On Implementing a Higher Order Generalized Finite Element Method

On Implementing a Higher Order Generalized Finite Element Method

Kai G. Schwebke

UNIVERSITÄT DER BUNDESWEHR MÜNCHEN

FAKULTÄT FÜR BAUINGENIEUR- UND VERMESSUNGSWESEN

Thema der Dissertation:
On Implementing a Higher Order
Generalized Finite Element Method

Verfasser: Kai Gerd Schwebke

Promotionsausschuss:
Vorsitzender: Univ.-Prof. Dr.-Ing. habil. Anton Heinen
1. Berichterstatter: Univ.-Prof. Dr.-Ing. Stefan Holzer
2. Berichterstatter: Univ.-Prof. Dr.-Ing. habil. Norbert Gebbeken

Tag der Prüfung: 18. April 2008

Mit der Promotion erlangter akademischer Grad:
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

Wildberg, den 5. Mai 2008

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet für Informationsverarbeitung im konstruktiven Ingenieurbau der Universität Stuttgart und am Institut für Mathematik und Bauinformatik der Universität der Bundeswehr München im Rahmen des DFG-Forschungsvorhabens *'Die verallgemeinerte Methode der finiten Elemente (GFEM) in der Strukturmechanik'*.

Mein besonderer Dank gilt Herrn Univ.-Prof. Dr.-Ing. Stefan Holzer für die Betreuung meiner Arbeit. Die zahlreichen inhaltlichen Anregungen und Diskussionen, verbunden mit großen Freiheiten bei der Umsetzung und Offenheit für neue Ideen, machten die Ausarbeitung in der vorliegenden Form erst möglich.

Herrn Univ.-Prof. Dr.-Ing. habil. Anton Heinen und Herrn Univ.-Prof. Dr.-Ing. habil. Norbert Gebbeken möchte ich für die Übernahme des Vorsitzes des Promotionsausschusses bzw. des Mitberichtes meinen Dank aussprechen.

Den Mitarbeitern und Doktoranden des Fachgebietes für Informationsverarbeitung im konstruktiven Ingenieurbau an der Universität Stuttgart und des Institutes für Mathematik und Bauinformatik an der Universität der Bundeswehr München danke ich für die gute und freundliche Zusammenarbeit und Unterstützung meiner Arbeit.

Darüber hinaus danke ich der deutschen Forschungsgemeinschaft für die Förderung des durchgeführten Vorhabens.

Abstract

The Generalized Finite Element Method (GFEM) was first introduced in [Mel95]. It combines desirable features of the standard Finite Element Method and the meshless methods.

The key difference of the GFEM compared to the traditional FEM is the construction of the ansatz space. Each node of the finite element mesh carries a number of ansatz functions, expressed in terms of the global coordinate system. Those ansatz functions are multiplied by a partition of unity and serve as element ansatz functions in the patch constituted by the elements incident at the node.

Using this technique to create the ansatz space allows for arbitrary ansatz functions. C^0-continuity is enforced by construction.

The ansatz is enriched using analytical functions or numerical approximations derived from side calculations containing a-priori knowledge of the solution close to singularities. The performance of GFEM with a higher order of polynomial ansatz functions is compared to traditional *h*-, *p*- and *hp*-extensions of the FEM.

Most of the efficient solvers, e.g. multi-grid or *cg*, cannot be applied to the semi-definite systems resulting from a GFEM discretization. Several solving strategies are evaluated for higher order GFEM.

The work concludes with a description of the implementation of the GFEM with a flexible object-oriented framework using C++.

Contents

Chapter 1

Introduction

The Finite Element Method (FEM) is a well-established tool for numerical simulation in mechanics, engineering and other fields of science. Reasons contributing to the ongoing success and further development of this method are its generality and relative simplicity.

The decomposition of the domain into elements of simple topology specifically allows the analysis of domains of complex shape which would be quite infeasible using, for example, finite differences. For some use-cases, however, compared to the computational time needed to perform the actual simulation, generating and checking a mesh is a time consuming and not fully automatic task.

The difficulties in the creation of a FE-mesh arise not from the fact that some mesh has to be created, but from the following requirements that such a mesh normally has to meet:

- The mesh should resolve the geometry of the domain.
- Discontinuities, like borders of loads, material interfaces, or changes of other relevant problem dependent parameters, should be aligned to an element edge in the mesh.

- The approximating functions have to meet continuity criteria.
- Depending on the method, the mesh should provide 'well shaped' triangles (measured by some criterion, e.g. the ratio of the largest inscribed circle to the smallest circumcircle).
- The mesh density function (average size of elements at a given point of the domain) should meet some given prerequisites. Often the mesh should be refined around singularities (e.g. re-entrant corners). Some methods—like *p*-FEM for plate-problems under certain boundary conditions—require refinements along the edges resolving a boundary layer.

To overcome these issues, meshless methods are subject to research in numerical mathematics and engineering disciplines. These methods replace the mesh by a set of points associated with a compact support surrounding the point and so avoid constructing a mesh, which is required for conventional FE methods. Instead of a mesh, a suitable set of uniformly or non-uniformly distributed points has to be constructed, e.g. with the methods described in [DGJ02]. Each point has a domain of influence (support) where ansatz functions can be applied. See [Dua95] and [BKO+96] to find reviews for meshless methods like *Moving Least Squares*, *Element-Free Galerkin* or *Smoothed Particle Hydrodynamics* methods.

While removing the need—and thereby removing the difficulties found in creating a mesh—new issues arise. These new issues may involve integration over domains of complex shape or implementing essential boundary conditions requiring additional care. E.g. [KS00] describes an octree-based representation of the discretization with a triangulation of boundary octants to gain an integration mesh.

[CQYY01] investigates the problem of numerical integration for Galerkin mesh-free methods in greater detail. To avoid the complexities arising from Gauss integration in this case, nodal integration methods employing Voronoi diagrams have been designed

for mesh-free methods. These methods require strain smoothing procedures or other stabilizing measures to avoid singular spurious modes. Some meshless methods, like *Moving Least Squares*, describe the ansatz in an implicit way. This leads to additional difficulties in efficiently determining derivatives and integrating the ansatz ([BRT00]). For a complete description of implementing a meshfree *Partition of Unity Method* including efficient solving and parallelization, see [GS00], [GS02c], [GS02d], [GS02b] and [GS02a].

An alternative is to combine ideas from meshless and classical Finite Element Methods into a new method. This method was first reported as a *Special FEM* in [BCO94] and then expanded into the *Partition of Unity Method* in [BM96] and [BM97]. Later on, the term *GFEM* originating from [Mel95] became commonly used. Polynomial approximations, as used in traditional finite element methods, require refinement of the mesh around singularities at corners and edges. This leads to additional difficulties in creating a suitable mesh and raises the number of elements needed. The key feature of Generalized Finite Element Methods is the use of a partition of unity, which is a set of functions whose values sum to the unity at each point in the domain. The partition of unity allows integration of a-priori knowledge of the nature of the searched solution into the discretization. While still requiring a mesh, anisotropic refinements are no longer needed if suitable enrichments are added to the ansatz. [DBO00] shows the integration of the handbook solution of the elasticity equations near a corner. Another property investigated there is the ability to produce seamless *hp*-FEM approximations with nonuniform h and p.

The fundamental difference of *GFEM* compared to other meshless methods is the choice of the partition of unity. In the *GFEM*, conventional finite element shape functions are used to create a partition of unity. *hp*-clouds[CAD95], in contrast, use circles (or n-dimensional spheres) to create a partition of unity. Using FEM shape functions as a partition of unity for *GFEM* leads to great

similarities implementing the method compared to *FEM*. Compared also to other mesh-free methods, like Moving Least Squares or Shepard's Interpolation, the numerical integration can easily be implemented over elements of simple shape. [BB07] provides on introduction to the evolvement of the Generalized Finite Element Method and its relations to classical *FEM* as well as meshless methods.

Another common term for this family of methods is the Extended Finite Element Method (*X-FEM*) as used in [BSMM00] with a focus on modelling discontinuities arising from cracks. The main benefit here is that crack evolvement does not require re-meshing of the domain. This is achieved combining asymptotic near tip field solutions to cover singularities and Haar functions to model discontinuities not resembled in the mesh. [BMMB05] refines the method by increasing the domain of enrichment, preconditioning the stiffness matrices to allow usage of conventional solvers and optimizing the numerical integration of enrichment functions.

Research in [MB02], [MGB02], [GMB02], [BXP03] and [BPM+03] covers alternative, implicit surface representations to further extend *X-FEM* in the context of crack growth for elastostatic problems. The resulting method requires no explicit representation of the crack—the crack and its growth are described entirely in forms of nodal data. [RGC05b] and [RGC05a] describe applying *X-FEM* to dynamic and time-dependant problems. For an overview of the development of the Extended Finite Element Method, see [Moe07].

GFEM implementations using higher order ansatz functions of the *p*-version of the FEM are described in [SZB04] and [SBCB03]. A conventional *p*-version of the FEM is enriched using the *GFEM* partition of unity with analytical and numerical handbook functions. [LPRS05] uses the term 'higher-order' *X-FEM* when a fixed area of influence is enriched with a special function during *h*-extension leading to successively more enriched nodes.

In contrast to these approaches, this work describes and analyses

the implementation of a pure higher order *GFEM* applying all ansatz functions using a partition of unity.

The linear partition of unity used by *GFEM* provides a framework for constructing C^0 continuous shape functions. [DKQ06] extends the partition of unity to allow for arbitrary smooth C^k continuous shape functions.

As shown in [DBO00], *GFEM* ansatz space contains linear dependencies arising from the fact that both the partition of unity and the basis of the ansatz functions are polynomial functions. [TYT06] investigates the problem in greater detail showing that, in addition, mesh topology and element type (quadrilateral or triangular) as well as element shape have a great impact on the linear dependencies. At the moment, these linear dependencies cannot be avoided for the general, higher order case. Therefore, different solving methods for the semi-definite linear equation system are investigated. [DBO00] proposes some solving strategies. One of them is perturbation and post-iteration of the stiffness matrix. Performing a *p*-extension, this method becomes more and more inefficient for higher order polynomial ansatz spaces. *GFEM* preserves, however, the banded structure of the stiffness matrix as described in [DBO00]. As a result, other alternatives to solve the linear equation system exploiting these properties will be compared in this work.

Imposing essential boundary conditions requires the construction of shape functions that vanish on the boundary of the domain or enfore Dirichlet boundary conditions through penalties as used in [ABCM02] in the context of Discontinuous Galerkin Methods.

[BBO02] suggests omitting the constant function from the space of ansatz functions at a boundary node. In addition, because higher order polynomial ansatz functions do not fulfil essential boundary conditions, these also cannot be used at the boundary of the domain, thus making the implementation of a higher order pure *GFEM* using this technique impossible. In [BBO02] the Penalty Method, Nitsche's Method and the Characteristic Function Method

are discussed in the context of other, non-*GFEM* shape functions like Reproducing Kernel Particle or Moving Least Squares. The Characteristic Function Method also proves to be suitable in imposing essential boundary conditions for higher order polynomial *GFEM*.

Chapter 2

GFEM

The Generalized Finite Element Method (GFEM) was first introduced in [Mel95]. A similar approach is described in [BSMM00] under the name *Extended Finite Element Method (X-FEM)*. The Generalized Finite Element Method (GFEM) shares many properties with meshless methods. Like the *hp*-cloud method [CAD95], approximation functions and enrichment of the approximation spaces can be done at each nodal point.

Unlike *hp*-clouds, the partition of unity (PUM) used in the GFEM is constructed on a regular mesh using linear finite element shape functions. This avoids the need to integrate over irregularly shaped subdomains like Ω_1 in Figure 2.1 resulting from intersecting arbitrary circular supports. Some meshless methods ignore this problem and use an integration mesh which is not aligned to support boundaries. According to [SBC98] this leads to numerical integration errors that are very difficult to control.

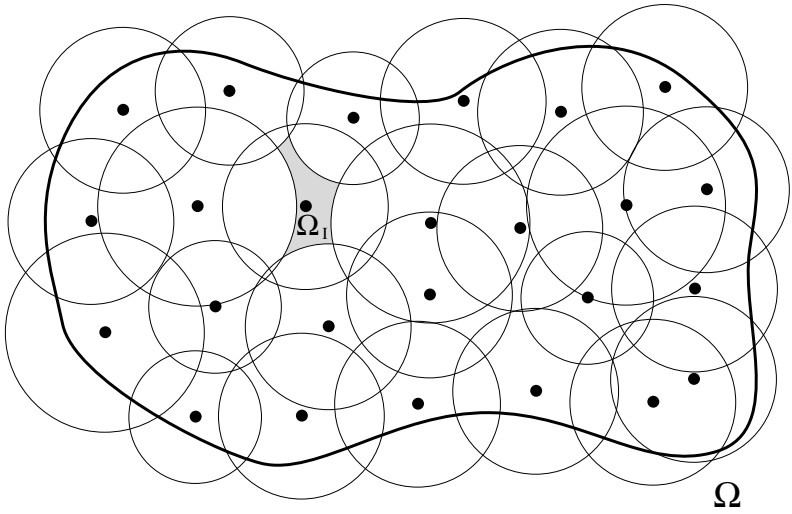

Figure 2.1: Circular support

2.1 Model Problems

To investigate the GFEM, the following two plane model problems are solved.

2.1.1 Plane Poisson Problem

For a plane poisson problem, the displacement field $u(x, y)$ is searched.

The governing partial differential equation of the domain Ω is:

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = c \tag{2.1}$$

The boundary Γ of Ω is subject to homogeneous Dirichlet boundary condition $u = 0$.

The transformation of Poisson's equation to the weak form leads to:

$$\int_\Omega \frac{\partial u}{\partial x} \cdot \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \cdot \frac{\partial v}{\partial y} \, d\Omega = \int_\Omega c\, v \, d\Omega \tag{2.2}$$

2.1.2 Plane Linear Elastostatic Problem

For a plane elastostatic problem, the displacement field $\mathbf{u}$ is searched:

$$\mathbf{u} = \begin{bmatrix} u_x(x, y) \\ u_y(x, y) \end{bmatrix} \tag{2.3}$$

The governing strain-displacement relations are:

$$\varepsilon_x(\mathbf{u}) = \frac{\partial u_x}{\partial x} \tag{2.4}$$

$$\varepsilon_y(\mathbf{u}) = \frac{\partial u_y}{\partial y} \tag{2.5}$$

$$\gamma_{xy}(\mathbf{u}) = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \tag{2.6}$$

With

$$\mathbf{D} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \tag{2.7}$$

the strain-displacement relation can be written as:

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \mathbf{D}\mathbf{u} \tag{2.8}$$

The stress tensor $\boldsymbol{\sigma}(\mathbf{u})$ corresponding to $\boldsymbol{\varepsilon}(\mathbf{u})$ is denoted by:

$$\boldsymbol{\sigma}(\mathbf{u}) = \begin{bmatrix} \sigma_x(\mathbf{u}) \\ \sigma_y(\mathbf{u}) \\ \tau_{xy}(\mathbf{u}) \end{bmatrix} \tag{2.9}$$

The stress-strain relationships are:

$$\boldsymbol{\sigma}(\mathbf{u}) = \mathbf{E} \cdot \boldsymbol{\varepsilon}(\mathbf{u}) \tag{2.10}$$

$\mathbf{E}$ is a symmetric positive definite matrix of material constants, also called material stiffness matrix. For isotropic materials, $\mathbf{E}$ is defined in terms of two material constants: Young's modulus E and Poisson's ratio ν:

$$\mathbf{E} = \begin{bmatrix} \frac{E}{1-\nu^2} & \frac{E\nu}{1-\nu^2} & 0 \\ \frac{E\nu}{1-\nu^2} & \frac{E}{1-\nu^2} & 0 \\ 0 & 0 & \frac{E}{2(1+\nu)} \end{bmatrix} \tag{2.11}$$

The part Γ_D of the boundary Γ of the domain Ω is subject to homogeneous Dirichlet boundary condition $\mathbf{u} = 0$.
The part Γ_N of the boundary is subject to Neumann boundary condition $\boldsymbol{\sigma} = \mathbf{T}$.
The transformation to the weak form leads to:

$$\int_\Omega \left((\mathbf{D}\,\boldsymbol{v})^T \; \mathbf{E}\,\mathbf{D}\,\mathbf{u} \right) \, d\Omega = \int_{\Gamma_N} \mathbf{T}\,\boldsymbol{v} \, d\Gamma_N \tag{2.12}$$

2.2 GFEM discretization

The domain Ω is subdivided using a regular mesh containing n nodes and m triangular linear finite elements. Associated to each node n_j is a patch $\omega_j \in \Omega$ constituted by all triangular elements incident to this node. The assemblage of linear shape functions of all elements of ω_j associated to n_j compose the Hat function $\mathcal{N}^j$ defined over the supporting patch ω_j (see Figure 2.2).

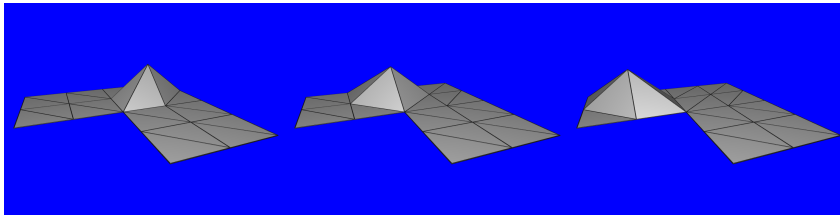

Figure 2.2: Hat functions

The set of functions $\left\{\mathcal{N}^j\right\}_{j=1}^{n}$ constitute a partition of unity:

$$\sum_{j=1}^{n} \mathcal{N}^j = 1 \quad \text{for each point } p \in \Omega \tag{2.13}$$

Using this partition of unity as an ansatz leads to the classical linear h-version of the Finite Element Method. Locally multiplying

the hat functions around each node with a set of shape functions (including the constant function $c(x) = 1$) results in a higher order ansatz of arbitrary polynomial degree. A-priori knowledge of local solution characteristics may be easily embedded in the ansatz as well.

The higher order GFEM shape functions are based on Legendre polynomials P_i:

$$P_0(x) = 1 \tag{2.14}$$

$$P_1(x) = x \tag{2.15}$$

$$(n+1)\,P_{n+1}(x) = (2n+1)\,x\,P_n(x) - n\,P_{n-1}(x), \tag{2.16}$$

$$n = 1, 2, 3, \ldots, p-1 \tag{2.17}$$

$L^j_{kl}(x, y)$ is a product of two Legendre polynomials centred around $n_j = (n^j_x, n^j_y)$ and scaled to the characteristic size h^j of the patch ω_j:

$$L^j_{kl}(x, y) = P_k\left(\frac{x - n^j_x}{h^j}\right) \cdot P_l\left(\frac{y - n^j_y}{h^j}\right) \tag{2.18}$$

Centering and scaling improves the numerical properties of the ansatz functions.

The GFEM shape functions result from the multiplication of the partition of unity with the polynomial enrichment functions:

$$\phi^j_{kl} = \mathcal{N}^j \cdot L^j_{kl} \tag{2.19}$$

A function space of degree p is spanned by the set of all shape functions ϕ_{kl} with $k = 0, 1, 2, \ldots, p$ and $l = 0, 1, 2, \ldots, p$.

If needed, the space is enriched by further functions multiplied with the partition of unity.

The resulting shape functions inherit the approximating properties of the enrichment functions and the compact support and C^0-continuity of the partition of unity.

Each node may carry an ansatz of different polynomial degree. This leads to an easy implementation of *p*-adaptive or *hp* methods.

