TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Multiphysics Modelling in the Context of Generative Programming

Ausgeführt am

Institut für Mikroelektronik
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Erasmus Langer
und
Dipl.-Ing. Dr.techn. Hajdin Ceric

durch

Karl Rupp

Eslarngasse 27/19
1030 Wien

Matrikelnummer 0325941
Studienkennzahl 066 439

Datum

Unterschrift

Abstract

A highly generic programming framework for the finite element method is presented in this work, building on top of former results from a previous work of the author. At first, a general domain decomposition strategy is presented that is fully decoupled from finite element algorithms. After that, a general mapping strategy allows the automatic construction of basis functions of arbitrary degree from the underlying geometry. For partial differential equations with constant coefficients, an analytical integration of local basis function integrals during compile time on simplex domains leads to excellent run time performance for higher order basis functions while full flexibility in the specification of the mathematical problem is preserved. The solution process for the resulting systems of linear equations is supported by multigrid capabilities of the framework. The applicability of the framework to multi-physics problems is shown at hand of three selected examples from the field of microelectronics: The first example covers the segregation model, which is used for material transport at interfaces. Existence and uniqueness of a solution of the underlying partial differential equation is shown. The second example investigates vacancy distributions at copper grain boundaries in interconnects during electromigration. For the modelling of grain goundaries, the segregation model was used. The deflection of a cantilever beam due to intrinsic strain is considered in the third example and readily shows the scaling difficulties of microelectromechanical systems.

Contents

List of Figures

List of Tables

Acknowledgement

Many thanks go to Hajdin Ceric, who kept my attention to practical issues of the finite element method during my work on the framework. Without his valuable input, I would not have realised the physical importance of coupling segments due to interface conditions. In most mathematical formulations, such couplings are often left unconsidered, because they often do not fit nicely into the beauty of mathematical abstractions.

Many thanks go to Ansgar Jüngel for valuable advice for the proof of Theorem 4. In this context, I also have to thank Philipp Dörsek for clearifying discussions about fractional Sobolev spaces.

I am indebted to Erasmus Langer and Siegfried Selberherr for a lot of scientific and creative freedom that allowed a continuation of my former work and ultimately resulted in the concepts and ideas presented in this thesis.

Notation

Symbol	Meaning
n	Spatial dimension
$\mathbb{R}$	Set of real numbers
$O(\cdot)$	Landau symbol
u	(Weak) Solution of a given problem
u_h	Discrete (weak) solution
Ω	Domain in $\mathbb{R}^n$
Γ, $\partial\Omega$	Boundary of Ω
$\boldsymbol{n}$	Outer unit normal vector
$\chi(M)$	Indicator function of set M
α	Multi-index
Δ	Laplace operator
∇	Nabla operator
$\nabla \cdot u = \operatorname{div}(u)$	Divergence of u
$\boldsymbol{A}, \boldsymbol{B}$	Matrices
$\boldsymbol{v}, \boldsymbol{w}$	Vectors
Π_n	Set of permutations of numbers $\{1, \dots, n\}$
π	Permutation
π_{id}	Identity permutation
$\mathcal{T}_h$	Triangulation of Ω with characteristic length h
$T_h^{(i)}$	i-th element of $\mathcal{T}_h$
$\mathcal{E}_k(\mathcal{T}_h)$	Set of elements of dimension k of a triangulation $\mathcal{T}_h$
$\mathbf{e}_k^{(i)}$	i-th element of $\mathcal{E}_k(\cdot)$
$\mathbf{v}_i$	Shorthand notation for $\mathbf{e}_0^{(i)}$ (vertices)
$\mathbf{e}_i$	Shorthand notation for $\mathbf{e}_1^{(i)}$ (edges)
$\mathcal{E}_k(\mathbf{e}_m^{(i)})$	Set of elements of dimension k of an element $\mathbf{e}_m^{(i)}$, $0 \leq k \leq m$
$L^p(\Omega)$	Space of (equivalence classes of) measurable functions f on Ω where $\lvert f\rvert^p$ is integrable
$H^{m,p}(\Omega)$	Sobolev space (functions with weak derivatives of up to order m in $L^p(\Omega)$)
$H^1(\Omega)$	Abbreviation for $H^{1,2}(\Omega)$
$H_0^{m,p}(\Omega)$	Completion of $C_0^\infty(\Omega)$ in $H(\Omega)^{m,p}$
$\varphi^{(i)}$	i-th global basis function
$\psi^{(i)}$	i-th local basis function
$x_0, x_1, \dots$	(Global) coordinates
$\xi_0, \xi_1, \dots$	Local coordinates of a reference element
$\boldsymbol{\varepsilon}$	strain tensor
$\boldsymbol{\sigma}$	stress tensor

Introduction

At the end of the 20th century, a revolution in programming took place. Object orientation became a widely accepted programming paradigm, allowing a much higher level of abstraction. Along with this revolution, other paradigms like generic programming and functional programming emerged. On the other hand, numerical algorithms have been implemented over decades in FORTRAN, drumming procedural programming into the brains of many mathematicians and engineers. This led to many textbooks describing "the procedural way" of doing numerics, so that students new to numerics are educated in this "procedural way".

Clearly, there are many algorithms that do not require a high level of abstraction from the point of programming (one may think of the Fast Fourier Transform, which inherently works on arrays of numbers), so that this "procedural way" is sufficient. There are, on the other hand, algorithms that can be formulated mathematically in an abstract way, but an implementation has to struggle with all the details that are hidden behind the mathematical abstractions. An archetypical example is the finite element method (FEM), that can mathematically be formulated for arbitrary dimensions, while an efficient implementation for two dimensions only is already challenging. Since the mathematical formulation works for arbitrary dimensions, one could expect the algorithmic part of a finite element implementation to work for arbitrary dimension as well. Looking at the features of some existing finite element packages (Tab. 1), one may wonder why most of them are restricted to a particular dimension (typically only dimensions up to three). It shows that the level of abstraction achieved at code level does not keep up with the mathematical abstraction.

A second look at Tab. 1 reveals that the number and type of PDEs that can be handled by a particular framework are often fixed. However, from the point of mathematics, a Galerkin scheme can be applied to many PDEs as soon as they are formulated in an integral (weak) way. One approach to achieve the full mathematical abstraction at code level is a separate preprocessor as provided by `FEniCS` [15], where the bilinear form specified in an external script file is translated into code. `Sundance` [28] allows to specify the variational formulation directly as source code. A full syntax tree with many optimisations is then built at run time. `deal.II` [13] follows a similar approach, but provides a lower level of abstraction, so that assembly routines have to be written to some degree by hand. However, most packages that allow for arbitrary equations rely on splitting the weak formulation into separate small chunks that can be handled by library functions. For example, `Getfem++` [17] provides library functions like

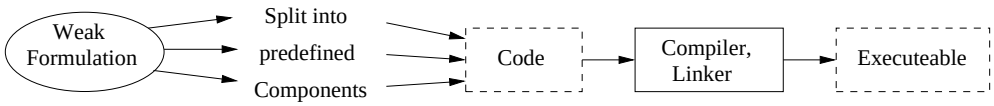
```
1 getfem::asm_stiffness_matrix_for_laplacian( /* parameters here */ );
2 getfem::asm_stiffness_matrix_for_linear_elasticity( /* parameters here */
    );
```

to define the equation. To be precise, `Getfem++` uses strings to hold the actual variational formulation and the two library functions above are convenience functions. Nevertheless, it is common practice in older FEM software to provide library functions for predefined tasks only.

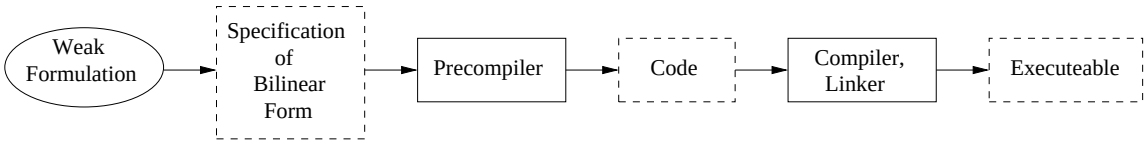
As is shown within this work, the use of abstractions provided by modern programming languages allows to *implement* the finite element method in such a way that it is independent from the spatial

	deal.II[13]	**FEniCS**[15]	**Free FEM Package** [16]	**Getfem++**[17]	**Kaskade**[20]	**Sundance**[28]
Spatial Dimensions	1, 2, 3	1, 2, 3	2	any	1, 2, 3	1, 2, 3
Equation Specification	Arbitrary (using run time objects)	Arbitrary, using separate precompiler	Four fixed equations	Predefined integral terms, user-defined integral terms possible	Four fixed problem classes	Arbitrary (using run time objects)
Cell shapes	Line, Quadrilaterals, Hexahedra	Simplex cells, minor support for other cell geometries	Triangles	Simplex cells, Prisms, Quadrilaterals, Hexahedra. Curved cells	Simplex cells	Simplex cells
Basis Functions	Several sets available, user-definable	Many sets available	Hard-coded	Many sets available	Hard-coded	Linear, quadratic and cubic (only 2d) Lagrange basis
Comments	FEM-assembly loops have to be implemented by the user	Separate compiler for the bilinear form	Most things have to be done by hand	Both analytical and exact integration on cells available	Additional user interface available	Variational problem as run time object

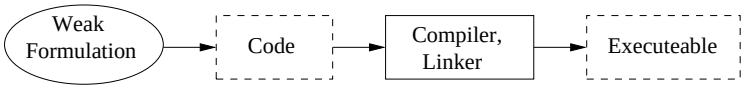
Table 1: Comparison of some existing FEM programming frameworks.



(a) Most FEM frameworks require the user to split a weak formulation into components (like mass matrix, stiffness matrix) supported by the framework.



(b) `FEniCS` automates this manual splitting using a separate precompiler.



(c) The presented framework allows to specify the full weak formulation *directly as program code*. A similar approach is only known to be used by `Sundance`, but no specialisations are used at compile time there.

Figure 1: Overview over several approaches to obtain an executable for a given weak formulation.

dimension of the underlying geometry. This is achieved by well defined interfaces for the interaction of the finite element algorithm with the underlying geometry. Using generative programming, it is even possible to reduce all FEM-related implementation necessary by an end-user of the programming framework to the specification of the weak formulation, which coincides with the mathematical level of abstraction. This way, we have eliminated the precompilation step used by `FEniCS`, so that the weak formulation is fully given within C++ code, similar to the possibilities provided by `Sundance`. While `Sundance` parses the variational problem at run time, the framework presented in this thesis analyses the problem at compile time, resulting in typically shortest execution times.

In chapter one the output of the author's work [26] is briefly summarised. The main idea of a compile time specification of the weak formulation showed up already there, but there was no clean distinction between the geometrical and the mathematical algorithms that are needed for the finite element method.

Chapter two presents a general domain decomposition strategy that is fully decoupled from FEM. The required iterations over elements within a mesh can be specified by type definitions, so that the compiler uses the best available implementation while building the executable.

The third chapter addresses a whole machinery of compile time constructs that are used to formulate the finite element method in arbitrary dimensions as abstract operations over the problem domain. The interfaces with the geometric representation are elaborated and clearly defined. Furthermore, the mapping of local basis function indices to global basis function indices is abstracted by means of the iterator concept. Similarly, iteration over all basis functions defined on a reference element is carried out by an iteration at compile time, enabling the analytic computation of the so-called *element matrices* during compile time.

Chapter four goes into the details that are hidden behind an abstract iteration over basis function indices. As it turns out, a general mapping scheme can be deduced from the construction of basis functions of arbitrary order on domain elements.

In chapter five, the availability of basis functions at compile time is used to compute integrals over the reference element. This results in much better run time efficiency compared to a full numerical

integration of these polynomials. The presented approach works for simplex cells "only", but fortunately this family of elements is probably the most popular for FEM when it comes to unstructured grids.

Multigrid capabilities are shown in chapter six, allowing for a fast solution of the resulting system of linear equations. Multigrid methods may fill a master thesis on its own, therefore only a few issues can be addressed.

Selected applications to problems in practice are shown in chapter eight. For the segregation model discussed first, existence and uniqueness of a solution are proven for small transport coefficients. The second application is an interconnect within integrated circuits, where the effect of electromigration at copper grain boundaries is investigated. Third, the deflection of a MEMS cantilever under prestrain arising from the manufacturing process is computed.

Finally, some future directions are given in chapter eight and a conclusion is drawn.

The reader of this thesis is required to be familiar with advanced concepts of C++ in order to follow the low-level implementations of the framework. A good coverage of modern programming techniques in C++ can be found in the literature, especially in the books of Abrahams and Gurtovoy [1], Alexandrescu [3], Czarnecki and Eisenecker [12] as well as Vandevoorde and Josuttis [30].

Chapter 1

Starting point for this Thesis

As already mentioned in the introduction, this thesis did not start from scratch, it was built on the corner stones of a previous work of the author [26]. The basic ideas demonstrated therein show up in this thesis in a regular basis, thus this chapter briefly summarises the most important concepts and results developed therein in order to introduce the reader into the fundamental concepts. No new results or ideas are developed in this chapter, so readers already familiar with the former work may skip this chapter.

1.1 Compile-Time Expressions

Many computer programs include a lot of information which is already available at compile time. For example, the number of vertices of a triangle is known to be three. Similarly, many mathematical expressions (e.g. polynomials) are needed for the formulation of an algorithm, but have to be either evaluated at run time (which might reduce run time efficiency), or are in some way evaluated by hand and the resulting value is then hard-coded.

Following the ideas of *Expression Templates* introduced into C++ by Veldhuizen [31], one can actually keep a representation of a polynomial, but do necessary evaluations at compile time, so that the resulting code might have an efficiency comparable to hand-tuned code.

One starts with the representation of an unknown x and its evaluation into code:

```
1  struct x_
2  {
3    double operator()(double value) { return value };
4  };
```

The square of such an unknown can be computed by means of introduction of a new type, together with an appropriate overload of the multiplication operator:

```
1  class Expression_mult
2  {
3    public:
4      Expression(x_ lhs, x_ rhs) : lhs(lhs_), rhs(rhs_) {};
5      double operator()(double value)
6      {
7        return lhs(value) * rhs(value);
8      };
9
10   private:
11     x_ lhs_;
12     x_ rhs_;
```

```
};

//the return-type for operator* can now be set to Expression_mult
Expression_mult operator*(x_ lhs, x_ rhs)
{
  return Expression_mult(lhs, rhs);
}
```

The same procedure can be carried out for the operations +, − and /. Using the operator as another template parameter, one arrives at code similar to

```
struct op_plus
{
  static double apply(double lhs, double rhs)
  { return lhs + rhs; }
};

//struct op_minus, op_mult, op_div in the same manner

template <typename LHS, typename RHS, typename OP>
class Expression
{
  public:
    Expression(LHS lhs, RHS rhs) : lhs(lhs_), rhs(rhs_) {};

    double operator()(double value)
    {
      return OP::apply(lhs(value), rhs(value));
    };

  private:
    LHS lhs_;
    RHS rhs_;
};

template <typename LHS, typename RHS>
Expression<LHS, RHS, op_plus>
  operator+(LHS lhs, RHS rhs)
{
  return Expression<LHS, RHS, op_plus>();
}

//and in the same way for operator-, operator* and operator/
```

In addition, scalars can be introduced in the same way:

```
template <long int_value>
struct ScalarExpression
{
    double operator() (double value) const { return int_value; }
};
```

With these helpers at compile time in hand, polynomials or fractions of polynomials can be written in the code as

```
x_ x;

//the polynomial x^2 + 42x - 3 evaluated at 1:
(x*x + 42 * x - 3)(1);

//the (1+x)(1-x)(2+x^2) polynomial evaluated at 2:
((1+x)*(1-x)*(2+x*x))(2);
```

A good compiler will then, in simple words, replace all occurrences of `x` with the evaluation argument and fully evaluate the polynomial at compile time.

For expressions with several unknowns, the above implementation of class `x_` can be extended. For mnemonic reasons, a character template parameter is used[1]:

```
template <char coord>
struct var {};
```

The final implementation for point types with member-function access to their coordinates is done in specialisations of the template:

```
template <>
struct var<'x'>
{
  template <typename Point>
  double operator()(const Point & p) const
  {
    return p.get_x();
  };
};

//similarly for var<'y'> and var<'z'>
```

The key point is now that one is even able to manipulate expressions. As shown in the author's former master thesis [26], differentiation of such compile-time expressions is possible, which renders them very attractive for the implementation of the finite element method (FEM), where usually precomputed element matrices are used.

1.2 Domain Management

The finite element method is – due to its abstract mathematical formulation – not restricted to a particular spatial dimension. A generic FEM-framework should therefore not be tailored to a particular spatial dimension, it should decouple the geometry from any issues specific for FEM.

The domain decomposition used as starting point of this master thesis is in more detail described in [26], for a two-dimensional domain it is shown in Fig. 1.1. The following elements of two- and three-dimensional domains have been introduced:

- *Point*: A `point` represents an arbitrary point whose coordinate number equals the spatial dimension.
- *Vertex*: A `vertex` is the lowest-dimensional element in a mesh. It can be seen as a class that holds a `point` and extends it with an ID.
- *Edge*: An `edge` connects two vertices.

[1] for a high number of unknowns, a template parameter of type `long` is preferred, see Chapter 3

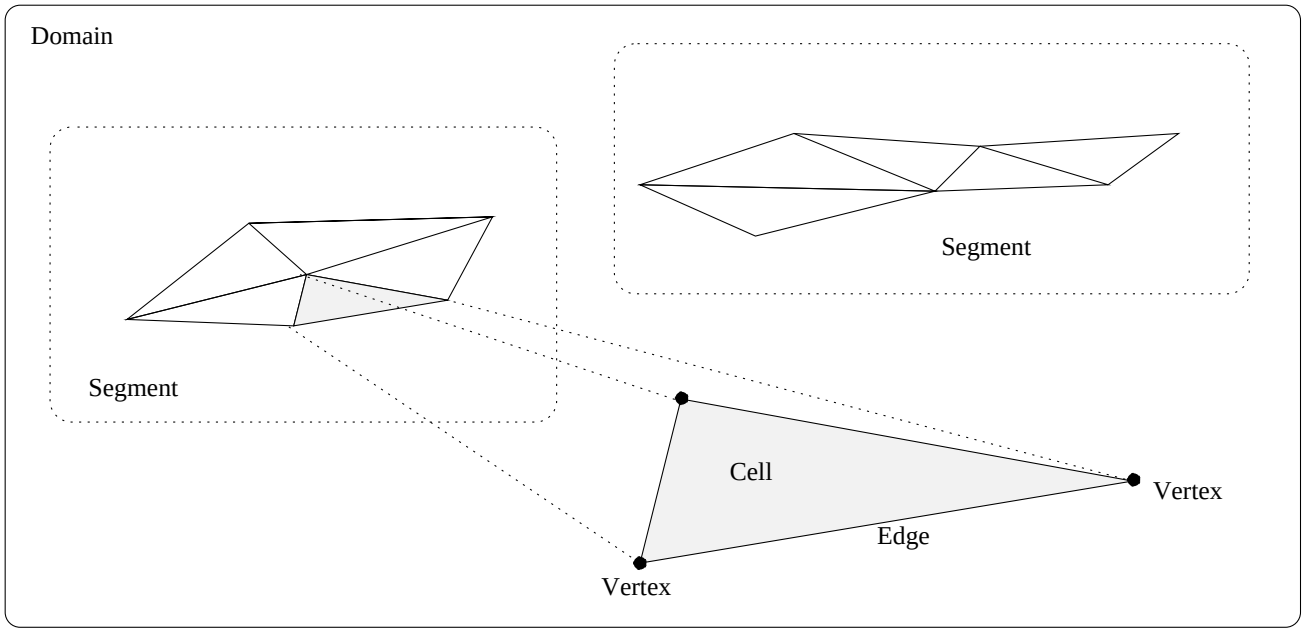


Figure 1.1: Decomposition of a two dimensional domain.

- *Facet*: A `facet` holds one of the lateral surfaces of a cell, for example a triangle in case of tetrahedral cells. In the two-dimensional case, a `facet` is equal to an `edge`.
- *Cell*: A `cell` is a subset of a problem domain Ω and carries the same dimension as Ω. In two dimensions, a cell is typically a triangle or a rectangle, in three dimensions tetrahedra, cubes and prisms are typical geometric shapes of a cell.
- *Segment*: A `segment` is a collection of cells (of the same type). Any vertex v within the segment can be reached from all other vertices of that segment by moving along edges that belong to the segment. However, the converse does not need to be true.
- *Domain*: Finally, a `domain` is a collection of segments of the same type.

Starting from two dimensions, a triangle was implemented as a `cell`-type. For the implementation of a tetrahedron as `cell`-type in three dimensions, the triangular two-dimensional `cell`-type was reused as a `facet`-type.

With the concept of *iterator*s, the dual behaviour of a triangle did not show up for users who required access to facets in three dimensions, but complicated implementation considerably. The following basic domain iterators were directly provided to end-users of the framework:

- On a `domain`: `SegmentIterator`
- On a `segment`: `VertexIterator`, `EdgeIterator`, `FacetIterator` and `CellIterator`
- On a `cell`: `VertexOnCellIterator`, `EdgeOnCellIterator` and `FacetOnCellIterator`
- On a `facet`: `VertexOnFacetIterator` and `EdgeOnFacetIterator`
- On an `edge`: `VertexOnEdgeIterator`

At this point some design problems showed up: With the use of a common `cell`-class for a triangle in two dimensions and a tetrahedron in three dimensions, a `VertexOnFacetIterator` in three dimensions is in some sense the same as a `VertexOnCellIterator` in two dimensions and analogously for an `EdgeOnFacetIterator` and an `EdgeOnCellIterator`. We do not go into further details at this point; it will become clear in Chapter 2 that direct translation of `vertex`, `edge`, `facet` and `cell` into classes is not the best way to go.

For convenient data storage on domain elements, a `QuantityManager` was introduced, which allows for data of arbitrary type to be stored at keys of arbitrary type on an element of the domain. For an end-user of the framework, the code is as short as

```
1 //write and read a double 5.0 with key "KeyString" (of type char[10]) on an
      object called vertex
2 vertex.storeQuantity("KeyString", 5.0);
3 vertex.retrieveQuantity<double>("KeyString");
4
5 //write and read a string "QuantityString" with key 'k' (of type char) on
     an object called cell
6 cell.storeQuantity('k', std::string("QuantityString"));
7 cell.retrieveQuantity<std::string>('k');
```

1.3 Assembly

The interface for the specification of the partial differential equation of interest uses compile-time expressions again. In fact, the weak formulation was in some sense directly written in code. For example, in two dimensions the formulation

$$\text{Find } u \in H_0^1(\Omega) \text{ such that } \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega (x^2 + y^2) v \,\mathrm{d}x \quad \text{for all } v \in H_0^1(\Omega) \tag{1.1}$$

with appropriate boundary conditions reads in the framework as

```
1 typedef gradient<TwoDimensionsTag, 1>              Gradient_u;
2 typedef gradient<TwoDimensionsTag, 2>              Gradient_v;
3
4 assembleMatrix(domain, matrix, Gradient_u() * Gradient_v(),
5                 QuadraticIntegrationTag());
```

and for the right-hand side

```
1 var<'x'> x;
2 var<'y'> y;
3
4 assembleRHS(domain, matrix, (x*x + y*y) * basefun<1>(),
5                 CubicIntegrationTag());
```

From the specification of the right-hand side, one will notice that the specification of the polynomial $x^2 + y^2$ reuses the compile-time expression concept. The class `basefun<1>` is a placeholder for a basis function of the test space. In an element-wise assembly, an iteration over all basis functions defined on each cell is carried out where `basefun<1>` is replaced with basis functions one after another. Since `basefun` is actually a placeholder for a bas*is* function, it was renamed to `basisfun` for the remainder of this thesis.

