, 197(19-20):1699–1714, 2008b. 11

E. Giner, N. Sukumar, J. E. Tarancòn, and F. J. Fuenmayor. An Abaqus implementation of the extended finite element method. *Engineering Fracture Mechanics*, 76(3):347–368, 2009. 4

P. Gottschling and A. Lumsdaine. The Matrix Template Library 4 Web page. `http://www.simunova.com/en/node/24`, 2011. 31

A. Gravouil, N. Moës, and T. Belytschko. Non-planar 3d crack growth by the extended finite element and level sets part ii: Level set update. *Int.J.Numer.Meth.Eng.*, 53(11):2569–2586, 2002. 20

S. Groß and A. Reusken. An extended pressure finite element space for two-phase incompressible flows with surface tension. *Journal of Computational Physics*, 224 (1):40–58, 2007. 10

R. Helmig. *Multiphase Flow and Transport in the Subsurface*. Springer-Verlag, Berlin, 1997. 10

M. A. Herox et al. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005. 31

D. J. Holdych, D. R. Noble, and R. B. Secor. Quadrature rules for triangular and tetrahedral elements with generalized functions. *International Journal for Numerical Methods in Engineering*, 73(9):1310–1327, 2008. 17

T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover, New York, 2000. 1, 21

A. R. Ingraffea and V. Saouma. Numerical modelling of discrete crack propagation in reinforced and plain concrete. In *Fracture Mechanics of Concrete*, pages 171–225. Martinus Nijhoff Publishers, Dordrecht, 1985. 1

P. Jäger, P. Steinmann, and E. Kuhl. Modeling three-dimensional crack propagation - a comparison of crack path tracking strategies. *International Journal for Numerical Methods in Engineering*, 76(9):1328–1352, 2008a. 18

P. Jäger, P. Steinmann, and E. Kuhl. On local tracking algorithms for the simulation of three-dimensional discontinuities. *Computational Mechanics*, 42: 395–406, 2008b. 18, 148

J. Jin. *The Finite Element Method in Electromagnetics*. John Wiley and Sons, Inc., New York, 2nd edition, 2002. ISBN 0471438189. 99

M. Jiràsek. Comparative study on finite elements with embedded discontinuities. *Computer methods in applied mechanics and engineering*, 188:307–330, 2000a. 2

M. Jiràsek. Conditions of uniqueness for finite elements with embedded cracks. In *Proceedings of the Sixth International Conference on Computational Plasticity*. Barcelona, Spain, 2000b. 2

M. Jiràsek and T. Belytschko. Computational resolution of strong discontinuities. In *Proceeding of Fifth World Congress on Computational Mechanics*. Vienna, Austria, 2002. 2

A. R. Khoei and M. Nikbakht. Contact friction modeling with the extended finite element method (X-FEM). *Journal of Materials Processing Technology*, 177(1-3): 58–62, 2006. 9

A. R. Khoei and M. Nikbakht. An enriched finite element algorithm for numerical computation of contact friction problems. *International Journal of Mechanical Sciences*, 49(2):183–199, 2007. 9

R. C. Kirby. FIAT: A new paradigm for computing finite element basis functions. *ACM Trans. Math. Software*, 30:502–516, 2004. 21, 32

R. C. Kirby. Optimizing FIAT with level 3 BLAS. *ACM Trans. Math. Softw.*, 32 (2):223–235, 2006. 21, 32, 56

R. C. Kirby. FIAT: numerical construction of finite element basis functions. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 13. Springer-Verlag, 2012a. 32, 56

R. C. Kirby. FErari, 2012b. URL: `http://launchpad.net/ferari/`. 32

R. C. Kirby. FIAT, 2012c. URL: `http://launchpad.net/fiat/`. 26, 31

R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006. 4, 22, 26, 27

R. C. Kirby and A. Logg. Efficient compilation of a class of variational forms. *ACM Transactions on Mathematical Software*, 33(3), 2007. 4, 22, 27, 51

R. C. Kirby and A. Logg. Benchmarking domain-specific compiler optimizations for variational forms. *ACM Transactions on Mathematical Software*, 35(2):1–18, 2008. 27, 51

R. C. Kirby and A. Logg. FErari: an optimizing compiler for variational forms. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 12. Springer-Verlag, 2012. 32