Unlike conventional higher order ansatz spaces, the higher order GFEM ansatz contains linear dependant elements. The resulting semi-definite equation system cannot be solved by almost any of the established solvers used for classic FEM methods like multigrid or CG solvers. Chapter Three will discuss various options for solving these semi-definite linear equation systems.

2.3 Essential Boundary Conditions

The GFEM shape functions have to fulfil essential (Dirichlet) boundary conditions or the approximate satisfaction of the boundary condition has to be ensured by other means. [BBO02] suggests the following methods:

1. The Penalty Method,
2. Nitsche's Method,
3. The Characteristic Function Method.

In the following, we will discuss the characteristic function method, which transforms the GFEM shape functions to directly fulfil essential boundary conditions.

2.3.1 Characteristic Function Method

For a patch ω_j around the node n_j and adjacent to a Dirichlet boundary Γ_D the ansatz $\{\phi^j\}$ is multiplied with a smooth function

Φ so that

$$\Phi > 0 \text{ in } \Omega, \tag{2.20}$$
$$\Phi = 0 \text{ on } \Gamma_D \tag{2.21}$$
$$\text{and } |\nabla\Phi| > 0 \text{ on } \Gamma_D \tag{2.22}$$

A natural candidate for a characteristic function is the Hat function N^j. It is smooth within each element, C^0-continous between elements and piecewise linear.

In the following example, the plane Poisson problem is solved on a circular domain. Figure 2.3 shows a mesh of four triangles. The edges on the boundary are mapped to follow the circular shaped domain boundary Γ_D using radial blending function mapping ([Gor71], [GH73b], [GH73a]—see Appendix D for a detailed description of the triangle mapping used).

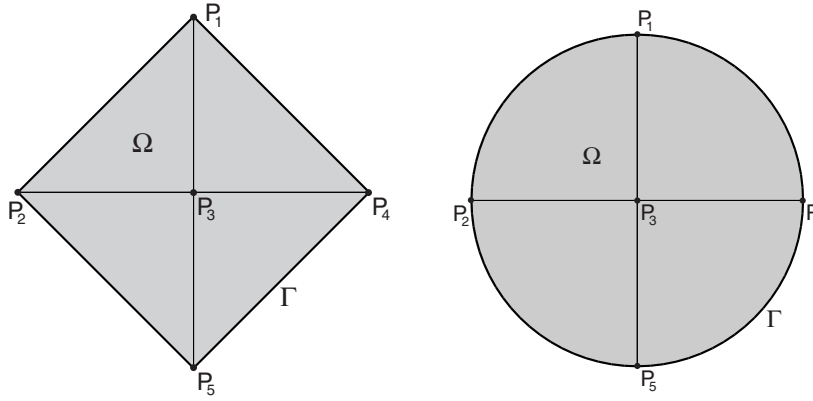

Figure 2.3: Linear and blended mesh

To avoid error cancellation due to symmetry effects, the node P_3 is shifted from the centre. Figure 2.5 shows the absolute local error of the displacement field. On the left, the characteristic function is unshifted, i.e. not aligned to the mesh. This leads to an error

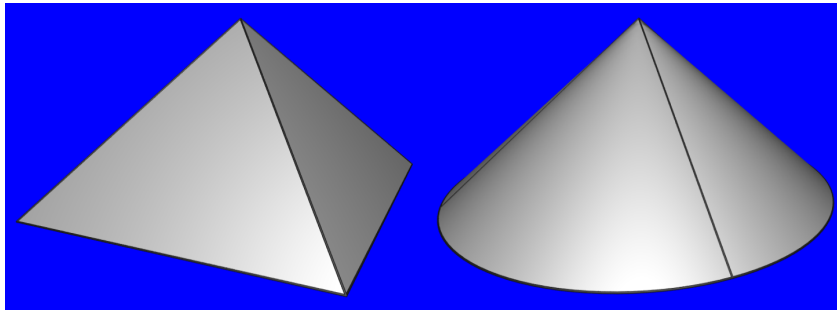

Figure 2.4: Linear and blended hat function

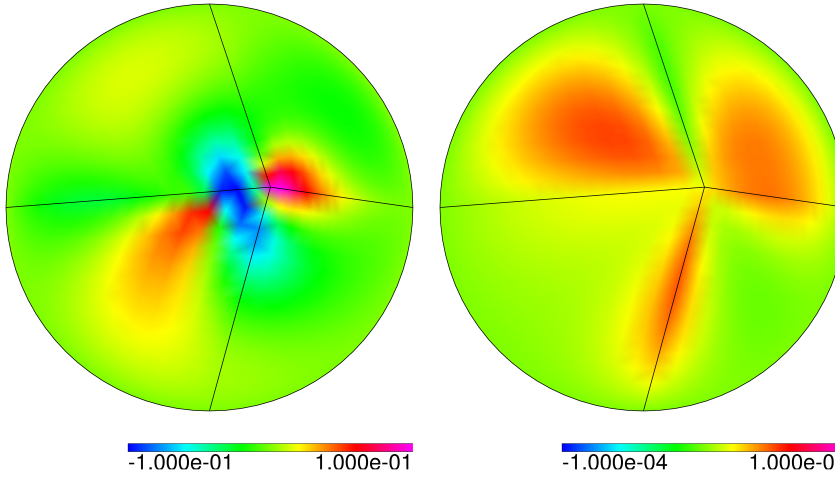

Figure 2.5: Absolute local error with unaligned and aligned characteristic function

which is four orders of magnitude higher than for the aligned case shown on the right hand side. Here, the Hat function $\mathcal{N}^3$ is used as a characteristic function, which is aligned to the underlying mesh by construction.

2.3.2 Convergence rates for *p*-extension

A plane elastostatic example problem is solved using *p*-GFEM and conventional *p*-version FEM. Figure 2.6 shows a mesh of a square with a circular hole which is fixed on the bottom and subject to loading on the top.

In the following example, the performance of GFEM for *p*-extension is compared to a conventional *p*-version FEM implementation. Figure 2.7 shows exponential convergence rates for both GFEM and traditional FEM.

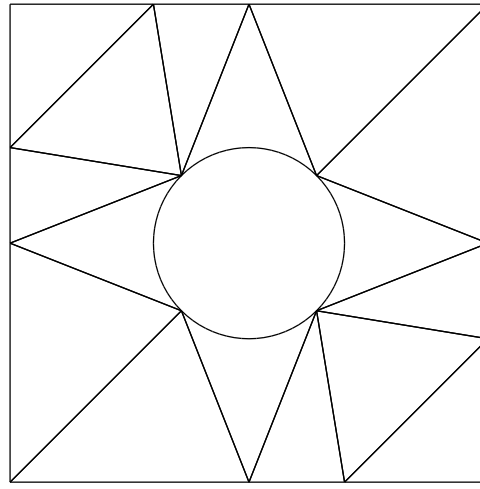

Figure 2.6: Curved domain example mesh

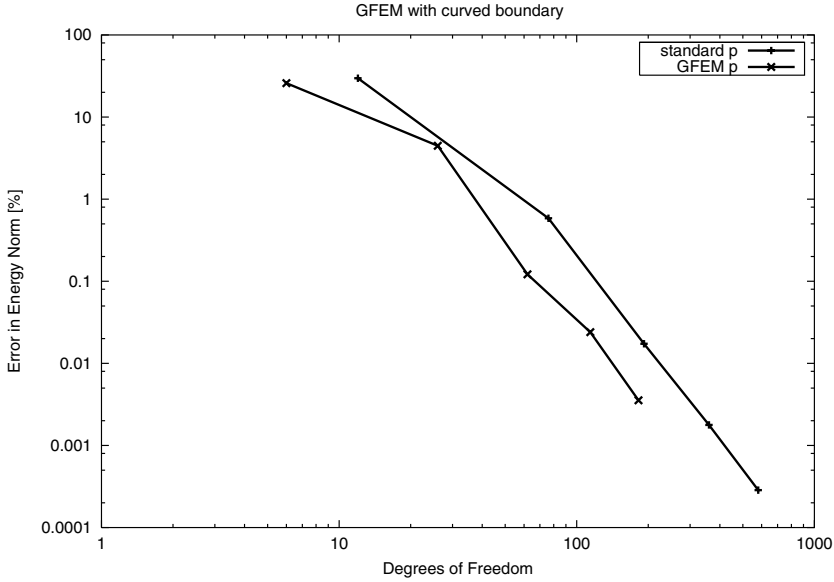

Figure 2.7: Error in energy norm for curved domain example

DOF	energy	relative error	β
12	1.942324	29.6804%	
76	2.745970	0.5853%	2.13
192	2.761660	0.0173%	3.80
360	2.762089	0.0018%	3.62
580	2.762130	0.0003%	3.83

Table 2.1: Domain with curved boundary, standard *p*-version

DOF	energy	relative error	β
6	2.046840	25.8965%	
26	2.638836	4.4640%	1.20
62	2.758780	0.1216%	4.15
114	2.761475	0.0240%	2.66
182	2.762040	0.0035%	4.09

Table 2.2: Domain with curved boundary, GFEM *p*-version

2.4 Enrichment for Singularities

2.4.1 Poisson problem for re-entrant corner with analytical enrichment

The solution of the Poisson problem on an L-shaped domain contains a singularity in the re-entrant corner. Such a singularity cannot be approximated very well using polynomials.
According to [BS92], the displacement u is of the form:

$$u = a_1 r^\lambda \cos(\lambda\Theta) + a_2 r^\lambda \sin(\lambda\Theta),\ \lambda \geqslant 0 \tag{2.23}$$

Imposing the essential boundary conditions of the re-entrant corner of an L-shape $u = 0$ for $\Theta = \pi$ and $\Theta = -\frac{\pi}{2}$, this yields to:

$$a_1 r^\lambda \cos(\lambda\pi) + a_2 r^\lambda \sin(\lambda\pi) = 0 \tag{2.24}$$

$$a_1 r^\lambda \cos\left(-\lambda\frac{\pi}{2}\right) + a_2 r^\lambda \sin\left(-\lambda\frac{\pi}{2}\right) = 0 \tag{2.25}$$

This system of two equations has nontrivial solutions for a_1, a_2 only if the determinant of the coefficient matrix vanishes:

$$\cos(\lambda\pi)\cdot\sin\left(-\lambda\frac{\pi}{2}\right) - \sin(\lambda\pi)\cdot\cos\left(-\lambda\frac{\pi}{2}\right) = 0 \tag{2.26}$$

Using $\lambda \geqslant 0$ this leads to:

$$\lambda = 0, \frac{2}{3}, \frac{4}{3}, 2, \ldots \tag{2.27}$$

Since only $r^{\frac{2}{3}}$ is singular, only $\lambda = \frac{2}{3}$ is used further to derive $a_1 = \sqrt{3}$ and $a_2 = 1$ leading to the singular term:

$$u = \sqrt{3}\, r^{\frac{2}{3}} \cos\left(\frac{2}{3}\Theta\right) + r^{\frac{2}{3}} \sin\left(\frac{2}{3}\Theta\right) \tag{2.28}$$

This function will be used to enrich the ansatz. Figure 2.8 shows the singular function on the left and the resulting GFEM ansatz after multiplication with the Hat function N on the right. The Figure

illustrates how the multiplication with $\mathcal{N}$ ensures inter-element C^0-continuity and restriction of the function to the compact support of the associated patch ω.

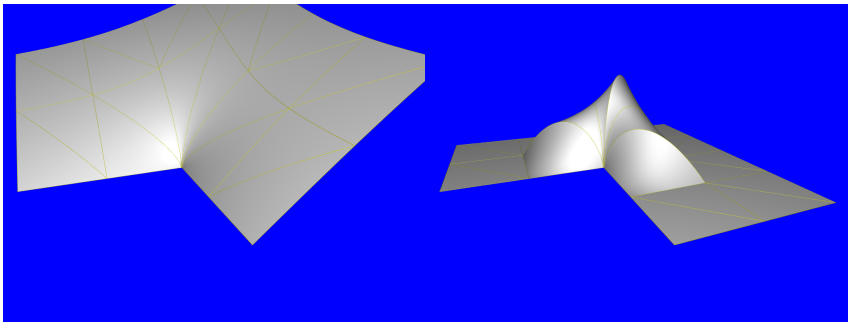

Figure 2.8: Analytical base function and resulting ansatz

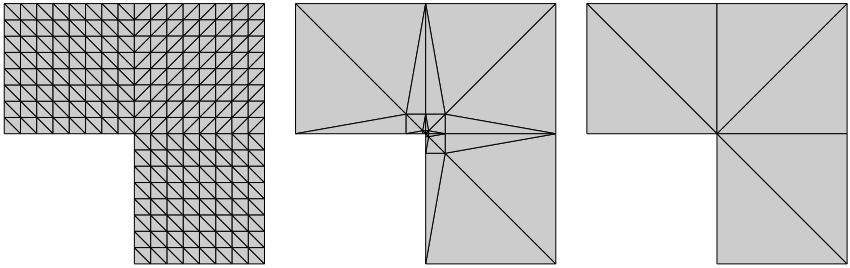

Figure 2.9: *h*-, *hp*- and *p*-version mesh

GFEM convergence rates are compared for the following cases:

1. *h*-extension using a linear ansatz (constant enrichment) and a successively refined mesh. In this case GFEM resembles the ordinary *h*-version FEM.

2. *p*-extension using a coarse mesh (see Figure 2.9 on the right) and a successively increased ansatz.

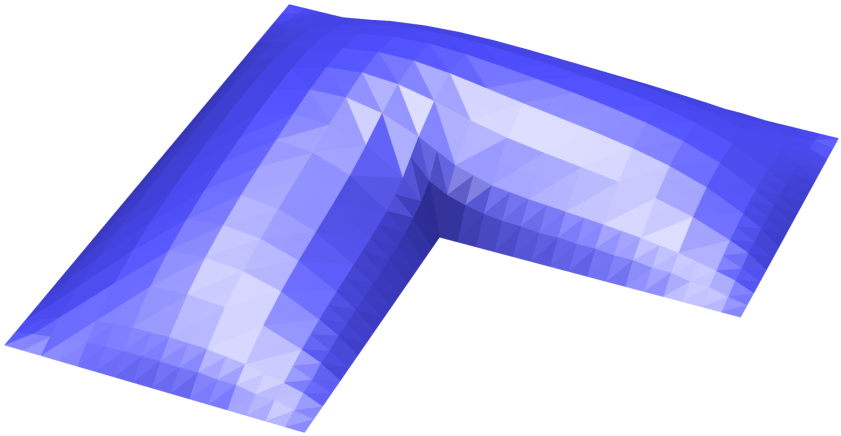

Figure 2.10: h-GFEM displacement

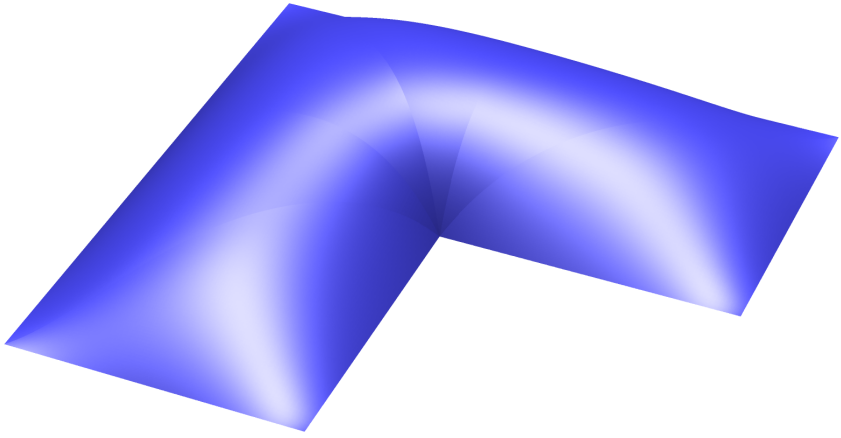

Figure 2.11: p-GFEM displacement

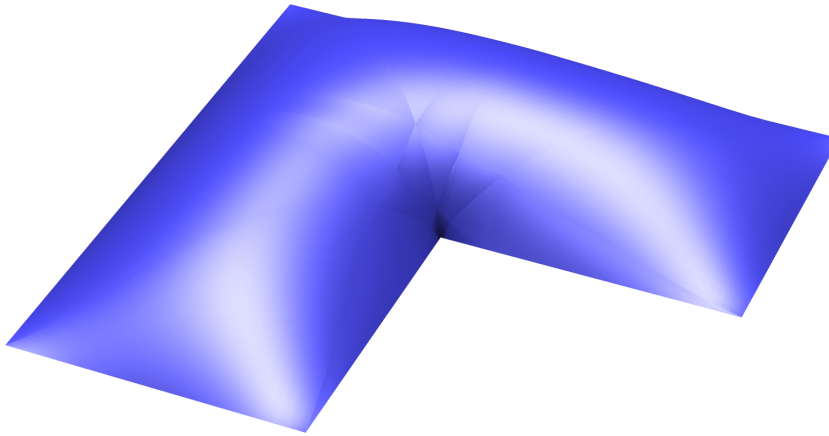

Figure 2.12: *hp*-GFEM displacement

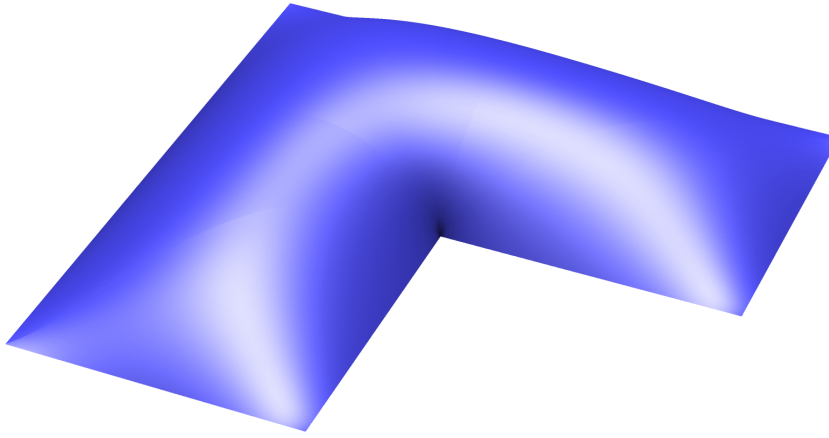

Figure 2.13: *p*-GFEM displacement with enrichment

3. *hp*-extension on a refined mesh using a recursive geometric progression of $s_9 = 0.15$ as advised in [BS92] and a linear ansatz on the smallest element. The degree of the ansatz increases by one on every larger element level.

4. *p*-extension like 2. enriched by the singular ansatz derived above.

The *h*-extension performs as expected showing algebraic convergence. Also, the *p*-extension shows only algebraic convergence due to the singularity. The *hp*-extension reaches an exponential convergence rate by adapting the mesh and the polynomial degree to the singularity. The *p*-extension with analytical enrichment reaches also an exponential convergence rate on the same coarse mesh used for *p*-extension.

Figures 2.10 – 2.13 illustrate the displacement of the resulting solution. These shaded pictures disclose discontinuity artefacts nicely allowing a visual assessment of the solution quality (as no smoothing post-processing is applied).