The `gradient<>` classes used for the specification of the system matrix are convenience functions. When multiplied together, they reduce to

```
basisfun<1, diff_x>()*basisfun<2, diff_x>() + basisfun<1, diff_y>()*
    basisfun<2, diff_y>()
```

in two dimensions and suitably in three dimensions. Therefore, the `basisfun<>` placeholders are used for substitution of derivatives of basis functions. The first template argument is used to distinguish between the test functions v and the solution u.

From a rather abstract point of view, the FEM assembly procedure can be written in pseudo-code as

```
for all cells c do
  for all basis functions bf_v defined on c_i do
    for all basis functions bf_u defined on c_i do
      matrix[map(bf_v), map(bf_u)] += \
                    integrate_over_cell(c_i, lhs_integrand(bf_v, bf_u));
    end
    rhs[map(bf_v)] += integrate_over_cell(c_i, rhs_integrand(bf_v, bf_u));
  end
end
```

so the choice of a basis function placeholder is inherently anchored in the FEM assembly algorithm. In the framework, two ways of managing basis functions are implemented, each with specific advantages and drawbacks:

1. The first uses a programming technique termed *type erasure*, that is, store all basis functions in a common wrapper class. The resulting objects can now be stored in an array, so iteration over basis functions is then equivalent to an iteration over an array. The price to pay is a run time dispatch: A look-up in the virtual function table has to be performed, which means that an evaluation of the basis function at a particular point is carried out, the run time logic has to find the object hidden behind the wrapper class first.

2. The second possibility uses so-called *type lists*. As the name suggest, it is a list of types, where the basis functions are encoded in. There is no run time dispatch needed, so all information about the basis function can be used during compilation to achieve best run time performance. However, since much more work has to be carried out by the compiler, one is limited to a number of about 20 basis functions per cell.

Additionally, the integrals arising from the substitution of basis function placeholders are computed on a reference cell, so an appropriate transformation of the integrand has to be done. The details of this procedure can be found in the literature (e.g. [4], [9], [32]).

1.4 Performance

With the advent of object oriented programming (OOP), programmers soon realised that in most cases an additional level of abstraction results in some run time penalty, the so-called *abstraction penalty*. With generic and generative programming it is much easier for the compiler to reduce any abstraction penalty, leading to run time efficiency comparable to that of hand-tuned code. As can be seen in Tab. 1.1, type lists are faster than type erasure by a factor of two, which shows up in three dimensions as well. It is remarkable that the additional run time dispatch has such a huge impact on performance. On the other hand, such dispatches might have lead to the rumor that object oriented programming is not suitable for high performance computing. In the original work, no comparisons with existing FEM-frameworks have been carried out. A full run time comparison is given in Chapter 5 and it turns out that the new framework is much faster.

Tetrahedra	DoF	Type Erasure [sec]	Type Lists [sec]	Ratio TE/TL
768	3925	0.8	0.5	1.6
29449	29449	5.9	2.6	2.3
49152	228241	49.2	20.5	2.4

Table 1.1: Run-Time comparison for a matrix assembly using cubic basis functions on different meshes for type erasure and type lists. (DoF = degrees of freedom)

	2D linear q1	2D quartic q7	3D linear q1	3D cubic q7
Type Lists	11s, 137MB	56s, 206MB	14s, 138MB	225s, 369MB
Type Erasure	10s, 135MB	14s, 147MB	11s, 134MB	16s, 151MB

Table 1.2: Compilation times and memory comparison two and three dimensions for different degrees of basis functions. (qX = quadrature rule exact up to degree X)

Since a lot more work has to be done by the compiler because of compile time expressions, one must not forget about time and memory needed for compilation. The measurement results in Tab. 1.2 show that for type lists the compilation times as well as memory consumption grow rapidly with increasing degree of the basis functions. However, modern computers do have enough computational power and memory so that the presented generic approach is not just of theoretical interest.

Chapter 2

A General Domain Decomposition Strategy

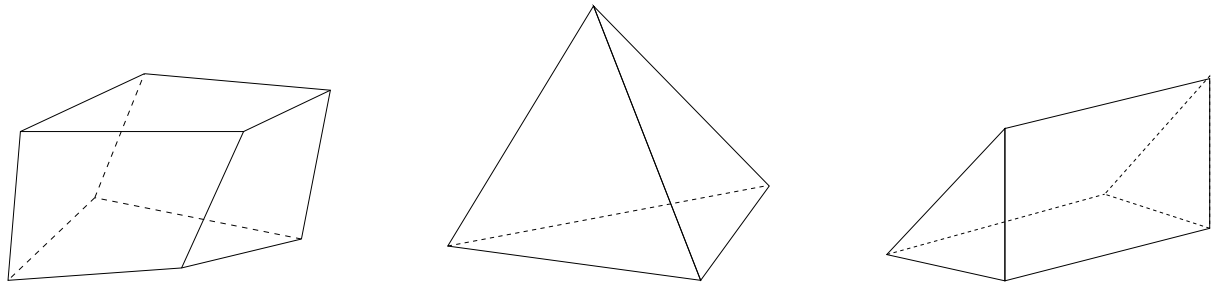
A programming framework for the finite element method (FEM) requires a clever handling of the underlying geometric informations, although this requirement cannot immediately be seen in the formal mathematical description. This chapter develops a domain management that is able to apply FEM in arbitrary spatial dimensions, while the domain management outlined in the previous chapter is more or less fixed to two and three spatial dimensions. For an overview of the mathematical terms used in this chapter, the reader is referred to Appendix A.

2.1 Elements as Container of Elements of Lower Dimension

The domain decomposition strategy is motivated by the most common cell geometries used for finite element computations: Triangular and quadrilateral shapes in two dimensions, tetrahedral and brick-like shapes in three dimensions. We assume the cell shapes to be regular in a sense that for all $k < n$ the k-elements associated with the cell are of the same family of geometric shapes. This poses certain restrictions for dimensions higher than two. In three dimensions for example, prisms with triangular basis cannot be used as cells, since they consist of both triangular and rectangular facets.

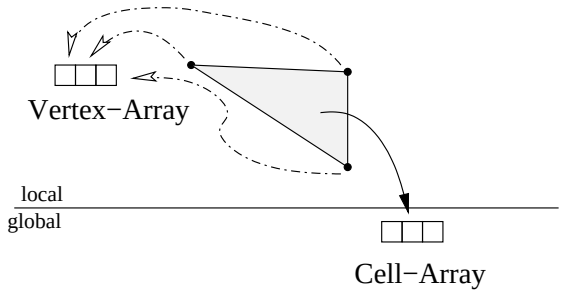
2.1.1 Mesh Handling Requirements

Let us start with a closer inspection of triangles: A triangle consists of three vertices and three edges. The three vertices fully define the shape of the triangle, the edges can be derived from vertices if a

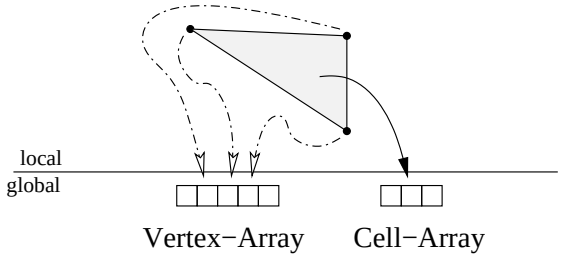


(a) Hexahedron. (b) Tetrahedron. (c) Prism.

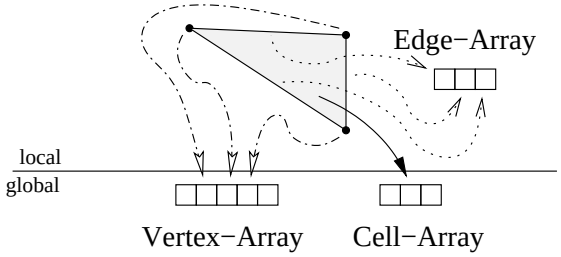
Figure 2.1: The proposed domain decomposition works for regular shapes like in 2.1(a) and 2.1(b). Irregular shapes as in 2.1(c) cannot be handled without further extensions.



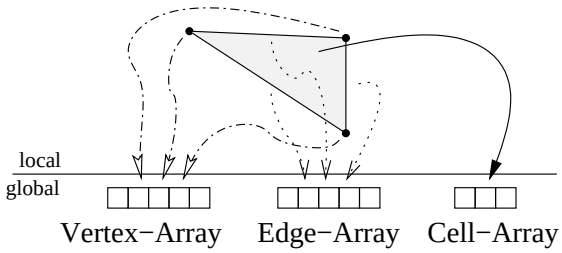
(a) Store vertices locally, do not store edges.



(b) Store vertices globally, do not store edges.



(c) Store vertices globally, store edges locally.



(d) Store vertices globally, store edges globally.

Figure 2.2: Several storage schemes for a triangle.

common reference orientation of the triangles is provided. Depending on the underlying algorithm, edges of the triangle may or may not be of interest.

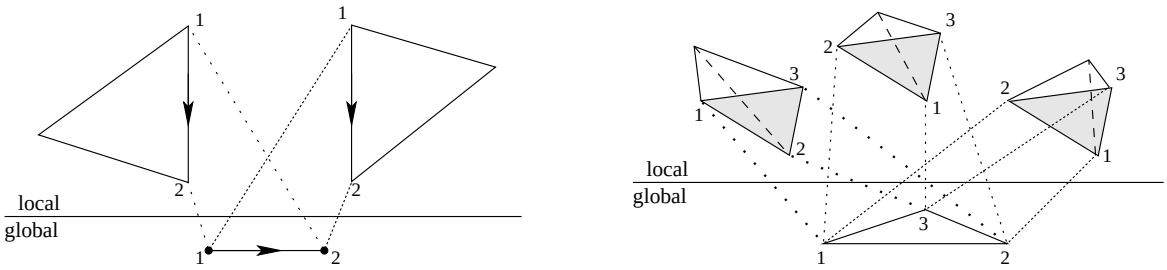
So let us assume we implemented a class `triangle`, which holds the three vertices only. A triangular mesh is then some array or list of `triangle`s and an algorithm `algo1` that works on vertices on a per-cell basis can be executed efficiently. However, let us assume that some other algorithm `algo2` needs to access all vertices of the mesh in a per-mesh basis (so that a vertex which is member of two triangles is accessed only once). For vertices stored per triangle, this would result in a lot of book-keeping and would slow the algorithm down.

It is therefore of advantage to store all vertices of a mesh globally, as well as all `triangle`s. This time, the triangles hold references (pointers) to vertices defining its shape only. This way, both `algo1` and `algo2` work efficiently.

Suppose another algorithm `algo3` needs access to edges on a per-mesh basis. Although access to edges on a per-cell basis can easily be realised from the vertex references stored on each triangle, a mesh-wide iteration over geometrically distinct edges would require again a lot of book-keeping, slowing down `algo3` considerably.

Just as for vertices, one may therefore require a class `mesh` to store vertices, edges and triangles. The `triangle` object holds references to its vertices and its edges. Furthermore, an edge can be oriented in two ways, so the triangle has to provide some means to compare the orientation of the edge as it is stored within the mesh (global orientation) and the local orientation within the triangle. Such an implementation is now capable of providing good run time efficiency for `algo1`, `algo2` and `algo3`. However, if only `algo1` or `algo2` has to be run on the mesh, a lot of memory is wasted, since all edges are set up without ever being used. Taking this into account, one may provide two classes `mesh_no_edges` and `mesh_with_edges` and suffer from a high amount of code duplication.

Let us consider the situation in three dimensions. For a class `tetrahedron` and the storage of its vertices and edges within a mesh, the same reasoning can be applied as for `triangle`s. Moreover, there are additional orientation issues for the storage of facets: Up to six different permutations of the vertex



(a) Edge. (b) Triangle.

Figure 2.3: A global element can have locally different orientations.

triple defining a triangular facet are possible. There is another question of how to actually store the facet: On the one hand, the `triangle` class can be reused for a such a facet, enabling the retrieval of edges of the facet triangle, on the other hand, if only the vertices of the facet are required, only a triple of vertex pointers is sufficient. Depending on the algorithms used, the most appropriate mesh storage scheme shall be chosen. Providing all possible realisations as different mesh and element classes is certainly a nightmare for code maintainers and the wrong way to go.

2.1.2 Specification of Mesh Requirements

Following the ideas of *policy classes* and *tagging*, we define small classes that indicate or realise a specific behaviour. For the concrete realisation, we stick to simplex elements, but the concept can be applied directly to other polyhedral shapes as well.

Starting with the simplest object within a mesh, a node can be associated with a simple tag that holds the topological level (dimension) of the element:

```
1 struct PointTag { enum { TopoLevel = 0 }; };
```

Similar tags are introduced for lines, triangles and tetrahedra:

```
1 struct LineTag        { enum { TopoLevel = 1 }; };
2 struct TriangleTag    { enum { TopoLevel = 2 }; };
3 struct TetrahedronTag { enum { TopoLevel = 3 }; };
```

The next step is to specify tags for the handling of each level of a cell's topology:

```
1 struct TopoLevelFullHandling {};
2 struct TopoLevelNoHandling {};
```

We stick with these two models:

- `TopoLevelFullHandling` indicates a full handling of that topology level. For example, in a tetrahedron this tag specified for the facets means that all facet triangles are set up at initialisation and stored within the cell.
- If `TopoLevelNoHandling` is used, the cell does not care about elements on that topology level.

A configuration class for each element type configures the desired implementation:

```
//declaration:
template <typename ElementTag_, long level>
struct TopologyLevel;

/******* partial specialisations define the behaviour: *********/

//topological description of a tetrahedron's vertices
template <>
struct TopologyLevel<TetrahedronTag, 0>
{
  typedef PointTag                ElementTag;
  typedef TopoLevelFullHandling   HandlingTag;

  enum{ ElementNum = 4 };         //4 vertices
};

//topological description of a tetrahedron's edges
template <>
struct TopologyLevel<TetrahedronTag, 1>
{
  typedef LineTag                 ElementTag;
  typedef TopoLevelNoHandling     HandlingTag;

  enum{ ElementNum = 6 };         //6 edges
};

//topological description of a tetrahedron's facets
template <>
struct TopologyLevel<TetrahedronTag, 2>
{
  typedef TriangleTag             ElementTag;
  typedef TopoLevelFullHandling   HandlingTag;

  enum{ ElementNum = 4 };         //4 facets
};

/**** similar for triangles and lines ********/
```

Above example shows the configuration for a tetrahedron that stores its vertices and facets, but does not store its edges. The actual realisation of the storage scheme for a triangle is then specified by the partial specialisation `TopologyLevel<TriangleTag, level>` for each topology level. This allows rather complex scenarios: Suppose an algorithms needs to iterate over all edges of all facets in the domain. Clearly, there is no need to store the edges on the cell then, it is sufficient to store edges on the facets then. Such behaviour can easily be configured with the above configuration method.

This way we can encode all the requirements for mesh handling into a class hierarchy that can be evaluated at compile time to select the desired implementations.

2.1.3 Implementation for one Layer is sufficient

Instead of packing all topological levels (or topological layers) into one single class, we provide one general configuration for a single layer and consider the following declaration:

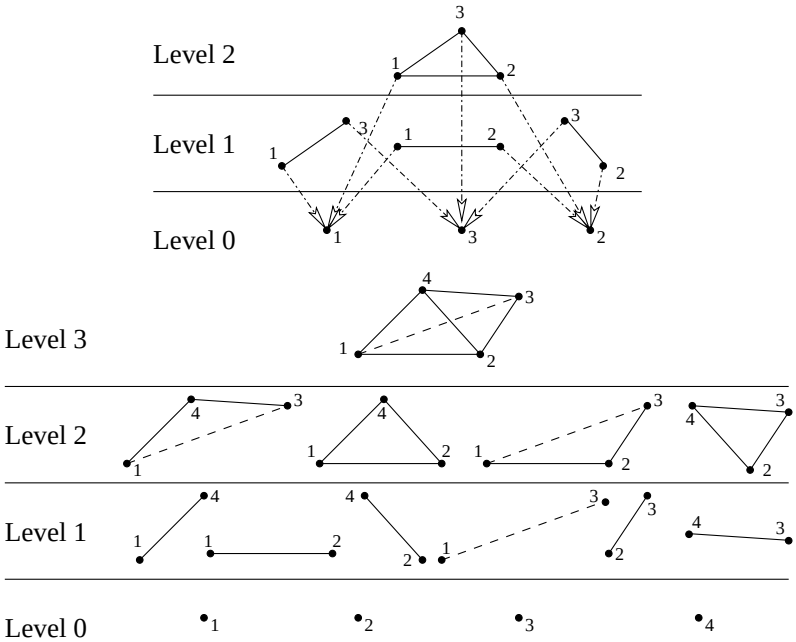


Figure 2.4: Each domain element is decomposed into several layers. Each layer can be configured separately: For example, a separate configuration applies to triangular facets of a tetrahedron again.

```
template <typename T_Configuration,
          typename ElementTag,
          unsigned long levelnum,
          typename HandlingTag = typename TopologyLevel<ElementTag,
                                                        levelnum
                                                       >::HandlingTag>
class lower_level_holder;
```

Here, `T_Configuration` denotes a domain-wide configuration that will be explained later, `ElementTag` is the tag for the cell, `levelnum` is the topology level of this layer and `HandlingTag` is either `TopoLevelFullHandling` or `TopoLevelNoHandling` and by default taken from the configuration class.

With partial specialisations of `lower_level_holder` we can now implement two different handling models for a topological level:

```
template <typename T_Configuration,
          typename ElementTag,
          unsigned long levelnum>
class lower_level_holder <T_Configuration, ElementTag,
                          levelnum, TopoLevelFullHandling>
{
    typedef TopologyLevel<ElementTag, levelnum>              LevelSpecs;

  protected:
    //SegmentType denotes the type of the segment this cell is member of
    void fillLevel(SegmentType & seg)
    {
      //write elements of this level to the segment and
      //store pointers in elements_
    };
```

```
16
17    private:
18      //for the moment, assume LevelElementType to be known from
            configuration class 'LevelSpecs'
19      LevelElementType * elements_[LevelSpecs::ElementNum];
20  };
```

In line 7 the configuration class is abbreviated with 'LevelSpecs' and all further configuration can now be retrieved. In particular, the type of elements an number of elements on this topology level (`LevelElementType`) can be derived from the tag `LevelSpecs::ElementTag`.

The partial specialisation for `TopoLevelNoHandling` is just a reduced variant of the one given above: The function `fillLevel(SegmentType & seg)` is now empty and the `elements_` array is also removed.

2.1.4 Recursive Inheritance

An implementation for one layer only is of no big use. However, minor changes make it very powerful: A cell within a mesh is itself member of a specific topological level n, which can be identified with the dimension of the domain Ω and can be seen as a container for elements of topological level $0 \leq k < n$. This motivates a recursive connection of topological layers:

```
1   template <typename T_Configuration,
2             typename ElementTag,
3             unsigned long levelnum>
4   class lower_level_holder <T_Configuration, ElementTag,
5                             levelnum, TopoLevelFullHandling>
6     : public lower_level_holder < T_Configuration, ElementTag, levelnum - 1 >
7   {
8       typedef lower_level_holder < T_Configuration,
9                                    ElementTag,
10                                   levelnum - 1 >               Base;
11
12    protected:
13      //SegmentType denotes the type of the segment this cell is member of
14      void fillLevel(SegmentType & seg)
15      {
16        Base::fillLevel(seg);
17
18        //write elements of this level to the segment and
19        //store pointers in elements_
20      };
21  };
```

A layer automatically inherits the next lower layer until the vertex level is reached. Additionally, `fillLevel()` has to do the initialisation of lower levels first, before the current level can be initialised.

In case `levelnum` equals zero, the recursion ends by writing another terminating partial specialisation. At vertex level we always store pointers to mesh vertices, since they specify the geometry of a cell.

Now we are ready for the final implementation of a domain element:

```
1   template <typename T_Configuration, typename ElementTag>
2   class element
3     : public lower_level_holder <T_Configuration, ElementTag, levelnum - 1>
4   {
5       typedef lower_level_holder<T_Configuration,
6                                  ElementTag,
7                                  ElementTag::TopoLevel - 1>     Base;
```

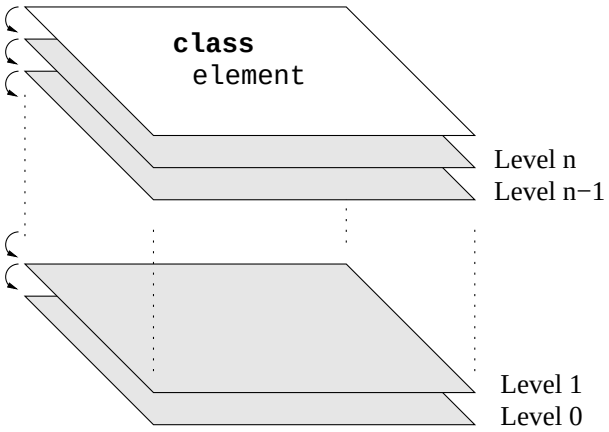


Figure 2.5: The final domain element type consist of several layers. Each layer inherits from the next lower level until the vertex layer (level 0) is reached.

```
8     typedef TopologyLevel<ElTag, 0>                                VertexSpecs;
9
10   public:
11     typedef T_Configuration                    Configuration;
12     typedef ElTag                              ElementTag;
13
14     void fill(SegmentType & seg) { Base::fillLevel(seg); }
15
16     void setVertices(VertexType **vertices)
17     {
18       for(int i=0; i<VertexSpecs::ElementNum; i++)
19         Base::vertices_[i] = vertices[i];
20     }
21 };
```

There are two interfaces at run time: a function `fill()` that fully initialises the cell from vertex level up to facet level. The second function `setVertices()` defines the geometry of the cell so that `fill()` can be called. Please note the access to the vertex specification of a cell through `TopologyLevel`: The number of vertices is known at compile-time, but still completely decoupled from the implementation of `element`.

Elements of lower topological levels still have to be set up. They do depend on the characteristic shape of the cell, thus the implementation has to take the cell tag and the lower level of interest into account. This is just the case in the configuration class `TopologyLevel`! For the edges of a triangle, an implementation can be realised as

```
1  template <>
2  struct TopologyLevel<TriangleTag, 1>
3  {
4    //same typedefs as introduced earlier
5
6    template <typename ElementType,
7              typename Vertices,
8              typename Segment>
9    static void fill(ElementType ** elements,
10                    Vertices ** vertices,
11                    Segment & seg)
12   {
```

```
    Vertices * edgevertices[2];
    ElementType edge;

    edgevertices[0] = vertices[0];
    edgevertices[1] = vertices[1];
    edge.setVertices(edgevertices);
    elements[0] = seg.template addElement<1>(edge);

    //and similarly for the edges with vertex indices 0-2 and 1-2
  }
};
```

Similar implementations have to be done for the setup of a tetrahedron. Some words have to be spent on the member function call to the segment: Since an edge (or any other element) is specified by its vertices, the element is first forwarded to the segment. Within the segment it is checked whether such an edge (with possibly different orientation) is already stored. If so, a pointer to the existing edge is returned, otherwise the edge is stored in the segment and a pointer to this newly created element is returned.