R. C. Kirby and L. R. Scott. Geometric optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 29:827–841, 2007. 32

R. C. Kirby, M. G. Knepley, A. Logg, and L. R. Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(6):741–758, 2005. 32

M. Klisinski, K. Runesson, and S. Sture. Finite element with inner softening band. *Journal of Engineering Mechanics*, 117(3):575–587, 1991. 2

A. Kuprat, D. George, G. Straub, and M. C. Demirel. Modeling microstructure evolution in three dimensions with grain3d and lagrit. *Computational Materials Science*, 28:199–208, 2003. 9

H. P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, 2003. 30

J. Larsson and R. Larsson. Finite-element analysis of localization of deformation and fluid pressure in an elastoplastic porous medium. *International Journal of Solids and Structures*, 37(48-50):7231 – 7257, 2000. 113

R. Larsson, K. Runesson, and S. Sture. Embedded localization band in undrained soil based on regularized strong discontinuity - theory and fe-analysis. *International Journal of Solids and Structures*, 33(20-22):3081–3101, 1996. 2, 7

A. Legay, J. Chessa, and T. Belytschko. An eulerian-lagrangian method for fluid-structure interaction based on level sets. *Computer Methods in Applied Mechanics and Engineering*, 195(17-18):2070–2087, 2006. 11

R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge University Press, UK, 2002. 21

S. Levy. *Two-Phase Flow in Complex Systems*. John Wiley and Sons, Inc., 1995. 10

F. Liu and R. I. Borja. Stabilized low-order finite elements for frictional contact with the extended finite element method. *Computer Methods in Applied Mechanics and Engineering*, 199(37-40):2456–2471, 2010a. 9

F. Liu and R. I. Borja. Finite deformation formulation for embedded frictional crack with the extended finite element method. *International Journal for Numerical Methods in Engineering*, 82(6):773–804, 2010b. 9

A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283–295, 2009. 31

A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010. 22, 27, 31

A. Logg, H. P. Langtangen, and X. Cai. Past and future perspectives on scientific software. In Aslak Tveito, Are Magnus Bruaset, and Olav Lysne, editors, *Simula Research Laboratory - by thinking constantly about it*, chapter 23, pages 321–362. Springer-Verlag, 2009. 23, 24, 148

A. Logg, K. A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*. Springer-Verlag, 2012a. 4

A. Logg, K. B. Ølgaard, M. Rognes, et al. FEniCS Form Compiler: FFC. `http://launchpad.net/ffc/`, 2012b. 6, 22, 26, 49, 50, 54, 130

A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. FFC: the FEniCS form compiler. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 17. Springer-Verlag, 2012c. 26

A. Logg, G. N. Wells, and J. Hake. DOLFIN: A C++/Python finite element library. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 10. Springer-Verlag, 2012d. 30, 31, 148

A. Logg, G. N. Wells, et al. Dynamic Object-oriented Library in FINite element method: DOLFIN. URL: `http://launchpad.net/dolfin/`, 2012e. 22, 63, 130

A. Logg, G. N. Wells, et al. FEniCS project. `http://feincsproject.org`, 2012f. 4, 5, 21

K. Long et al. Sundance. URL: `http://www.math.ttu.edu/~klong/Sundance/html/`, 2012. 21

K. R. Long, R. C. Kirby, and B. G. van Bloemen Waanders. Unified embedded parallel finite element computations via software-based frechet differentiation. *SIAM Journal on Scientific Computing*, 32(6):3323–3351, 2010. 21

H. R. Lotfi and P. B. Shing. Embedded representation of fracture in concrete with mixed finite elements. *International Journal for Numerical Methods in Engineering*, 38(8):1307–1325, 1995. 2, 7

U. M. Mayer, A. Gerstenberger, and W. A. Wall. Interface handling for three-dimensional higher-order xfem-computations in fluid-structure interaction. *International Journal for Numerical Methods in Engineering*, 79(7):846–869, 2009. 11

U. M. Mayer, A. Popp, A. Gerstenberger, and W. A. Wall. 3D fluid-structure-contact interaction based on a combined xfem fsi and dual mortar contact approach. *Computational Mechanics*, 46(1):53–67, 2010. 11

A. R. Mitchell and D. F. Griffiths. *The finite Difference Methods in Partial Differential Equations*. John Willy, NewYork, 1979. 21