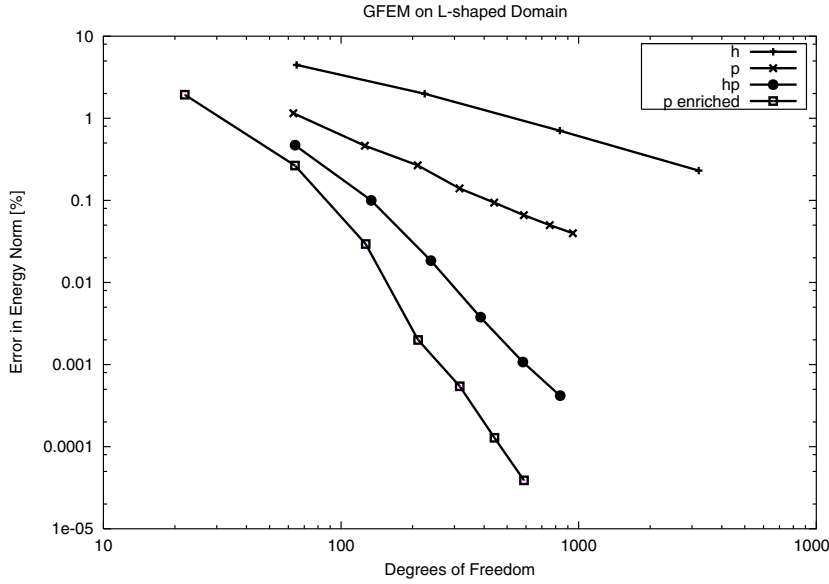

Figure 2.14: Error in energy norm for singular domain example

Table 2.3: L-shaped domain, h-version

DOF	energy	relative error	β
8	59.495192	11.0667%	
21	63.680628	4.8103%	0.86
65	63.909366	4.4684%	0.07
225	65.566258	1.9917%	0.65
833	66.427336	0.7046%	0.79
3201	66.744192	0.2309%	0.83
12545	66.848280	0.0753%	0.82

Table 2.4: L-shaped domain, p-version

DOF	energy	relative error	β
21	63.680628	4.8103%	
63	66.128845	1.1507%	1.30
126	66.589322	0.4624%	1.32
210	66.719740	0.2675%	1.07
315	66.804980	0.1401%	1.60
441	66.835929	0.0938%	1.19
588	66.854478	0.0661%	1.22
756	66.865210	0.0500%	1.11
945	66.872080	0.0397%	1.03

Table 2.5: L-shaped domain, *hp*-version

DOF	energy	relative error	β
64	66.584786	0.4692%	
134	66.831690	0.1001%	2.09
239	66.886361	0.0184%	2.93
386	66.896148	0.0038%	3.31
582	66.897953	0.0011%	3.06
834	66.898392	0.0004%	2.62

Table 2.6: L-shaped domain, enriched *p*-version

DOF	energy	relative error	β
22	65.602828	1.9370%	
64	66.720910	0.2657%	1.86
127	66.878978	0.0294%	3.21
211	66.897338	0.0020%	5.30
316	66.898308	0.0005%	3.22
442	66.898586	0.0001%	4.30
589	66.898646	0.0000%	4.17

2.4.2 Plane Elasticity and enrichment using numerical side calculation

For many problems, analytical solutions for singular points are not available. In the following example, a numerical side calculation provides an approximate solution. A plane elastiticity problem on the discretized domain in Figure 2.15 is solved. All corners of the rectangular holes lead to singularities of the exact solution.

Two side calculations using *hp*-GFEM under different loading conditions (see Figure 2.16) provide a numerical approximation close to the singular points. The two different loading conditions resemble the two modes known from plane fracture mechanics.

The resulting displacement fields of the side calculation are mapped and used as ansatz functions. Figure 2.17 illustrates the mapping for one singular point. The mapping is performed for all points reusing the characteristics of the singular solution determined once.

Figures 2.18 and 2.19 show the σ_v stress component of the solution. The results show a slightly better convergence using numerical enrichment. Unlike the analytical enrichment for the poisson problem, there is no improvement from algebraic to exponential convergence. In addition, the choice of numerical integration influences the accuracy and stability of the results using a numerical side calculation heavily. This leads to an additional performance penalty which overcompensates the small improvement in convergence rate.

2.4.3 Triangle Location

Evaluating a numerical side calculation requires locating the triangle of the side calculation domain containing a given point mapped from the main domain. Randomly distributed point sets require elaborate algorithms, like space partitioning trees or directed search strategies. Here, the point set resulting from a

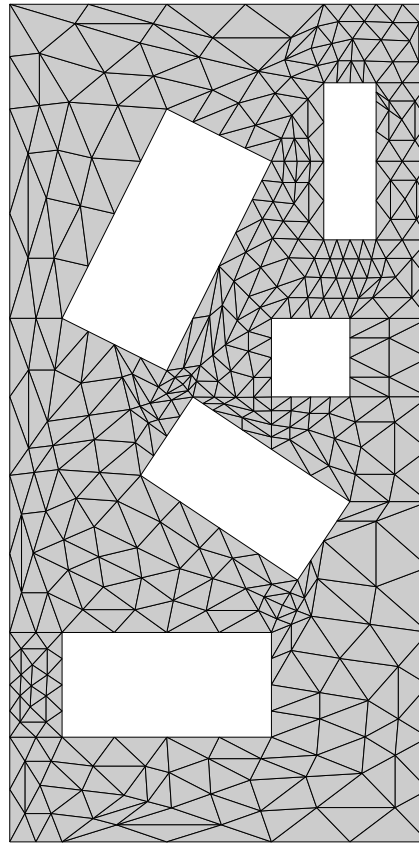

Figure 2.15: GFEM discretization mesh

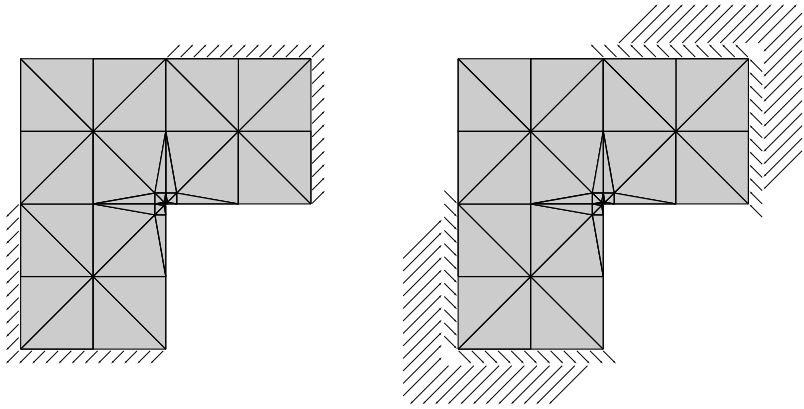

Figure 2.16: GFEM side calculations

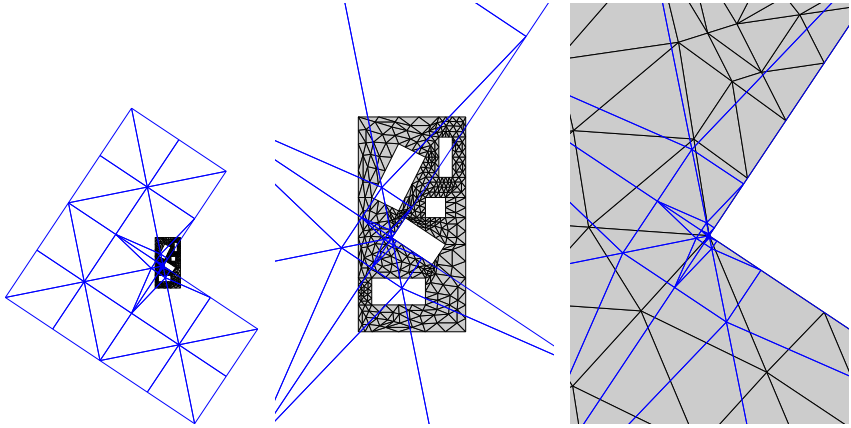

Figure 2.17: GFEM discretization mesh and one of the side calculation meshes

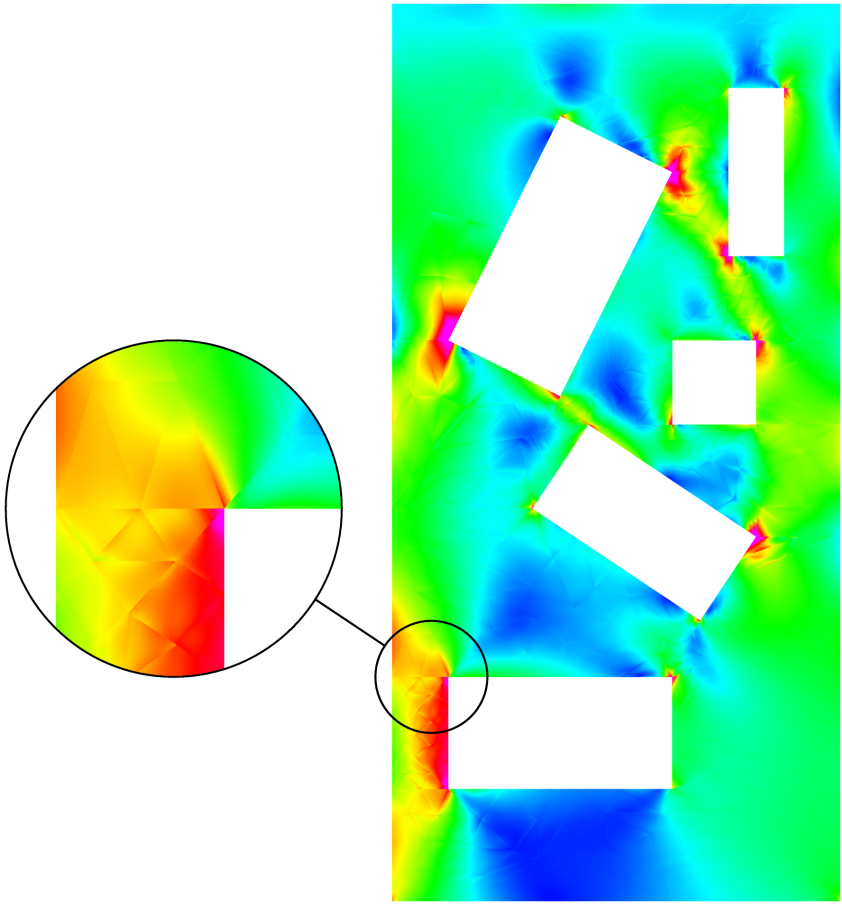

Figure 2.18: p-GFEM result: σ_v

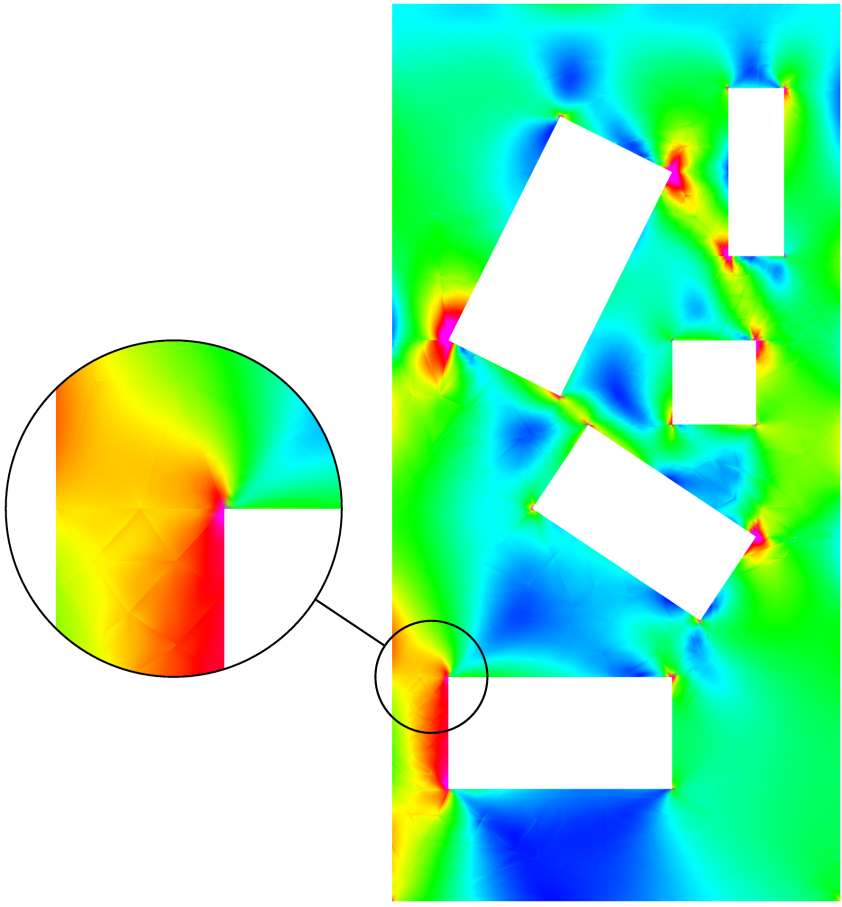

Figure 2.19: p-GFEM with num. side calc. result: σ_v

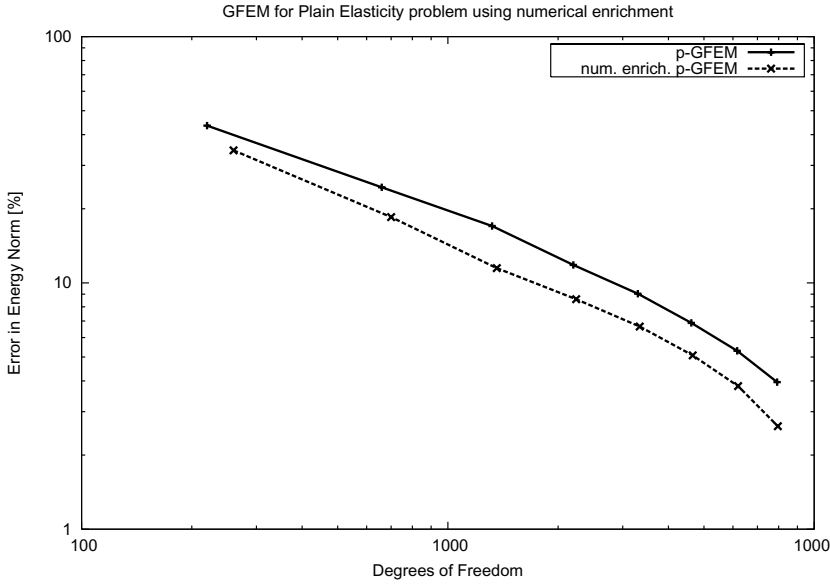

Figure 2.20: Error in energy norm for plane elasticity and numerical enrichment

Table 2.7: Plane elasticity, *p*-version

DOF	energy	relative error	β
220	1.156310	43.50%	
660	1.340900	24.45%	0.52
1320	1.384890	17.01%	0.52
2200	1.406210	11.83%	0.71
3300	1.414500	9.042%	0.66
4620	1.419410	6.880%	0.81
6160	1.422150	5.303%	0.91
7920	1.423930	3.954%	1.17

Table 2.8: Plane elastitcity, *p*-version with numerical side calculation

DOF	energy	relative error	β
260	1.256270	34.51%	
700	1.377350	18.50%	0.63
1360	1.407360	11.48%	0.72
2240	1.415640	8.589%	0.58
3340	1.419860	6.646%	0.64
4660	1.422480	5.080%	0.81
6200	1.424090	3.810%	1.01
7960	1.425180	2.621%	1.50

numerical integration scheme has the property that two successive points are located typically at a close distance and, therefore, with a high probability, contained by the same triangle or a small set of triangles.

This property is exploited using a LRU (least recently used) list. All triangles of a side calculation are referred using a linked list. To locate a triangle containing a point, the list is searched from the beginning until the containing triangle is found. If the triangle containing the point is not referred from the first element of the list, it is relocated to the beginning, so the least recently used triangles become the first elements of the list.

To measure the performance of this location algorithm, the average hit rate is calculated:

$$r_h = \frac{n_{\text{point locations}}}{n_{\text{triangles searched}}} \tag{2.29}$$

This simple strategy performs surprisingly well, leading to hit rates which are typically much greater than 50%.

2.4.4 Numerical Integration

All implemented integration schemes are based on the Gaussian Quadrature rule, which approximates the integral by a weighted sum at n evaluation points such that a polynomial of degree $p = 2n - 1$ is integrated exactly.

Tensor Product Gauss Quadrature

Approximating the area integrals of triangular elements using a Gaussian Quadrature on natural triangle coordinates (see Appendix D) leads to tensor product Gauss Quadrature. This integration scheme is well suited for polynomial ansatz functions on triangular elements.

Arbitrary shaped elements are implemented using the blending function method (see Appendix D.2). If blending is applied to an element, the Gauss Quadrature no longer integrates exactly. In this case, the number of evaluation points has to be increased until the numerical integration leads to a sufficiently exact approximation. As the blending is smooth, the Gaussian Quadrature converges well.

h-adaptive Integration Scheme

Integrating non-smooth ansatz functions, e.g. analytical or numerical enrichments, the Gaussian Quadrature becomes ineffective as these functions cannot be approximated very well using smooth polynomials.

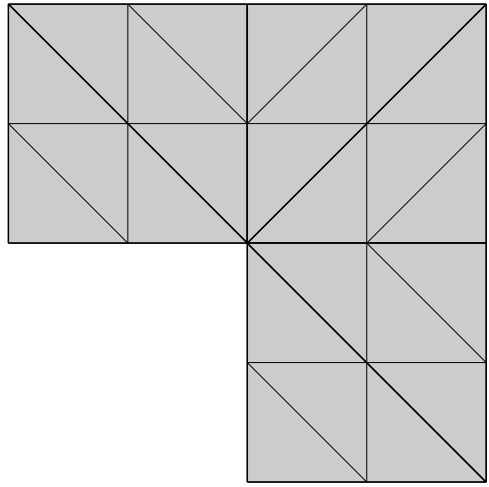

Figure 2.21: *p*-GFEM discretization using *h*-adaptive integration

As a general purpose integration method, an *h*-adaptive integration rule is implemented. Using a specified number of evaluation

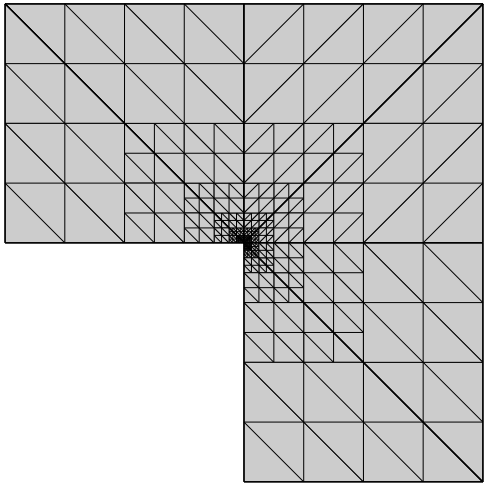

Figure 2.22: p-GFEM discretization with analytical enrichment using h-adaptive integration

points and a specified approximation accuracy, at least two numerical integrations are performed: one at an elemental level and one at a refined level, subdividing the element into four smaller sub elements. Using Richardson extrapolation, the exact result and the integration error is estimated. If the specified approximation accuracy is not reached, the specified approximation accuracy is distributed across the sub elements and the integration procedure is applied recursively until the accuracy target or a maximal number of refinements is reached.