In the final implementation there is an additional argument for the function `fill`: An orientation array that links the local orientation of vertices to the global orientation is also supplied and appropriately set up at the segment. Orientation issues are treated in much more detail in Chapter 4.

2.1.5 Segments

A segment is the top level container for all domain elements that arise from cells of that segment. Similar as for cells, a segment may or may not store elements of a particular topology level, depending on the algorithm that is intended to be used. One may therefore use another configuration class, but this would result in illegal configuration combinations: Assume that triangles are configured to store their edges globally. This forces a segment type to provide storage for edges, a configuration such that segments do not store edges would be illegal. Therefore, the storage scheme for segments is derived from the configuration class.

Similar to cells, a segment is recursively constructed via layers for each topology level. A simplified implementation using these concepts is:

```
template <typename T_Configuration,
          unsigned long levelnum = T_Configuration::CellTag::TopoLevel,
          typename HandlingTag =
                      typename GET_SEGMENT_CONFIG<T_Configuration,
                                                  levelnum>::ResultType>
class segment;

// specialisation for TopoLevelFullHandling
template <typename T_Configuration, unsigned long levelnum>
class segment <T_Configuration, levelnum, TopoLevelFullHandling>
  : public segment <T_Configuration, levelnum - 1>
{
  //some typedefs here as for class element

  public:
    //Return type of addElement omitted
    addElement(long pos, LevelElementType & elem)
    {
      //implementation goes here
```

```
20      }
21
22    private:
23      std::map< ElementKey<LevelElementType>, LevelElementType >   elements;
24  };
```

The `HandlingTag` is derived from a meta-function that checks the configuration class `TopologyLevel` and returns the appropriate handling tag. The member function `addElement` was already used for the setup of lower level elements. The elements are stored in a map with a separate `ElementKey` which compensates for element orientations.

There are two special cases: At vertex level and at cell level, arrays are used instead of maps, because the number of vertices and cells is usually known from the input mesh file and no orientation issues show up.

2.2 The Transformation Layer

The evaluation of mathematical expressions on a domain element can be done on a reference element followed by an appropriate transformation of the intermediate result. Especially for FEM, integrals are evaluated on a reference element. Since we aim at the construction of a powerful domain handling, element transformations must not be omitted.

According to the transformation rules for integrals it is sufficient to know the functional determinant of the mapping. However, for FEM purposes the full Jacobian of the transformation has to be available because the transformation of derivatives of functions requires the knowledge of individual entries of the Jacobian.

The mathematical background, demonstrated for a triangle, is as follows: For an arbitrary triangle T with corner points $\mathbf{v}_0 = (x_0^0, x_1^0)$, $\mathbf{v}_1 = (x_0^1, x_1^1)$, $\mathbf{v}_2 = (x_0^2, x_1^2)$ in space, we consider the integral

$$\int_T f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\ . \tag{2.1}$$

For a reference triangle $\tilde{T}$ with corner points $\tilde{\mathbf{v}}_0 = (0,0)$, $\tilde{\mathbf{v}}_1 = (1,0)$ and $\tilde{\mathbf{v}}_2 = (0,1)$, one can write the transformation from $\tilde{T}$ to T for a point $\boldsymbol{x} = (x_0, x_1)$ as

$$\boldsymbol{x} = \mathbf{v}_0 + \xi_0(\mathbf{v}_1 - \mathbf{v}_0) + \xi_1(\mathbf{v}_2 - \mathbf{v}_0)\ . \tag{2.2}$$

The Jacobian of this mapping with $\boldsymbol{\xi} = (\xi_0, \xi_1)$ is

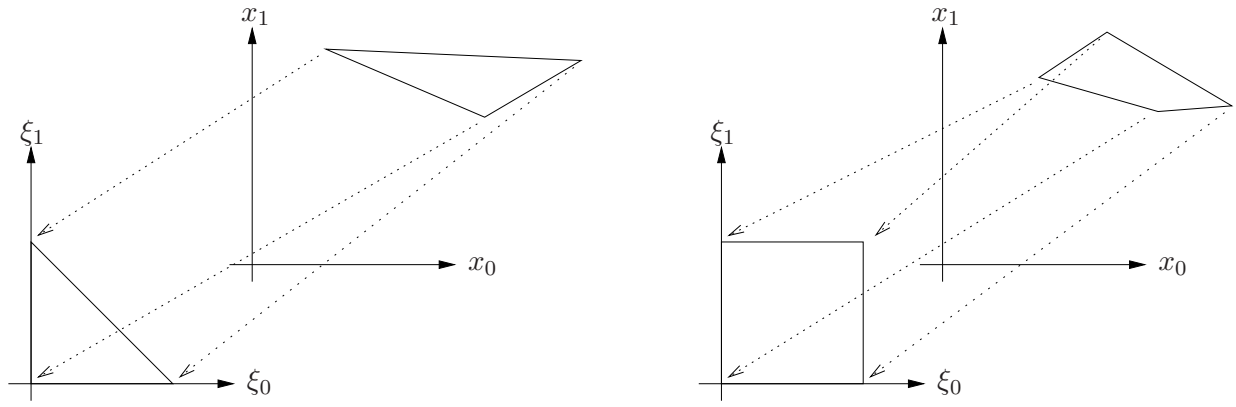
$$\left(\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{\xi}}\right) = \begin{pmatrix} x_0^1 - x_0^0 & x_0^2 - x_0^0 \\ x_1^1 - x_1^0 & x_1^2 - x_1^0 \end{pmatrix} \tag{2.3}$$

and the Jacobian of the inverse mapping (its entries are needed for the transformation of derivatives) is

$$\left(\frac{\mathrm{d}\boldsymbol{\xi}}{\mathrm{d}\boldsymbol{x}}\right) = \left(\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{\xi}}\right)^{-1} \tag{2.4}$$

and can thus be computed by a matrix inversion. Especially for small matrices, *Cramer's rule* is typically applied to obtain the explicit form of the inverse. With the map

$$\boldsymbol{F} : \begin{cases} \tilde{T} \to T \\ \boldsymbol{\xi} \mapsto \boldsymbol{x} \end{cases} \tag{2.5}$$



(a) Triangle. (b) Quadrilateral.

Figure 2.6: Transformation of an element located arbitrarily in the mesh to a reference element.

given by (2.2), the integral (2.1) is computed as

$$\int_T f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_{\tilde{T}} f(\boldsymbol{F}(\boldsymbol{\xi}))\left|\frac{\mathrm{d}\boldsymbol{F}}{\mathrm{d}\boldsymbol{\xi}}\right|\mathrm{d}\boldsymbol{\xi} = \int_{\tilde{T}} f(\boldsymbol{F}(\boldsymbol{\xi}))\left|\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{\xi}}\right|\mathrm{d}\boldsymbol{\xi}\ . \tag{2.6}$$

The Jacobian of $\boldsymbol{F}$ is not necessarily a constant matrix. For example, all quadrilaterals apart from parallelograms have a non-constant Jacobian, complicating the analytic computation of local element matrices. For simplex elements, the Jacobian has constant entries only, which can be exploited for the computation of element matrices. We will come back to this in Chapter 5.

Just as for the topological handling of lower level elements of a cell, several handling strategies for the entries and the determinant of the Jacobian are possible:

- Compute the Jacobian and its determinant at the initialisation of the cell and store the results, so that repeated access at later times is as fast as possible. This leads to high memory consumption, since for an element of dimension n, n^2 entries plus one determinant have to be stored per element.
- Compute all entries of the Jacobian and its determinant only on access but do not store any results. This requires no additional memory at all, but results in poor run time performance. However, if every single byte of memory is needed to avoid the use of swap memory, this strategy can in the end be faster than others.
- Compute and store the determinant of the Jacobian at initialisation. This accelerates the computation of the entries of the Jacobian at access to some extend and consumes memory for one floating point number only.
- Compute and store all entries of the Jacobian and its determinant at first access. As soon as another cell's Jacobian is accessed, the values are overwritten. This strategy has insignificant memory requirements while achieving run time efficiency comparable to the first method. However, extra care has to be taken when it comes to parallelization, since static memory is used for storing the Jacobian and its determinant.

Again, we introduce tags to refer to each of these strategies as ordered above:

```
1 struct DtDxStoreAll {};          //entries and determinant stored
2 struct DtDxOnAccess {};          //everything computed on-the-fly
3 struct DtDxStoreDetOnly {};      //only determinant is stored
4 struct DtDxStoreStatically {};   //compute and store at first access
```

For the reasons just given, the entries and the determinant of the Jacobian are available as member functions of the domain element, because it is an intrinsic property of the element's geometry. Following the idea of assembling the final `element` class by inheritance from topological layers, we introduce another layer we refer to as *transformation layer*:

```
template <typename T_Configuration,
          typename ElementTag,
          typename HandlingTag = typename ElementTag::DtDxHandler>
struct dt_dx_handler;

//sample specialisation for DtDxStoreDetOnly of a triangle:
template <typename T_Configuration>
struct dt_dx_handler<T_Configuration, TriangleTag, DtDxStoreDetOnly>
{
    typedef typename T_Configuration::CoordType           ScalarType;

  public:
    ScalarType get_dt_dx(int i, int j) const
    {
      //implementation
    }

    ScalarType get_det_dF_dt() const { return det_dF_dt; }

    void update_dt_dx()
    {
      //implementation for initialisation of this layer
    };

  private:
    ScalarType det_dF_dt;            //determinant of Jacobian matrix
};
```

This transformation layer has to be located at a higher level than `lower_level_holder`, since access to the point coordinates of the cell is needed. Therefore, `element` directly inherits from `dt_dx_handler` only, which then inherits from `lower_level_holder` layers. In the above code snippet, one has to modify the partial specialisation to

```
template <typename T_Configuration>
struct dt_dx_handler<T_Configuration, TriangleTag, DtDxStoreDetOnly>
  : public lower_level_holder<T_Configuration,
                              TriangleTag,
                              TriangleTag::TopoLevel - 1>
{
  //as before
};
```

Still, each transformation strategy has to be implemented separately, but there are no direct implementation dependencies with other layers, which prevents a combinatorial explosion of code.

It has to pointed out that not only cells can be transformed: Since for a full topological handling, facets and edges (and any other topological layers in between in spatial dimensions higher than three) are again some realisations of the template class `element`, one gets all the transformation capabilities for lower level elements as well! This is especially important for handling boundary integrals that occur in the weak formulation of a PDE.

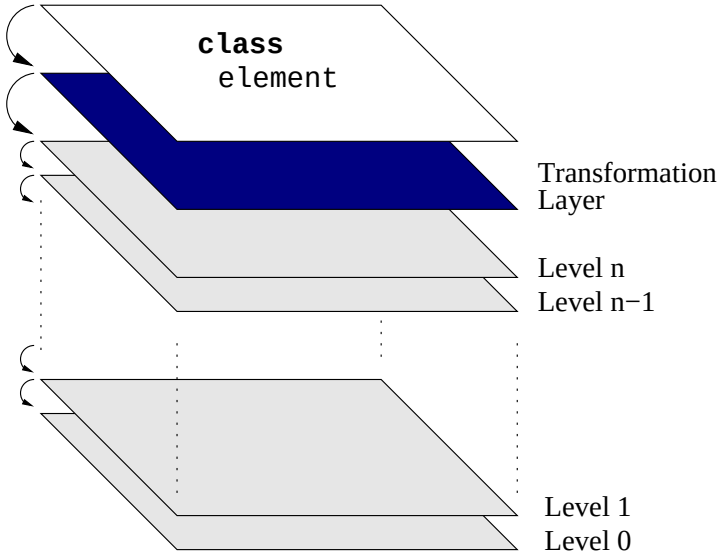


Figure 2.7: The transformation layer is located on top of the topological layers.

2.3 Iteration over Domain Elements

So far we are able to build up domain elements layer by layer and store lower level elements on a segment if desired. What is still missing is a means of iteration over elements of a certain topology level. The necessary implementation is explained in this section.

Iteration over all elements of a certain domain level on a segment will be implemented first. Since segments are constructed by inheritance on a per-layer basis, access from the outside is not straightforward. Because all layers of a segment are built from the same class, only the top level layer can be accessed from the outside and member functions from lower levels are overwritten by their higher level equivalents. Therefore, the top level layer of a segment has to provide well-defined access to lower levels as well as providing its own functionality.

Let us tackle the problem from the viewpoint of an end-user of the framework. Given an segment object `segment` and following the conventions of the Standard Template Library (STL), an end-user may have to iterate over all edges and would therefore write code like

```
1 for (EdgeIterator eit = segment.edges_begin();
2       eit != segment.edges_end();
3       ++eit)
4 {
5   //do something
6 }
```

However, such an implementation is not feasible, because each layer would have to provide a member function `edges_begin()`, since they all belong to the same (template) class. Looking at the problem in n dimensions, an end-user may have the need to iterate over all elements of level k. The member function name is the same for all levels, thus the level number could be provided as argument:

```
1 for (Level_k_Iterator lkit = segment.begin(k);
2       lkit != segment.end(k);
3       ++lkit)
4 {
5   //do something
6 }
```

This time it is possible to find the correct level: At top level, the member functions check whether k is equal to their layer level. If so, they are able to return some iterator, if not, they call the member function with the same name of their base class. This recursion ends as soon as the correct layer is found.

Still, there are some problems with the outlined approach: First, the return types of the member functions have to coincide at all levels. This requires some interface class and virtual functions, slowing down run time performance considerably in a worst case scenario. Furthermore, if lowest level elements are accessed, all higher levels have to be traversed first, even if the topological level on which iteration takes place is already known at compile time.

The suggested solution to the problems stated above is to use template member functions. From an end-user's perspective, the code is

```
1  //obtain type of LevelIterator here
2
3  for (LevelIterator lkit = segment.getLevelIteratorBegin<k>();
4       lkit != segment.getLevelIteratorEnd<k>();
5       ++lkit)
6  {
7    //do something
8  }
```

Instead of the long member function name `getLevelIteratorBegin` one could also use a short `begin`, but from the author's point of view a more expressive member function name supports readability of the code. One may object that the level of iteration *must* be known at compile time, however, for finite element calculations this is typically the case.

The appropriate type of the `LevelIterator` has to be determined, which is is the task of another meta function. We will not go into the details of this meta function here, since the discussion of its implementation requires many technical details. In the end, the type retrieval reduces to an absolute minimum set of input arguments: the type of the `segment` and the level index:

```
1  typedef IteratorTypes<SegmentType, k>::ResultType          LevelIterator;
```

Actually, `SegmentType` has to be provided only because we will reuse `IteratorTypes` for iterators on domain elements as well.

The implementation of the `segment`'s member functions still needs to be done. The crucial step is to do a full compile time look-up of the final member function. In order to do so, we introduce two tags first:

```
1  struct LessType {};
2  struct EqualType {};
```

Using overloading of member functions, we can now obtain the desired behaviour for the retrieval of an iterator for level `j` with the current level `levelnum`:

```
1  //implementation if j is smaller than levelnum
2  template <long j>
3  typename IteratorTypes< segment<T_Configuration>, j>::ResultType
4  getLevelIteratorBegin(LessType)
5  {
6    return Base::template getLevelIteratorBegin<j>();
7  }
8
9  //implementation if j equals levelnum
10 template <long j>
```

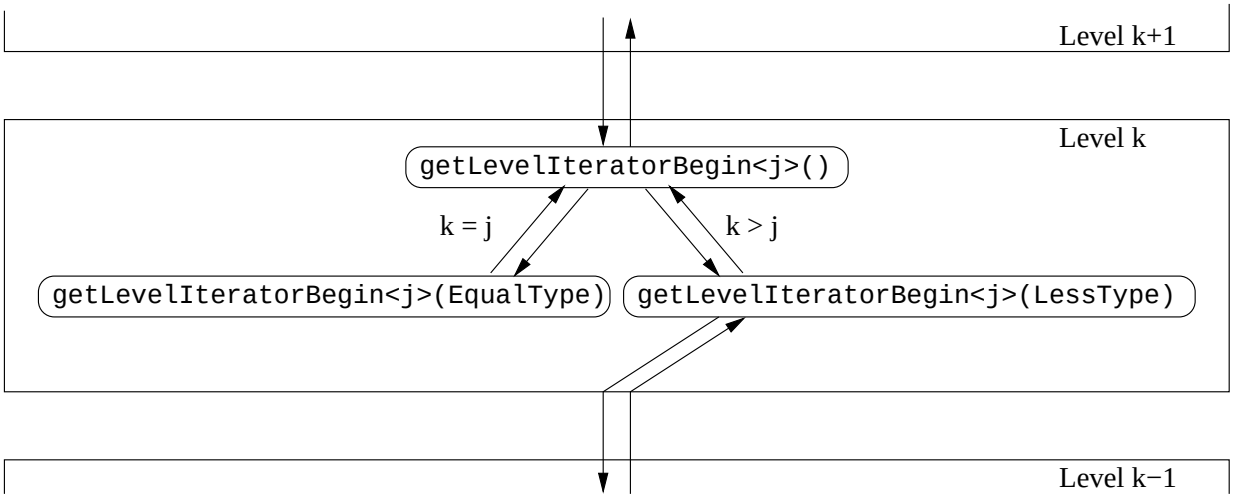


Figure 2.8: Schematic view of a `LevelIterator` instantiation.

```
11 typename IteratorTypes< segment<T_Configuration>, j>::ResultType
12 getLevelIteratorBegin(EqualType)
13 {
14   typedef typename IteratorTypes< segment<T_Configuration>,
15                                   j>::ResultType    IteratorType;
16   return IteratorType(elements.begin());
17 }
18
19 //public interface:
20 template <long j>
21 typename IteratorTypes< segment<T_Configuration>, j>::ResultType
22 getLevelIteratorBegin()
23 {
24   //LevelDiscriminator explained in text below
25   return getLevelIteratorBegin<j>(
26             typename LevelDiscriminator<levelnum, j>::ResultType() );
27 }
28
29 //and similar implementation for getLevelIteratorEnd()
```

The meta function `LevelDiscriminator` returns `LessType` if the required level `j` is less than the current level `levelnum` and `EqualType` if `levelnum` equals `j`. An intentional compilation error is provoked if `j` is larger then `levelnum`, because layers can only be traversed from top to bottom levels. This way, the appropriate member function is called and the compiler is now able to fully inline the member functions, so that a full traversal of all topology levels at compile time results (or at least may result) in one single function call at run time.

The same programming pattern is now applied to domain elements. The type retrieval facility `IteratorTypes` is reused in order to have one single iterator type facility only. The first argument is the type of the element on which iteration has to be performed and the second argument is again the topology level of interest. The syntax is equivalent to iteration on a segment level. Note that even nested iterations are possible: For iteration over all edges of a cell and iteration over all vertices on each edge, one can write:

```
typedef IteratorTypes<CellType, 1>::ResultType    EdgeIterator;
typedef IteratorTypes<EdgeIterator::value_type,
                      0>::ResultType              VertexOnEdgeIterator;

for (EdgeIterator eit = cell.getLevelIteratorBegin<1>();
                  eit != cell.getLevelIteratorEnd<1>();
                ++eit)
{
  for (VertexOnEdgeIterator voeit = eit->getLevelIteratorBegin<0>();
                            voeit != eit->getLevelIteratorEnd<0>();
                          ++voeit)
  {
    //do something
  }
}
```

For compatibility reasons, the limiting cases of iteration over all vertices of a vertex, over all edges of an edge (and so on), are supported as well. However, iterations over higher level elements sharing a particular domain element are not natively supported (e.g. iteration over all triangles sharing a particular edge), but can be manually provided using the quantity storage facility presented in the next section.

2.4 Quantity Storage

As mentioned in the Chapter 1, a convenient way of storing quantities on each domain element was already made available in the primal work [26]. However, it was already mentioned there that considerable speed improvements can be achieved, since the proposed `QuantityManager` used two nested maps for data storage: The first map used the domain element's address as a key, the second map (returned by the first map) used the key supplied by the user for finding the desired data.

Although logarithmic access times are guaranteed for the used map (taken from the STL), this is still too slow if constant access times can be achieved. To be more specific: Assume that each domain element carries an ID that is unique among all objects of the same type, then the first map used above (the one for the element's address) can be replaced by a vector, leading to much faster access times. Nevertheless, we cannot assume all domain elements to carry an ID, so the implementation should automatically pick the storage scheme suited best.

2.4.1 Handling of IDs

Since the handling of element IDs is intimately connected with quantity management, a closer look is necessary. Let us introduce two tiny classes solely responsible for the handling of IDs:

```
class NoID        //this class does not provide IDs
{
  public:
    //for compatibility with the interface of ProvideID:
    void setID(long id) { };
    const NoID * getID() const { return this; };
};

class ProvideID   //this class provides an ID
{
  public:
    void setID(long id) { id_ = id; };
```

```
13      long getID() const { return id_; };
14
15    protected:
16      long id_;
17  };
```

There is a comment on the code necessary: `NoID` provides the same interfaces as does `ProvideID`. The reason for this is that a derived class should have a defined interface no matter if it is derived from `NoID` or `ProvideID`.

Assume now that for a class `A` a `QuantityManager` and ID handling should be provided. One possible implementation is

```
class A : public QuantityManager<A>,
          public ProvideID
{
  //implementation here
};
```

However, this way the `QuantityManager` does not know about the existence of an ID of `A`. Even if it knew, there would not be a general way to access the member function `getID()` of `A`. For this reasons, an implementation

```
class A : public QuantityManager<A, ProvideID>
{
  //implementation here
};

template <typename T, typename IDHandler>
class QuantityManager : public IDHandler
{
  //implementation here
};
```

is of advantage. Now, two partial specialisations for the template parameter `IDHandler` of `QuantityManager` can be provided so that the most suitable implementation is chosen automatically. Nevertheless, a word of caution is necessary: While access to a map using a key does not fail in general, access to an array with an invalid ID leads to segmentation faults. Furthermore, the size of the array should be known in advance, so that an (eventually existing) automatic memory management does not need to do unnecessary copying of data because of reallocation. Therefore another member function (apart from `storeQuantity` and `retrieveQuantity` presented already in Section 1.2) is provided:

```
template <typename T, typename KeyType>
void reserveQuantity(KeyType const & key)
{
  //code to resize the vector/array
}
```

The current `QuantityManager` is working on a per-segment basis, so it can access the total number of elements stored on the segment and resize the quantity array appropriately. The introduction of `reserveQuantity` actually allows to trade flexibility for speed: Quantities that are stored in an array *must* call `reserveQuantity` for the desired key prior to first access.

There is another situation where maps are of advantage: Assume that some quantity is stored only on a small number of cells within the domain. Clearly, allocating memory for all cells is a waste of resources then. This has finally led to the decision that the default storage scheme is still a map.

2.4.2 Fast Access for Selected Keys

In situations where the risk of segmentation faults can be controlled and maximum quantity retrieval speed is needed, the vector storage scheme can still be enabled for selected access keys by means of a meta function. First, two tags are defined that specify the desired storage scheme:

```
struct DenseStorageTag {};       //use vector[ID]
struct SparseStorageTag {};      //use map[element address]
```

A meta function can enable the faster vector-access (provided that an ID is available at all):

```
template <typename KeyType, typename IDHandler>
struct QuantityManagerStorageScheme
{
  typedef SparseStorageTag        ResultType;
};

//for example, enable fast access for keys of type std::string
template <>
struct QuantityManagerStorageScheme< std::string, ProvideID>
{
  typedef DenseStorageTag        ResultType;
};
```

Two implementations for the selected storage schemes are done in `QuantityManager_impl`, an internal class of `QuantityManager` that actually holds the data:

```
template <typename ElementType, typename StorageTag, typename KeyType,
   typename T>
class QuantityManager_impl;

//implementation for map[element address]
template <typename ElementType, typename KeyType, typename T>
class QuantityManager_impl<ElementType, SparseStorageTag, KeyType, T>
{
  public:
    //public member functions here
  private:
    std::map<ElementType *, std::map<KeyType, T> > map_;
};

//implementation for vector[ID]
template <typename ElementType, typename KeyType, typename T>
class QuantityManager_impl<ElementType, DenseStorageTag, KeyType, T>
{
  public:
    //public member functions here
  private:
    std::vector< std::map<KeyType, T> > vec_;
};
```

`QuantityManager_impl` is embedded in `QuantityManager` in a sort of *Singleton Pattern* working on a per-segment basis.