N. Moës and T. Belytschko. Extended finite element method for cohesive crack growth. *Engineering Fracture Mechanics*, 69(7):813–833, 2002. 8

N. Moës, J. Dolbow, and T. Belytschko. A finite element method for crack growth without remeshing. *International Journal for Numerical Methods in Engineering*, 46(1):131–150, 1999. 3, 7, 8, 17

N. Moës, A. Gravouil, and T. Belytschko. Non-planar 3d crack growth by the extended finite element and level sets part i: mechanical model. *Int.J.Numer.Meth.Eng.*, 53(11):2549–2568, 2002. 20

M. Moumnassi, S. Belouettar, E. Béchet, S. P. A. Bordas, D. Quoirin, and M. Potier-Ferry. Finite element analysis on implicitly defined domains: An accurate representation based on arbitrary parametric surfaces. *Computer Methods in Applied Mechanics and Engineering*, 200(5-8):774 – 796, 2011. 20

J. C. Nédélec. Mixed finite elements in $\mathbb{R}^3$. *Numer. Math.*, 35(3):315–341, 1980. 27, 101

T. D. Nguyen. *Discontinuous Galerkin formulations for thin bending problems*. PhD thesis, Delft university of technology, The Netherlands, 2008. 38

M. Nikbakht and G. N. Wells. Automated modelling of evolving discontinuities. *Algorithms*, 2(3):1008–1030, 2009. 49, 59, 82, 83

M. Nikbakht and G. N. Wells. Modelling evolving discontinuities. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 30. Springer-Verlag, 2012a. 83

M. Nikbakht and G. N. Wells. Partition of Unity Solver Library. `http://www.launchpad.net/dolfin-pum`, 2012b. 130

M. Nikbakht and G. N. Wells. Partition of Unity Form Compiler. `http://www.launchpad.net/ffc-pum`, 2012c. 130

J. T. Oden. *Applied Functional Analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1979. 21

K. B. Ølgaard and G. N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software*, 37(1), 2010. 28, 51, 56, 57

K. B. Ølgaard and G. N. Wells. Quadrature representation of finite element variational forms. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 7. Springer-Verlag, 2012a. 28, 51

K. B. Ølgaard and G. N. Wells. Applications in solid mechanics. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 26. Springer-Verlag, 2012b. 39

K. B. Ølgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous Galerkin methods. *SIAM Journal on Scientific Computing*, 31(2): 849–864, 2008. 36, 94

J. Oliver. Modelling strong discontinuities in solid mechanics via strain softening constitutive equations. part 1: Fundamentals. part 2: Numerical simulation. *International Journal for Numerical Methods in Engineering*, 39:3575–3624, 1996. 2, 7

M. Ortiz, Y. Leroy, and A. Needleman. A finite element method for localized failure analysis. *Computer Methods in Applied Mechanics and Engineering*, 61 (2):189–214, 1987. 1, 2

S. Osher and R. P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2003. 18

L. Piegl and W. Tiller. *The NURBS Book (2nd ed)*, volume XIV of *Monographs in Visual Communication*. Springer, 1997. 20

O. Pironneau, F. Hecht, A. L. Hyaric, and K. Ohtsuka. FreeFEM. URL: `http://www.freefem.org/`, 2010. 21

H. Prautzsch, W. Boehm, and M. Paluszny. *Bezier and B-Spline Techniques*, volume XIV of *Mathematics and Visualization*. Springer, 2002. 20

C. Prudhomme. Life: Overview of a unified C++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In *In Proceedings of the International Conference on Applied Parallel Computing. Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, Germany, 2007. 21

B. M. A. Rahman and J. B. Davies. Penalty function improvement of waveguide solution by finite elements. *IEEE Trans. Microw. Theory and Techni.*, MTT-32 (8):922–928, 1984. 98

P. A. Raviart and J. M. Thomas. A mixed finite element method for 2nd order elliptic problems. pages 292–315. Lecture Notes in Math., Vol. 606, 1977. 27

J. N. Reddy. *Applied Functional Analysis and Variational Methods in Engineering*. Krieger Pub Co, Malabar, Florida, 1991. 21

J. J. C. Remmers, R. de Borst, and A. Needleman. A cohesive segments method for the simulation of crack growth. *Computational Mechanics*, 31(1-2 SPEC.):69–77, 2003. 8