Figure 2.21 shows the adaptive integration scheme applied to a polynomial ansatz. As the polynomials are integrated exactly using Gaussian quadrature, no adaptive refinement is performed. In Figure 2.22, the ansatz is enriched using an analytical singular function. The Gaussian quadrature performs poorly for non-smooth integrands, so an adaptive refinement towards the singular point of the analytical ansatz function is performed. Figure 2.23

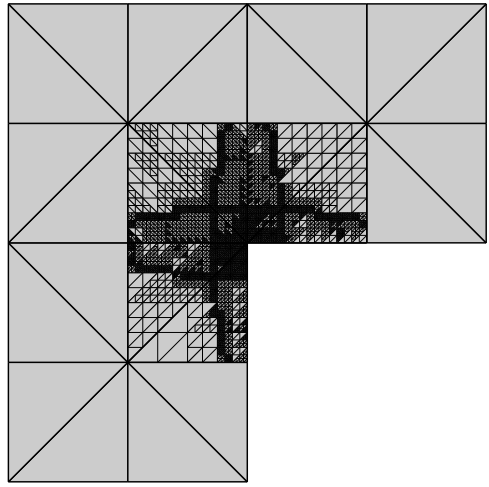

Figure 2.23: p-GFEM discretization with numerical enrichment using h-adaptive integration

shows adaptive integration for an ansatz function resulting from a numerical side calculation. This side calculation is non-smooth between elements of the side calculation mesh, so the adaptive refinement resembles the side calculation mesh inter-element boundaries. As the integration elements are not aligned with the elements of the side calculation, hard to control integration errors may nevertheless occur (see [DB99]).

hp-refined Integration Scheme

To integrate analytical ansatz functions more efficiently, a third integration scheme is implemented. Resembling the *hp*-extension of the FEM, a refinement combined with a reduction of polynomial degree is performed in a non-adaptive way towards the singular point of the analytical ansatz. This leads to good approximations comparable to h-adaptive integration, but is more efficient as less

evaluation points are needed in the many small elements close to the singular point of the ansatz function.

Chapter 3

Solving the linear Equation System

The solution of the linear equation systems arising from problems discretized using the General Finite Element Method poses a serious problem. Especially for a higher polynomial degree ansatz, the increasing number of resulting linear dependencies prohibits the usage of efficient methods, like multi-grid, Krylov subspace methods and other iterative solving algorithms.

Some alternatives for solving the linear system are pointed out in the following sections. Another approach would be to avoid the linear dependencies by construction, using a modified discretization like a mixed cell complex.

3.1 Perturbed Matrix and Post Iteration

This method is proposed in [DBO00] to solve a linear equation system where $\mathbf{K}$ is a semidefinite matrix:

$$\mathbf{K}\,\mathbf{u} = \mathbf{r} \tag{3.1}$$

The equation system is scaled with a transformation matrix $\mathbf{T}$:

$$\hat{\mathbf{K}} = \mathbf{T}\,\mathbf{K}\,\mathbf{T} \tag{3.2}$$
$$\hat{\mathbf{u}} = \mathbf{T}^{-1}\,\mathbf{u} \tag{3.3}$$
$$\hat{\mathbf{r}} = \mathbf{T}\,\mathbf{r} \tag{3.4}$$

with

$$T_{i,j} = \frac{\delta_{ij}}{\sqrt{K_{i,j}}} \tag{3.5}$$

where δ_{ij} is denoting Kronecker's delta-function, leading to

$$\hat{\mathbf{K}}\,\hat{\mathbf{u}} = \hat{\mathbf{r}} \tag{3.6}$$

For the transformed matrix $\hat{\mathbf{K}}$ holds $\hat{K}_{i,i} = 1$. The scaled matrix is perturbed:

$$\hat{\mathbf{K}}_\varepsilon = \hat{\mathbf{K}} + \varepsilon\,\mathbf{I} \text{ with } \varepsilon > 0 \tag{3.7}$$

where $\mathbf{I}$ denotes the identity matrix. $\hat{\mathbf{K}}_\varepsilon$ is a positive definite matrix which leads to a linear equations system solveable with every standard method:

$$\hat{\mathbf{K}}_\varepsilon\,\hat{\mathbf{u}}_0 = \hat{\mathbf{r}} \tag{3.8}$$

$\hat{\mathbf{u}}_0$ is not a solution for the unperturbed equation system 3.6. The remaining defect is determined to:

$$\mathbf{d}_0 = \hat{\mathbf{r}} - \hat{\mathbf{K}}\,\hat{\mathbf{u}}_0 \tag{3.9}$$

The solution of the unperturbed system could be gained from the calculated one, if one could calculate the necessary correction $\boldsymbol{e}_0$:

$$\hat{\mathbf{u}} = \hat{\mathbf{u}}_0 + \boldsymbol{e}_0 \tag{3.10}$$

This correction is directly associated to the defect $\mathbf{d}_0$:

$$\begin{aligned} \boldsymbol{e}_0 &= \hat{\mathbf{u}} - \hat{\mathbf{u}}_0 && (3.11)\\ \hat{\mathbf{K}}\,\boldsymbol{e}_0 &= \hat{\mathbf{K}}\,\hat{\mathbf{u}} - \hat{\mathbf{K}}\,\hat{\mathbf{u}}_0 && (3.12)\\ &= \hat{\mathbf{r}} - \hat{\mathbf{K}}\,\hat{\mathbf{u}}_0 && (3.13)\\ &= \mathbf{d}_0 && (3.14) \end{aligned}$$

With $\hat{\mathbf{K}}_\varepsilon\,\mathbf{d}_0 \approx \hat{\mathbf{K}}\,\mathbf{d}_0$ an approximation $\boldsymbol{e}'_0$ for the correction $\boldsymbol{e}_0$ can be computed solving again the linear system (3.8) with a different right-hand-side:

$$\boldsymbol{e}'_0 \text{ from } \hat{\mathbf{K}}_\varepsilon\,\boldsymbol{e}_0 \approx \mathbf{d}_0 \tag{3.15}$$

If the solution to (3.8) was calculated using a factorization of $\hat{\mathbf{K}}_\varepsilon$, the approximate correction $\boldsymbol{e}'_0$ can be determined easily performing a second back substitution. Now, a new approximate correction to the previous approximate correction can be calculated, leading to the following iteration:

$$\begin{aligned} \mathbf{d}_i &= \mathbf{d}_{i-1} - \hat{\mathbf{K}}\,\boldsymbol{e}'_{i-1} && (3.16)\\ \boldsymbol{e}'_i &\text{ from } \hat{\mathbf{K}}_\varepsilon\,\boldsymbol{e}_i \approx \mathbf{d}_i && (3.17)\\ \hat{\mathbf{u}}_i &= \hat{\mathbf{u}}_{i-1} + \boldsymbol{e}'_{i-1} && (3.18) \end{aligned}$$

The iteration is performed until the corrections become sufficiently small:

$$\left| \frac{\boldsymbol{e}'_i\,\hat{\mathbf{K}}\,\boldsymbol{e}'_i}{\hat{\mathbf{u}}_i\,\hat{\mathbf{K}}\,\hat{\mathbf{u}}_i} \right| < \varepsilon_{th} \tag{3.19}$$

The solution of the original system 3.1 is then given by

$$\mathbf{u} = \mathbf{T}\,\hat{\mathbf{u}} \tag{3.20}$$

ansatz degree p	*DOF*	*iterations*	*solution defect* d^2	*iteration t./ solving t.*
0	42	1	$5.1 \cdot 10^{-28}$	0%
1	136	1	$7.0 \cdot 10^{-27}$	0%
2	298	1	$1.7 \cdot 10^{-26}$	0%
3	542	1	$2.5 \cdot 10^{-25}$	8%
4	882	983	$7.0 \cdot 10^{-17}$	99%
5	1332	379	$4.2 \cdot 10^{-16}$	96%
6	1906	567	$7.5 \cdot 10^{-17}$	96%
7	2618	590	$1.5 \cdot 10^{-16}$	95%
8	3482	457	$1.3 \cdot 10^{-16}$	93%
9	4512	632	$1.3 \cdot 10^{-16}$	94%
10	5722	1078	$2.3 \cdot 10^{-16}$	96%

Table 3.1: Example for perturbed matrix + post iteration solution method

Table 3.1 shows a numerical example of this solving strategy. The solved linear system arises from a plane elasticity problem on a L-shaped domain, using a *hp*-refined discretization. The polynomial degree p is equal to the number of mesh refinements around the re-entrant corner.

Factorization is performed using a Cholesky decomposition. The decomposition, as well as the matrix-vector-products, are performed applying banded storage (see 3.2.4).

The addition ε to the scaled main diagonal entries equals to 10^{-10}. Iteration is continued until the correction (3.19) drops below $\varepsilon_{th} = 10^{-12}$. For small polynomial degrees ($p < 4$), the computational costs of post iteration steps required are negligible. This resembles the results found in [DBO00].

For higher polynomial degrees, however, the number of post iteration steps necessary increases dramatically. The time spent during iteration is over ten times larger than the time required for

factorization. However, the defect $d^2 = (\mathbf{r} - \mathbf{K}\,\mathbf{u})^2$ of the solution finally found is acceptably small.

3.2 Givens *QR* Factorization

A decomposition of a matrix $\mathbf{K}$ into an orthogonal matrix $\mathbf{Q}$ ($\mathbf{Q}^T = \mathbf{Q}^{-1}$) and an upper triangular matrix $\mathbf{R}$, such that $\mathbf{K} = \mathbf{Q}\,\mathbf{R}$ is called a *QR* factorization.

This factorization has many applications in numerics, e.g. eigenvalue problems, linear equation systems, and least squares problems. The *QR* factorization of a quadratic matrix exists independently of the matrix rank, so it can be used to solve a rank deficient equation system.

To perform a *QR* factorization, several algorithms can be used, e.g. Householder or Givens transformation ([GL89]). All of these methods have the computational complexity of $O(n) = n^3$ typical for non-iterative linear solvers. Compared to the Gaussian elimination, however, a *QR* factorization using Givens transformation is ~ 10 times more expensive ([Sch97]). A parallel implementation is therefore highly desirable. While being slightly more efficient, Householder reflections are not so well suited for parallelization, because they affect more matrix elements in each step. Givens rotations allow for a finer granularity, and are therefore often used for dense and sparse parallel *QR* factorization implementations ([SK78], [CR86], [CD94], [TDZ96]).

3.2.1 Overview of the Solution Process

To solve the linear equation system

$$\mathbf{K}\,\mathbf{u} = \mathbf{r} \tag{3.21}$$

the matrix $\mathbf{Q}$ is left-factorized from both sides leading to

$$\mathbf{R}\,\mathbf{u} = \tilde{\mathbf{r}} \tag{3.22}$$

with $\tilde{\mathbf{r}} = \mathbf{Q}^T \mathbf{r}$ and $\mathbf{R} = \mathbf{Q}^T \mathbf{K}$ using $\mathbf{Q}^T = \mathbf{Q}^{-1}$. Only $\mathbf{R}$ and $\tilde{\mathbf{r}}$ are necessary in determining $\mathbf{u}$ using back-substitution, so $\mathbf{K}$ and $\mathbf{r}$ can be overwritten during the transformation process (in-place algorithm). $\mathbf{Q}$ is not needed for this application in an explicit form.

3.2.2 Givens Rotation

A Givens rotation is the elementary step of the transformation. Two rows (i, j) of the matrix $\mathbf{K}$ are multiplied in step t with an orthogonal two-dimensional rotation matrix $\mathbf{U}_t$, such that in column k the element $\mathbf{K}_{j,k}$ becomes zero.
In step t, the rotation matrix $\mathbf{U}_t$

$$\mathbf{U}_t = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \tag{3.23}$$

is determined from the matrix entries

$$c = \frac{\mathbf{K}_{i,k}}{\sqrt{\mathbf{K}_{i,k}^2 + \mathbf{K}_{j,k}^2}}, s = \frac{\mathbf{K}_{j,k}}{\sqrt{\mathbf{K}_{i,k}^2 + \mathbf{K}_{j,k}^2}} \tag{3.24}$$

So one of Givens rotation reads as:

$$\begin{pmatrix} \mathbf{K}_{i,1\ldots n} & \mathbf{r}_i \\ \mathbf{K}_{j,1\ldots n} & \mathbf{r}_j \end{pmatrix}_{t+1} = \mathbf{U}_s \cdot \begin{pmatrix} \mathbf{K}_{i,1\ldots n} & \mathbf{r}_i \\ \mathbf{K}_{j,1\ldots n} & \mathbf{r}_j \end{pmatrix}_t \tag{3.25}$$

Given that

$$\mathbf{K}_{i,l} = \mathbf{K}_{j,l} = 0 \text{ for } l = 1 \ldots k-1 \tag{3.26}$$

after the transformation holds

$$\mathbf{K}_{i,l} = \mathbf{K}_{j,l} = \mathbf{K}_{j,k} = 0 \tag{3.27}$$

introducing one additional zero.

3.2.3 Sequential Givens Transformation

The order in which the single rotations are performed cannot be chosen freely. (3.26) has to be satisfied to avoid introducing non-zero elements left of row k.

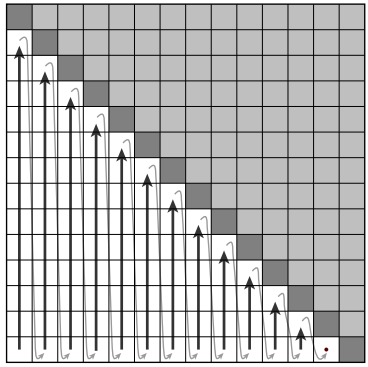

Figure 3.1: Possible sequence of Givens rotations

Figure 3.1 shows a possible sequence of rotations leading to complete factorization of a dense matrix. In each step, the row containing the element $\mathbf{K}_{j,k}$ to become zero can be combined with any row above this element up to the main diagonal.

3.2.4 Banded Matrix

The matrices arising from a GFEM problem are not fully populated. Like FEM matrices, the non-zero elements are typically close to the main diagonal, if a suitable numbering of unknowns is chosen. Such a numbering is usually gained by performing a bandwidth optimization using the graph of the discretization mesh or the matrix connectivity graph—here a reverse Cuthill-McKee ([CM69], [Sch84]) reordering is applied. Figure 3.3 shows on the left the

profile of the optimized matrix resulting from the discretization in Figure 3.2.

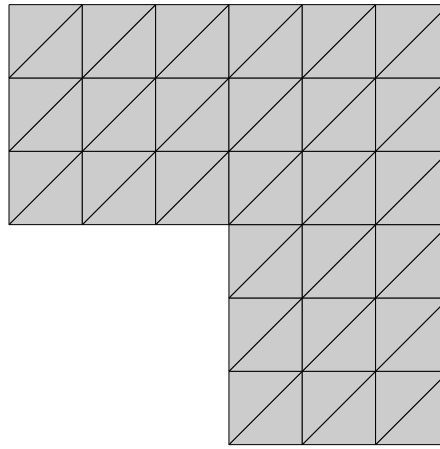

Figure 3.2: Example domain (p=8, plane Poisson problem)

Memory space and computation time can be saved exploiting this banded structure. Instead of storing the entire matrix, only a vector of sub-rows is stored. Every sub-rows extends from the first to the last non-zero element, omitting all out-of-band zeros.

The Givens transformation can easily take advantage of this band structure by omitting unnecessary rotations of zero elements. Some out-of-band zeros are populated after the transformation, however. Figure 3.3 shows on the right hand side the increase of bandwidth in the transformed matrix $\mathbf{R}$ (the newly created zeros below the main diagonal still consume storage).

If an element $\mathbf{K}_{i,j}$ below the main diagonal ($i > j$) is in the band, all sub-rows $k = j \ldots i-1$ above this element up to the main diagonal have to be expanded, if necessary (i.e. if they are shorter), to the length of sub-row i. This allows the combination of any two sub-rows needed to perform the Givens rotations that are necessary.

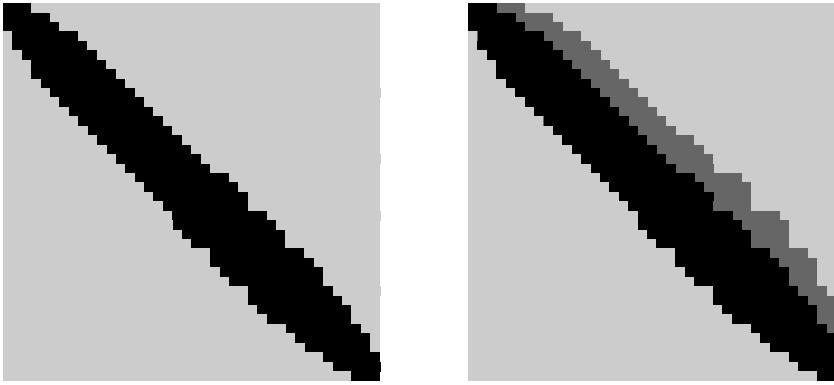

Figure 3.3: Global matrix before and after Givens transformation (1800 DOF)

3.2.5 Hybrid Parallelization of Givens Rotations

Each Givens rotation affects only two rows. This allows for a parallel implementation of the algorithm. Instead of performing one rotation after another in the current column, as illustrated in Figure 3.1, the sub-rows containing non-zero elements are divided into a certain number of subsets. On each subset, Givens rotations can be applied in parallel, until every subset has only one non-zero element in the current column left. Then, the left subset rows are combined with the row holding the main diagonal.

This parallelization idea can be realized using the distributed memory paradigm. Every node holds a subset of the matrix. The bulk of necessary Givens rotations can be performed locally. Only during the last combination step does the remaining sub-row have to be exchanged.

If the nodes themselves support parallelization using the shared memory paradigm (i.e. multithreading) or vectorization, the Givens rotations can be parallelized further locally. The machine may

provide an efficient (vectorized) implementation for the matrix-matrix product $\mathsf{U} \cdot \mathsf{K}$ or independent rotations can be performed locally in parallel.

Distributed Memory Parallelization using MPI

If an algorithm can be parallelized in a way that avoids frequent exchange and large amounts of interchanged data (weakly coupled problem), it can be effectively parallelized using message passing. The main advantage of the message passing paradigm is the availability of suitable hardware—efficient implementation is possible on distributed and shared memory machines as well. Thread-based multiprocessing in contrast can only be implemented efficiently on shared memory hardware.

MPI (Message Passing Interface) is a widely available programming interface for implementing parallel algorithms using message passing. It provides a set of standardized library subroutines for *C* and *Fortran* and some implementation dependent tools to administrate the parallelized programs.

During a parallel computation, a certain number of concurrent processes, identified by a unique ID, are running. These processes can only interact by interchanging messages. Using MPI, one often writes one program, which is started multiple times to create the required number of processes.

MPI is used to implement the top-level parallelization of the Givens transformation. A master-slave concept is used: one master process holding the matrix to be factorized and responsible for coordinating the distributed calculation interacts with a number of slaves performing the actual computational work.

Parallel Givens Transformation Algorithm using MPI The following steps describe the parallel Givens transformation using n_p MPI processes.

1. The master (process 0) assigns every sub-row of the matrix to a slave (process $1 \ldots n_p - 1$) in a striped pattern:

sub-row	1	2	3	4	5	6	7	...
process	1	2	3	...	$(n_p - 1)$	1	2	...

2. The sub-rows are distributed to the respective slaves.

3. Master and slaves exchange their roles (master, participating slave holding the main diagonal element, ordinary participating slave, uninvolved slave) for the current column.

4. Participating slaves eliminate all but one non-zero of the current column in the assigned rows.

5. The participating slaves perform a tree-reduce operation, as described in [Pac97], with the remaining rows:

   An ID is assigned to every participating slave. The slave process holding the main diagonal element gets the ID 0. Further participating slaves are numbered ascending.

   Depending on the count of participating processes, a number of stages are performed in descending order.

   During stage s, a process performs a master rotation with process ID $+ 2^s$ (i.e. sends away its last non-zeroed sub-row and receives an updated one) if its ID $< 2^s$ and the slave process is existent.