As run time performance analysis showed, access to the data members is considerably faster than before, but still not the best: Assume that data is stored on an element at keys `"foo"` and `"bar"`, both of type `std::string`. On data retrieval, the map first has to do a string comparison before returning

the desired data. Such a comparison has to be done even if only one key of type `std::string` is used throughout the whole program, leading to a run time penalty and a waste of memory for the map with one key only.

For FEM algorithms, the global basis function numbers are stored on domain elements, thus access to these numbers is crucial for good run time performance. To achieve best performance, a separate type used only as key for basis function numbers is introduced:

```
template <long id>
struct MappingKeyType {};
```

The template parameter allows for multiple keys of the same name, but different `id`. With the guarantee that `MappingKeyType` is used only for mapping indices, we can add another specialisation to `QuantityManager_impl`:

```
template <typename ElementType, long id, typename T>
class QuantityManager_impl<ElementType, DenseStorageTag, MappingKeyType<id
    >, T>
{
  public:
    //public member functions here
  private:
    std::vector< T > vec_;
};

//similar for SparseStorageTag
```

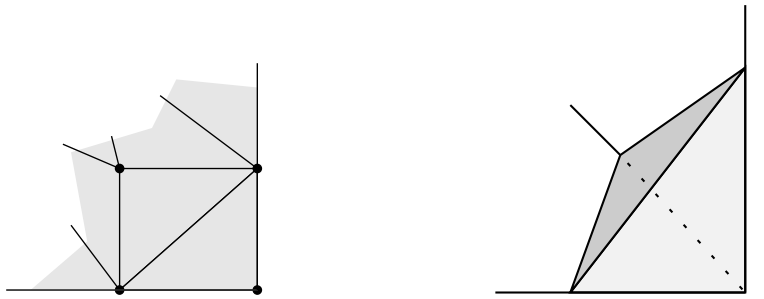
Such a specialisation should be done for every key type that turns out to be a bottleneck for run time performance. To be on the safe side, the default storage scheme is the implementation with two nested maps, because it does not bring any risks for segmentation faults along and still offers a fairly good performance at run time.

2.4.3 The QuantityManager for Domain Elements

The quantity storage concept developed so far still has to be integrated into the existing domain elements. This is again achieved by the use of inheritance, for which we still have to find the best location within the existing inheritance hierarchy. In fact, since no direct dependencies of the `QuantityManager` with other layers of an `element` exist, the most attractive choice is to let `element` directly inherit from `QuantityManager` and from `dt_dx_handler`. This way, a code maintainer can directly see that `element` supports arbitrary quantity storage.

```
template <typename T_Configuration, typename ElTag>
class element
  : public QuantityManager< T_Configuration, ElTag >,
    public dt_dx_handler< T_Configuration, ElTag >
{
  //implementation
}
```

The specification of the desired ID handler can now be given in the cell tags. For example, to use IDs for edges, but not for triangles, a relevant configuration is



(a) Two dimensional case.

(b) Three dimensional case.

Figure 2.9: Lower level elements with all vertices on the boundary are not necessarily part of the boundary.

```
1  struct LineTag
2  {
3    typedef ProvideID         IDHandler;
4
5    //more type definitions here
6  };
7
8  struct TriangleTag
9  {
10   typedef NoID              IDHandler;
11
12   //more type definitions here
13 };
```

so that the class definition of `QuantityManager` becomes

```
1  template <typename DomainConfiguration, typename ElementTag>
2  class QuantityManager : public ElementTag::IDHandler
3  {
4    //implementation
5  };
```

This way the compiler will build up a full customisable class hierachy based on a few type definitions supplied from an end-user of the framework! The compiler will automatically select the fastest implementation available for a given domain element configuration solely based on the type definitions in the element configuration classes, so that best run time efficiency both in terms of execution time and memory requirements can be achieved while full flexibility at code level is preserved.

2.5 Boundary Detection

If a mesh is read from a file and the mesh's cells are specified by the coordinates of their vertices only, information about elements on the boundary is not immediately available. This additional information may be provided directly with the mesh file, but such information is typically redundant and therefore not necessarily provided by the file. In case no boundary information is supplied, a separate algorithm has to determine whether a domain element is located on the boundary, which is not as easy as it seems at first sight.

It is shown in Fig. 2.9 that the knowledge of all boundary vertices is not sufficient to distinguish

between edges on the boundary and those located inside the mesh. However, in two dimensions, edges on the boundary are exactly those who are member of one cell only. As soon as all edges on the boundary are found, all vertices on the boundary are known. Therefore, from the knowledge of all boundary edges, all boundary vertices can be determined, but the converse is not true in general.

In three dimensions, the knowledge of boundary vertices is again not sufficient to distinguish between boundary edges and interior edges because of the same reasoning as in the previous paragraph. Also, an edge may be a member of several cells, even if it is located on the boundary (Fig. 2.9). Therefore, we have to find all faces within the mesh that are member of one cell only. Then, all lower level elements of a boundary face are also located on the boundary and can be tagged appropriately.

Such a boundary detection algorithm can now easily be extended to n dimensions, using the topological property that facets on the boundary belong to one cell only, while all interior facets are member of two cells. Therefore, adding a boolean member variable to class `element` that is toggled using the member function `toggleOnBoundary()` and read via `isOnBoundary()`, the following boundary detection algorithm for a segment can be implemented (note that the code is close to a pseudo-code description):

```
template <typename Segment>
void detectBoundary(Segment & seg)
{
  //some typedefs for CellIterator, FacetIterator
  //and FacetOnCellIterator here

  for (CellIterator cit =
            seg.template getLevelIteratorBegin<CellTag::TopoLevel>();
       cit != seg.template getLevelIteratorEnd<CellTag::TopoLevel>();
       ++cit)
  {
    for (FacetOnCellIterator focit =
              cit->template getLevelIteratorBegin<CellTag::TopoLevel-1>();
         focit != cit->template getLevelIteratorEnd<CellTag::TopoLevel-1>()
             ;
         ++focit)
    {
      focit->toggleOnBoundary();
    }
  }

  //iterate over all facets and tag all lower level topological elements
  //on facets that belong to the boundary:
  for (FacetIterator fit =
            seg.template getLevelIteratorBegin<CellTag::TopoLevel-1>();
       fit != seg.template getLevelIteratorEnd<CellTag::TopoLevel-1>();
       ++fit)
  {
    if (fit->isOnBoundary())
    {
      //set all lower level elements of this facet onto the boundary:
      BoundarySetter<Facet, CellTag::TopoLevel-2>::apply(fit);
    }
  }
}
```

The class `BoundarySetter` results in a compile-time loop that iterates recursively over all topology levels smaller or equal to the second template argument and sets the boundary member variable.

It has to be pointed out that this boundary detection algorithm works in arbitrary dimensions with

any polytopic cell shapes! This is achieved by the powerful mechanism of providing the topological level as template argument to the iterator retrieval member function, so no extra virtual functions have to be used within class `element`, as it were the case with pure object oriented programming at run time.

To be honest, the above algorithm works only if the facet level is actually stored globally on the segment. Otherwise, all facets can be set up temporarily as vertex tuples. Tuples that appear twice are removed then. The remaining tuples represent the boundary facets, from which lower level elements on the boundary can be obtained.

There is another (minor) improvement possible for the algorithm given above: So far, all domain elements carry a boolean flag that indicates whether this element is on the boundary. However, most elements within a mesh are usually located in the inside, thus it is sufficient to use the quantity manager with a default storage scheme, since only elements on the boundary require memory in the map then. However, a boolean flag has a very small memory footprint, so this optimisation is probably not worth the effort in the end.

2.6 Top Level Domain Configuration and Summary

Up to now we have developed code for highly flexible domain elements in this chapter. What is missing is that we still have to configure the top level characteristics of the domain. Such characteristics are:

- *Underlying coordinate system*: Even though in most cases the problem domain is a subset of $\mathbb{R}^n$ with non-empty interior, a coordinate system with discrete point coordinates may be required. For example, the problem domain may be $\{0,1,2,3,4,5\}^2$. Even if this is a rare requirement, the implemented domain management allows such a scenario as well.

- *Spatial dimension*: The desired spatial dimension is specified by the use of tags:

```
1 struct ThreeDimensionsTag{   enum{ dim = 3 };  };
2 struct TwoDimensionsTag  {   enum{ dim = 2 };  };
3 struct OneDimensionTag   {   enum{ dim = 1 };  };
```

  Instead of using tags, a single enumeration could also be used, but tags make the code more expressive and readable (at least in the author's opinion).

- *Cell Tag*: The top level cell shape has to be specified by the use of one of the element tags introduced in the beginning of this chapter. Note that the spatial dimension cannot be derived from the top level cell in general: One can think of a configuration where triangular cells in three dimensions are used to form the surface of some object.

- *Boundary Specification*: Determines whether the boundary is read from some input file or directly specified as code.

A sample domain configuration for a domain in three dimensions with coordinates of type `double` and tetrahedral cells is

```
1 struct DomainConfiguration3DTetrahedra
2 {
3   typedef double                              CoordType;
4   typedef ThreeDimensionsTag                  DimensionTag;
5   typedef TetrahedronTag                      CellTag;
6   typedef NoBoundaryRead                      BoundaryReadTag;
7 };
```

As closing example for this chapter we summarise the domain handling possibilities we have available now, using the configuration `DomainConfiguration3DTetrahedra`:

```
typedef DomainConfiguration3DTetrahedra          DomainConfig;
typedef domain<DomainConfig>                     Domain;
Domain sample_domain;

//add a segment to the domain:
typedef segment<DomainConfig>                    Segment;
Segment & seg = sample_domain.addSegment();

//read the segment's cells from file 'mesh.sgf':
//(the template argument FEMConfig will be explained in the next chapter)
readSegment<FEMConfig>(seg, "mesh.sgf");

//detect boundary facets:
detectBoundary(seg);

//iterate over vertices and store '42' with key "The answer" on each vertex
    :
for (VertexIterator vit = seg.getLevelIteratorBegin<0>();
     vit != seg.getLevelIteratorEnd<0>();
     ++vit)
  vit->storeQuantity(std::string("The answer"), 42);

//iterate over all edges on each facet and print the edge's ID
typedef DomainConfig::CellTag          CellTag;
for (FacetIterator fit = seg.getLevelIteratorBegin<CellTag::TopoLevel-1>();
     fit != seg.getLevelIteratorEnd<CellTag::TopoLevel-1>();
     ++fit)
{
  for (EdgeOnFacetIterator eofit = fit->getLevelIteratorBegin<1>();
       fit != fit->getLevelIteratorBegin<1>();
       ++fit)
  {
    std::cout << eofit->getID() << std::endl;
  }
}
```

If we replace the top level domain configuration by a different configuration for a triangular domain in two dimensions, say `DomainConfiguration2DTriangles`, only the first type definition has to be changed, the remaining code remains untouched! Clearly, the mesh file `mesh.sgf` has to provide a triangular mesh then, and the `EdgeOnFacetIterator` reduces to a trivial iterator over all edges of a triangle's edge (since facets and edges are identical in two dimensions).

Chapter 3

FEM Compile Time Specification

The main innovation in the fundamental work for this thesis [26] compared to traditional programming frameworks for the finite element method (FEM) was the use of a compile time expression engine for the specification of the underlying weak formulation. A brief summary was already given in Chapter 1, the topic of this chapter is the enhancement of these concepts.

3.1 Compile Time Expressions in Arbitrary Dimensions

The initial expression engine was designed for a use in up to three dimensions. This poses certain restrictions for the formulation of a finite element method for arbitrary dimensions, therefore we remove the restriction to three (spatial) dimensions in a first step.

Given the placeholders

```
var<'x'> x_;
var<'y'> y_;
var<'z'> z_;
```

a natural way for a generalisation is to replace the mnemonic template parameters `'x'`, `'y'` and `'z'` with a parameter of type `unsigned long`. This allows for a sufficiently high number of dimensions, but we have to introduce the following convention: The first spatial coordinate is associated with a template parameter zero, the second spatial coordinate is associated with a template parameter one, and so on. This convention is fortunately very common in computer sciences, therefore we can expect that a certain mnemonic association is preserved. The implementation of such a generalised placeholder does not require any specialisations for `'x'`, `'y'` or `'z'` anymore:

```
template <unsigned long id>
struct var
{
  //evaluation at a point:
  template <typename Point>
  typename Point::CoordType operator()(const Point & p) const
  {
    return p.getCoordinate(id);
  };

  //some more members implemented in a similar way
};
```

We require that we can access a `Point`'s coordinates using the member function `getCoordinate`. One may object that a careless use of for example `var<1000>` in three dimensions may lead to a segmentation

fault during run time: Suppose that an object of class `Point` stores its coordinates in an array, such that `id` is used as array index upon access. In such a case, a good compiler issues a warning, if `id` is larger than the `Point`'s coordinate array, because both `id` and the `Point`'s coordinate array size are known at compile time.

The existing differentiation routines need minor corrections for this generalised variable type only, because the only relevant information is whether two `var` template classes carry the same template parameter or not. Much more changes have to be applied for the differentiation tags that occur within the `basisfun` placeholder (which is the renamed analogue of `basefun`): The three differentiation tags

```
struct diff_x;
struct diff_y;
struct diff_z;
```

have to be replaced by more general tags. We even have to find differentiation tags for second and third order derivatives (like `diff_xx` and `diff_xyz`), which should also be covered by a generalised differentiation tag as well.

Let us first consider the following generalisation:

```
//first approach for a generalised differentiation tag:
template <unsigned long coord>
struct diff;
```

Here, `coord` denotes differentiation with respect to the `coord`-th coordinate. This way, single derivatives of basis functions could be defined as

```
basisfun<1, diff<0> >  //differentiation with respect to x
basisfun<1, diff<1> >  //differentiation with respect to y
basisfun<1, diff<2> >  //differentiation with respect to z
```

but second and third order derivatives would not directly fit into this pattern. Even though such an approach is sufficient for most second order PDEs, such a situation is not fully satisfactory. Let us therefore consider the following generalisation:

```
//second approach for a generalised differentiation tag:
template <long coord1, long coord2 = -1, long coord3 = -1>
struct diff;
```

Up to third order derivatives could now be covered, so typically PDEs of order up to six can be assembled. This approach, however, has a drawback when it comes to the transformation of basis functions from the reference element to the element in space. Let us consider a triangle in a global coordinate system (x_0, x_1) and local coordinates (ξ_0, ξ_1). For a basis function φ with local representation ψ, the second partial derivative $\frac{\partial^2 \varphi}{\partial x_0^2}$ is given as (function arguments omitted for the sake of brevity)

$$\begin{aligned}
\frac{\partial^2 \varphi}{\partial x_0^2} &= \frac{\partial}{\partial x_0}\left(\frac{\partial \psi}{\partial \xi_0}\frac{\partial \xi_0}{\partial x_0} + \frac{\partial \psi}{\partial \xi_1}\frac{\partial \xi_1}{\partial x_0}\right) \\
&= \frac{\partial}{\partial x_0}\left(\left[\frac{\partial^2 \psi}{\partial \xi_0^2}\frac{\partial \xi_0}{\partial x_0} + \frac{\partial^2 \psi}{\partial \xi_1^2}\frac{\partial \xi_1}{\partial x_0}\right]\frac{\partial \xi_0}{\partial x_0} + \left[\frac{\partial^2 \psi}{\partial \xi_0^2}\frac{\partial \xi_0}{\partial x_0} + \frac{\partial^2 \psi}{\partial \xi_1^2}\frac{\partial \xi_1}{\partial x_0}\right]\frac{\partial \xi_1}{\partial x_0}\right) \\
&= \frac{\partial}{\partial x_0}\left(\frac{\partial \psi}{\partial \xi_0}\right)\frac{\partial \xi_0}{\partial x_0} + \frac{\partial}{\partial x_0}\left(\frac{\partial \psi}{\partial \xi_1}\right)\frac{\partial \xi_1}{\partial x_0},
\end{aligned} \tag{3.1}$$

where we already used that $\frac{\partial \xi_0}{\partial x_0}$ and $\frac{\partial \xi_1}{\partial x_0}$ are scalars in case of a triangle. The crucial point here is that the second order derivative can be reduced to linear combinations of first order derivatives (with respect to x_0). This first order derivative with respect to x_0 has to applied to the first order derivative of the

local representation, thus leading to a natural nesting of differentiation tags. Such a nesting is not included in the differentiation tag above, therefore one has to implement all such transformations by hand or by some tricky meta functions. What complicates the implementation further is the fact that there are two different basis function models, one for type erased basis functions and one for fully type encoded basis functions ("type listed" basis functions). For example, the evaluation of the derivative with respect to the first spatial coordinate of a basis function at point `p` is:

```
bf.diff_0(p);                       //for a type erased basis function bf
differentiate< var<0> >(bf)(p);     //for a type listed basis function bf
```

Such a distinction is (unfortunately) necessary, since type listed basis functions do not exist as objects at run time, while type erased basis functions do.

Let us come back to the design of a general differentiation tag for the basis function placeholder. Motivated by nested differentiations as discussed earlier, we consider the following template class:

```
template <long coord, typename EXPR = diff_none>
struct diff;
```

The default template argument is `diff_none`, which is the trivial differentiation tag (i.e. no differentiation). The choice of this default argument will become clear soon. With this nested differentiation tag, basis function placeholders look like

```
basisfun<1>                          // no differentiation
basisfun<1, diff<0> >                // d/dx
basisfun<1, diff<1> >                // d/dy
basisfun<1, diff<0, diff<0> > >      // d^2/dx^2
basisfun<1, diff<0, diff<1> > >      // d^2/(dx dy)
```

There is only one member function of this generalised differentiation tag:

```
template <long coord, typename EXPR>
struct diff
{
  template <typename CellType, typename BasisFun, typename Point>
  static typename Point::CoordType apply(CellType & cell,
                                         BasisFun const & bf,
                                         const Point & p)
  {
    return DIFF_TRANSFORMER<Point::DimensionsTag::dim,
                            diff<coord, EXPR> >::apply(cell, bf, p);
  }
};
```

The meta function `DIFF_TRANSFORMER` is responsible for the creation of transformations like the one given in (3.1). Its declaration is

```
template <long dim, typename DIFF_TAG,
          typename EVAL_MODIFIER = diff_none,
          long current_index = 0>
struct DIFF_TRANSFORMER;
```

Here, `dim` denotes the spatial dimension, `DIFF_TAG` is the differentiation tag, `EVAL_MODIFIER` is a helper type and `current_index` is a loop counter from zero to `dim-1`. The iteration core is

```
template <long dim, long coord, typename DIFF_TAG,
          typename EVAL_MODIFIER, long current_index>
struct DIFF_TRANSFORMER<dim, diff<coord, DIFF_TAG>,
                        EVAL_MODIFIER, current_index>
{
  template <typename CellType, typename BasisFun, typename Point>
  static typename Point::CoordType apply(CellType & cell,
                                         BasisFun const & bf,
                                         const Point & p)
  {
    return DIFF_TRANSFORMER<dim,
                            DIFF_TAG,
                            diff<current_index, EVAL_MODIFIER>
                           >::apply(cell, bf, p)
                           * cell.get_dt_dx(current_index, coord)
         + DIFF_TRANSFORMER<dim,
                            diff<coord, DIFF_TAG>,
                            EVAL_MODIFIER,
                            current_index + 1 >::apply(cell, bf, p);
  }
};
```

The code is best understood by looking at (3.1): The first term in the return statement can be seen as $\frac{\partial\psi}{\partial\xi_0}\cdot\frac{\partial\xi_0}{\partial x_0}$, where eventually another derivative acts on $\frac{\partial\psi}{\partial\xi_0}$, therefore another call to `DIFF_TRANSFORMER` with the outer differentiation tag shifted to `EVAL_MODIFIER`. The second summand in the return statement proceeds to the next local variable, i.e. from ξ_0 to ξ_1 in (3.1). If `DIFF_TAG` equals `diff_none`, the basis function is evaluated according to `EVAL_MODIFIER`. There, a distinction between type erased and type listed basis functions is made, but these specialisations are implemented in a rather straight-forward way which does not provide further insight, therefore further details are omitted here.

The proposed transformation algorithm requires that the partial derivatives of local coordinates with respect to global coordinates is constant. However, these partial derivatives may also be functions of the location in space (as it is the case for quadrilateral cells) and thus have non-vanishing derivatives. Such a case can also be included in the above compile time transformation, but the code becomes rather technical then. To keep the attention to the underlying algorithm, this simplified transformation was presented, which is still valid for all simplex cell geometries. Nevertheless, since most PDEs of interest are of second order, this simplified illustration requires an additional point argument `p` for a call to `get_dt_dx` only: Since no additional derivatives of the Jacobian are necessary then, the transformation is valid for all cell geometries.

3.2 Integrals and the Weak Formulation

The matrix assembly function `assembleMatrix` in Section 1.3 implicitly assumes integration of the integrand over the whole domain, so for the specification of boundary integral contributions another assembly function would be required. Instead of providing a function `assembleBoundary`, which would (most likely) result in a duplication of code from `assembleMatrix`, we aim at providing a uniform assembly interface.

Let us have another look at the translation of the weak formulation for the matrix entries into code:

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x \tag{3.2}$$

translates into

```
assembleMatrix(domain, matrix, rhs,
               Gradient_u() * Gradient_v(), QuadraticIntegrationTag());
```

Here, `assembleMatrix` is used both as an ordinary function starting the assembly process and as an implicit integral over Ω. The next step thus is to split these two meanings into separate code constructs.

With the view of integration as a functional on integrable functions, the integration result depends on the integration domain and the integrand. If the integral is evaluated numerically, the result may additionally depend on the quadrature formula used, hence we start with the following template class:

```
template <typename IntDomain, typename Integrand, typename IntTag>
struct IntegrationType
{
  typedef IntDomain             IntegrationDomain;
  typedef Integrand             IntegrandType;
  typedef IntTag                IntegrationTag;
};
```

For the specification of the integration domain, we introduce two tags:

```
struct Omega {};

template <long id>
struct Gamma {};
```

As the class names suggest, `Omega` represents integration over the whole domain. The additional template parameter of `Gamma` accounts for the possibility that the boundary $\partial\Omega$ is decomposed into several arcs $\Gamma_0, \Gamma_1, \ldots$ of the boundary and integration has to be carried out only in one such arc.

We cannot use `IntegrationType` directly within `assembleMatrix`, since the arguments of a function have to be objects, while `IntegrationType` requires the type of the integrand. However, a conversion can be accomplished with the following utility function:

```
template <typename IntDomain, typename Integrand, typename IntTag>
IntegrationType<IntDomain, Integrand, IntTag>
integral(Integrand const &, IntTag const &)
{
  return IntegrationType<IntDomain, Integrand, IntTag>();
}
```

An additional overloading of `integral` also provides a default integration rule. With this in hand, we can write from an end-user's perspective

```
assembleMatrix(domain, matrix, rhs,
     integral<Omega>(Gradient_u() * Gradient_v(), QuadraticIntegrationTag()
        )
               );
```

and the internals of `assembleMatrix` can then process the supplied integration domain. This code snippet now fully accounts for the left-hand side of the weak formulation given in (3.2).