A. Reusken. Analysis of an extended pressure finite element space for two-phase incompressible flows. *Computing and Visualization in Science*, 11(4-6):293–305, 2008. 10

M. E. Rognes, R. C. Kirby, and A. Logg. Efficient assembly of H(div) and H(curl) conforming finite elements. *SIAM Journal on Scientific Computing*, 36 (6):4130–4151, 2009. 96, 101

S. Ross. *Tacoma Narrows 1940.* MCGrow Hill, 1984. 11

W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics*. Kitware, Inc. publishers, New York, 4th edition, 2006. 65

J. A. Sethian. *Level Set Methods and Fast Marching Methods (2nd edn)*. Cambridge University Press, 1999. 18

Y. Shapira. *Solving PDEs in C++: Numerical Methods in a Unified Object-Oriented Approach*. SIAM, Society for Industrial and Applied Mathematics, 2006. 63

J. C. Simo and J. Oliver. A new approach to the analysis and simulation of strain softening in solids. In Z. P. Bažant, Z. Bittnar, M. Jiràsek, and J. Mazars, editors, *Proceeding of Fracture and Damage in Quasibrittle Structures, E and FN Spon*. London, UK, 1994. 2

J. C. Simo, J. Oliver, and F. Armero. An analysis of strong discontinuities induced by strain-softening in rate-independent inelastic solids. *Computational Mechanics*, 12(5):277–296, 1993. 2, 7

A. Simone, C. A. Duarte, and E. Van der Giessen. A generalized finite element method for polycristals with discontinuous grain boundaries. *International Journal for Numerical Methods in Engineering*, 67(8):1122–1145, 2006. 9, 10

O. Skavhaug. Viper. `http://launchpad.net/fenics_viper/`, 2012. 32

L. J. Sluys and A. H. Berends. Discontinuous failure analysis for mode-i and mode-ii localization problems. *International Journal of Solids and Structures*, 35(31-32): 4257 – 4274, 1998. 2

G. D. Smith. *Numerical solution of partial differential equations: finite difference methods, Third edition*. Oxford University Press, UK, 1985. 21

G. S. Smith. *An Introduction to Classical Electromagnetic Radiation*. Cambridge University Press, 1997. 99

M. Stolarska, D. L. Chopp, N. Moës, and T. Belytschko. Modelling crack growth by level sets in the extended finite element method. *International Journal for Numerical Methods in Engineering*, 51:943–960, 2001. 19, 20

T. Strouboulis, I. Babuška, and K. Copps. The design and analysis of generalized finite element method. *Computer Methods in in Applied Mechanics and Engineering*, 181:43–69, 2000a. 16

T. Strouboulis, K. Copps, and I. Babuška. The generalized finite element method: An example of its implementation and illustration of its performance. *International Journal for Numerical Methods in Engineering*, 47(8):1401–1417, 2000b. 4, 7

T. Strouboulis, K. Copps, and I. Babuška. Computational mechanics advances. the generalized finite element method. *Computer Methods in Applied Mechanics and Engineering*, 190(32-33):4081–4193, 2001. 4, 7

N. Sukumar, N. Moës, B. Moran, and T. Belytschko. Extended finite element method for three-dimensional crack modelling. *International Journal for Numerical Methods in Engineering*, 48(11):1549–1570, 2000. 8, 17

N. Sukumar, D. J. Srolovitz, T. J. Baker, and J. Prévost. Brittle fracture in polycrystalline microstructures with the extended finite element method. *International Journal for Numerical Methods in Engineering*, 56(14):2015–2037, 2003. 9

D. V. Swenson and A. R. Ingraffea. Modeling mixed-mode dynamic crack propagation using finite elements: Theory and applications. *Computational Mechanics*, 3(5):381–397, 1988. 1

C. Taylor and P. Hood. A numerical solution of the Navier-Stokes equations using the finite element technique. *Internat. J. Comput. and Fluids*, 1(1):73–100, 1973. ISSN 0045-7930. 46, 113

A. R. Terrel, L. R. Scott, M. G. Knepley, R. C. Kirby, and G. N. Wells. Finite elements for incompressible fluids. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 17. Springer-Verlag, 2012. 47

M. G. A. Tijssens, L. J. Sluys, and E. Van der Giessen. Numerical simulation of quasi-brittle fracture using damaging cohesive surfaces. *European Journal of Mechanics, A/Solids*, 19(5):761–779, 2000a. 1