   A process performs a slave rotation (i.e. receives a non-zeroed sub-row, performs a Givens rotation using the received row so that its own last non-zeroed row becomes zero and sends the updated sub-row back) with process ID $- 2^s$ if $2^s \leqslant$ ID $< 2^{s+1}$.

   Figure 3.4 shows an example of the tree-reduce operation for seven processes ($0 \ldots 6$) using 3 stages ($3 \ldots 0$).

   After the tree reduce operations, all remaining sub-rows below the main diagonal carry a zero in the current column.

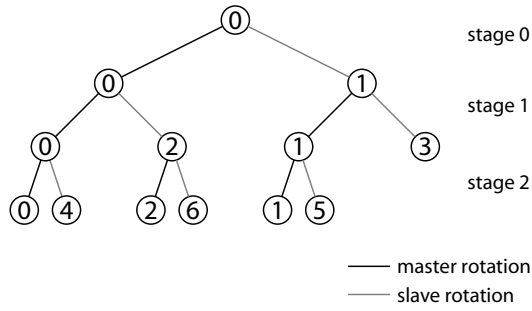

Figure 3.4: Tree reduce operation example

6. Continue with step three for the next column or next step if no columns are left.
7. Master process collects matrix rows from slaves.

Speedup of parallel Givens Factorization Figure 3.5 shows the speedup $S(n, n_p)$ of the MPI parallel Givens factorization for a *p*-refined calculation sequence of a plane elasticity problem on a quadrilateral domain discretized with $10 \cdot 10 \cdot 2$ triangular elements. The number of unknowns stating the problem size n range from $n = 1404$ for $p = 3$ to $n = 10746$ for $p = 9$. Performance measurements were performed from one ($n_p = 1$) up to ten processors ($n_p = 10$). $t_s(n)$ denotes the factorization time on a uniprocessor, $t(n, n_p)$ on a multiprocessor machine.

$$S(n, n_p) = \frac{t_s(n)}{t(n, n_p)} \tag{3.28}$$

The speedup of the parallel factorization depends mainly on the bandwidth of the matrix. The bandwidth depends mostly on the

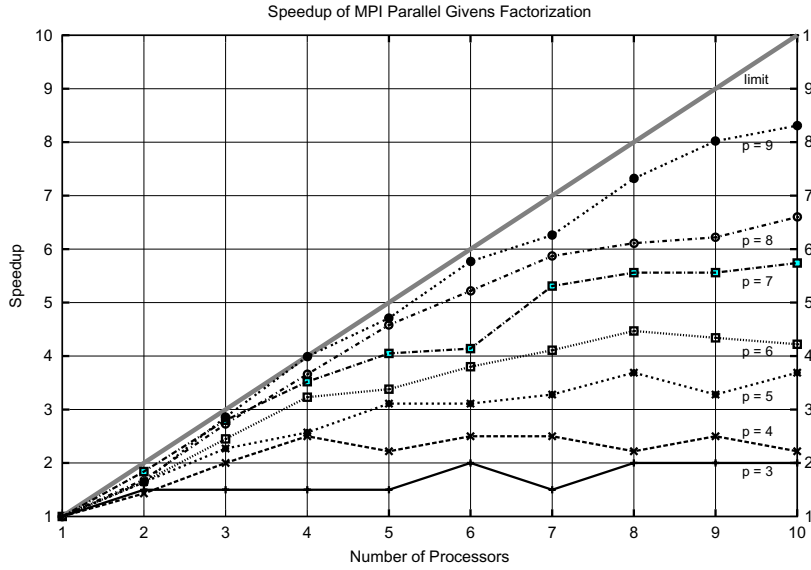

Figure 3.5: Speedup of MPI parallel Givens factorization

used ansatz degree. For a higher order ansatz, the algorithm scales pretty well, following very closely the theoretical upper speedup limit up to a number of six processors, while for a lower order ansatz multiple processors have hardly any effect on computation time.

3.3 Sparse Multifrontal Gaussian Elimination/*HSL MA27*

Some commercially available direct solvers for indefinite linear systems can also be used to solve the equation systems arising from a GFEM discretization. One of the solvers also proposed in [DBO00] is the *HSL MA27* available from [hsl]. This method is further described in [DR83]. An overview on the topic of sparse direct solvers can be found in the review article [Liu92].

3.4 Comparison of the Different Methods

All of the methods examined share (theoretically) a computational complexity of $O(n) = n^3$. This holds even for banded storage if the increased number of degrees of freedom n arises from a *p*-refinement as the bandwidth is increased in this case.

Figure 3.6 shows a side-by-side comparison of the timed needed to solve an indefinite system using the three different methods discussed. A plane elasticity problem with a point singularity arising from a re-entrant corner was discretized using a *hp*-graded mesh. The polynomial degree p indicated in the diagram refers to the unrefined elements. Subsequent refinement levels use a lower ansatz degree down to $p = 0$—thus p is also the number of mesh refinements towards the singularity.

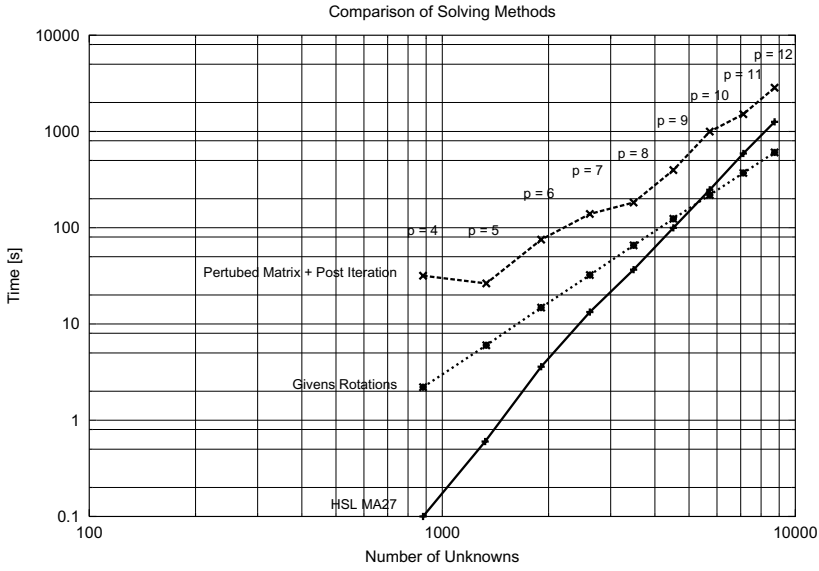

Figure 3.6: Performance comparison of different solving Methods

For a small and medium polynomial degree, *HSL MA27* performs much better than Givens rotations. For large polynomial degrees however, Givens rotations catch up. If the scalability on a parallel computer is taken into account, this breakeven will be reached even earlier.

The first method, involving post iteration on the solution of a modified linear system, performs poorly already on medium polynomial degree discretizations.

Table 3.2 gives a numerical estimate derived from the observed runtimes in two successive steps for the polynomial complexity $O(n) = n^p$. While the post iteration scheme behaves rather irregularly, a clear conclusion can be drawn for the other two methods. Givens Rotations behave as expected, showing a complexity below three for the used *hp*-refinement—one would expect a value around three for a pure *p*-refinement. *HSL MA27* exhibits a much higher estimated complexity which ranges close to four. This is evidence of increasing difficulty dealing with linear dependencies.

DOF	*Perturbed Matrix/ Post Iteration*		*HSL MA27*		*Givens Rotations*	
	t [s]	p_{est}	t [s]	p_{est}	t [s]	p_{est}
882	32		0		2	
1332	26	-0.4	1	4.3	6	2.4
1906	75	2.9	4	5.0	15	2.5
2618	139	1.9	13	4.1	32	2.4
3482	183	1.0	37	3.6	66	2.5
4512	397	3.0	100	3.9	124	2.5
5722	998	3.9	248	3.8	219	2.4
7126	1514	1.9	591	4.0	371	2.4
8738	2846	3.1	1258	3.7	605	2.4

Table 3.2: Performance comparison and polynomial complexity estimation

Chapter 4

Implementation of the GFEM

The programming language chosen to implement GFEM is *C++* [Str00]. While providing many object oriented features, this compiled language provides a good compromise between abstraction and performance. As a hybrid language, the programmer can bias his design in a wide range between these two contradicting concepts.

The programmatic design is primarily headed towards being as general as possible and easily extensible. Performance was only a secondary issue, but care has been taken not to trade more than necessary for design reasons. The impact of some of the decisions is profiled and discussed in this chapter.

For an alternative implementation of *GFEM* realized in *FORTRAN95* see [SCB02]. The implementation described there uses lower order polynomial approximations and allows for meshes which share no common boundary with the domain focussing on geometries containing large numbers of voids or cracks.

4.1 Function Classes

The key advantage of GFEM is the ability to include almost arbitrary ansatz functions into the approximation. To provide the possibility of inserting and examining analytical and numerical derived special functions, the program is based on an object-oriented notion of functions, an idea also proposed in [SBC98].

4.1.1 Scalar Multivariate Function Interface— `SFunction`

As a strongly typed language, *C++* requires the usage of common base types if different classes of objects should be used in a uniform manner[1]. Also not differentiated by the syntax of the *C++* language[2] we can distinguish two different concepts for such a base type:

- Specification of an interface.
- Implementation of common functionality.

An interface only states the protocol, i.e. the names of the messages and their arguments an object shall provide. The implementation of this protocol is completely left up to the implementing objects. The interface itself does not provide any functionality. We implement interfaces in *C++* via classes containing solely pure virtual methods.

One of the basic building blocks of our object-oriented function concept is a scalar function with a variable number of real arguments (*C++*'s `double` base type). Figure 4.1 shows the interface

[1] For example, *Smalltalk* [GR83] as an untyped language, does not impose this requirement. Missing methods will only be detected at run-time in this case, however. It is very controversial if this additional compile-time check is worth the additional amount of coding effort.

[2] *Java* [Gos95] provides the special notion of an `interface` keyword.

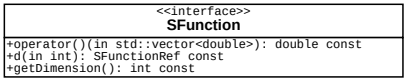

Figure 4.1: `SFunction` interface

that such an object shall implement. The function call operator (`(...)`) is used to evaluate the function at the given n-dimensional point. The dimensionality can be queried using the `getDimension` method. `d(n)` is used to query the function $f(\mathbf{x})$ for its partial derivative $\frac{\partial f}{\partial x_n}$ with respect to the nth component of the argument vector $\mathbf{x}$. In case of an incorrect dimensioned argument or a partial derivative of a wrong dimension, the behaviour of the object is unspecified. An empty reference is returned if the requested derivative is not available.

4.1.2 Smart Reference to scalar Function Object—`SFunctionRef`

C++ provides no means of automatic memory management. In some cases, the manual management of object lifetimes is an easy task, but for function objects, something more elaborate is required. As we will see later, functions can be combined in arithmetic expressions leading to new functions and are excessively passed around between objects. Often, a function object has more than one owner referring to it.

We use the technique of reference counting as implemented in [Jos99] `CountedPtr` class. `CountedPtr` resembles an ordinary *C++* pointer, but keeps track of the number of `CounterPtr` instances referring to a certain object. The object is deleted when it's no longer referred from any `CountedPtr` instance.

`SFunctionRef` (Figure 4.2) implements a smart reference (in the

sense of reference counting and the ability of assignment) to a `SFunction` object.

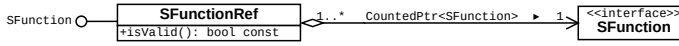

Figure 4.2: `SFunctionRef` smart reference

It achieves this by using an instance of `CountedPtr<SFunction>` delegating all methods of the `SFunction` interface to the referred object. Using the `isValid` method, one can check if the reference is empty or if it points to a valid object.

In the program, all instances of `SFunction` are managed by `SFunctionRef` smart references. `SFunctionRef` is therefore the result and the argument type used in all methods dealing with `SFunction`s.

4.1.3 Vector-valued Function Interface—`Function`

In general, vector-valued functions are implemented as an object behaving like an array of `SFunction`s. `Function` (Figure 4.3) pro-

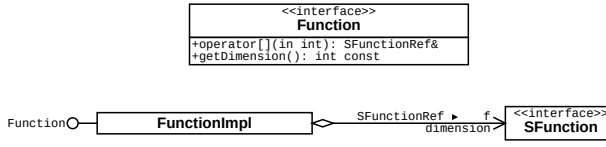

Figure 4.3: `Function` interface and `FunctionImpl` implementation

vides an array access operator (`[...]`) returning the corresponding scalar as a `SFunction`. Again, dimensionality can be queried using the `getDimension` method.

A straight-forward implementation of this interface is provided with `FunctionImpl`. This class holds an array of `SFunctionRef`s. Proper resource management for parameter passing, copy and assignment operations is provided by the smart references.

4.1.4 Arithmetic

To perform arithmetic operations with functions, objects representing the sum $(g + h)$ or product $(g \cdot h)$ of two functions are required (Figure 4.4). The respective operators ($\star$, $+$) are over-

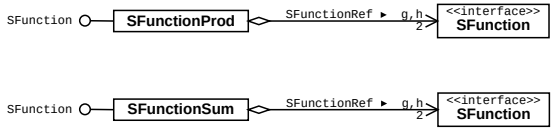

Figure 4.4: Objects representing products and sums of functions

loaded to create new instances of these classes and returning them as `SFunctionRef`.

If the associated functions are able to deliver derivatives, the arithmetic expression provides derivatives, too. For example, an instance of `SFunctionProd` creates a new expression representing the derivative according to the product rule ($(g \cdot h)' = g' \cdot h + g \cdot h'$) on demand.

As illustrated in Figure 4.5, the objects represent the evaluation tree of the arithmetic expression. To aid debugging of the resulting expressions, a textual representation of this tree can be printed on the console (see 4.1.10).

4.1.5 Function Proxy

Up until now, we have only discussed how to manage and associate existing objects implementing the `SFunction` interface. An

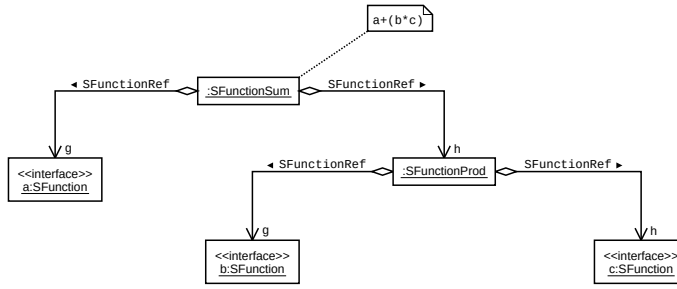

Figure 4.5: Objects are representing an evaluation tree

obvious way to create such an object is deriving from `SFunction` and providing the required methods.

This approach turned out to be very inconvenient, however. Consider, for example, an object representing a mapping between two coordinate systems. The components of forward and backward mapping should be available in the form of `SFunction` objects. When deriving the coordinate system from `SFunction`, this object represents only one function component, so some helper classes are required. These helpers could query the mapping object for the respective value and return it via the `SFunction` interface.

The natural way of implementing a function is to provide a method calculating the desired value. The `SFunctionProxy` templates store a pointer to our object and a pointer to the methods implementing the function.

To enable such a proxy object to deliver derivatives, the proxy object contains an array of `SFunctionRef`s. The methods implementing derivatives are also wrapped in `SFunctionProxy` objects and are assigned.

Depending on the dimensionality of the wrapped methods, different versions of the `FunctionProxy` are required. Figure 4.6 shows a proxy for two-dimensional functions.

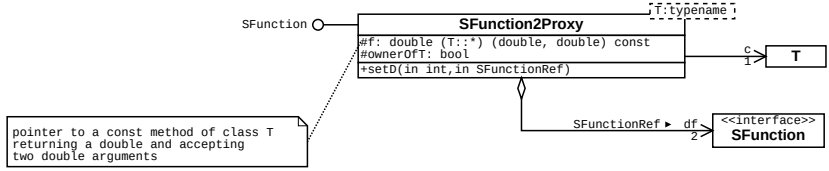

Figure 4.6: `SFunction2Proxy` for two-dimension methods

The flag `ownerOfT` indicates whether the proxy shall destroy the object it is pointing to at the end of its own life.

4.1.6 Polynomials

4.1.7 Univariate Polynomials—`Poly`

Univariate polynomials are represented in the class `Poly` (Figure 4.7) using an array of `double`s to store the coefficients c_i.

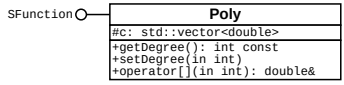

Figure 4.7: Univariate polynomial

$$p(x) = \sum_{i=0}^{n} c_i \cdot x^i$$

This coefficient representation can be efficiently evaluated using a

nested multiplication scheme:

$$\begin{aligned} p_n &= c_n \\ p_i &= c_i + x \cdot p_{i+1}, \quad i = n-1, n-2, \ldots, 2, 1, 0 \\ &\textit{with} \\ p(x) &= p_0 \end{aligned}$$

`Poly` directly implements the `SFunction` interface.
Since the arithmetic operations ($+$, $-$, $*$) and the derivation ($\frac{dp}{dx}$) can be performed explicitly, the corresponding operators are appropriately overloaded returning a new `Poly`[3].

4.1.8 Bivariate Polynomials—`PolyProduct`

`PolyProduct` implements a bivariate polynomial using two univariate polynomials for the respective directions (Figure 4.8). To

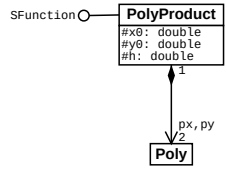

Figure 4.8: Bivariate polynomial

ease the usage of `PolyProduct` as an ansatz function, a basic coordinate system transformation (originating at (x_0, y_0) and scaled with factor h) is embedded:

$$u(x, y) = p_x((x - x_0)h) \cdot p_y((y - y_0)h)$$

`PolyProduct` directly implements the `SFunction` interface, too.

[3]These operators perform directly on `Poly`, not on smart references (`SFunctionRef`).

Partial derivatives of a `PolyProduct` ($\frac{\partial p}{\partial x}$, $\frac{\partial p}{\partial y}$) are again `PolyProduct`s and dynamically created on demand using the appropriate operations of `Poly`.

4.1.9 Set of *Legendre* Polynomials—`Legendre`

The `Legendre` class creates the set of *Legendre* polynomials up to a given degree `p`. When creating an instance of `Legendre`, *Bonnet*'s

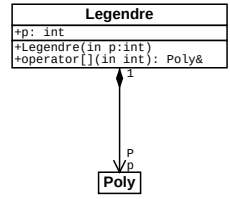

Figure 4.9: Set of *Legendre* polynomials

recursion formula is used to fill the array of polynomials:

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ (n+1)\,P_{n+1}(x) &= (2n+1)\,x\,P_n(x) - \\ &\qquad n\,P_{n-1}(x), \quad n = 1, 2, 3, \ldots, p-1 \end{aligned}$$

4.1.10 Analysing and Debugging

To assist inspecting and debugging the internal structures hidden behind `SFunctionRef` or `FunctionImpl`, the following routines,

- `SFunctionRef analyse(SFunctionRef f)` and
- `FunctionImpl analyse(FunctionImpl f)`,

are provided. A tree representation (see Figure 4.5) of the referenced object is printed on the console. The original reference is passed through to enable the usage of the debugging routine inside expressions.

This example output shows a two-dimensional vector-valued function. The first component consists of a product of two polynomials multiplied with an (unknown) object member function wrapped in a proxy. The second component is an empty reference.

```
FunctionImpl(dimension=2)
   SFunctionProd
      PolyProduct
         Poly x: 1
         Poly y: -0.5 + 0x + 1.5x^2
      SFunctionProxy
   invalid/empty function
```

4.2 GFEM in two Dimensions

4.2.1 Overview

After providing an overview of the general structure, this section will explain the implementation of the GFEM code in detail.