From the mathematical point of view, an integral behaves like other scalar expressions: Integral expressions can be accumulated, multiplied with scalars and much more. Consequently, `IntegrationType` is another member of the expression engine. For `operator+`, the code is

```
template <typename EXPR1, typename DOM1, typename INTTAG1,
          typename EXPR2, typename DOM2, typename INTTAG2>
Expression< ExpressionDefaultScalarType,
            IntegrationType<EXPR1, DOM1, INTTAG1>,
            IntegrationType<EXPR2, DOM2, INTTAG2>,
            op_plus<ExpressionDefaultScalarType>
          >
operator+(IntegrationType<EXPR1, DOM1, INTTAG1> const & lhs,
          IntegrationType<EXPR2, DOM2, INTTAG2> const & rhs)
{
  return Expression< ExpressionDefaultScalarType,
                     IntegrationType<EXPR1, DOM1, INTTAG1>,
                     IntegrationType<EXPR2, DOM2, INTTAG2>,
                     op_plus<ExpressionDefaultScalarType>
                   > (lhs, rhs);
}
```

In order to support the sum of three integrals, the sum of `IntegrationType` with `Expression` has to be overloaded as well. This leads to several code blocks very similar to the one just shown. One has to keep in mind that `operator+` is by default not commutative with respect to the types of its arguments, therefore two operator overloads have to be given if two different types are involved.

Summing up, we can now write for the assembly of $\int_\Omega \nabla u \cdot \nabla v + uv \,\mathrm{d}x$:

```
//typedefs for Gradient_u, Gradient_v, u and v here

assembleMatrix(domain, matrix, rhs,
               integral<Omega>(Gradient_u() * Gradient_v()
                               + u()*v(),
                               QuadraticIntegrationTag())
             );

//or alternatively:
assembleMatrix(domain, matrix, rhs,
               integral<Omega>(Gradient_u() * Gradient_v(),
                               QuadraticIntegrationTag()) +
               integral<Omega>(u()*v(),
                               QuadraticIntegrationTag())
             );
```

where the second form allows control over the quadrature rules used for each summand.

For the assembly of the right-hand side vector, the same extensions for `assembleRHS` can be realised as for `assembleMatrix`, so that $\int_\Omega 1 \cdot v \,\mathrm{d}x$ transferred to code finally reads

```
typedef basisfun<1>        v;
assembleRHS(domain, rhs,
            integral<Omega>(v(), QuadraticIntegrationTag())
           );
```

We still have to discuss the internals of `IntegrationType`: The interface for the evaluation of an integral is nothing but a redirection to integration rules implemented in the `ElementTag` of the current element over which integration is carried out:

```
template <typename IntDomain, typename Integrand, typename IntTag>
struct IntegrationType
{
  //typedefs as before

  template <typename VectorType, typename Element,
            typename BF1, typename BF2>
  static double evaluate(VectorType const & prev_res1,
                         VectorType const & prev_res2,
                         Element & elem, BF1 const & bf1, BF2 const & bf2)
  {
   return Element::ElementTag::integrate(Integrand(prev_res1, prev_res2),
                                         elem, bf1, bf2, IntTag());
  }
};
```

The first two arguments of `evaluate` are used for `evalResult`, which is a placeholder in the weak formulation for already computed solutions from preprocessing steps. This placeholder was already discussed in the former work of the author [26] and since its functionality is unchanged, we are not going into further details of `evalResult` here. The argument `elem` is the domain element over which integration is carried out, while `bf1` and `bf2` is the pair of basis functions that has to be substituted into the `Integrand`. The final implementation of `IntegrationType` provides a second member function `evaluate`, which takes only one basis function argument and is used for setting up the right-hand side vector.

Thanks to the template mechanism, the template member function `evaluate` works for boundary integrals as well, since an integration is directly provided by the element type. For example, an integration rule for tetrahedra and exact for constant and linear polynomials is implemented in `TetrehedronTag` as

```
struct TetrahedronTag
{
  //other members and typedefs untouched

  template <typename EXPR, typename Element,
            typename Basisfun1, typename Basisfun2>
  static double integrate(EXPR const & expr, Element & elem,
                          Basisfun1 const & bf1,
                          Basisfun2 const & bf2, LinearIntegrationTag)
  {
    typedef typename Element::Configuration                DomainConfig;
    typedef typename DomainTypes<DomainConfig>::PointType  PointType;
     return elem.get_det_dF_dt() *
            Element::ElementTag::getRefVolume() *
            expr(cell, bf1, bf2, PointType(1.0/4.0, 1.0/4.0, 1.0/4.0));
  }
};
```

Such an integration mechanism was already available from the fundamental work [26].

3.3 Equation Specification and Rearrangement

Specifying the left-hand side and the right-hand side of the weak formulation in two different functions is not fully satisfactory: On the one hand, a certain degree of code duplication becomes necessary,

since both require an iteration over all basis functions. On the other hand it does not mnemonically represent the weak formulation, whose core is an *equation*.

Let us therefore introduce an `EquationType` that holds the types of the left-hand side and the right-hand side of an equation:

```
template <typename LHS, typename RHS>
struct EquationType
{
  typedef LHS  LHSType;
  typedef RHS  RHSType;
};
```

For a mnemonic representation of equations, the equality sign is overloaded for `IntegrationType` and `Expression`, so that it becomes possible to use a single assembly function named `assemble`:

```
//typedefs for Gradient_u, Gradient_v and v
assemble(domain, matrix1, rhs1,
         integral<Omega>( Gradient_u() * Gradient_v(),
                          QuinticIntegrationTag()) =
         integral<Omega> ( v(), QuinticIntegrationTag() )
        );
```

This way, the necessary code fully reflects the mathematical formulation and simplifies the implementation of a new weak formulation as much as possible. Since the integration tags are optional arguments, the absolute minimum of code for the specification of this particular weak formulation is

```
//typedefs for Gradient_u, Gradient_v and v
assemble(domain, matrix1, rhs1,
         integral<Omega>( Gradient_u() * Gradient_v() ) =
         integral<Omega>( v() ) );
```

There are two more modifications necessary to match the final implementation: First, the function `assemble` expects an additional configuration template parameter. The configuration for the finite element method is done in analogy to the domain configuration in the previous chapter. A single class holds the necessary type definitions:

```
struct Laplace_Config
{
  typedef ScalarTag                       ResultDimension;
  typedef TypeErasureTag                  BasisfunTreatmentTag;
  typedef NoBoundaryMappingTag            MappingTag;

  typedef QuadraticBasisfunctionTag       TestSpaceTag;
  typedef QuadraticBasisfunctionTag       TrialSpaceTag;

  typedef SegmentConnectionKey<0>         SegmentConnection;

  typedef MappingKeyType<0>               MappingKey;
  typedef BoundaryKeyType<0>              BoundaryKey;
  typedef BoundaryKeyType<11>             BoundaryData;
};
```

Some of the type definitions showed up in the fundamental work together with the domain configuration. The present implementation fully decouples the domain configuration (shown in the previous chapter) and the FEM configuration shown here. The meaning of the type definitions are:

- `ResultDimension`: Whether the underlying PDE has a scalar-valued (`ScalarTag`) or vector-valued (`VectorTag`) solution.
- `BasisfunTreatmentTag`: Basis functions are internally handled using a full type encoding (`TypeListTag`) or as type erased objects (`TypeErasureTag`). See Section 1.3 for a short description of these two concepts.
- `MappingTag`: Whether basis functions on boundary elements are first assembled into the matrix and then eliminated (`FullMappingTag`) or directly written to the right-hand side during assembly (`NoBoundaryMappingTag`).
- `TestSpaceTag`: The polynomial degree of the basis functions of the test space.
- `TrialSpaceTag`: The polynomial degree of the basis functions of the trial space.
- `SegmentConnection`: The key used for coupled segments (will be explained in Section 3.5)
- `MappingKey`: The type used for storing mapping indices using `QuantityManager` on elements (will be explained in Chapter 4).
- `BoundaryKey`: The type used for storing Dirichlet boundary conditions using `QuantityManager`.
- `BoundaryData`: The key used for Dirichlet boundary data (which might be vector-valued) stored in `QuantityManager`.

The second modification to the `assemble` function given above is that the first argument is a segment instead of the whole domain. The final assembly instruction using `Laplace_Config` thus is

```
1 //typedefs for Gradient_u, Gradient_v and v
2 assemble<Laplace_Config>(segment, matrix, rhs,
3                           integral<Omega>( Gradient_u() * Gradient_v() ) =
4                           integral<Omega>( v() )
5                          );
```

3.4 Logicals

The full decoupling of domain configuration and FEM configuration allows to solve several PDEs on the same mesh – if required even simultaneously. This very pleasant constellation has one cumbersome side-effect: One has to store the boundary conditions for each PDE of interest in some way. Although there is a possibility to provide the boundary conditions directly with the mesh file, one is then limited to partial differential equations with the same boundary conditions. Alternatively, one has to write a special function for a particular PDE that sets the boundary conditions appropriately. Doing so, one spends a lot more time in setting the boundary conditions than in specifying the weak formulation. To overcome such hassle, this section establishes a generic way of applying boundary conditions to meshes whose geometric shape is known at compile time and discusses possibilities to finally relax this constraint.

Let us go back to the initial construction of compile time expressions: The placeholder `var<0>` is used for the first coordinate of a `point`. For such a placeholder, operations `+`, `-`, `*`, and `/` are defined for its objects (i.e. `var<0>()`). For such an object `x_`, evaluation of `(x_ + x_)(p)` for some point `p` is equal to `x_(p) + x_(p)`, so the result for `p` equal to a point $(1, 2, 3)$ is `1 + 1 = 2`. However, for scalar types, many more operators are defined, like `<`, `>` and `&&`. Because of this, the expression engine is extended to support logical operations as well.

Following the ideas from the previous paragraph, let us introduce new operator tags:

```
struct op_and
{
  template<typename T, typename U>
  static bool apply(T const & lhs, U const & rhs)
  { return lhs && rhs; }
};

struct op_equal
{
  template<typename T, typename U>
  static bool apply(T const & lhs, U const & rhs)
  { return lhs == rhs; }
};

//and in a similar way:
//op_or, op_unequal, op_greaterthan, op_lessthan
```

and a new class:

```
template <typename LHS, typename RHS, typename OP>
struct LogicalExpression
{
  LogicalExpression(LHS const & lhs, RHS const & rhs)
   : lhs_(lhs), rhs_(rhs) {};

  template <typename Point>
  bool operator()(Point const & p) const
  {
    return OP::apply( lhs_(p), rhs_(p) );
  }
};
```

Although `LogicalExpression` has similar functionality like `Expression`, we will still keep two different classes to emphasize that the first returns a boolean (i.e. either `true` or `false`), while the second returns a scalar (e.g. a `double`).

The operator overloads are straight-forward (but lengthy) and similar to the ones shown in Section 1.1. With this we can now write code like

```
var<0> x_; var<1> y_; var<2> z_;
Point p(1.0, 2.0, 3.0);

(x_ == y_)(p);          //result: false
(x_ < y_)(p);           //result: true
((x_ + y) == z_)(p)     //result: true
```

As can be seen from the last code line, the existing expression engine is seamlessly integrated into the new logical expressions. One has to be aware that the use of == does not necessarily work as expected due to round-off errors: A better choice is to use a fuzzy check for equality using two inequalities – we will come back to this later.

The correct handling of scalars with logical expressions is still missing. For example, the following code is still invalid:

```
(x_ == 1.0)(p)          //error: no matching function for '1.0(p)'
```

The reason for the break-down is that `p` is applied to both sides of the operand. However, the compiler does not know how to handle the call to the parenthesis operator for a floating point variable. The remedy is to use a helper class that calls the parenthesis operator for all types except scalars:

```
template <typename T>
struct PointEvaluator
{
  template <typename Point>
  static double apply(T const & expr, Point const & p)
  {
    return expr(p);
  }
};

template <>
struct PointEvaluator<double>
{
  template <typename Point>
  static double apply(double value, Point const & p)
  {
    return value;
  }
};

//similar for float, long and other scalar types
```

With this helper class, all we have to do is to replace the return statement in the parenthesis operator of `LogicalExpression` with

```
    return OP::apply( PointEvaluator<LHS>::apply(lhs_, p),
                      PointEvaluator<RHS>::apply(rhs_, p) );
```

As closing remark for this section it has to be mentioned that it is possible to use logical expressions within the specification of the weak formulation. For the weak formulation

$$\text{Find } u \in H_0^1([0,1]) \text{ such that } \int_0^1 \nabla u \cdot \nabla v \,\mathrm{d}x = \int_0^1 f v \,\mathrm{d}x \quad \text{for all } v \in H_0^1([0,1]), \tag{3.3}$$

$$f(x) = \begin{cases} 0, & x \leq 0.5\,, \\ 1, & x > 0.5\,, \end{cases} \tag{3.4}$$

one possible assembly instruction is

```
//typedefs for Gradient_u, Gradient_v and v
var<0> x_;
assemble<Laplace_Config>(segment, matrix, rhs,
                         integral<Omega>( Gradient_u() * Gradient_v() ) =
                         integral<Omega>( (x_ > 0.5) * v() ) );
```

This (ab)uses the conversion rules for booleans: `false` is always converted to 0, while `true` is always converted to 1. Nevertheless, logical expressions can be of much greater use when it comes to the specification of boundaries and boundary conditions.

3.5 Boundary Specification

Since we are able to use floating point numbers within logical expressions, we can even provide functions that set the boundary values on a mesh. For example, setting all elements of a segment `seg1` with first

coordinate equal to 1.0 to a Dirichlet boundary value of 0.0 reads

```
var<0> x_;
setDirichletBoundary<FEMConfig>(seg1, (x_ == 1.0), 0.0 );
```

The template argument is the FEM configuration, where the key type for the `QuantityManager` is defined. The second argument is the logical expression and the third argument is the boundary value. It is also possible to use an expression instead of a value as third argument.

We have been quite fuzzy with "all domain elements with first coordinate equal to 1.0": Only for vertices the meaning is clear. However, for all higher level domain elements, the strategy is to check all vertices and the barycenter together with the boundary flag for the condition on the x coordinate. If one of these checks fails, the element is not considered for Dirichlet boundary data.

Bearing the finite floating point precision in mind, it is advantageous to specify intervals instead of hard comparisons, so

```
//method 1:
setDirichletBoundary<FEMConfig>(seg1, (x_ < 1.01) && (x_ > 0.99, 0.0 );

//method 2:
setDirichletBoundary<FEMConfig>(seg1, (x_ == 1.0), 0.0 );
```

are expected to yield the same results for a sufficiently coarse mesh, but the first method is much more robust against numerical noise.

With the use of logical expressions, it is now much easier to set the boundary data for different PDEs: We can use different FEM configurations (e.g. a different key for the mapping numbers and the boundary data) and call `setDirichletBoundary` with the particular configuration class and set appropriate boundary data.

Similar to the specification of Dirichlet boundaries, Neumann boundaries can be specified:

```
template <long Id, typename Segment, typename KEY>
void setBoundaryArc(Segment & seg, KEY const & key);
```

All boundary facets for which the expression `key` evaluates to `true` at the facet's midpoints are tagged with `Gamma<id>`, which can then be used as integration domain. Unlike for `setDirichletBoundary`, there is no FEM configuration parameter necessary: Neumann boundary conditions are incorporated into the weak formulation directly. This is similar to the choice of mathematical spaces: Dirichlet boundary conditions are usually incorporated into the underlying Sobolev spaces, while Neumann boundaries do not show up in these spaces.

For the treatment of multi-segment problems, where additional boundary effects have to be incorporated, a coupling of segments is necessary. In particular, each facet on the interface of two segments has to be coupled algorithmically, which is achieved by storing a pointer to the twin facet (that is the facet with the same geometric location, but stored on the other segment). First, we introduce another tag:

```
template <long id>
struct Interface;
```

which is more or less identical to `Gamma`, except that the integrand is expected to be made up from functions of both segments of the interface. Although there is no mathematical distinction between `Gamma` and `Interface`, it forces end-users of the framework to clearly distinguish between coupled and uncoupled boundary integrals. Alternatively, one could have provided a modified basis function placeholder like for example

```
basisfun<2, coupled_tag>
```

but since such a type is likely to be hidden behind a type definition at the specification of the weak formulation as code, so that a distinction between coupled and uncoupled boundary integrals may not be immediate anymore.

A boundary element coupling is provided by

```
template <long Id, typename Segment, typename KEY>
void setInterface(Segment & seg1, Segment & seg2, KEY const & key)
```

The template parameter `Id` allows to have several types with name `Interface` to store the pointer to the twin element. The internals of `setInterface` iterate over all boundary facets whose midpoints evaluate to `true` when applied to `KEY`. The third argument `KEY` is optional: It speeds up the interface detection considerably if chosen properly. However, if it is not supplied, all boundary facets of `seg1` are compared with all boundary facets of `seg2`. For every pair of facets with the same geometric location, the pointer to the other member of the pair is stored. To allow for `QuantityManager` to operate on a per-segment basis, additionally the pointer to interfacing segment is stored on each facet. Since the number of facets on an interface is typically much smaller than the total number of cells, the additional memory requirements are negligible. On the other hand, it is possible that each facet of a segment is connected to a different segment, thus storing a segment pointer on each facet separately is indeed necessary.

In cases where uncoupled boundary terms at interfaces appear, an additional helper function that tags all facets of a particular interface is provided:

```
template <long GammaId, long InterfaceID, typename Segment>
void setBoundaryArcAtInterface(Segment & seg);
```

All facets of a segment `seg` that carry an `Interface<InterfaceID>` tag are also tagged with `Gamma <GammaId>`. Such a situation is rather common: Coupled segments may lead to both coupled and uncoupled boundary integral contributions. An application is given in Chapter 7.

There is another possibility for the coupling of segments: Common degrees of freedom on the interface. One such scenario is presented in Section 7.2. For the finite element method this means that facets with the same geometric location have to carry the same global basis function numbers. How this distribution of global basis function numbers is actually done is shown in the next chapter, we only mention the top level function for future reference:

```
template <typename FEMConfig, typename Segment, typename KEY>
void setSegmentConnection(Segment & seg1, Segment & seg2, KEY const & key)
```

This function tags all facets, which have a midpoint fulfilling `KEY` on the common interface of `seg1` and `seg2`, with a key whose type is given by `FEMConfig::SegmentConnection` (see also Section 3.3). Just as for `setInterface`, pointers to the other facet and its lower level elements are stored. When the global basis function numbers are assigned to all elements of the segment, connected elements are considered only once. Again, the third argument is optional.

3.6 The FEM Core

So far we have discussed the classes used for the specification of the variational problem. In the current section we have look at the internals of the FEM assembly routines and see how the weak formulation is used for the assembly of the system matrix and the right-hand side vector. The general domain decomposition strategy introduced in the foregoing chapter and the extensions to the expression engine allow us to formulate the assembly routines sufficiently general, so that topics from the following chapters can be incorporated easily. While going through the implementation, the reader is advised to have a look at Fig. 3.1 while going through the code snippets.

Let us begin with the top level interface function `assemble`, which reads

```
template <typename FEMConfig, typename Segment,
          typename MatrixType, typename VectorType,
          typename EquationArray>
void assemble(Segment & seg, MatrixType & matrix, VectorType & rhs,
              EquationArray const & eqnarray,
              VectorType & prev_result1, VectorType & prev_result2)
{
  // typedefs for CellTag and BasisfunTreatmentTag omitted

  // First step: Collect all contributions from first
  // integration domain and assemble according to the assembly tag:
  typedef typename ASSEMBLY_GET_FIRST_INT_DOMAIN<
                       EquationArray>::ResultType  IntDomain;
  typedef typename ASSEMBLY_EXTRACT_DOMAIN<EquationArray,
                       IntDomain>::ResultType      CurrentEquation;

  assemble_impl<FEMConfig>(seg, matrix, rhs,
                           CurrentEquation(),
                           prev_result1, prev_result2,
                           BasisfunTreatmentTag());

  // Second step: call this function again
  // with already assembled contributions removed:
  typedef typename ASSEMBLY_REMOVE_DOMAIN<EquationArray,
                        IntDomain>::ResultType     RemainingEquation;

  assemble<FEMConfig>(seg, matrix, rhs,
                      RemainingEquation(),
                      prev_result1, prev_result2);
}
```

In a recursive manner an iteration over all integration domains is performed. These integration domains are `Omega`, `Gamma<>` and `Interface<>`, so that the assembly is performed over each integration domain separately (note that for example `Gamma<1>` and `Gamma<2>` are distinct integration domains). The meta function `ASSEMBLY_EXTRACT_DOMAIN` extracts all integrals in the supplied `EquationArray` that have a given integration domain `IntDomain` and passes these terms to `assemble_impl`. After that, only terms with integration domains other than `IntDomain` continue the recursion. The recursion stops if all integration domains are assembled by means of overloading `assemble` with respect to a type `EmptyEquation`, which represents the trivial statement $0 = 0$.

Next, we have a look at how each integration domain is assembled within `assemble_impl`. There are two almost identical implementations: One is for type erased basis functions, which require a run time loop over all basis functions defined on a cell, whereas type listed basis functions require a loop at compile time. We consider the type listed case here:

```
template <typename FEMConfig, typename Segment,
          typename MatrixType, typename VectorType,
          typename EquationArray>
void assemble_impl(Segment & seg,
                   MatrixType & matrix, VectorType & rhs,
                   EquationArray const & eqnarray,
                   VectorType & prev_result1, VectorType & prev_result2,
                   TypeListTag)
{
  //all typedefs omitted
```

```
11
12    //Iterate over all AssemblyCells:
13    for (AssemblyCellIterator acit = seg.template getLevelIteratorBegin<
          AssemblyCellTag::TopoLevel>();
14         acit != seg.template getLevelIteratorEnd<AssemblyCellTag::TopoLevel
              >();
15         ++acit)
16    {
17      //iterate over elements in the integration domain only, skip others:
18      if ( IntegrationDomainChecker<IntDomain>::apply(*acit) == true)
19      {
20        MapIterator_v mapit_v(*acit);
21        MapIterator_u mapit_u(*acit);
22
23        //let MatrixIterator do the job now:
24        MatrixIterator<FEMConfig, AssemblyCellType, Equation,
25                       BFIterator, BFIterator
26                      >::assemble(matrix, rhs, *acit,
27                                  prev_result1, prev_result2,
28                                  mapit_v, mapit_u);
29      }
30    } //for
31  };
```

We have omitted all type definitions to keep the code snippet short. `AssemblyCellIterator` is the iterator over all elements of the topology level specified by the integration domain. In this way, we are able to reuse the same code for boundary integrals. `IntegrationDomainChecker` assures that only elements are assembled which are part of the integration domain. Of high importance from the point of abstraction are `MapIterator_u` and `MapIterator_v` together with `BFIterator` (a type iterator over all basis functions defined on the reference element). They provide the global basis function numbers associated with the current basis function defined on the local reference element.