M. G. A. Tijssens, E. Van Der Giessen, and L. J. Sluys. Modeling of crazing using a cohesive surface methodology. *Mechanics of Materials*, 32(1):19–35, 2000b. 1

G. Ventura. On the elimination of quadrature subcells for discontinuous functions in the extended finite-element method. *International Journal for Numerical Methods in Engineering*, 66(5):761–795, 2006. 17

E. Vitali and D. Benson. Kinetic friction for multi-material arbitrary lagrangian eulerian extended finite element formulations. *Computational Mechanics*, 43(6): 847–857, 2009. 9

E. Vitali and D. J. Benson. An extended finite element formulation for contact in multi-material arbitrary lagrangian-eulerian calculations. *International Journal for Numerical Methods in Engineering*, 67(10):1420–1444, 2006. 9

J. Walter et al. uBLAS web page. `http://www.boost.org/`, 2012. 31

H. Wang, J. Chessa, W. K. Liu, and T. Belytschko. The immersed/fictitious element method for fluid-structure interaction: Volumetric consistency, compressibility and thin members. *International Journal for Numerical Methods in Engineering*, 74 (1):32–55, 2008. 11

X. Wang. *Fundamentals of Fluid-Solid Interactions: Analytical and Computational Approaches*. Monograph Series on Nonlinear Science and Complexity. Elsevior B.V., Amsterdam, 2008. 10

G. N. Wells and L. J. Sluys. A new method for modelling cohesive cracks using finite elements. *International Journal for Numerical Methods in Engineering*, 50 (12):2667–2682, 2001a. 8, 76

G. N. Wells and L. J. Sluys. Three-dimensional embedded discontinuity model for brittle fracture. *International Journal of Solids and Structures*, 38(5):897–913, 2001b. 2, 7

M. Westlie, K. A. Mardal, and M. S. Alnæs. Instant: Inlining of C/C++ in Python, 2012. URL: `http://launchpad.net/instant`. 32

S. Weyer, A. Fröhlich, H. Riesch-Oppermann, L. Cizelj, and M. Kovac. Automatic finite element meshing of planar voronoi tessellations. *Engineering Fracture Mechanics*, 69:945–958, 2002. 9

I. M. Wilbers, K. A. Mardal, and M. S. Alnæs. Instant: just-in-time compilation of C/C++ in Python. In A. Logg, K. A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, chapter 14. Springer-Verlag, 2012. 32

P. Wriggers. *Computational Contact Mechanics*. Springer-Verlag, Berlin, 2nd edition, 2006. 8

A. Yazid, N. Abdelkader, and H. Abdelmadjid. A state-of-the-art review of the x-fem for computational fracture mechanics. *Applied Mathematical Modelling*, 33(12): 4269–4282, 2009. 7

G. Zi and T. Belytschko. New crack-tip elements for xfem and applications to cohesive cracks. *International Journal for Numerical Methods in Engineering*, 57 (15):2221–2240, 2003. 8

O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method: Sixth Edition*. Elsevier, Burlington, MA, 2005. 1, 21

List of Figures

List of Tables

Summary

Although computers were invented to automate tedious and error-prone tasks, computer programming is a tedious and error-prone task itself. This is a well-known paradox in the field of computational mathematical modelling. Recently, automatic code generation has been proposed to solve this paradox. In this approach, a required code to model physical problems is generated by compiling an input file which mimics mathematical notations.

In this thesis, the automatic code generation has been extended to support developing models for problems with discontinuities. Examples of this kind of problems in real world are cracks, slip planes and singularities in materials as well as phase interfaces in multiphase flows.

This framework is designed in the context of the FEniCS project, an open source project in the Automation of Computational Mathematical Modelling (ACMM). The automated framework has been implemented in two packages which are licensed as open source software and they can be downloaded for free.

- A compiler (in Python) for generating C++ low-level code to model discontinuities from the high-level code close to mathematical notations `https://launchpad.net/ffc-pum`

- A solver library (in C++) to use the generated code from the PUM compiler in combination with other reusable components of DOLFIN to model discontinuities `https://launchpad.net/dolfin-pum`

This framework provides required tools and functionalities to fast and efficient development of the partition of unity models for physical problems represented by partial differential equations in domains with discontinuity surfaces. Developing such models especially for coupled problems, in which different combinations of continuous/discontinuous spaces may exist, is a time consuming and difficult task. Using the automated framework moves the focus from implementation to modeling. Therefore, different models can be simulated and tested quickly with minimum reworks.