Figure 4.10 shows the main objects used to perform a GFEM calculation. `Domain` serves as a container class representing a discretized two-dimensional domain governed by the differential equation described in `DiffEq`. `Material` holds default properties for the material.

`Domain` aggregates sets of `Vertex` and `Triangle` objects. Every `Triangle` references its three corner vertices. Besides these three associations, no further relations are necessary to perform a GFEM calculation. This contrasts to conventional *p*-FEM, where the additional entity *edge* and more topological relations are required

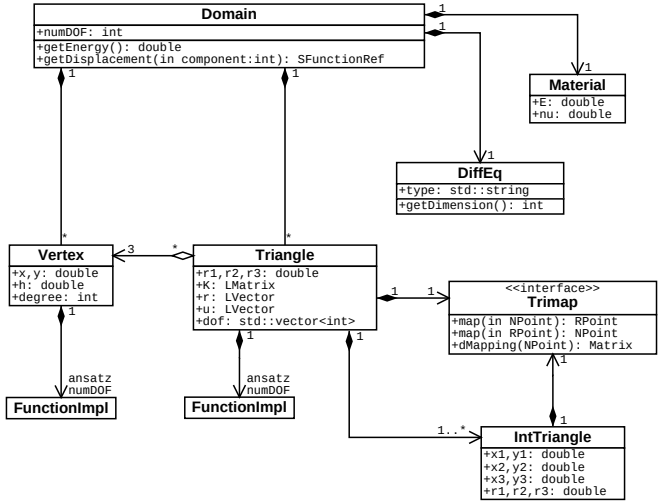

Figure 4.10: Overview of the GFEM implementation in two dimensions

to build a suitable discretization complicating the data structure significantly.

Figure 4.11 gives an overview of the calculation of one domain. A reader with conventional FEM experience will notice strong analogies to the steps normally performed in such codes.

`Vertex` initialization

`Vertex` contains an array of `n`-dimensional ansatz functions. In the first loop, every `Vertex` of the domain is initialized. The array is filled with an `n`-dimensional tensor product ansatz space build from multiplied *Legendre* polynomials up to the given degree `p` of the respective `Vertex`. Additional special functions—e.g. analytical partial solutions and numerical side calculations—are added as well. A unique (which respect to the domain) global identifier (global DOF id) is associated which each ansatz function.

`Triangle` initialization

The second loop initializes the `Triangle`s. Each `Triangle` queries its three corner vertices for the set of ansatz functions associated with this `Vertex`. After multiplying these functions with the respective hat function, they are added to the array containing the ansatz functions of the `Triangle`. The assigned global DOF ids a stored as well.

After this step, the `Triangle`s contain all information necessary to perform integration of the weak form. While performing this integration, the results are locally stored.

Assembling and Solving

The last loop assembles the local stiffness matrices into the global linear equation system. After solving the global system, the resulting displacement components are redistributed to the corre-

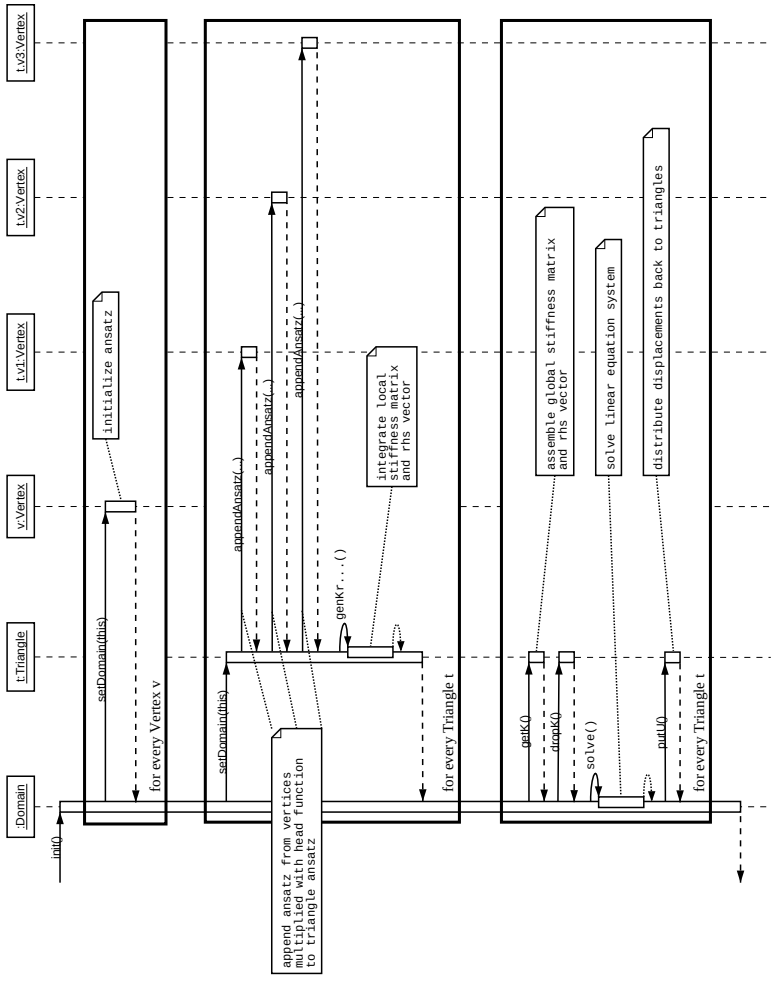

Figure 4.11: Outline of the calculation for a domain

sponding `Triangle`s. Now `Domain` (and `Triangle`s) are ready to be queried about properties of the solution, like displacement field, global energy and stresses.

4.2.2 Input and Output

XML Parser Concepts

Interfacing the program to other tools and to provide persistence involves loading and storing data from and to permanent storage as needed. *XML* is chosen because it is a well-standardized format supported by many ready-to-use tools and libraries.

Another advantage of *XML* (*eXtensible Markup Language*) is the easy realization of optional fields in the data file. Also, later extensions of the format are forward and backward compatible, if done with some care.

XML parsers are available in two main flavours:

1. Event-driven parsers
2. Object oriented parsers

Object-oriented parsers—e.g. *DOM*—read the entire document building a tree of objects. These objects can be traversed, queried and modified.

Event-driven parsers generate events (e.g. opening tag, closing tag and content) out of the read stream contents. The main advantage of the event-driven approach is the ability to directly create and modify custom data structures. With *DOM* on the other hand, one would have to work with two groups of objects—*DOM* and custom objects.

Parser Generator

To ease the task of interfacing an object-oriented custom data structure to an event driven *XML* parser, a code generator was

developed. A tree-like structure of objects with their respective properties is described using an *XML* file. A set of interfaces mainly consisting of `set...` and `get...` methods, which have to be implemented by the custom objects, is generated.

The generated parser code creates new instances of custom objects and sets the values of properties while reading the file. To enable the parser to create custom objects, an instance of an object factory [GHJV94] must be provided. Construction of each object is finished with a call to its `init` function.

The user is required to divert from the conventional constructor style, which creates complete objects in a single step. Here, first a default object is created; then none, some or all properties are modified as needed; finally, the creation process is completed with a call to the `init` function.

Serialization of the data is possible if the object state is fully defined by the properties known to the parser generator.

For more information on generated classes and their relations see Appendix B: 'xmlom — XML Object Manager.'

Document Data Structure

Besides properties holding primitive types (e.g. `double`, `string`, `bool`), an object can aggregate complex types (other objects). The parser generator supports the following aggregating data structures:

- multiple child-objects identified by a numeric ID (`int`), stored in a `map`
- multiple ordered anonymous child-objects, stored in a `list`
- one embedded child-object, which may be missing

The *C++* parser generator implements the associations to aggregated child-objects via pointers.

Further associations between custom objects should be implemented using a property holding the ID of the referred object stored in a `map`. This ensures the ability to load and store the association. To improve performance, the pointer returned from the `map`-lookup may be cached during the `init`-process.

While the main structure of custom data objects using generated aggregating associations is tree-like[4] in order to ensure easy serialization[5], the user may arbitrarily add further ID-based associations allowing general object structures (e.g. cyclic references).

The generated aggregations determine the lifetime of the document objects. On destruction of the document, all contained objects are deleted.

GFEM Document Specification

After pointing out the general concept of serialization, the specific GFEM file format is explained.

Example Figure 4.12 shows a discretization of a quadrilateral domain using two triangles. The border of the domain is subject to homogeneous Dirichlet boundary conditions. Furthermore, the domain is subject to a constant area loading with $p = -1$.

The XML document representing the discretization is given here:

```
<?xml version="1.0"
      encoding="iso-8859-1"?>
<gfem>
   <domain id="1">

      <diffeq>
         <type>planepoisson</type>
      </diffeq>
```

```
      <vertex id="1">
         <x>0</x> <y>0</y> <h>10</h>
         <degree>9</degree>
         <mulfunc>
            <element>1</element>
            <x1>10</x1><y1>10</y1>
            <x2>0</x2><y2>10</y2>
            <x3>0</x3><y3>0</y3>
```

[4] tree: connected acyclic graph

[5] to serialize an object: convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object[Lee02]

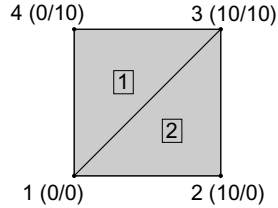

Figure 4.12: Simple GFEM discretisation example

```
   </mulfunc>
   <mulfunc>
      <element>2</element>
      <x1>10</x1><y1>10</y1>
      <x2>0</x2><y2>0</y2>
      <x3>10</x3><y3>0</y3>
   </mulfunc>
</vertex>

<vertex id="2">
   <x>10</x> <y>0</y> <h>10</h>
   <degree>9</degree>
   <mulfunc>
      <x1>10</x1><y1>10</y1>
      <x2>0</x2><y2>0</y2>
      <x3>10</x3><y3>0</y3>
   </mulfunc>
   <mulfunc>
      <x1>0</x1><y1>0</y1>
      <x2>10</x2><y2>0</y2>
      <x3>10</x3><y3>10</y3>
   </mulfunc>
</vertex>

<vertex id="3">
   <x>10</x>
   <y>10</y>
   <h>10</h>
   <degree>9</degree>
   <mulfunc>
      <element>1</element>
      <x1>0</x1><y1>0</y1>
      <x2>10</x2><y2>10</y2>
      <x3>0</x3><y3>10</y3>
   </mulfunc>
   <mulfunc>
      <element>2</element>
      <x1>0</x1><y1>0</y1>
      <x2>10</x2><y2>0</y2>
      <x3>10</x3><y3>10</y3>
   </mulfunc>
</vertex>

<vertex id="4">
   <x>0</x> <y>10</y> <h>10</h>
   <degree>9</degree>
   <mulfunc>
      <x1>10</x1><y1>10</y1>
      <x2>0</x2><y2>10</y2>
      <x3>0</x3><y3>0</y3>
   </mulfunc>
   <mulfunc>
      <x1>0</x1><y1>0</y1>
      <x2>10</x2><y2>10</y2>
      <x3>0</x3><y3>10</y3>
   </mulfunc>
</vertex>


<triangle id="1">
   <n1>1</n1>
   <n2>3</n2>
   <n3>4</n3>
   <areaload>
      <p>-1</p>
   </areaload>
</triangle>

<triangle id="2">
   <n1>1</n1>
   <n2>2</n2>
   <n3>3</n3>
```

```
      <areaload>
         <p>-1</p>
      </areaload>
   </triangle>
```

```
   </domain>
</gfem>
```