`MatrixIterator` is responsible for the iteration over all basis function pairs. The name is derived from local element matrices, which are also obtained by an iteration over all basis function pairs defined on the local reference element. The implementation for type erased basis functions is

```
1  template < /* several template parameters */ >
2  struct MatrixIterator
3  {
4    template < /* several template parameters */ >
5    static void assemble( /* several parameters */ )
6    {
7      //some typedefs omitted
8
9      for (long i=0; mapit_v.valid(); ++i, ++mapit1)
10     {
11       //check for Dirichlet boundary conditions
12       if (*mapit_v == -1)
13         continue;
14
15       for (long j=0; mapit_u.valid(); ++j, ++mapit2)
16       {
17         if (*mapit_u != -1)
18           //write matrix entry here
19         else
20           //write Dirichlet boundary conditions to rhs-vector here
```

```
      }
      //write RHS here
    }
  }
};
```

The program statements for writing the entries (either to the matrix or to the right-hand side vector) are rather lengthy, but consist of one call to another static function each. For writing a matrix entry, this statement is

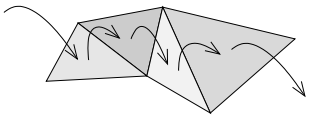
```
EntryWriter
< TypeListIterator< typename Equation::LHSType,
                    ResultDimension::dim,
                    ResultDimension::dim>
>::apply( matrix,
          ResultDimension::dim * (*mapit_v),   //global index for v
          ResultDimension::dim * (*mapit_u),   //global index for u
          cell,
          basisfuns_v[i],      //basis function for u
          basisfuns_u[j],      //basis function for v
          prev_result1, prev_result2 );
```

Here, `EntryWriter` is responsible for the support of vector-valued weak formulations and iterates by means of a `TypeListIterator` over all components of the weak formulation. For such vector-valued solutions, the vector-valued test functions are built from scalar basis function by a multiplication with the (vector-valued) unit vectors, so that for each scalar-valued basis function, `ResultDimension::dim` vector-valued basis functions emerge.

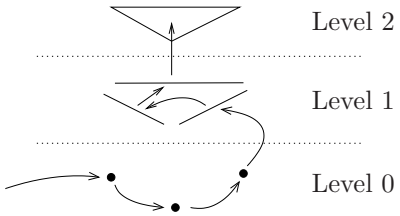
The implementation of `EntryWriter` is again very short, but the number of template and function arguments is rather large.

```
template <typename TListIter>
struct EntryWriter
{
  template < /* several template parameters */ >
  static void apply( /* several function parameters */ )
  {
    typedef typename TListIter::ResultType      ExpressionType;

    //write the matrix entry:
    matrix(rowindex + TListIter::rowindex,
           colindex + TListIter::colindex)
        += ExpressionType::evaluate(prev_result1, prev_result2,
                                    cell, bf_v, bf_u);

    //next entry in the weak formulation:
    EntryWriter<typename TListIter::IncrementType
                >::apply( /* same function paramters again */ );
  }

  //similarly for entries to the right-hand side vector
};
```

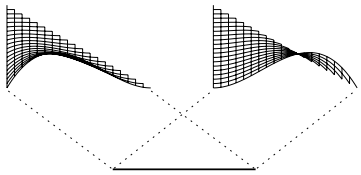
Thanks to the abstraction of an iteration over all entries of the weak formulation using `TListIter`, the implementation of `EntryWriter` is very short. The call to `ExpressionType::evaluate` evaluates the mathematical expression given by `ExpressionType`, which is in the scalar-valued case just the left-hand



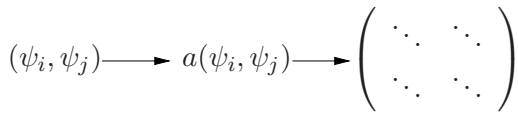
(a) Iterate over all cells.



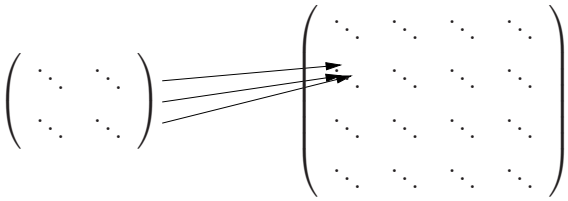
(b) Iterate over all topology levels and elements of the cell.



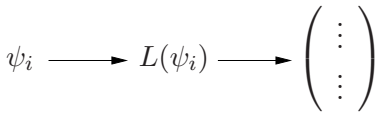
(c) Iterate over all (scalar) basis functions associated with each element. (Here: Edge with two associated basis functions)



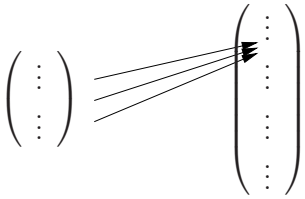
(d) For all pairs (ψ_i, ψ_j) of basis functions obtained from separate iterations in steps (b) and (c), evaluate the (possibly matrix-valued) bilinear form.



(e) Write the (possibly matrix-valued) result of the bilinear form to the system matrix, where the row and column indices are obtained from the `MappingIterator`s associated with each basis function.



(f) For all basis functions ψ_i of the test space, evaluate the (possibly vector-valued) linear form L.



(g) Write the (possibly vector-valued) result of the linear form to the global right-hand side vector, where the row index is obtained from the `MappingIterator` associated with ψ_i.

Figure 3.1: Overview over the FEM assembly core components for the weak form $a(u, v) = L(v)$. For vector-valued basis functions, the intermediate result vectors and matrices in (d) and (f) do NOT correspond to the element matrices typically given in the literature for scalar problems.

side of the weak formulation, on the cell `cell` with basis functions `bf_v` and `bf_u`. This evaluation was already outlined in Section 3.1, therefore the description of the FEM assembly core is complete.

In Fig. 3.1 the necessary steps are illustrated. In the code snippets given in this section, direct associations with the iteration steps can be given: `AssemblyCellIterator` corresponds to Fig. 3.1(a), `MapIterator_v` and `MapIterator_u` resemble Fig. 3.1(b) and Fig. 3.1(c) for the mapping numbers of the basis functions, which are provided one after another by `BFIterator`. This double-iteration is unified in `MatrixIterator`. The actual evaluations and matrix entries in Fig.s 3.1(d) to 3.1(g) are carried out by `EntryWriter`, where the intermediate results are not set up explicitely, instead they are written directly to the system matrix and the right-hand side vector.

Chapter 4

A General Mapping Strategy

In the early days of FEM, mostly piecewise linear basis functions have been used. Even nowadays, most engineering FEM tools use at most piecewise quadratic basis functions. Keeping the analogy with adaptive integration schemes in mind, there is in fact no need to use higher order basis functions in the vicinity of singularities of the solution, which is well confirmed by the mathematical theory. However, in regions where the solution is smooth, a substantial amount of unknowns can be saved by the use of higher order basis functions, which give much better approximations. This idea has led to the development of the so-called *hp*-FEM, with usually best convergence properties among adaptive refinement strategies.

While the degrees of freedom can be associated with a mesh's vertices for piecewise linear basis functions and with vertices and edges for piecewise quadratic basis functions, there is typically no direct one-to-one association with particular domain elements for basis functions of higher degree. For spatial dimensions greater than one, additional orientation issues have to be taken into account. For example, an edge can be oriented in two different ways, so adjacent triangles sharing this edge may use a different local orientation of that edge. So, if there are two or more degrees of freedom associated with this edge, its orientation does matter. This gets even more complicated for triangular facets in three dimensions.

Since basis functions are usually defined on a reference element with some predefined orientation, it is from the software implementation's point of view very appealing to have an iterator available that returns the numbers of the global basis functions just in accordance to the sequence of the locally defined basis functions. In this chapter a way to iterate over global basis function numbers is presented, which is not taylored to a particular geometry of the underlying cell. The presented algorithm finally works for the family of Lagrange and Lobatto basis functions on simplex geometries and brick-like geometries in arbitrary dimensions! The algorithm can also be adapted to other families of basis functions with only minor modifications.

4.1 Construction of Basis Functions of Arbitrary Degree

As first step we look at the construction of basis functions of arbitrary degree on the reference elements. We follow an H^1-conforming construction as given by Ŝolín et al. [27].

Let us start with the one dimensional case: Here, cells are straight lines. For a single cell and piecewise linear basis functions, the local basis functions associated to that cell are straight lines that evaluate to 1 in one node $\mathbf{v}_0$ and to 0 in the other node $\mathbf{v}_1$. For the reference element $\mathbf{c} = [0, 1]$, the two local basis functions are thus

$$\psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi) = 1 - \xi, \quad \psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi) = \xi. \tag{4.1}$$

One way to define higher order basis functions on this reference cell is to use products of these vertex functions. For quadratic basis functions, a possible basis is

$$\psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi) = 1 - \xi, \quad \psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi) = \xi, \quad \psi_{\mathbf{c}}^{\mathbf{c}}(\xi) = \psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi)\psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi) = (1-\xi)\xi. \tag{4.2}$$

More generally, basis functions of degree d can in one dimension be constructed as

$$\psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi) = 1 - \xi, \tag{4.3}$$
$$\psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi) = \xi, \tag{4.4}$$
$$\psi_{k,\mathbf{c}}^{\mathbf{c}}(\xi) = \left(\psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi)\right)^{d-k-1}\left(\psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi)\right)^{(k+1)} = (1-\xi)^{d-k-1}\xi^{k+1}, \quad 0 \le k < d-1. \tag{4.5}$$

Thus, basis functions in the interior of the one dimensional reference cell can be associated with nodal powers on that element. This motivates the following definition:

Definition 1. *On a reference cell* $\mathbf{c}$ *with vertices* $\mathbf{v}_0, \mathbf{v}_1, \dots \mathbf{v}_{n-1}$ *a polynomial* $\psi_{\mathbf{c}}^{\mathbf{v}_i}$ *that fulfills*

$$\psi_{\mathbf{c}}^{\mathbf{v}_i}(\mathbf{v}_j) = \delta_{ij} \tag{4.6}$$

is called vertex function *of a vertex* $\mathbf{v}_i$*.*

The number of vertex functions on a cell is infinite. However, for the construction of a nodal basis we only need one vertex function for each vertex, so that linear and, if required, quadratic vertex functions are chosen for a nodal basis. Of particular interest are basis that emerge from product of a nodal basis:

Definition 2. *On a cell* $\mathbf{c}$ *with a set of vertices* $\mathcal{E}_0(\mathbf{c}) = \{\mathbf{v}_0, \mathbf{v}_1, \dots \mathbf{v}_{n-1}\}$ *and a set of vertex functions* $\left\{\psi_{\mathbf{c}}^{\mathbf{v}_0}, \psi_{\mathbf{c}}^{\mathbf{v}_1}, \dots, \psi_{\mathbf{c}}^{\mathbf{v}_{n-1}}\right\}$*, a* nodal product basis *is a basis for some function space that consists of basis functions of the form*

$$\left(\psi_{\mathbf{c}}^{\mathbf{w}_0}\right)^{k_0}\left(\psi_{\mathbf{c}}^{\mathbf{w}_1}\right)^{k_1} \cdots \left(\psi_{\mathbf{c}}^{\mathbf{w}_{m-1}}\right)^{k_{m-1}}, \qquad 0 < m \le n \tag{4.7}$$

only, where $k_i \in \mathbb{N}^+$*, for* $i = 0, \dots, m-1$ *and the pairwise distinct* $\mathbf{w}_0, \dots \mathbf{w}_{m-1} \in \mathcal{E}_0(\mathbf{c})$ *are in case* $m < n$ *the vertices of a lower level element of* $\mathbf{c}$*.*

However, a nodal product basis leads to a poor matrix condition number for the Laplace operator, hence *Lobatto shape functions* $l_k(\tilde{\xi})$ are used in one dimension on the reference element $[-1, 1]$ with $\tilde{\xi} := 2\xi - 1$. The key observation now is that l_k, a polynomial of degree $k > 1$, can be written as

$$l_k(\tilde{\xi}) = l_0(\tilde{\xi}) l_1(\tilde{\xi}) \phi_{k-2}(\tilde{\xi}) \tag{4.8}$$

where $\phi_{k-2}(\tilde{\xi})$ is a so-called *kernel function.*

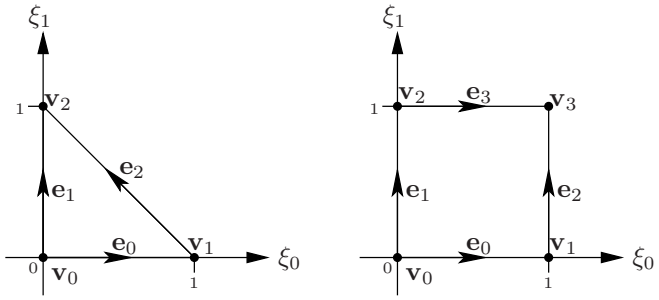
Since $l_0(\tilde{\xi})$ and $l_1(\tilde{\xi})$ vanish on one end of the reference interval and equal 1 on the other end, we can relate nodally constructed basis functions with Lobatto shape functions in the following way:

$$\psi_{k,\mathbf{c}}^{\mathbf{c}}(\xi) = \psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi)\psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi)\left(\psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi)\right)^{d-k-2}\left(\psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi)\right)^{k} =: \psi_{\mathbf{c}}^{\mathbf{v}_0}(\xi)\,\psi_{\mathbf{c}}^{\mathbf{v}_1}(\xi)\,\hat{\phi}_k(\xi), \quad 0 \le k < d-1 \tag{4.9}$$

Now, since the transition from nodally constructed basis functions to (transformed) Lobatto basis functions is a replacement of kernel functions $\hat{\phi}_k(\xi)$ with $\phi_k(\tilde{\xi})$ only, it is sufficient to construct a mapping scheme for a nodally constructed basis.

In higher dimensions, a similar construction can be done: For example, on a triangle $\mathbf{t}$ with labels and orientations given in Fig. 4.1(a), the vertex functions are again linear functions that vanish on all vertices of the triangle except for one:

$$\psi_{\mathbf{t}}^{\mathbf{v}_0}(\boldsymbol{\xi}) = 1 - \xi_0 - \xi_1\ , \tag{4.10}$$
$$\psi_{\mathbf{t}}^{\mathbf{v}_1}(\boldsymbol{\xi}) = \xi_0\ , \tag{4.11}$$
$$\psi_{\mathbf{t}}^{\mathbf{v}_2}(\boldsymbol{\xi}) = \xi_1\ . \tag{4.12}$$



(a) Reference triangle. (b) Reference quadrilateral.

Figure 4.1: Labels and orientations of two reference elements.

Then, $d-1$ edge functions $\psi^{\mathbf{e}_1}_{k,\mathbf{t}}$ of degree d can be constructed on each edge from vertex functions that do not vanish on that edge:

$$\psi^{\mathbf{e}_0}_{k,\mathbf{t}} = (\psi^{\mathbf{v}_0}_{\mathbf{t}})^{k+1}\,(\psi^{\mathbf{v}_1}_{\mathbf{t}})^{d-k-1}, \tag{4.13}$$

$$\psi^{\mathbf{e}_1}_{k,\mathbf{t}} = (\psi^{\mathbf{v}_0}_{\mathbf{t}})^{k+1}\,(\psi^{\mathbf{v}_2}_{\mathbf{t}})^{d-k-1}, \tag{4.14}$$

$$\psi^{\mathbf{e}_2}_{k,\mathbf{t}} = (\psi^{\mathbf{v}_1}_{\mathbf{t}})^{k+1}\,(\psi^{\mathbf{v}_2}_{\mathbf{t}})^{d-k-1}, \tag{4.15}$$

where $0 \le k < d-1$. Again, such a nodally constructed basis leads to a poor matrix condition number, therefore a basis that accounts for Lobatto shape functions is used in practice:

$$\psi^{\mathbf{e}_0}_{k,\mathbf{t}} = \psi^{\mathbf{v}_0}_{\mathbf{t}}\,\psi^{\mathbf{v}_1}_{\mathbf{t}}\,\phi_k(\lambda_{1,\mathbf{t}} - \lambda_{0,\mathbf{t}}), \tag{4.16}$$

$$\psi^{\mathbf{e}_1}_{k,\mathbf{t}} = \psi^{\mathbf{v}_0}_{\mathbf{t}}\,\psi^{\mathbf{v}_2}_{\mathbf{t}}\,\phi_k(\lambda_{2,\mathbf{t}} - \lambda_{0,\mathbf{t}}), \tag{4.17}$$

$$\psi^{\mathbf{e}_2}_{k,\mathbf{t}} = \psi^{\mathbf{v}_1}_{\mathbf{t}}\,\psi^{\mathbf{v}_2}_{\mathbf{t}}\,\phi_k(\lambda_{2,\mathbf{t}} - \lambda_{1,\mathbf{t}}), \tag{4.18}$$

where $0 \le k < d-1$ and ϕ_k are again the Lobatto kernel functions and the $\lambda_{i,t}$ are affine coordinates (aka. barycentric coordinates) associated with the triangle such that $\lambda_{i,t}(\mathbf{v}_j) = \delta_{ij}$.

Just as in one dimension, one can define kernel functions for the nodal basis functions and then relate them to Lobatto kernel functions. For the construction of triangular *bubble functions* $\psi^{\mathbf{t}}_{k_0,k_1,\mathbf{t}}$, all three vertex functions are multiplied together with appropriate powers. For Lobatto shape functions, a product of two kernel functions now shows up. However, since $\psi^{\mathbf{t}}_{k_0,k_1,\mathbf{t}}$ has two varying indices k_0 and k_1, it is again possible to establish a connection between nodal kernel functions and Lobatto kernel functions.

Similar constructions hold for tetrahedra, rectangles and bricks and can be found in much more detail in Ŝolín et al. [27]. Just as it was shown for one dimension and for a triangle in two dimensions, it is possible to relate nodal basis functions to hierarchic Lobatto basis functions in a unique way by relating appropriate kernel functions.

4.2 Basisfunction-ID and Exponent Vector

From a programmer's point of view, nodal basis functions are much easier to handle, since a basis function $\psi_{\boldsymbol{k},\mathbf{c}}$ of a cell $\mathbf{c}$ with n vertices $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ can be written in the form

$$\psi_{\boldsymbol{k},\mathbf{c}} := (\psi^{\mathbf{v}_0}_{\mathbf{c}})^{k_0} \cdot (\psi^{\mathbf{v}_1}_{\mathbf{c}})^{k_1} \cdots (\psi^{\mathbf{v}_{n-1}}_{\mathbf{c}})^{k_{n-1}} \tag{4.19}$$

with integer exponents $0 \le k_0, k_1, \ldots, k_{n-1}$. Thus, in order to identify a basis function on a cell, it is tempting to store the *exponent vector* $\boldsymbol{k} = (k_0, k_1, \ldots, k_{n-1})$ in some array. For the construction of basis functions, the set of exponent vectors of the same length and the same degree is of interest:

Definition 3. *The set $\mathcal{K}_n^d$ is the set of all exponent vectors $\boldsymbol{k}$ of size n (i.e. with n components) and modulus $|\boldsymbol{k}| = \sum_{i=0}^{n-1} k_i = d$*

Considering a single set $\mathcal{K}_n^d$ per cell is not very appealing: For a given exponent vector, permutations of the entries do not necessarily yield another valid exponent vector on that cell. This complicates iteration over all valid exponent vectors considerably. Let us consider a rectangle $\mathbf{r}$ as shown in Fig. 4.1(b): The quadratic edge functions are $\psi_{\mathbf{r}}^{\mathbf{v}_0}\psi_{\mathbf{r}}^{\mathbf{v}_1}$, $\psi_{\mathbf{r}}^{\mathbf{v}_0}\psi_{\mathbf{r}}^{\mathbf{v}_2}$, $\psi_{\mathbf{r}}^{\mathbf{v}_1}\psi_{\mathbf{r}}^{\mathbf{v}_3}$ and $\psi_{\mathbf{r}}^{\mathbf{v}_2}\psi_{\mathbf{r}}^{\mathbf{v}_3}$, but the functions $\psi_{\mathbf{r}}^{\mathbf{v}_0}\psi_{\mathbf{r}}^{\mathbf{v}_3}$ and $\psi_{\mathbf{r}}^{\mathbf{v}_1}\psi_{\mathbf{r}}^{\mathbf{v}_2}$ are not from the set of nodal basis functions, even though they are a product of two vertex functions. For such a rectangle, any product of three vertex functions is not from the set of basis functions either, therefore using an array to store the exponents also includes the geometry of the cell to some extend.

Let us have a different view at the construction of basis functions: Given some set of vertex functions, the edge functions can be built by taking all edges of a cell and associating the vertices of each edge with the corresponding vertex function. Then, all basis functions for this particular edge are constructed by a multiplication of appropriate powers of the two vertex functions. This scheme is applicable in general: Basis functions of an element $\mathbf{e}_l^{(i)}$ of a topological level l are constructed by taking all vertex functions defined for the element's vertices and creating products thereof. This can be seen as setting up the bubble functions of each level only.

At this point, we have to recall the domain decomposition strategy from Chapter 2: We broke down cells into lower level elements: Given the cell's vertices, lower level elements of the cell are constructed by taking appropriate subsets of the set of vertices. The construction of basis functions follows this pattern: Given a set of vertex functions, construct the bubble functions of all topological levels of the cell. To account for this topological similarity, we use the following basis function identifier:

```
1 template <long topolevel_, long element_id_, long bf_id_>
2 struct BasisFunctionID
3 {
4   enum{ topolevel  = topolevel_,
5         element_id = element_id_,
6         bf_id      = bf_id_ };
7 };
```

Here, `topolevel` denotes the topological level of the basis function (i.e. the level of the element this basis function is a bubble function of), `element_id` is the number of the element on this topological level, and `bf_id` is the index of the bubble function of that element.

For example, for the basis functions of a triangle $\mathbf{t}$ as given in (4.10) to (4.15), there holds the correspondence

$$\texttt{BasisFunctionID} < 0, \texttt{i}, 0 > \longleftrightarrow \psi_{\mathbf{t}}^{\mathbf{v}_i},$$
$$\texttt{BasisFunctionID} < 1, \texttt{i}, \texttt{k} > \longleftrightarrow \psi_{k,\mathbf{t}}^{\mathbf{e}_i}.$$

Since the code now directly holds mathematical parameters, let us mix mathematical notation with code for an illustrative explanation: We have for a cell $\mathbf{c}$ with tag `CellTag`:

$$\begin{aligned}
\texttt{topolevel} &\in [0, \texttt{CellTag::TopoLevel}],\\
\texttt{element_id} &\in [0, \texttt{TopologyLevel<CellTag, topolevel>::ElementNum}),\\
\texttt{bf_id} &\in [0, \texttt{BasisFuncNum<CellTag, BasisFunctionTag, topolevel>::ReturnValue}),
\end{aligned}$$

where `BasisFunctionTag` denotes the type of basis functions (and their degree respectively). The number of basis functions on a particular element is configured as

```
template <typename ElementTag, typename BasisFunctionTag, long levelnum>
struct BasisFuncNum
{
  //Default behaviour: assume no basis function on that element:
  enum{ ReturnValue = 0 };
};

//1 basis function per vertex:
template <typename ElementTag, typename BasisFunctionTag>
struct BasisFuncNum<ElementTag, BasisFunctionTag, 0>
{
  enum{ ReturnValue = 1 };
};

//Quadratic basis function have one DOF on each edge:
template <typename ElementTag>
struct BasisFuncNum<ElementTag, QuadraticBasisfunctionTag, 1>
{
  enum{ ReturnValue = 1 };
};

//and so on for other basis function tags and topological levels
```

and has to be set up manually to some extent at present. With such an identifier it is easy to iterate over all locally defined basis functions: One starts with `BasisFunctionID<0,0,0>` and increments the third template parameter until an "overflow" occurs: As soon as all basis functions on an element are traversed, one increments `element_id` and resets `bf_id`. Similarly, if all elements on a topology level are traversed, `topolevel` is increased and both `element_id` and `bf_id` are reset to zero. The end of the iteration is reached if `topolevel` exceeds the topology level of the cell.