The examples, presented in this thesis, were limited to non-branching discontinuity surfaces with the Heaviside enrichment function. A novel algorithm is also proposed to keep track of three-dimensional propagating surfaces. However, the solver library

is designed such that it can be relatively easily extended to support other types of problems.

Samenvatting

Hoewel computers zijn uitgevonden om lastige, foutgevoelige taken te automatiseren, is computer programmeren zelf een lastige en foutgevoelige onderneming. Dit is een bekende paradox in het domein van de numerieke modellering. Automatische code-generatie is recent voorgesteld als uitweg uit deze paradox. Met deze aanpak wordt een model voor het oplossen van fysische problemen gegenereerd door een invoerbestand in abstracte wiskundige notatie automatisch te compileren.

In dit proefschrift is de automatische code-generatie uitgebreid voor het ontwikkelen van modellen voor problemen met discontinuiteiten. Voorbeelden hiervan in de praktijk zijn scheuren, slip-vlakken en singulariteiten in materialen en eveneens fase-grenzen in meerfase stromingen.

Dit raamwerk is ontworpen in de context van het FEniCS project, een open source project in de *Automation of Computational Mathematical Modeling* (ACMM). Het geautomatiseerde raamwerk is geïmplementeerd in twee pakketten die als open source software zijn gelicenseerd en die vrij te downloaden zijn.

- Een compiler (in Python) die laag-niveau code voor het modelleren van discontinuiteiten genereert vanaf een hoog-niveau code die bij benadering gelijk is aan wiskundige notatie `https://launchpad.net/ffc-pum`
- Een solver library (in C++) om de gegenereerde code in combinatie met andere herbruikbare componenten te kunnen gebruiken voor het modelleren van discontinuiteiten `https://lanchpad.net/dolfin-pum`

Dit raamwerk levert de nodige tools en functionaliteiten voor het snel en efficiënt ontwikkelen van *partition of unity* modellen voor fysische problemen die beschreven kunnen worden met partiele differentiaalvergelijkingen in domeinen met discontinuiteiten. Het ontwikkelen van zulke modellen is een veeleisende en moeilijke taak, in het bijzonder voor gekoppelde problemen waar verschillende combinaties van continue en discontinue ruimtes kunnen voorkomen. Het geautomatiseerde raamwerk verschuift de inspanning van de implementatie naar het modelleren. Daarom kunnen verschillende modellen snel doorgerekend en getest worden met een minimum aan dubbel werk.

De voorbeelden die in dit proefschrift worden gepresenteerd beperken zich tot niet-vertakkende discontinuiteiten met een Heaviside *enrichment*-functie. Er wordt ook een nieuw algoritme voorgesteld waarmee de groei van oppervlakken in drie

dimensies bijgehouden kan worden. De solver library is zodanig ontworpen dat deze relatief eenvoudig uit te breiden is om ook andersoortige problemen op te lossen.

Propositions

1. Although computers were invented to automate tedious and error-prone tasks, computer programming itself is a tedious and error-prone task. This paradox in the field of computational mathematical modelling can be overcome largely by automatic code generation.

2. Utilizing code of others helps software developers to avoid spending their time on developing software which has already been developed. Why reinventing a wheel which already exists?

3. Object oriented programming languages provide the required equipment to design well organized, easily expandable finite element software packages.

4. The structure of computational mathematical modelling software must follow the same structure and abstractions used for mathematics.

5. Learning programming languages is a steady and trial and error process. One cannot be a good programmer by just reading books – experience has no shortcut.

6. Implementing mathematical models takes a lot of time of PhD students in computational mechanics subjects. Automation can help them to rapidly develop finite element models by shifting the focus from implementation to modelling.

7. Behind any successful computational model lies strong mathematics.

8. "Human being are members of a whole,
   In creation of one essence and soul.
   If one member is afflicted with pain,
   Other members uneasy will remain.
   If you've no sympathy for human pain,
   The name of human you cannot retain!" – Sa'adi; Persian poet, thinker, and philosopher, 13th century.

9. "*Free software* is a matter of liberty, not price. To understand the concept, you should think of *free* as in *free speech*, not as in *free beer*." – Richard M. Stallman, the founder of GNU.