Minimal Document A minimal document has to contain a header specifying the type (e.g. version) and one pair of matching `gfem` tags surrounding the contents (which are intentionally not present in this example).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<gfem>
</gfem>
```

GFEM Document Structure A document is built from an arbitrary number of *discretized* domains.

Every aspect of the discretization—like mesh structure, degree and properties of the ansatz function, numerical integration procedures and boundary conditions—is specified in the input file. Figure 4.13 shows a structural overview of the domain description.

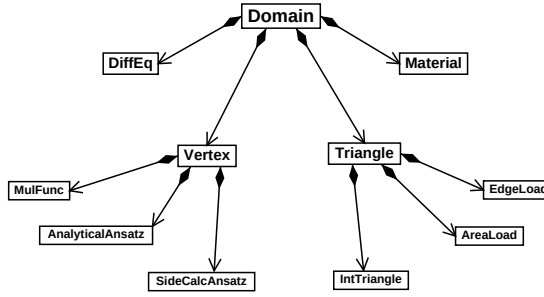

Figure 4.13: GFEM document overview

Domain			
Subtypes			
Tag	*Description*	*Storage*	*optional*
`material`	material properties	single	o[6]
`diffeq`	governing differential equation	single	
`vertex`	vertex	map	
`triangle`	triangular element	map	

Table 4.1: Domain Tag

Domains The individual domains are identified with a numeric ID given in the opening tag. This ID is used to refer to a domain in case its solution is used as a side calculation for another domain. Every domain is governed by a specified partial differential equation and is subject to the stated load case[7].

Vertices The vertices are holding—beside their geometric location—all information concerning the ansatz. The base of the ansatz is a set of Legendre polynomials up to the degree `p` originating from (V_x, V_y) and scaled to the characteristic size `h` to improve numerical characteristics.

It is possible to enrich the ansatz with special functions. These functions may be determined analytically or numerically.

Essential boundary conditions are imposed using the characteristic function method[BBO02]. This characteristic function is described by a set of functions multiplied onto (almost[8]) every

[6] required for plane elasticity

[7] Multiple load cases for one domain are not supported—multiple domains have to be used instead.

[8] numerical or analytical special functions should fulfil essential boundary conditions by construction and are therefore not subject to characteristic function multiplication

Vertex			
Properties			
Tag	*Description*	*Type*	*optional*
`x`	V_x coordinate	double	
`y`	V_y coordinate	double	
`h`	characteristic size `h`	double	
`degree`	polynomial ansatz order `p`	int	
Subtypes			
Tag	*Description*	*Storage*	*optional*
`mulfunc`	characteristic function components	list	•
`sidecalcansatz`	numerical side calculation ansatz	list	•
`analyticalansatz`	analytical ansatz	list	•

Table 4.2: Vertex Tag

function of the ansatz.

Triangles A triangle refers to its three nodes by ID. This is all of the topological information needed to construct a two-dimensional GFEM discretization.

The edges of the triangular element may be circular arcs. For $r_i > 0$, the edge bends outwards, for $r_i < 0$, the edge bends inwards towards the triangle. The resulting functions cannot be integrated exactly with the used Gauss'ian tensor product quadrature scheme. In this case one has to specify the *additional* number of integration points to be used.

The standard Gauss'ian integration scheme performs badly for singular special functions. In this case, one should use the *hp*-refined recursive integration algorithm. This method is enabled by inserting `<inttype>hp</inttype>`. The integration triangle is recursively refined towards node 2 using the number of levels specified with the `hpintdepth` tag. On the coarsest level, the same number of integration points as in the case of standard Gauss'ian integration is used. On each subsequent level the number of integration points decreases by one. If necessary, this number is increased on the coarsest level not to become zero or less at the finest one.

Optionally, a uniform *h*-refinement may be applied using the `uintdepth` tag. In this case *hp*-refinement is applied after the specified depth of uniform refinement. Setting `hpintdepth` or `uintdepth` to zero, a pure *h*- or pure *hp*-refined integration is performed.

Using the `<inttype>adaptive</inttype>` keyword, an adaptive *h*-refined integration scheme is enabled. In this case, the integration triangle is recursively refined until the error estimated using the results of two subsequent levels drops below the value

[9] required for *hp*-integration
[10] required for *h-adaptive*-integration

Triangle			
Properties			
Tag	*Description*	*Type*	*optional*
`n1`	node 1	int	
`n2`	node 2	int	
`n3`	node 3	int	
`r1`, `r2`, `r3`	edge radii	double	•
`overint`	additional number of integration points	int	•
`inttype`	integration type	string	•
`hpintdepth`	depth of recursive *hp*-integration	int	○[9]
`uintdepth`	depth of recursive *h*-integration	int	•
`accuracy`	accuracy of adaptive *h*-integration	int	○[10]
Subtypes			
Tag	*Description*	*Storage*	*optional*
`inttriangle`	integration triangulation	list	•
`areaload`	area loading (plane poisson)	list	•
`edgeload`	edge traction loads (plane elasticity)	list	•

Table 4.3: Triangle Tag

specified using `accuracy` or the maximal level of refinements is reached.

Normally, the whole area covered by the triangular element is integrated. To integrate over only a part of this area (e.g. to discretize a hole in the domain without representation in the element triangulation), one can specify that an integration triangulation is to be used instead.

DiffEq			
Properties			
Tag	*Description*	*Type*	*optional*
`type`	Name of partial differential equation	string	

Table 4.4: PDE Tag

Governing Partial Differential Equation Two types of governing partial differential equations are supported:

1. plane isotropic Poisson problem: `<type>planepoisson</type>`
2. plane isotropic elasticity problem: `<type>planestress</type>`

Material			
Properties			
Tag	*Description*	*Type*	*optional*
`e`	Young's modulus E	double	
`nu`	Poisson ratio ν	double	

Table 4.5: Material Tag

Material For an isotropic elastic problem, Young's modulus and the Poisson ratio of the used material must be specified.

MulFunc			
Properties			
Tag	*Description*	*Type*	*optional*
`x1`, `x2`, `x3`, `y1`, `y2`, `y3`	plane defining coordinates	double	
`r1`, `r2`, `r3`	edge radii	double	•
`type`	type of plane	string	•
`element`	affected element[11]	int	•

Table 4.6: Characteristic Function Component

Characteristic Functions The characteristic function used to enforce essential boundary conditions is built from parts of the hat functions used by the chosen partition of unity.

In the simplest case, a linear function is used. x_i and y_i coordinates of three points have to be specified. The function is determined to fulfil $f(x_2, y_2) = f(x_3, y_3) = 0$, $f(x_1, y_1) = 1$.

To describe a piecewise linear function, all pieces are listed. Effects of every function part are limited to the triangular element stated in the `element` tag. Care should be taken that the described function is C_0 continuous.

To enforce essential boundary conditions on curved edges, a blended hat function part can be used by specifying the respective radii r_i. Contrary to a linear function, the blended hat function part cannot be evaluated in the entire $\mathbb{R}^2$, only inside the blended triangle described by the points $V_{1...3}$.

[11] if missing all elements adjacent to respective node are affected

SideCalcAnsatz			
Properties			
Tag	*Description*	*Type*	*optional*
`domain`	ID of side calculation domain	int	
`x1`, `x2`, `x3`, `y1`, `y2`, `y3`	points in side calculation domain	double	
`x1s`, `x2s`, `x3s`, `y1s`, `y2s`, `y3s`	points in ansatz (current) domain	double	

Table 4.7: Side Calculation Ansatz

Side Calculations The displacement field of another domain's solution can be used as an ansatz function. The mapping used to place the side calculation is defined by three point-pairs. The points (x_1, y_1), (x_2, y_2), (x_3, y_3) located in the side calculation are mapped to the points (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3) in the using domain, respectively.

Analytical Enrichment An analytical solution is mapped into the domain the same way as a numerical side calculation.

The following analytical functions can be selected with the respective identifier:

- `EdgeSingularity`: $f(r, \theta) = \sqrt{3}\, r^\lambda \cos(\lambda\,\theta) + r^\lambda \sin(\lambda\,\theta)$ with $\lambda = \frac{2}{3}$. Singularity of Poisson's problem at two edges subject to Dirichlet boundary conditions with included angle $\varphi = 90°$.

[12] required for *hp*-integration
[13] required for *h-adaptive*-integration

AnalyticalAnsatz			
Properties			
Tag	*Description*	*Type*	*optional*
`type`	Identifier of analytical ansatz function	string	
`x1`, `y1`, `x2`, `y2`, `x3`, `y3`	points in analytical function coordinate system	double	
`x1s`, `y1s`, `x2s`, `y2s`, `x3s`, `y3s`	points in ansatz (current) domain	double	

Table 4.8: Analytical Ansatz

Integration Sub-Triangles If the integration should not be performed over the whole triangular element area (default), a list of sub-triangles covering the area to be integrated is specified. Type and accuracy of the integration scheme have to be specified for every integration triangle in the same way as for a normal, fully integrated triangle. The integration type arguments of the triangular element are ignored in this case.

Loading For plane elasticity, constant traction loads at boundaries can be applied by stating the load components σ_x and σ_y at the affected triangular element edges.

Plane Poisson problems can be loaded by a constant `p` on an element level basis.

IntTriangle			
Properties			
Tag	*Description*	*Type*	*optional*
`x1`, `x2`, `x3`, `y1`, `y2`, `y3`	integration triangle points	double	
`r1`, `r2`, `r3`	edge radii	double	•
`inttype`	identifier of analytical ansatz function	string	
`inttype`	integration type	string	•
`hpintdepth`	depth of recursive *hp*-integration	int	○[12]
`uintdepth`	depth of recursive *h*-integration	int	•
`accuracy`	accuracy of adaptive *h*-integration	int	○[13]

Table 4.9: Integration Triangulation

AreaLoad			
Properties			
Tag	*Description*	*Type*	*optional*
`p`	scalar area loading component `p`	double	

Table 4.10: Area Loading

EdgeLoad			
Properties			
Tag	*Description*	*Type*	*optional*
x, y	edge traction load components σ_x, σ_y	double	

Table 4.11: Edge Loading

Chapter 5

Summary

A complete framework for two dimensional *h*-, *p*- and *hp*-extended GFEM was developed. Both analytical functions and numerical side calculations can be used to enrich the ansatz. Using different numerical integration methods, smooth polynomial ansatz functions as well as non-smooth numerical side calculations can be integrated in an appropriate and error-controlled fashion. The resulting semi-definite linear equation system requires special solving methods. Some methods proposed for *h*-extended GFEM turned out to be infeasible for higher order ansatz functions. To exploit high performance hardware, the code was parallelized for shared and distributed memory architectures.

Analytical enrichment leads to exponential convergence rates on unrefined meshes. Numerical enrichment improves convergence rates, but the gain turned out to be too small to be used for this reason alone.

Extending the partition of unity further to a mixed cell complex cover may lead to equation systems which are definite and thus solvable by a broader range of better performing algorithms. As many of the performance increasing algorithms, like Gauss-Lobatto-Quadrature or sum factorization, are not implemented,

the method cannot compete with classical FEM for the problems investigated. For special applications—like crack propagation—the method could be a promising option. Another approach could be incorporating the GFEM only for enrichment of an otherwise classic FEM ansatz.

Appendix A

UML — Unified Modeling Language

For diagrammatic representation of object oriented software designs, UML (*Unified Modeling Language*) has become a de facto standard [Obj01]. Instead of inventing yet another notion, this work will take advantage of UML.

Because a complete UML reference would be beyond the scope of this document, this appendix will explain only the subset of syntax elements actually used. A more in-depth introduction to history and concepts of UML can be found in one of the many textbooks available (e.g. [HK03]).

A.1 Static Syntax Elements

The UML elements used in this document can be divided into two groups: static and dynamic elements.

Static elements describe structures and dependencies of the design already visible at compile-time. Dynamic elements describe

behaviour, interaction and collaboration during run-time.

Other groups of elements, e.g. concerning the packaging and deployment of the system, are supported in UML, but will not be described here.

A.1.1 Class Diagrams

A class diagram describes classes with member variables and methods as well as the class hierarchy.

Class Hierarchy

Figure A.1 shows a simple class hierarchy: `Triangle` is derived from `Element`. `Element` is a virtual class indicated by displaying the class' name italic. Details like member variables or methods

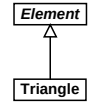

Figure A.1: Class hierarchy

may be partially or completely omitted. However, this does not imply their non-existence.

Interfaces

An interface is like a special kind of abstract base class which owns only virtual[1] functions. It describes a functionality with a

[1] Don't confuse *virtual* with C++'s notation of `virtual`: in this context virtual means that no implementation is provided. In C++ this is called a *pure virtual* function.
C++'s keyword `virtual` however just turns on *polymorphic* behavior (which should be the default at least from an object oriented point of view).

set of functions. A class can implement an arbitrary number of

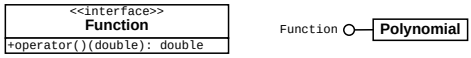

Figure A.2: Interface

interfaces. To document the interface itself, one uses the same notation as for ordinary classes, but the name is supplemented with the keyword «*interface*». To indicate that a class implements a specific interface, we use the so-called *Lollipop-notation* such as in Figure A.2 where `Polynomial` implements the `Function` interface.

Member Variables and Methods

Figure A.3 is a class diagram with member variables and methods. Class `Point` contains two public members of type `double` which default to 0.0. The visibility of class elements is indicated with the

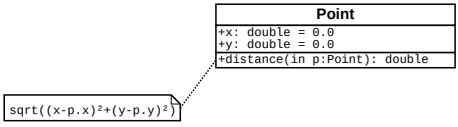

Figure A.3: Members and methods

following symbols:

- + public
- # protected
- − private

`Point` provides a method `distance` which accepts an argument of type `Point` named `p` and returns the distance between itself and the supplied `Point`. The argument's direction of data-flow

may be indicated as *in* or *out*. Again, details may be omitted if appropriate.

To indicate that a method is *virtual* an italic font is used for the name. Note that methods of interfaces (see A.1.1) are not displayed in italics despite their virtual character.

Diagram parts can be annotated using notes connected with a dashed line. In this example, a note is used to describe the formula chosen to calculate the returned distance.

A.1.2 Association

An association describes a relationship between objects. A binary association relating two objects is expressed by a line connecting one object to the other.

The association can be named as in Figure A.4. The direction the name is to be read is expressed with a small filled arrow.

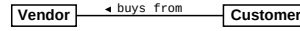

Figure A.4: Association

An arrow at the end of an association indicates a direction. In Figure A.5 an `Appointment` is related to a `Document` which may provide additional information. A `Document` can be found (if present) for a given `Appointment`, but not the other way around.

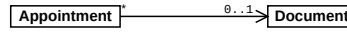

Figure A.5: Directed association

If an association is not qualified with any arrow the direction is left unspecified. It is convenient however to agree that in this case

the association shall be bi-directional. We will use this convention throughout this document.

At both ends of an association the multiplicity can be stated. In Figure A.5, an `Appointment` can point to zero or one `Document`s; a `Document` may be referenced from an arbitrary number of `Appointment`s.

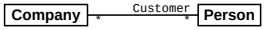

Figure A.6: Role

Like multiplicity, a role can be stated at both ends of an association. Figure A.6 shows the association between a `Company` and a `Person`. To indicate the `Person`'s role as a customer in the modeled association, this syntactic facility is used.

A.1.3 Aggregation

Aggregation is a special kind of association. It expresses a part-of relation between two unequal partners: a master side—denoted with a rhombus—and a slave side.

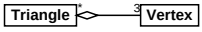

Figure A.7: Aggregation

Note that aggregation does not by itself imply semantics on lifetime or existence of objects. Such restrictions can be expressed by explicitly stating the multiplicity of the relationship. The example in Figure A.7 shows a `Triangle` composed of three `Vertex` objects. On the side of the `Vertex` a multiplicity of 3 indicates that a `Triangle` is composed out of exactly three `Vertices`. On the other hand, on the `Triangle`'s side, a wildcard states that a `Vertex`

may be associated to zero, one or more triangles. This implies that a `Vertex` can exist without `Triangles`, but a `triangle` needs exactly three `Vertices`.

A.1.4 Composition

Composition is a stronger form of aggregation. While multiple aggregating associations to a child object are allowed, only one composite association to a child can exist. The composite association can coexist with all non-composite associations, however. The master side of the composite association is denoted by a filled rhombus.

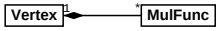

Figure A.8: Composition

Like in the case of aggregation, the composition implies nothing concerning the lifetime of the participating objects.

Figure A.8 shows `MulFunc` objects which are part of a `Vertex`. Here, a multiplicity of one at the `Vertex` side states that a `MulFunc` is always a composite part of a `Vertex`. This case represents the intuitive meaning of composition—the child object's lifetime depends on its master.

A.2 Dynamic Syntax Elements

A.2.1 Object Diagram

An object diagram shows a graph of instances at a specific point in time. There is no separate kind of diagram exclusively for objects. Instead, the class diagram allows the representation of objects, too. Classes and objects may be present in the same

diagram. Sometimes the terms *class diagram* and *object diagram* are used interchangeably. We will use the term *class diagram* only if no object instances are present and the term *object diagram* otherwise. This type of diagram is especially useful in documenting

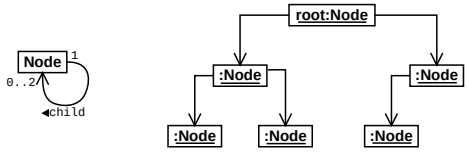

Figure A.9: Class and object diagram

data structures. Figure A.9 shows the class diagram for a binary tree's `Node` class and an object diagram for an example tree. To distinguish objects from classes, their name and type (class name) are underlined. The name is separated with a colon from the type. As always, the name or type can be omitted if appropriate.

Appendix B

xmlom — XML Object Manager

This chapter will explain the parser generator used to load and store the GFEM discretization document (see 'Parser Generator', 4.2.2) in greater detail. It is not supposed to be read independently, but will continue where 4.2.2 left off. The actual GFEM document is used as an example. However, the parser generator could be used for many other kinds of documents as well.

B.0.2 Type Maps

Whenever a document object embeds child objects, the user of the document as well as the parser itself requires means to access these objects. The functionality needed to accomplish this is stated in a number of `...Map` classes. Figure B.1 shows the hierarchy of the generated type maps.

The composite associations used to embed child objects are implemented using bare pointers, `map`s of pointers or `list`s of pointers.

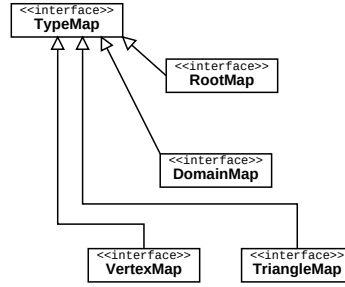

Figure B.1: Type Maps

As seen in this example the `...Map` classes basically declare an interface for accessing functions to these associations:

```
class DomainMap
   :public TypeMap
{
   public:
      virtual MaterialBase*& getMaterial() = 0;
      virtual DiffEqBase*& getDiffEq() = 0;
      virtual std::map<int, VertexBase*>&
         getVertexMap() = 0;
      virtual std::map<int, TriangleBase*>&
         getTriangleMap() = 0;
};
```

All type map classes are derived from the base class `TypeMap`. This empty class is used internally as a data type for storing different type maps.

B.0.3 Document Base Types

Figure B.2 shows the generated document object structure.

The root of this structure is the class `XMLObjectManager`. This class represents the document. It contains the associations to the top-level child objects. The generated code to load and store the document resides here.

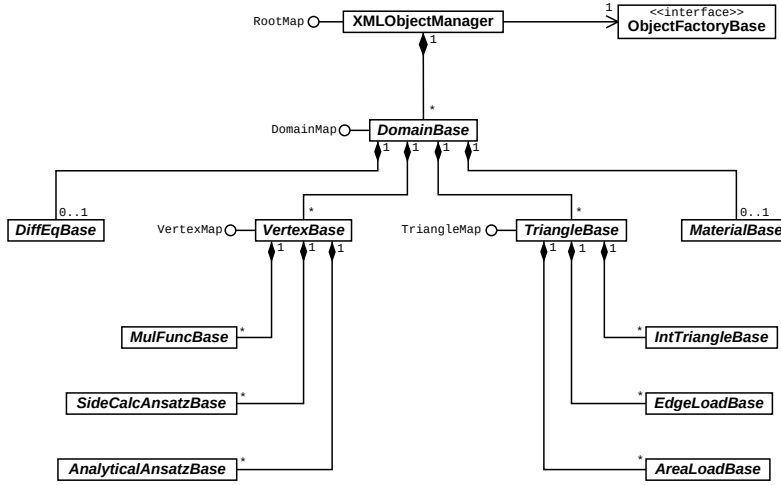

Figure B.2: Generated Document Structure

`XMLObjectManager` contains an association to an object instance implementing the `ObjectFactory` interface. This association is set during construction of `XMLObjectManager`.

The user supplied `ObjectFactory` is responsible for creating the custom subclasses of document base type objects.

The generated document base types below `XMLObjectManager` are abstract, even though they implement the respective `...Map` interfaces. Besides custom functionality, the ability to load and store properties is declared virtually, but not implemented. By implementing the `...Map` interfaces, these base objects already own the associations required to form the tree-like document structure. The user provides subclasses to these base objects

implementing properties (pairs of `...Set` and `...Get` functions) and the desired custom functionality.

Appendix C

List of Symbols

Symbol	Name
$a, b, c, \ldots$	Scalars
$A, B, C, \ldots$	Points
$\mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots$	Vectors
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$	Matrices
$\mathbb{R}$	Field of real numbers

Appendix D

Natural Triangle Coordinates

D.1 Standard triangular Element

The standard (or reference) element used is (P_1, P_2, P_3) with

$$P_1 = \begin{pmatrix} x_1 = 0 \\ y_1 = 0 \end{pmatrix} \tag{D.1}$$

$$P_2 = \begin{pmatrix} x_2 = 1 \\ y_2 = 0 \end{pmatrix} \tag{D.2}$$

$$P_3 = \begin{pmatrix} x_3 = 0 \\ y_3 = 1 \end{pmatrix} \tag{D.3}$$

in Cartesian coordinates.

For arbitrarily shaped triangles, we introduce the notion of natural (or area) coordinates denoted as triplets of the form $S(s_1, s_2, s_3)$. These coordinates are 1 at a certain node and decay linearly to 0

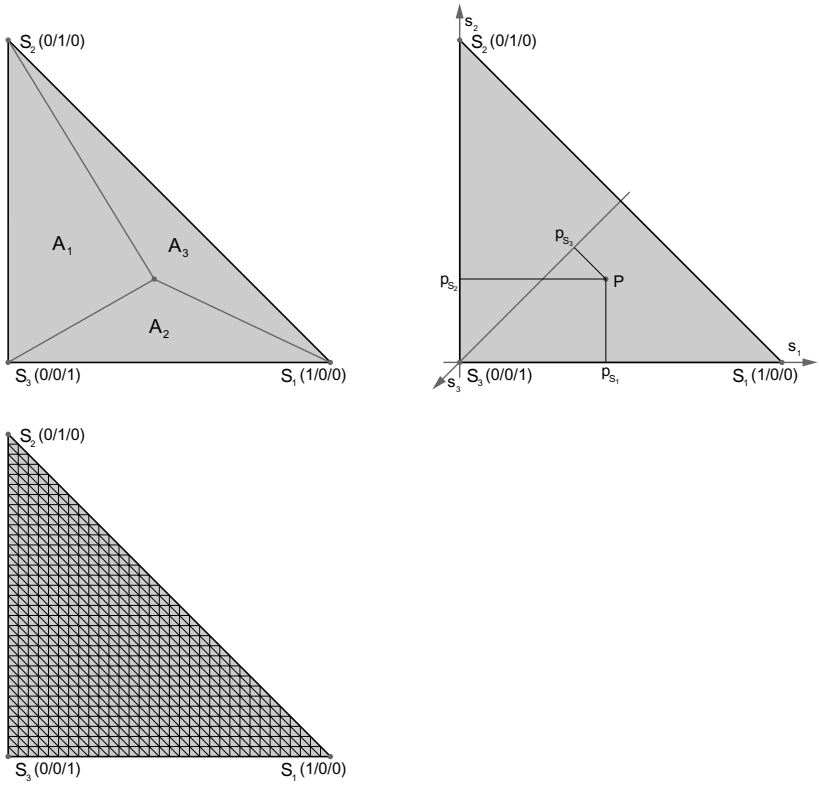

Figure D.1: Standard triangular element in natural coordinates

at the opposite edge. As a special case for our standard element holds:

$$s_1 = x \tag{D.4}$$
$$s_2 = y \tag{D.5}$$
$$s_3 = 1 - x - y \tag{D.6}$$

In the general case, the following relations apply:

$$s_i = \frac{A_i}{A} \tag{D.7}$$

$$A = \sum_{i=1}^{3} A_i \tag{D.8}$$

$$\sum_{i=1}^{3} s_i = 1 \tag{D.9}$$

$$P = \sum_{i=1}^{3} P_i \cdot s_i \tag{D.10}$$

where A is the area of the entire triangle and $A_1 ... A_3$ are the areas of the sub-triangles resulting from connecting a point to the three vertices of the triangle. The cartesian coordinates of a point given in natural coordinates can be determined from the sum of the cartesian coordinates of the triangle vertices weighted with the natural coordinates of the point as denoted in D.10.

D.2 Blending Function Method

Using D.10, the standard element can already be mapped to an arbitrarily triangle with straight edges. To map to an arbitrary triangle with curved edges, the blending function method is used. The curved edge is represented as a vector function $\nu_i(t_i), t_i \in$

$[-1, 1]$ with $\boldsymbol{\nu}_i(0) = \boldsymbol{\nu}_i(1) = \mathbf{0}$ describing the difference between the straight line connecting the incident vertices and the edge (Figure D.2). Therefore, $\boldsymbol{\nu}_i = \mathbf{0}$ in the case of a straight edge.

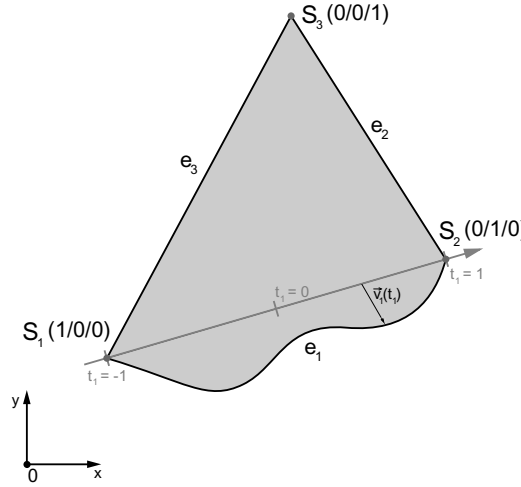

Figure D.2: Blended triangle

The parameters t_i can be determined from a point given in natural coordinates using the following relations:

$$t_1 = 2 \cdot \frac{s_2}{s_1 + s_2} - 1 \tag{D.11}$$

$$t_2 = 2 \cdot \frac{s_3}{s_2 + s_3} - 1 \tag{D.12}$$

$$t_3 = 2 \cdot \frac{s_1}{s_3 + s_1} - 1 \tag{D.13}$$

The linear mapping D.10 is complemented with a linear blending

term

$$\begin{aligned} \mathbf{b}(s_1, s_2, s_3) &= \boldsymbol{\nu}_1(t_1) \cdot (1 - s_3) + && \text{(D.14)} \\ &\quad \boldsymbol{\nu}_2(t_2) \cdot (1 - s_1) + && \text{(D.15)} \\ &\quad \boldsymbol{\nu}_3(t_3) \cdot (1 - s_2) && \text{(D.16)} \end{aligned}$$

which fades the influence of the curved edge from 1 to 0 at the opposite vertex. The complete mapping is given by

$$P = \sum_{i=1}^{3} P_i \cdot s_i + \mathbf{b} \tag{D.17}$$

While the linear mapping D.10 could be inverted analytically, this is no longer possible for the blended mapping. The inverse function has to be determined numerically, e.g. with some Newton-Raphson steps.

D.2.1 Blending to a circular shaped Edge

The blending vector function $\boldsymbol{\nu}$ can be constructed in many ways. However, some care has to be taken to choose a function with advantageous properties.

In the following two ways of constructing a blending function for a circular edge will be discussed.

Normal Blending

One idea of blending to a circular edge is to add the additional normal height between the straight edge and the circular arc to a point on the edge as illustrated in Figure D.3.

With

$$(h + |\boldsymbol{\nu}|)^2 + \left(\frac{1}{2} \cdot a \cdot t\right)^2 = r^2$$

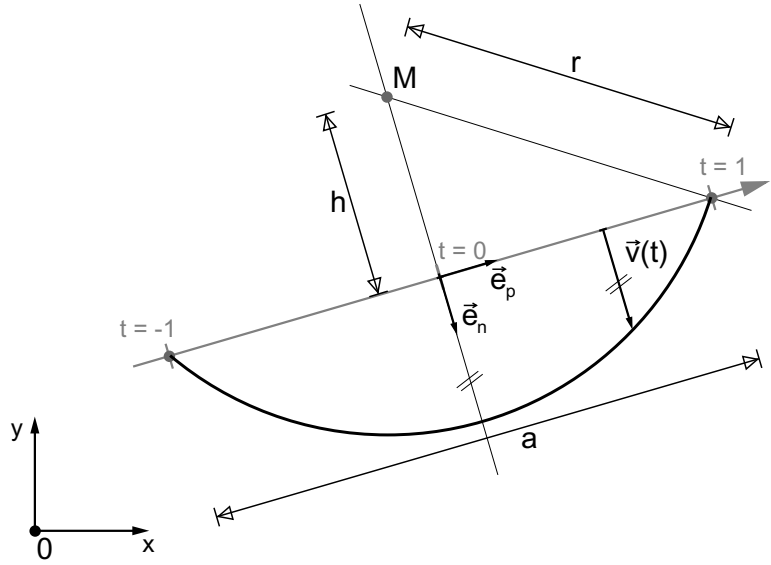

Figure D.3: Normal blending to a circular edge

we get

$$\boldsymbol{v}(t) = \boldsymbol{e}_n \cdot \left(\sqrt{r^2 - \frac{a^2 t^2}{4}} - \sqrt{r^2 - \frac{a^2}{4}} \right) \cdot \operatorname{sgn}(r) \tag{D.18}$$

Radial Blending

Another possibility is to radially map a point on the edge to the arc as illustrated in Figure D.4. The associated blending function can be determined to:

$$\boldsymbol{v}(t) = \left(M + \begin{pmatrix} \sin(\alpha + \varphi) \\ \cos(\alpha + \varphi) \end{pmatrix} \cdot r \right) - \left(C + \boldsymbol{e}_p \cdot t \cdot \frac{a}{2} \right) \tag{D.19}$$

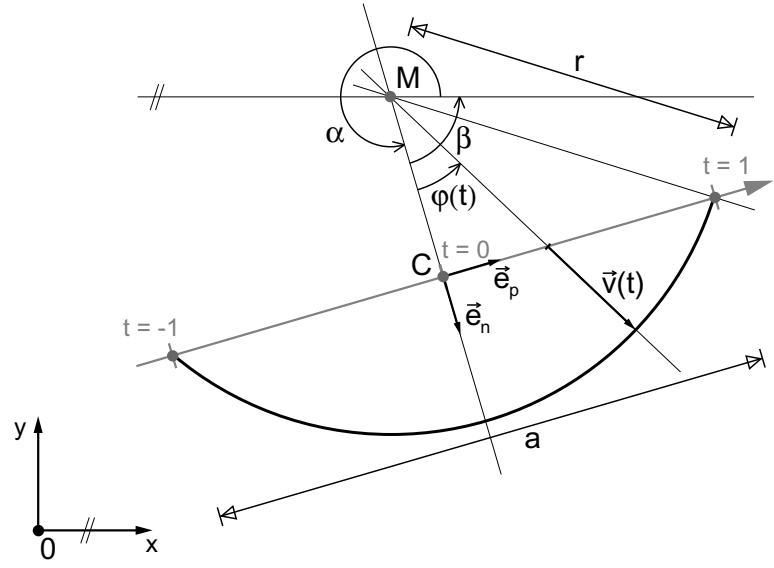

Figure D.4: Radial blending to a circular edge

D.2.2 Effects of well- or ill-chosen Parametrization

Figure D.5 shows an example of both normal (left column) and radial (right column) blending. In the first row, an equidistant grid is mapped, the second row shows the size and direction of $\frac{\partial P}{\partial s_1}$ and $\frac{\partial P}{\partial s_2}$. The heavy distortion and large derivatives of normal blending can be seen clearly.

Diagram D.6 shows as a numerical experiment, the error in approximating the area $\int 1 \, d\Omega$ of the mapped triangle using a Gauss-Legendre tensor product integration scheme. As one would expect, radial mapping performs much better using this integration scheme because the square root terms of normal mapping cannot be approximated well using this quadrature algorithm.

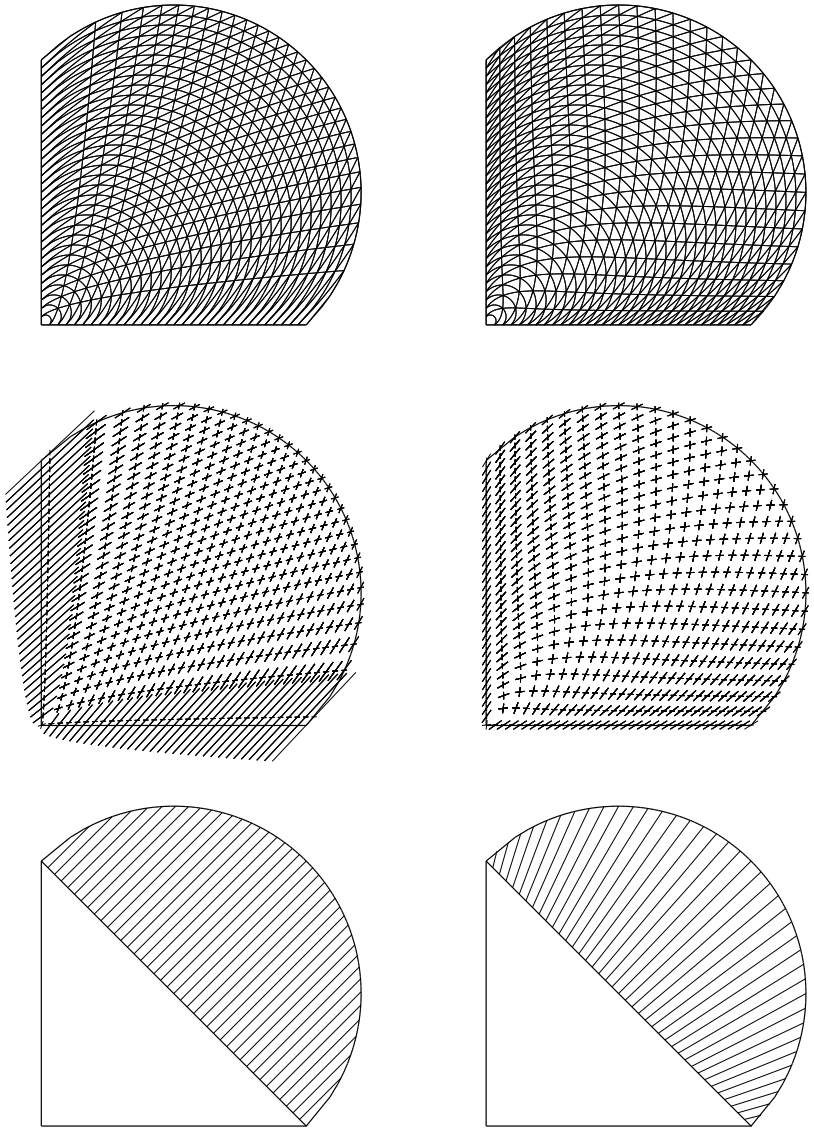

Figure D.5: Mapped triangle

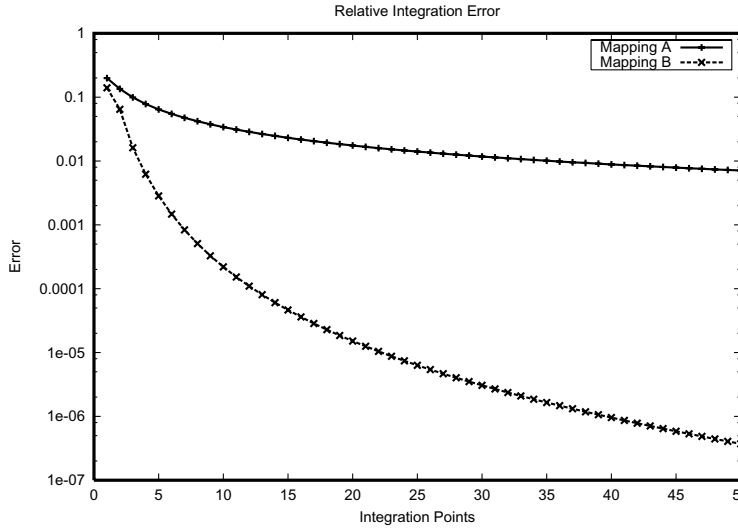

Figure D.6: Integration error

Bibliography

[ABCM02] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39:1749–1779, 2002.

[BB07] I. Babuska and U. Banerjee. The finite element method for elliptic pdes and its generalizations. *iacm expressions*, 21:14–17, 2007.

[BBO02] I. Babuška, U. Banerjee, and J. E. Osborn. Meshless and generalized finit element methods: A survey of some major results. Technical Report 02-03, Texas Institute of computational and applied mathematics, University of Texas at Austin, 2002.

[BCO94] I. Babuska, G. Caloz, and J. E. Osborn. Special finite element methods for a class of second order elliptic problems with rough coefficients. *SIAM Journal on Numerical Analysis*, 31:945–981, 1994.

[BKO+96] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering*, 139:3–47, 1996.

[BM96] I. Babuška and J. M. Melenk. The partition of unity finite element method: Basic theory and applications. *Computer Methods in Applied Mechanics and Engineering*, 139:289–314, 1996.

[BM97] I. Babuška and J. M. Melenk. The partition of unity method. *International Journal for Numerical Methods in Engineering*, 40:727–758, 1997.

[BMMB05] E. Béchet, H. Minnebo, N. Moes, and B. Burgardt. Improved implementation and robustness study of the x-fem for stress analysis around cracks. *International Journal for Numerical Methods in Engineering*, 64:1033–1056, 2005.

[BPM+03] T. Belytschko, C. Parimi, N. Moes, N. Sukumar, and S. Usui. Structure extended finite element methods for solids defined by implicit surfaces. *International Journal for Numerical Methods in Engineering*, 56:609–635, 2003.

[BRT00] P. Breitkopf, A. Rassineux, and G. Touzout. Explicit form and efficient computation of mls shape functions and their derivatives. *International Journal for Numerical Methods in Engineering*, 48:451–466, 2000.

[BS92] I. Babuska B. Szabo. *Finite Element Analysis*. John Wiley & Sons, New York, 1st edition, 1992.

[BSMM00] T. Belytschko, N. Sukumar, N. Moes, and B. Moran. Extended finite element method for three-dimensional crack modelling. *International Journal for Numerical Methods in Engineering*, 48:1549–1570, 2000.

[BXP03] T. Belytschko, S. P. Xiao, and C. Parimi. Topology optimization with implicit functions and regularization. *International Journal for Numerical Methods in Engineering*, 57:1177–1196, 2003.

[CAD95] J. T. Oden C. Armando Duarte. Hp clouds–a meshless method to solve boundary-value problems. Technical report, TICAM, University of Texas at Austin, 1995. TICAM Report 95-05.

[CD94] M. Cosnard and E. M. Daoudi. Optimal algorithms for parallel givens factorization on a coarse-grained pram. *Journal of the Association for Computing Machinery*, 41(2):399–421, 1994.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM National Conference*, pages 157–172, 1969.

[CQYY01] J.-S. Chen, C.-T. Qu, S. Yoon, and Y. You. A stabilized conforming nodal integration for galerkin mesh-free methods. *International Journal for Numerical Methods in Engineering*, 50:435–466, 2001.

[CR86] M. Cosnard and Y. Robert. Complexity of parallel qr factorization. *Journal of the Association for Computing Machinery*, 33(4):712–723, 1986.

[DB99] J. Dolbow and T. Belytschko. Numerical integration of the galerkin weak form in meshfree methods. *Computational Mechanics*, 23:219–230, 1999.

[DBO00] C. A. Duarte, I. Babuška, and J. T. Oden. Generalized finite element methods for three-dimensional structural mechanics problems. *Computers and Structures*, 77:215–232, 2000.

[DGJ02] Q. Du, M. Gunzburger, and L. Ju. Meshfree, probabilistic determination of point sets and support regions for meshless computing. *Computer Methods in Applied Mechanics and Engineering*, 191:1349–1366, 2002.

[DKQ06] C. A. Duarte, D.-J. Kim, and D. M. Quaresma. Arbitrarily smooth generalized finite element approximation. *Computer Methods in Applied Mechanics and Engineering*, 196:33–56, 2006.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, pages 302–325, 1983.

[Dua95] C. A. Duarte. A review of some meshless methods to solve partial differential equations. Technical Report Technical Report 95-06, Texas Institute of Computational and Applied Mathematics, University of Texas at Austin, 1995.

[GH73a] W. J. Gordon and C. A. Hall. Construction of curvilinear co-ordinate systems and applications to mesh generation. *International Journal for Numerical Methods in Engineering*, 7:461–477, 1973.

[GH73b] W. J. Gordon and C. A. Hall. Transfinite element methods: Blending-function interpolation over arbitraty curved element domains. *Numerische Mathematik*, 21:109–129, 1973.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, New York, 1994.

[GL89] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[GMB02] A. Gravouil, N. Moes, and T. Belytschko. Non-planar 3d crack growth by the extended finite element and level sets—part ii: Level set update. *International Journal for Numerical Methods in Engineering*, 53:2569–2586, 2002.

[Gor71] W. J. Gordon. Blending-function methods of bivariate and multivariate interpolation and approximation. *SIAM Journal on Numerical Analysis*, 21:109–129, 1971.

[Gos95] J. Gosling. Java: an overview. Technical report, Sun Microsystems, 1995. http://java.sun.com/people/jag/.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, New York, 1983.

[GS00] M. Griebel and M. A. Schweitzer. A particle-partition of unity method for the solution of elliptic, parabolic and hypercolic pdes. *SIAM Journal on Scientific Computing*, 22:853–890, 2000.

[GS02a] M. Griebel and M. A. Schweitzer. *Geometric Analysis and Nonlinear Partial Differential Equations*, chapter A particle-partition of unity method — Part V: Boundary Conditions. 2002.

[GS02b] M. Griebel and M. A. Schweitzer. *Lecture Notes in Computational Science and Engineering*, chapter A particle-partition of unity method — Part IV: Parallelization, pages 161–192. 2002.

[GS02c] M. Griebel and M. A. Schweitzer. A particle-partition of unity method — part ii: Efficient cover construction and reliable integration. *SIAM Journal on Scientific Computing*, 23:1655–1682, 2002.

[GS02d] M. Griebel and M. A. Schweitzer. A particle-partition of unity method — part iii: A multilevel solver. *SIAM Journal on Scientific Computing*, 24(2):377–409, 2002.

[HK03] M. Hitz and G. Kappel. *UML@Work: von der Analyse zur Realisierung*. dpunkt, Heidelberg, 2003.

[hsl] The HSL archive. http://hsl.rl.ac.uk/archive/hslarchive.html.

[Jos99] N. Josuttis. *The C++ Standard Library. A Tutorial and Reference*. Addison-Wesley, New York, 1999.

[KS00] O. Klaas and M. S. Shepard. Automatic generation of octree-based three-dimensional discretizations of partition of unity methods. *Computational Mechanics*, 25:296–304, 2000.

[Lee02] R. Lee. The JNDI tutorial. Technical report, Sun Microsystems, 2002. http://java.sun.com/products/jndi/tutorial/.

[Liu92] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, mar 1992.

[LPRS05] P. Laborde, J. Pommier, Y. Renard, and M. Salaün. High-order extended finite element method for cracked domains. *International Journal for Numerical Methods in Engineering*, 64:354–381, 2005.

[MB02] N. Moes and T. Belytschko. Extended finite element method for cohesive crack growth. *Engineering Fracture Mechanics*, 69:813–833, 2002.

[Mel95] J. M. Melenk. *On generalized finite element methods*. PhD thesis, University of Maryland, College Park, MD, 1995.

[MGB02] N. Moes, A. Gravouil, and T. Belytschko. Non-planar 3d crack growth by the extended finite element and level sets—part i: Mechanical model. *International Journal for Numerical Methods in Engineering*, 53:2549–2568, 2002.

[Moe07] N. Moes. A look back at the extended finite element method and a peek ahead... *iacm expressions*, 21:10–13, 2007.

[Obj01] Object Management Group, Needham, MA, USA. *OMG Unified Modeling Language Specification*, 1.4th edition, September 2001.

[Pac97] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[RGC05a] J. Réthoré, A. Gravouil, and A. Combescure. A combined space-time extended finite element method. *International Journal for Numerical Methods in Engineering*, 64:260–284, 2005.

[RGC05b] J. Réthoré, A. Gravouil, and A. Combescure. An energy-conserving scheme for dynamic crack growth using the extended finite element method. *International Journal for Numerical Methods in Engineering*, 63:631–659, 2005.

[SBC98] T. Strouboulis, I. Babuška, and K. Copps. The design and analysis of the generalized finite element method. *Computer Methods in Applied Mechanics and Engineering*, 1998.

[SBCB03] F. L. Stazi, E. Budyn, J. Chessa, and T. Belytschko. An extended finite element method with higher-order elements for curved cracks. *Computational Mechanics*, 31:38–48, 2003.

[SCB02] T. Strouboulis, K. Copps, and I. Babuška. The generalized finite element method. *Computer Methods in Applied Mechanics and Engineering*, 190:4081–4193, 2002.

[Sch84] H. R. Schwarz. *Methode der finiten Elemente*. B.G. Teubner, Stuttgart, 1984.

[Sch97] H. R. Schwarz. *Numerische Mathematik*. B.G. Teubner, Stuttgart, 4th edition, 1997.

[SK78] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *Journal of the Association for Computing Machinery*, 25(1):81–91, 1978.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, New York, special edition, 2000.

[SZB04] T. Strouboulis, L. Zhang, and I. Babuska. p-version of the generalized fem using mesh-based handbooks with applications to multiscale problems. *International Journal for Numerical Methods in Engineering*, 60:1639–1672, 2004.

[TDZ96] J. Tourino, R. Doallo, and E. L. Zapata. Sparse givens QR factorization on a multiprocessor, May 1996.

[TYT06] R. Tian, G. Yagawa, and H. Terasaka. Linear dependence problems of partition of unity-based generalized fems. *Computer Methods in Applied Mechanics and Engineering*, 195:4768–4782, 2006.

Index