4.3 Mapping Local to Global Basis Functions

With the introduction of `BasisFunctionID`, basis functions are handled for each domain element of a cell. However, there are still orientation issues on an element: An edge can be oriented in two ways, a triangular face in six ways. Clearly, one could handle orientation specific issues separately for each geometric shape, but this would require a lot of tedious work, when a new element type is introduced.

Since, according to Section 4.1, it is sufficient to solve the orientation problems for nodal product basis functions, let us consider an element $\mathbf{e}$ with $n_\mathbf{e}$ vertices $\mathbf{v}_0, \ldots, \mathbf{v}_{n_\mathbf{e}-1}$ in reference orientation of $\mathbf{e}$ on a cell $\mathbf{c}$. These vertices are a subset of the vertices of $\mathbf{c}$. The local ordering of the vertices of $\mathbf{e}$ is generally different from the global ordering within $\mathbf{c}$, say $\mathbf{v}_{\pi^\mathbf{e}_\mathbf{c}(0)}, \ldots, \mathbf{v}_{\pi^\mathbf{e}_\mathbf{c}(n_\mathbf{e}-1)}$ with a permutation $\pi^\mathbf{e}_\mathbf{c}$ that is determined by the local orientation of $\mathbf{e}$ within $\mathbf{c}$. If we now identify bubble functions on element $\mathbf{e}$ with a nodal exponent vector $\boldsymbol{k} = (k_0, \ldots, k_{n_\mathbf{e}-1})$, the underlying basis function $\psi^\mathbf{e}_{\boldsymbol{k},\mathbf{c}}$ is then given as

$$\psi^\mathbf{e}_{\boldsymbol{k},\mathbf{c}} = \prod_{i=0}^{n_\mathbf{e}-1} (\psi^{\mathbf{v}_i}_\mathbf{c})^{k_i}. \tag{4.20}$$

One possible mapping strategy thus is to store the underlying mapping number in a map (whose keys are the exponent vectors $\boldsymbol{k}$) on $\mathbf{e}$. For an assembly on a per cell basis as it is typically done with FEM, a cell $\mathbf{c}$ actually requests $\mathbf{e}$'s basis function numbers with an exponent vector in the cell's local orientation. Hence, if the global basis function number with exponent vector $\boldsymbol{k}$ is requested from $\mathbf{e}$, the cell requests

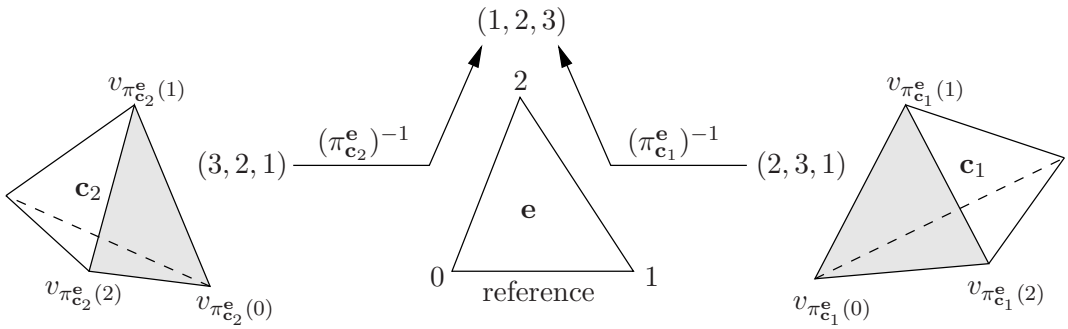


Figure 4.2: Generally, lower level elements have a different orientation within the cell than their global representative. An appropriate permutation of the entries in the exponent key is therefore necessary, shown here for triangular facets.

$\pi^{\mathbf{e}}_{\mathbf{c}}(\boldsymbol{k}) = (k_{\pi^{\mathbf{e}}_{\mathbf{c}}(0)}, k_{\pi^{\mathbf{e}}_{\mathbf{c}}(1)}, \ldots, k_{\pi^{\mathbf{e}}_{\mathbf{c}}(n-1)})$. Therefore, the cell has to apply the inverse permutation $(\pi^{\mathbf{e}}_{\mathbf{c}})^{-1}$ to the exponent vector $\pi^{\mathbf{e}}_{\mathbf{c}}(\boldsymbol{k})$ first, in order to get the correct basis function number from $\mathbf{e}$.

From the cell's point of view, it is sufficient to have $(\pi^{\mathbf{e}}_{\mathbf{c}})^{-1}$ available. There is not even a need for storing this permutation on the cell: If we define the reference ordering of an element to be such that the first vertex is the one with the lowest address and the orientation is determined from the adresses of the neighbouring vertices in ascending order, the vertex pointers stored on the cell are sufficient to derive $(\pi^{\mathbf{e}}_{\mathbf{c}})^{-1}$.

Storing a map (e.g. `std::map`) on an element $\mathbf{e}$ with exponent vector keys returning the map index is not very appealing, because a map would use far too much memory: Assume that we have to store three numbers (with one byte of size) on a triangular facet. Each number is then associated with a key consisting of three exponents each, making a total of nine bytes. Additionally, there is an additional memory consumption due to the internal data structure of the map, so that we have a memory overhead of a factor of more than three in this case.

As first step towards the elimination of such a memory overhead, an ordering relation among all exponent vectors of equal modulus $|\boldsymbol{k}|$ is introduced:

Definition 4. *An exponent vector* $\boldsymbol{k}^1 = (k^1_0, \ldots, k^1_{n-1})$ *is said to be* smaller *than another exponent vector* $\boldsymbol{k}^2 = (k^2_0, \ldots, k^2_{n-1})$ *(denoted as* $\boldsymbol{k}^1 < \boldsymbol{k}^2$*), if*

$$\exists s \in \{0, 1, \ldots, n-1\} : \quad k^1_s < k^2_s \quad \wedge \quad k^1_i = k^2_i \quad \forall i = 0, 1, \ldots, s-1 \ . \tag{4.21}$$

Let us illustrate this ordering for bubble functions of degree 6 on a tetrahedron (identified by all exponent vectors with length 4 and modulus 6). The ordered $\boldsymbol{k}$-vectors, starting with the smallest, are

$$\begin{gathered}(1,1,1,3)\\(1,1,2,2)\\(1,1,3,1)\\(1,2,1,2)\\(1,2,2,1)\\(1,3,1,1)\\(2,1,1,2)\\(2,1,2,1)\\(2,2,1,1)\\(3,1,1,1)\end{gathered}$$

A traversal through all exponent vectors in $\mathcal{K}_n^d$ with $d > n$ from the smallest to the largest can be done as follows:

Algorithm 1. *1. Start with the smallest vector $\boldsymbol{k}_{\min} \in \mathcal{K}_n^d$ given by $k_i = 1$ for $i = 0, \dots n-2$ and $k_{n-1} = d - n + 1$. Output this vector.*

2. Set $j = 0$.

3. Increment k_{n-2-j} and decrement k_{n-1}. If $k_{n-1} < 1$, set $k_{n-2-j} = 1$, $k_{n-1} = d - \sum_{i=0}^{n-2} k_i$, increment j and repeat this step.

4. Output the current vector.

5. If $k_0 = d - n + 1$, iteration is finished. Otherwise, continue with step 2 to obtain the next vector.

Given such an ordering of the exponent vectors, it is sufficient to store an array containing the global basis function indices only. However, it is rather tricky to get the correct array entry for a given key without iterating through all possible keys.

4.4 Fast Access to Global Basis Function Numbers

In this section we derive a general way to determine the position $p_n^d(\boldsymbol{k})$ of a given vector $\boldsymbol{k} \in \mathcal{K}_n^d$ subject to the ordering relation introduced with Def. 4. Once the number of vectors smaller than $\boldsymbol{k}$ is known, the global basis function index is obtained from the basis function index array that is stored on the element.

With the convention that $p(\boldsymbol{k}_{\min}) = 0$ for the the smallest vector $\boldsymbol{k}_{\min}$ of the set $\mathcal{K}_n^d$, we can give a more mathematical formulation of the problem: For a given exponent vector $\boldsymbol{k} \in \mathcal{K}_n^d$, we define

$$p_n^d(\boldsymbol{k}) = \left| \{ \tilde{\boldsymbol{k}} \in \mathcal{K}_n^d : \tilde{\boldsymbol{k}} < \boldsymbol{k} \} \right| \ . \tag{4.22}$$

From an algorithmic point of view it is too costly to iterate over all vectors of $\mathcal{K}_n^d$ and count all traversed vectors until finally $\boldsymbol{k}$ is reached, because each vector-comparison includes typically several integer-comparisons. Especially for a large number of large vectors, look-up times become unacceptable.

A more efficient way for the evaluation of $p_n^d(\boldsymbol{k})$ is therefore desired.

Theorem 1. *For an exponent vector $\boldsymbol{k} \in \mathcal{K}_n^d$ there holds*

$$p_n^d(\boldsymbol{k}) = \sum_{j=0}^{n-2} \left(\sum_{i=1}^{k_j - 1} \left| \mathcal{K}_{n-1-j}^{d - \sum_{l=0}^{j-1} k_l - i} \right| \right) \ . \tag{4.23}$$

Proof. Let us rearrange:

$$\begin{aligned} p_n^d(\boldsymbol{k}) &= \left| \{ \tilde{\boldsymbol{k}} \in \mathcal{K}_n^d : \tilde{\boldsymbol{k}} < \boldsymbol{k} \} \right| \\ &= \left| \{ \tilde{\boldsymbol{k}} \in \mathcal{K}_n^d : \tilde{k}_0 < k_0 \} \right| + \left| \{ \tilde{\boldsymbol{k}} \in \mathcal{K}_n^d : (\tilde{k}_0 = k_0) \wedge (\tilde{\boldsymbol{k}} < \boldsymbol{k}) \} \right| \\ &= \sum_{i=1}^{k_0 - 1} \left| \mathcal{K}_{n-1}^{d-i} \right| + \left| \{ \tilde{\boldsymbol{k}} \in \mathcal{K}_n^d : (\tilde{k}_0 = k_0) \wedge (\tilde{\boldsymbol{k}} < \boldsymbol{k}) \} \right| \ . \end{aligned}$$

Now, if we denote $\boldsymbol{k}[i{:}j]$ as the vector $(k_i, k_{i+1}, \ldots, k_{j-1}, k_j) \in \mathcal{K}_{j-i+1}^{d-k_0-\ldots-k_{i-1}-k_{j+1}-\ldots-k_{n-1}}$ with entries from $\boldsymbol{k}$, we have

$$p_n^d(\boldsymbol{k}) = \sum_{i=1}^{k_0-1} \left|\mathcal{K}_{n-1}^{d-i}\right| + p_{n-1}^{d-k_0}(\boldsymbol{k}[1:(n-1)]) \ .$$

This is now a recursion for $p_n^d(\cdot)$ in n and d. Substitution for $p_{n-1}^{d-k_0}(\cdot)$ leads to

$$p_n^d(\boldsymbol{k}) = \sum_{i_0=1}^{k_0-1} \left|\mathcal{K}_{n-1}^{d-i_0}\right| + \sum_{i_1=1}^{k_1-1} \left|\mathcal{K}_{n-2}^{d-k_0-i_1}\right| + p_{n-2}^{d-k_0-k_1}(\boldsymbol{k}[2:(n-1)])$$

and a full unwrapping of this recursion with termination condition $p_1^d(\cdot) = 0$ for all integers d results in

$$p_n^d(\boldsymbol{k}) = \sum_{j=0}^{n-2} \left(\sum_{i=1}^{k_j-1} \left|\mathcal{K}_{n-1-j}^{d-\sum_{l=0}^{j-1} k_l - i}\right| \right) \ .$$

□

The evaluation of $p_n^d(\boldsymbol{k})$ is then reduced to a computation of $|\mathcal{K}_{\hat{n}}^{\hat{d}}|$ for $0 < \hat{n} < n$ and $0 < \hat{d} < d$. These numbers can be computed easily:

Theorem 2. *For the number of elements $|\mathcal{K}_n^d|$ of $\mathcal{K}_n^d$ with integers $d > n > 1$ there holds*

$$|\mathcal{K}_n^d| = |\mathcal{K}_n^{d-1}| + |\mathcal{K}_{n-1}^{d-1}| \ . \tag{4.24}$$

Furthermore, $|\mathcal{K}_n^n| = 1$ for all integers $n > 0$ and $|\mathcal{K}_1^d| = 1$ for all integers $d > 0$.

Proof. $|\mathcal{K}_1^d| = 1$ is obvious.

Next, since for an exponent vector $\boldsymbol{k} \in \mathcal{K}_n^n$ all entries k_i are positive integers, the only way to satisfy $\sum_{i=0}^{n-1} k_i = n$ is to have all n summands equal to one, thus $|\mathcal{K}_n^n| = 1$.

To prove the recursion for $d > n > 0$, let us decompose a vector $\boldsymbol{k} \in \mathcal{K}_n^d$ as

$$\boldsymbol{k} = [\hat{\boldsymbol{k}}, k_{n-1}],$$

where $\hat{\boldsymbol{k}} \in \mathcal{K}_{n-1}^{\hat{d}}$ with $\hat{d} = d - k_{n-1}$ (so $n \le \hat{d} < d$) and the brackets used as vector concatenation. Now, iteration over all possible integer values for k_i allows a decomposition of the form

$$|\mathcal{K}_n^d| = \sum_{i=1}^{d-n+1} |\mathcal{K}_{n-1}^{d-i}|, \tag{4.25}$$

since by construction $\mathcal{K}_{n-1}^{d_1} \cap \mathcal{K}_{n-1}^{d_2} = \emptyset$ for $d_1 \neq d_2$. Extraction of the first summand leads to

$$|\mathcal{K}_n^d| = |\mathcal{K}_{n-1}^{d-1}| + \sum_{i=2}^{d-n+1} |\mathcal{K}_{n-1}^{d-i}|.$$

A shift of indices for the sum on the right-hand side and the use of (4.25) finally yields

$$|\mathcal{K}_n^d| = |\mathcal{K}_n^{d-1}| + |\mathcal{K}_{n-1}^{d-1}|.$$

□

What is now very appealing from the point of run time efficency is that $|\mathcal{K}_{\hat{n}}^{\hat{d}}|$ for $0 < \hat{n} < n$ and $0 < \hat{d} < d$ only need to be computed once in an array of size $(n-1) \times (d-1)$. In almost all cases, this array will not have more than 50 entries[1]. For this reason, one can also store the sums

$$\sum_{i=1}^{c} \left| \mathcal{K}_a^b \right|$$

in an three dimensional array with parameters (a, b, c) that show up in (4.23) with $a = n - 1 - j, b = d - \sum_{l=0}^{j-1} k_l$ and $c = k_j - 1$. The memory consumption for this array is typically negligible, so $p_n^d(\boldsymbol{k})$ can be computed with $n - 1$ additions and memory look-ups. Alternatively, it is also possible to run through the full recursion for $p_n^d(\boldsymbol{k})$ given by (4.23) and (4.24), which is attractive for small values of n and d.

4.5 From Basisfunction ID to the Full Basis Function

With the introduction of `BasisFunctionID` we are able to iterate over all basis functions on a cell. Using exponent vectors, we can enumerate bubble functions defined on a particular element. The topic of this section is the construction of the basis functions as defined on a cell, that is, to transform a `BasisFunctionID` to a compile time representation of the basis function. This is achieved in three steps:

1. `BasisFunctionID` is transformed into an exponent vector defined locally on the element.
2. The local exponent vector is transformed to an exponent vector on the cell.
3. This cell exponent vector then forms the basis function as an expression at compile time.

Given `BasisFunctionID<t,e,i>`, where `t` is the topology level, `e` the element identifier and `i` the basis function identifier, we have to find the `i`-th basis function defined on the `e`-th element on topology level `t`. This can already be achieved with Algorithm 1. The algorithm was implemented to evaluate at compile time, so there is no run time penalty for a linear search through all identifiers. For the implementation, the following types have been introduced:

```
template <long a0>
struct compressed_bf_1
{
  enum { ARG_NUM = 1, Degree = a0 };
};

template <long a0, long a1>
struct compressed_bf_2
{
  enum { ARG_NUM = 2, Degree = a0 + a1 };
};

//a lot more types compressed_bf_X, X=3,4,5,...
```

Since the current C++ standard does not support a variable number of template arguments, we have to go the tedious way of keeping separate classes for different number of arguments. The name `compressed_bf` arises from the fact that it holds the bubble functions of the particular element, thus it is an exponent vector.

[1] For quadrilateral facets, $n = 4$ and basis functions of degree 10, we have for example $3 \cdot 9 = 27$ entries only

We will not go into all implementation details of the current implementation since a new C++ standard that allows so-called *variadic templates* will enable much simpler implementations with the benefit of faster compilation times. For the moment, the meta function

```
template <long vertex_num, long bf_id, long bf_degree>
struct GET_COMPRESSED_BF_FOR_ELEMENT
{
  //implementation here
};
```

can be used as a black-box function. `vertex_num` is the number of vertices of the element (not the cell!), `bf_id` is the third template parameter of `BasisFunctionID` and `bf_degree` is the desired basis function degree. To establish a link with Section 4.2, the parameters `vertex_num` and `bf_degree` can be associated with n and d of $\mathcal{K}_n^d$. Thus, `GET_COMPRESSED_BF_FOR_ELEMENT` returns the vector $\boldsymbol{k}$ (at compile time using a type definition) such that $p_n^d(\boldsymbol{k}) =$`bf_id`.

Now, this local representation of the basis function has to be transformed to a cell exponent vector. However, since the element's vertices are known from the element's identifier, an embedding does the job.

```
template <typename ElementTag,
          typename CellTag,
          typename CompressedBf,
          long element_id>
struct EMBED_ELEMENT_BF_TO_CELL_BF;
```

All the embeddings are now specialisations. For example, the following specialisation embeds an exponent vector of an edge with identifier 2 into a tetrahedral cell:

```
template <long a0, long a1>
struct EMBED_ELEMENT_BF_TO_CELL_BF<LineTag,
                                   TetrahedronTag,
                                   compressed_bf_2<a0, a1>,
                                   2>
{
  typedef compressed_bf_4<a0, 0, 0, a1>   ResultType;
};
```

Note that it is used that the edge with identifier 2 (thus, the third edge of the tetrahedron) is the one connecting the first and the fourth vertex of the tetrahedron. For this example, the reference orientation is determined in `TopologyLevel<TetrahedronTag, 1>::fill()`, which is the cell topology description and turned up in Section 2.1.4 already. For the moment, the embeddings are implemented by hand, but since the number of spatial dimensions of a mesh hardly exceeds five, such effort can be justified.

Now, the final step is to build the compile time expression for the basis function. This is done by another meta function:

```
template< typename CompressedBf, typename AssemblyCellTag >
struct COMPRESSED_BF_TO_FULL_BF
{
  //implementation
};
```

Effectively, this meta function translates the exponent vector into a template class

```
template <long num, long denum,
          typename T1, long pow1,
          typename T2, long pow2,
```

```
          typename T3 = CompoundUnused, long pow3 = 0,
          typename T4 = CompoundUnused, long pow4 = 0
        >
struct CompoundExpression;
```

representing

$$\frac{\texttt{num}}{\texttt{denum}}(\texttt{T1})^{\texttt{pow1}}(\texttt{T2})^{\texttt{pow2}}(\texttt{T3})^{\texttt{pow3}}(\texttt{T4})^{\texttt{pow4}} \ . \tag{4.26}$$

and thus directly (4.19) within code and is explained in more detail in Section 5.5. Now, the template parameters only have to filled appropriately:

```
template< typename CompressedBf, typename AssemblyCellTag >
struct COMPRESSED_BF_TO_FULL_BF
{
  //get Compound-Expression:
  typedef typename BFStock<AssemblyCellTag>::CompoundType     CompoundType;
  //set exponents:
  typedef typename COMPRESSED_BF_TO_FULL_BF_IMPL
                   <CompressedBf, CompoundType>::ResultType   ResultType;
};
```

Here, `BFStock` provides the vertex functions of the reference cell:

```
template <typename CellTag>
struct BFStock {};

//sample specialisation for a triangle:
template <>
struct BFStock<TriangleTag>
{
  typedef Expression < ExpressionDefaultScalarType,
                       ScalarExpression<1>,
                       Expression< ExpressionDefaultScalarType,
                                   var<0>,
                                   var<1>,
                                   op_plus<ExpressionDefaultScalarType>
                                  >,
                       op_minus<ExpressionDefaultScalarType> >
                          OneMinusXY;

  typedef CompoundExpression<1, 1, OneMinusXY, 0,
                             var<0>, 0, var<1>, 0>     CompoundType;
};
```

Then, the exponent vector can be transformed to a full compile time expression:

```
template< typename CompressedBf, typename CompoundType >
struct COMPRESSED_BF_TO_FULL_BF_IMPL {};

//sample specialisation for a cell with two vertices:
template <long a0, long a1,
          long num, long denum,
          typename Expr1, long alpha1,
          typename Expr2, long alpha2,
          typename Expr3, long alpha3,
```

```
          typename Expr4, long alpha4
         >
struct COMPRESSED_BF_TO_FULL_BF_IMPL
         <compressed_bf_2<a0, a1>,
          CompoundExpression<num, denum, Expr1, alpha1, Expr2, alpha2,
                             Expr3, alpha3, Expr4, alpha4>
         >
{
  typedef CompoundExpression<num, denum,
                             Expr1, a0, Expr2, a1,
                             Expr3, alpha3, Expr4, alpha4>
                                ResultType;
};
```

As soon as all specialisations are implemented, all basis functions defined on a cell can be obtained. Together with the mapping indices that can be retrieved from a suitable permutation of the bubble function's exponent vectors from the vector stored on the cell, evaluation of the integrals arising from the weak formulation can be performed.

The general strategy is to evaluate the integrals over the cell after a transformation to a reference cell numerically. However, for PDEs with constant coefficients, integrals over the reference cell can be computed analytically and then appropriately scaled, which is the topic of Chapter 5.

4.6 The Mapping Iterator

The mapping strategy described in the previous sections has complicated internals, but boils down to providing the global basis function indices in the order at which the `BasisFunctionID` is traversed.