10. Life has its own governing partial differential equation (PDE) with its boundary conditions which may change in time. The more you experience, the better you know your own PDE and more accurate you can predict the boundary conditions. .

11. To stay motivated during a PhD-project, it is important to divide the project into small sub projects which can help to measure progress in each stage.

12. Although communication tools have been progressed considerably in recent years (e.g. emails and phones), none of them can replace direct and face-to-face meetings.

The propositions are regarded as opposable and defendable, and have been approved as such by the supervisors, Prof. dr. ir. L. J. Sluys and Dr. G. N. Wells.

Stellingen

1. Hoewel computers zijn uitgevonden om lastige, foutgevoelige taken te automatiseren, is computer programmeren zelf een lastige en foutgevoelige onderneming. De oplossing voor deze paradox van de numerieke modellering kan voor een groot deel gevonden worden in automatische code-generatie.

2. Door code van anderen te gebruiken kunnen software-ontwikkelaars hun tijd besteden aan het ontwikkelen van programmatuur die reeds ontwikkeld is. Waarom het wiel opnieuw uitvinden?

3. Objectgeoriënteerde programmeertalen leveren het noodzakelijke gereedschap om goed gestructureede, eenvoudig uit te breiden eindige-elementenprogramma's te ontwerpen.

4. De structuur van numerieke modelleringssoftware behoort de structuur en abstracties van de onderliggende wiskunde te volgen.

5. Het leren van programmeertalen is een gestaag proces van vallen en opstaan. Iemand wordt nooit een goed programmeur door het lezen van boeken – er is geen kortere weg dan die van de ervaring.

6. Veel tijd van promovendi in de numerieke mechanica wordt besteed aan het implementeren van wiskundige modellen. Automatisering kan hen helpen om snel eindige-elementenmodellen te ontwikkelen waardoor de inspanning kan verschuiven van implementatie naar modellering.

7. Aan elk geslaagd computermodel ligt sterke wiskunde ten grondslag.

8. "Ieder mens is lid van een geheel,
   Zo geschapen als één ziel.
   Als één lid getroffen wordt door leed,
   Behoudt de rest onmogelijk de vrede.
   Als je niet meevoelt met andermans smart,
   ben je het niet waard mens genoemd te worden." –Sa'adi; Perzische dichter, denker en filosoof, 13e eeuw.

9. *Free software* is een kwestie van vrijheid, niet van geld. Om het concept te doorgronden kan beter gedacht worden aan *free speech* dan aan *free beer*.

10. Het leven heeft zijn eigen partiële differentiaalvergelijking (PDV) met randvoorwaarden die kunnen veranderen in de tijd. Hoe meer je meemaakt, hoe beter je je eigen PDV leert kennen en hoe nauwkeuriger je de randvoorwaarden kunt voorspellen.

11. Om gemotiveerd te blijven gedurende een promotieproject is het belangrijk het project te verdelen in kleine sub-projecten die kunnen helpen om in ieder stadium voortgang te kunnen meten.

12. Hoewel communicatiemiddelen in de laatste jaren een enorme vooruitgang geboekt hebben (bijv. emails en telefoons) kan geen van deze middelen directe ontmoetingen vervangen.

Deze stellingen worden opponeerbaar en verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotoren, Prof. dr. ir. L. J. Sluys en Dr. G. N. Wells.

Curriculum vitae

August 1st, 1980	Born in Mianeh, Iran, as Mehdi Nikbakht
1994 – 1998	Diploma in Mathematics and Physics, Roshd High School, Tehran, Iran.
1998 – 2002	Bachelor of Science in Civil Engineering, Sharif University of Technology, Tehran, Iran.
2002 – 2004	Master of Science in Structural Engineering, Sharif University of Technology, Tehran, Iran.
2004 – 2005	Structural Engineer, Namvaran Engineering and Management Company, Tehran, Iran.
2006 – 2011	PhD candidate at the Computational Mechanics group in the Faculty of Civil Engineering and Geosciences, Delft University of Technology, The Netherlands.
2008 – 2010	Visiting PhD candidate at the Mechanics, Materials and Design division of the Engineering Department, the University of Cambridge, The United Kingdom.
Since 2011	Metrology Design Engineer at ASML, Veldhoven, The Netherlands.