This actually means that we *iterate* over all global basis function indices in a sequence determined by the orientation of the cell. Therefore, on a programming level it is sufficient to provide an `Iterator` that hides the internals:

```
template <typename FEMConfig, typename CellType>
class MappingIterator
{
  public:
    MappingIterator(CellType & cell);

    //some more implementation dependent members omitted
};
```

The first template argument `FEMConfig` is the configuration class for the desired finite element method, the second template argument `CellType` is the domain element type, from which the number of elements and levels of the cell can be obtained. With this in hand, the code for printing all mapping indices in the order of the locally defined basis functions becomes

```
CellType cell;
//some code for writing the
//global basis function indices to that cell

//iterate over global basis function indices on 'cell'
for (MappingIterator<FEMConfig, CellType> mappingiterator(cell);
     mappingiterator.valid();
     ++mappingiterator)
{
  //print current mapping index:
```

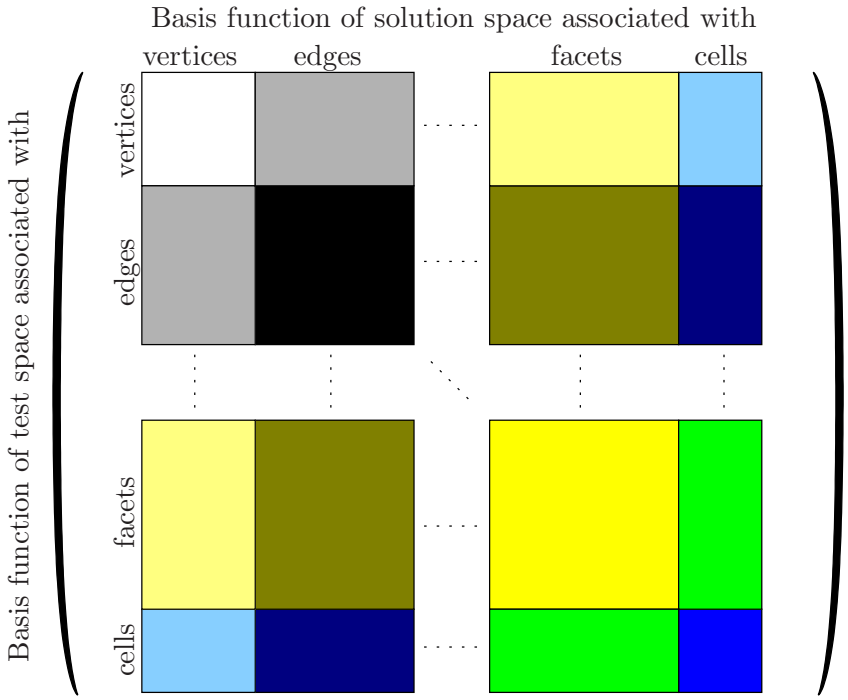


Figure 4.3: Structure of the global system matrix. Vertex functions are encoded in white, edge functions in black, facets in yellow and cells in blue. The colors in the off-diagonal blocks arise from a mixture of the corresponding colors.

```
11    std::cout << *mappingiterator << std::endl;
12  }
```

The internals of `MappingIterator` are given in descriptive pseudo-code, the crucial steps have already been discussed in the previous sections.

```
1  for all topology levels 'l' of 'cell' do
2    for all elements 'e' of 'cell' at level 'l' do
3      for all basis functions on 'e' do
4        - set up the exponent vector 'k' of the 'i'-th
5          basis function at 'e' with respect to the orientation of 'c'
6        - transform 'k' to 'k2' according to the reference orientation of 'e'
7        - return the p_n^d(k2)-th number stored
8          in the mapping array of element 'e'
9      end
10   end
11 end
```

Here, `p_n^d` refers to $p_n^d(\cdot)$ as defined in (4.22). Since the number of topology levels is typically small, the outermost loop is unrolled at compile time, which allows for various specialisations of the inner loop done in a class with name `MappingIterator_impl`:

- If only one basis function is defined per element on a particular topology level, the two innermost loops simplify to accessing a *single basis function index* per element only, thus eliminating the need for the innermost loop and all handling of exponent vectors. This is especially important for piecewise linear basis functions, since each vertex is then associated with one global basis function. Unnecessary book-keeping would decrease run time performance considerably.

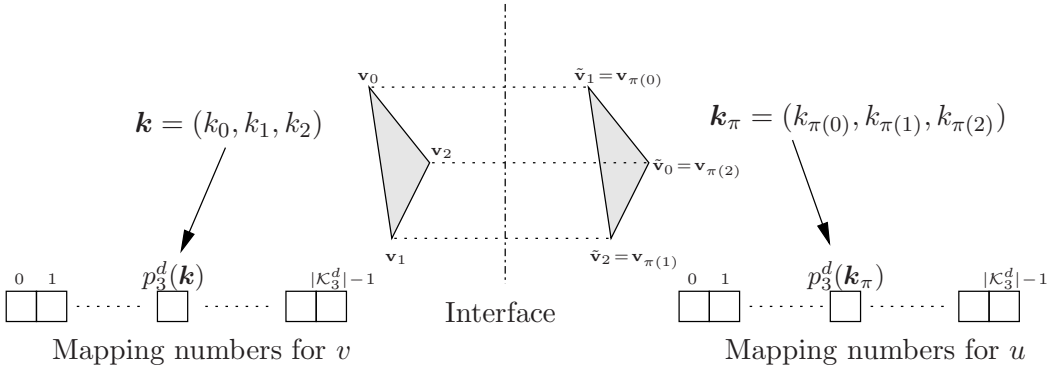


Figure 4.4: The associated mapping index on the interface twin is obtained by permutation of the exponent vector, shown for triangular facets.

- For bubble functions of the cell, there is no need to do any rearrangement: The two inner loops are replaced by a (linear) iteration over the mapping indices stored on the cell.
- If there are no basis functions associated with a topology level at all, there is no need to start with the second loop.

After a translation of these optimisations into partial specialisations, the compiler will then "automatically" choose the best implementation for a given set of basis functions.

4.7 A Mapping Strategy for Coupled Segments

For the simulation of physical effects, several segments are coupled by flux terms that mathematically show up as additional boundary integrals in the weak formulation. A physically motivated example is given in Chapter 7, for the moment we consider the assembly of the boundary integral

$$\int_\Gamma u_1 v_2 \mathrm{d}x, \quad \Gamma = \partial\overline{\Omega}_1 \cap \partial\overline{\Omega}_2 \tag{4.27}$$

arising from the coupling of two segments Ω_1 and Ω_2 for some u_1 sufficiently regular on Ω_1 and v sufficiently regular on Ω_2 (typically: $u_1 \in H^1(\Omega_1), v_2 \in H^1(\Omega_2)$) and continuous continuation to the common boundary Γ. Following the standard FEM formulation, $u_1|_\Gamma$ and $v_2|_\Gamma$ can be seen as the traces of basis functions defined on Ω_1 and Ω_2.

Recalling the construction of basis functions on a per-element basis from Section 4.1, we can view the boundary facet as the new cell type within the boundary integral. Thus, the local basis function ID on an element of the boundary facet can also be mapped to the boundary facet instead of the enclosing cell. This requires that the traces of basis functions defined on the cell show up in the lower dimensional basis of the boundary elements. For nodal product basis functions and Lobatto basis functions, this is the case.

For coupled segments, we have one single integration domain from the mathematical point of view, but there are two instances of this element in the memory: One element is member of the segment representing Ω_1, the other element is member of the segment representing Ω_2. These two instances are coupled by storing a pointer on each element that points to the element in the other segment at the same location. This is performed during the interface specification using `setInterface` (see Section 3.5). What is left for a mapping of coupled segments is to determine the permutations of the local vertex enumeration.

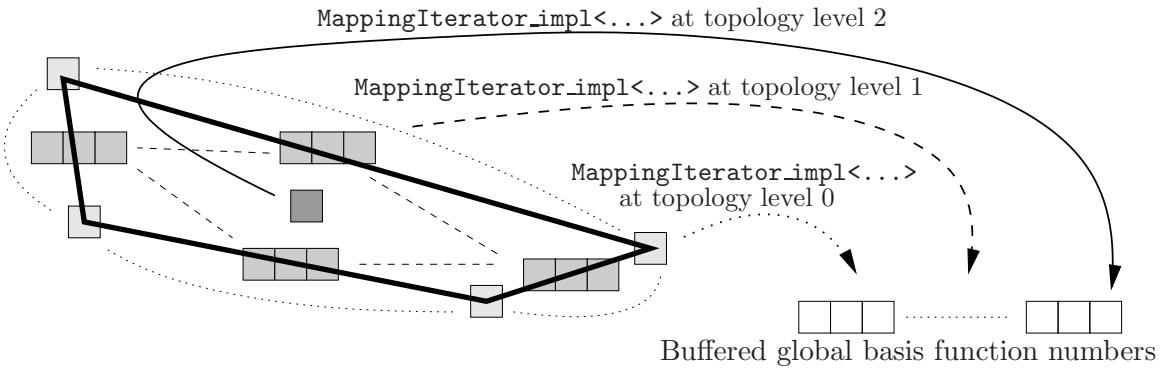


Figure 4.5: The final `MappingIterator` buffers all mapping indices obtained from `MappingIterator_impl` for each topology level. Here this is shown for a quadrilateral with one basis function per vertex, three basis functions per edge and one bubble function. The squares denote basis function numbers stored on the elements, the thin lines denote a collection of these arrays that is handled by `MappingIterator_impl`.

Let us consider the assembly of the boundary facet $\mathbf{f}$. The facet $\tilde{\mathbf{f}}$ from the other segment with the same geometric location is in the following called *interface twin*. Let $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ be the vertices of $\mathbf{f}$ and $\tilde{\mathbf{v}}_0, \tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_{n-1}$ those of $\mathbf{f}$. By a brute force any-to-any comparison[2], we can set up a permutation π, such that $\mathbf{v}_{\pi(j)} = \tilde{\mathbf{v}}_j$ for $j = 0, \ldots, n-1$, where equality is to be understood in a geometrical sense.

This permutation π is sufficient to find the mapping indices on the coupled segment: For a local exponent vector $\boldsymbol{k} = (k_0, k_1, \ldots, k_{n-1}) \in \mathcal{K}_n^d$ given on $\mathbf{f}$ (in reference orientation, therefore a permutation from the cell's orientation to $\mathbf{f}$ has already been applied), the exponent vector of the corresponding basis function on the interface twin $\tilde{\mathbf{f}}$ is obtained as $\boldsymbol{k}_\pi = (k_{\pi(0)}, k_{\pi(1)}, \ldots, k_{\pi(n-1)})$. The basis function can then be obtained from the vector holding the mapping numbers (stored on $\tilde{\mathbf{f}}$) at index $p_n^d(\boldsymbol{k}_\pi)$.

To account for the additional requirements for coupled segments, the final `MappingIterator` type is extended for additional template arguments:

```
template <typename FEMConfig, typename CellType,
          typename IntegrationDomain, bool map_u = false,
          typename BFTag = typename FEMConfig::BasisfunctionTag>
class MappingIterator
{
  //implementation here
};
```

The first two arguments are the same as in the previous chapter. Additionally, `IntegrationDomain` expects one type out of `Omega`, `Gamma<>` and `Interface<>`. Together with `map_u`, a flag that indicates whether the mapping has to be done for test functions v or the unknown u, a partial specialisation for `IntegrationDomain` equals `Interface<id>` for some `id` and `map_u` equals `true` enables interface mapping as described before. The fifth parameter `BFTag` is used for a performance gain: While the default implementation buffers all mapping indices obtained from each topology level in a separate array, such a buffer is not needed in case of linear basis functions, because the mapping numbers are directly accessible from the vertices.

[2] for most problems, the number of vertices on each facet is smaller than five, thus an any-to-any comparison is fast enough

Chapter 5

Analytic Integration at Compile Time

In this chapter a method for the analytic computation of integrals over basis functions defined on a reference cell at compile time is presented. Unlike for the previous chapters, where the concepts can be used for several classes of cell geometries, an analytic integration at compile time is for the moment restricted to simplex geometries, especially triangles in two dimensions and tetrahedra in three dimensions. This restriction arises from the requirement that the transformation to (and from) the reference cell has to be affine, which is the case for simplex geometries, but generally not for other cell geometries typically used for unstructured grids.

5.1 Motivation

Assembly of the system matrix arising from FEM is typically done per cell. The integral contributions on that cell are computed on a reference cell and then transformed appropriately to take the cell's geometry into account. Although the transformation of integrals to the reference cell is documented in most books about FEM, we repeat the calculation for a triangle here to motivate the necessary steps needed for an integration at compile time.

Let us consider the integral $\int_T \nabla\varphi_u \nabla\varphi_v \,\mathrm{d}x_0\mathrm{d}x_1$ over a triangle $\mathbf{T}$. Such an integral shows up in the assembly of the stiffness matrix and is computed over a reference triangle $\mathbf{T}^{\text{ref}}$. With basis functions $\varphi_u = \varphi_u(x_0, x_1)$, $\varphi_v = \varphi_v(x_0, x_1)$ defined on $\mathbf{T}$ and the transformed pair $\psi_u(\xi_0, \xi_1) := \varphi_u(\boldsymbol{F}(\xi_0, \xi_1))$, $\psi_v(\xi_0, \xi_1) := \varphi_v(\boldsymbol{F}(\xi_0, \xi_1))$, where $\boldsymbol{F}(\xi_0, \xi_1) = (x_0, x_1)$ is the affine transformation from $\mathbf{T}^{\text{ref}}$ (with parametrisation (ξ_0, ξ_1)) to $\mathbf{T}$, the transformation reads

$$\begin{aligned}\int_{\mathbf{T}} \nabla\varphi_u \nabla\varphi_v \,\mathrm{d}x_0\mathrm{d}x_1 &= \int_{\mathbf{T}} \frac{\mathrm{d}\varphi_u}{\mathrm{d}x_0}\frac{\mathrm{d}\varphi_v}{\mathrm{d}x_0} + \frac{\mathrm{d}\varphi_u}{\mathrm{d}x_1}\frac{\mathrm{d}\varphi_v}{\mathrm{d}x_1} \\ &= \int_{\mathbf{T}^{\text{ref}}} \left[\left(\frac{\partial\psi_u}{\partial\xi_0}\frac{\partial\xi_0}{\partial x_0} + \frac{\partial\psi_u}{\partial\xi_1}\frac{\partial\xi_1}{\partial x_0}\right)\left(\frac{\partial\psi_v}{\partial\xi_0}\frac{\partial\xi_0}{\partial x_0} + \frac{\partial\psi_v}{\partial\xi_1}\frac{\partial\xi_1}{\partial x_0}\right) + \right. \\ &\qquad \left.\left(\frac{\partial\psi_u}{\partial\xi_0}\frac{\partial\xi_0}{\partial x_1} + \frac{\partial\psi_u}{\partial\xi_1}\frac{\partial\xi_1}{\partial x_1}\right)\left(\frac{\partial\psi_v}{\partial\xi_0}\frac{\partial\xi_0}{\partial x_1} + \frac{\partial\psi_v}{\partial\xi_1}\frac{\partial\xi_1}{\partial x_1}\right)\right] \left|\frac{\partial \boldsymbol{F}(\xi_0, \xi_1)}{\partial(\xi_0, \xi_1)}\right| \mathrm{d}\xi_0\mathrm{d}\xi_1 \,.\end{aligned} \tag{5.1}$$

At this point we use that the mapping $\boldsymbol{F}$ is affine: The partial derivatives $\frac{\partial\xi_0}{\partial x_0}$, $\frac{\partial\xi_0}{\partial x_1}$, $\frac{\partial\xi_1}{\partial x_0}$, $\frac{\partial\xi_1}{\partial x_1}$ and the Jacobian $\frac{\partial F(\xi_0,\xi_1)}{\partial(\xi_0,\xi_1)}$ are constant and can be pulled in front of the integral. The remaining integrals are

$$\begin{aligned}&\int_{\mathbf{T}^{\text{ref}}} \frac{\partial\psi_u}{\partial\xi_0}\frac{\partial\psi_v}{\partial\xi_0}\mathrm{d}\xi_0\mathrm{d}\xi_1 \,, \quad \int_{\mathbf{T}^{\text{ref}}} \frac{\partial\psi_u}{\partial\xi_0}\frac{\partial\psi_v}{\partial\xi_1}\mathrm{d}\xi_0\mathrm{d}\xi_1 \,, \\ &\int_{\mathbf{T}^{\text{ref}}} \frac{\partial\psi_u}{\partial\xi_1}\frac{\partial\psi_v}{\partial\xi_0}\mathrm{d}\xi_0\mathrm{d}\xi_1 \,, \quad \int_{\mathbf{T}^{\text{ref}}} \frac{\partial\psi_u}{\partial\xi_1}\frac{\partial\psi_v}{\partial\xi_1}\mathrm{d}\xi_0\mathrm{d}\xi_1 \,.\end{aligned} \tag{5.2}$$

Since the basis functions ψ_u and ψ_v are explicitely constructed on the reference cell, the integrals are known at compile time. Typically, each of these integrals is evaluated for each pair (ψ_u, ψ_v) of basis functions defined on the reference cell, leading to four so-called *element matrices*. These element matrices can be found in most books about FEM and are in most cases hard-coded in the final code. The disadvantages of hard-coding these matrices have already been discussed in a previous work of the author [26].

Nevertheless, evaluation of these integrals at run time leads to an overhead that cannot be accepted for high performance computing. The next sections describe a compile time expression engine to fully compute the integrals in (5.2) during compilation, leading to improved run time performance.

5.2 Compile Time Transformation to the Reference Element

The first step for any analytic computation is to transform the integrands from the weak formulation such as in (5.1) to the reference element, since we are only able to do an integration at compile time there. Such a transformation can be seen as replacing all derivatives of basis functions by a term determined by the chain rule. In general, for global coordinates x_i, $i = 0, \ldots, n-1$ and local coordinates ξ_i, $i = 0, \ldots, n-1$, the transformation of derivatives for a function φ is given as

$$\frac{\partial \varphi}{\partial x_j} = \sum_{i=0}^{n-1} \frac{\partial \psi}{\partial \xi_i} \frac{\partial \xi_i}{\partial x_j} \,, \tag{5.3}$$

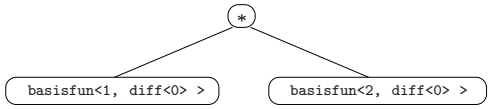
where ψ is the transformed function. Additionally, the Jacobian's determinant of the transformation has to be appended to the integrand, but since we have assumed an affine transformation, this determinant is a scalar, thus it can be pulled in front of the integral and does not contribute to the integrand.

Since integrals can only be evaluated at compile time if the integrand does not consist of any space dependent functions apart from basis functions, the expression tree for the integrand prior to transformation consists of at most two template types: `ScalarExpression` and `basisfun`. The former remains unchanged during a transformation, while the latter is replaced by the chain rule by means of the following meta function:

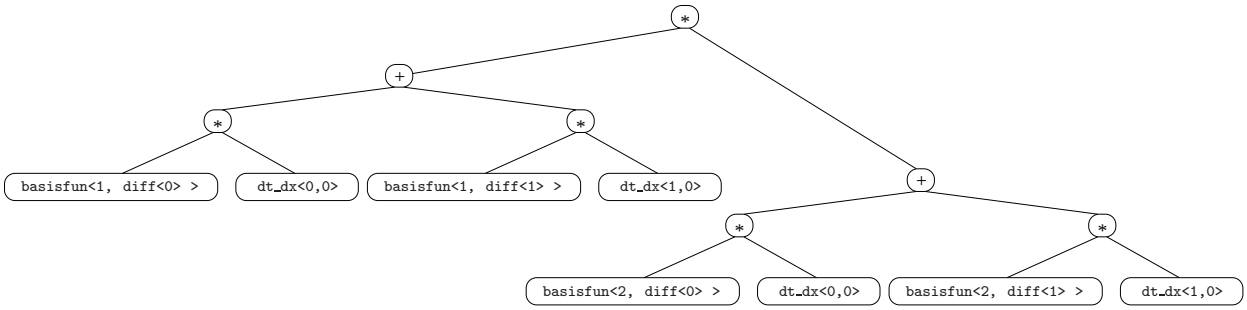
```
template <typename EXPR, long dim>
struct EXPRESSION_TRANSFORM_TO_REFERENCE_ELEMENT;
```

The replacement of `basisfun` with the chain rule given in (5.3) is implemented in a recursive manner as

```
template <long i, long j, long dim>
struct EXPRESSION_TRANSFORM_TO_REFERENCE_ELEMENT<basisfun<i, diff<j> >, dim
    >
{
  typedef Expression< ExpressionDefaultScalarType,
                typename EXPRESSION_TRANSFORM_TO_REFERENCE_ELEMENT
                          < basisfun<i, diff<j> >, dim - 1 >::ResultType,
                Expression< ExpressionDefaultScalarType,
                            basisfun<i, diff<dim - 1> >,
                            dt_dx<dim-1,j>,
                            op_mult<ExpressionDefaultScalarType>
                          >,
                op_plus<ExpressionDefaultScalarType>
                      >                                  ResultType;
};

//end recursion:
```



(a) Initial expression tree.



(b) Expression tree after transformation.

Figure 5.1: Transformation of $\partial\varphi_u/\partial x_0 \cdot \partial\varphi_v/\partial x_0$ to a two-dimensional reference cell as expression tree.

```
17 template <long i, long j>
18 struct EXPRESSION_TRANSFORM_TO_REFERENCE_ELEMENT<basisfun<i, diff<j> >, 1>
19 {
20   typedef Expression< ExpressionDefaultScalarType,
21                       basisfun<i, diff<0> >,
22                       dt_dx<0,j>,
23                       op_mult<ExpressionDefaultScalarType>
24                     >                                ResultType;
25 };
```

Here we used another placeholder class for partial derivatives:

```
1 template <long localindex, long globalindex>
2 struct dt_dx
3 {
4   template <typename CellType>
5   double operator()(CellType & cell) const
6   {
7     return cell.get_dt_dx(localindex, globalindex);
8   }
9 };
```

The placeholder returns the entry at the `localindex`-th row and the `globalindex`-th column of the Jacobian of the transformation of `cell` to its reference cell. Each domain element provides a member function `get_dt_dx` due to its transformation layer described in Section 2.2.

5.3 A Brute Force Approach to Analytic Integration

Given a basis function polynomial as compile time expression tree, let us have a look at the computation of

$$\int_0^1 \int_0^{1-\xi_0} \nabla(\xi_0^2)\nabla(\xi_0\xi_1)\, \mathrm{d}\xi_1 \mathrm{d}\xi_0, \tag{5.4}$$

where we use (ξ_0, ξ_1) as coordinates for the reference triangle with corners at $(0,0)$, $(1,0)$ and $(0,1)$.

Although (5.4) can be computed in a straight-forward manner, let us still do the evaluation step by step to see which manipulations have to be done by the compiler:

$$\begin{aligned}
\int_0^1 \int_0^{1-x} \nabla(\xi_0^2)\nabla(x\xi_1)\,\mathrm{d}\xi_1\mathrm{d}\xi_0 &= \int_0^1 \int_0^{1-\xi_0} 2\xi_0\xi_1\,\mathrm{d}\xi_1\mathrm{d}\xi_0 \\
&= \int_0^1 2\xi_0 \left.\frac{\xi_1^2}{2}\right|_0^{1-\xi_0} \mathrm{d}\xi_0 \\
&= \int_0^1 2\xi_0 \frac{(1-\xi_0)^2}{2}\,\mathrm{d}\xi_0 \\
&= \int_0^1 \xi_0(1-\xi_0)^2\,\mathrm{d}\xi_0 \\
&= \int_0^1 \xi_0(1-2\xi_0+\xi_0^2)\,\mathrm{d}\xi_0 \\
&= \int_0^1 \xi_0\,\mathrm{d}\xi_0 - \int_0^1 2\xi_0^2\,\mathrm{d}\xi_0 + \int_0^1 \xi_0^3\,\mathrm{d}\xi_0 \\
&= \left.\left(\frac{\xi_0^2}{2} - \frac{2\xi_0^3}{3} + \frac{\xi_0^4}{4}\right)\right|_0^1 \\
&= \left(\frac{1}{2} - \frac{2}{3} + \frac{1}{4}\right) = \frac{1}{12}
\end{aligned}$$

Let us summarise the steps necessary to evaluate the integral:

1. Differentiation of polynomials.
2. Finding the anti-derivative of a power of the integration variable.
3. Substitution of the integral bounds.
4. Full expansion of an expression.
5. Term-wise integration.
6. Simplification of terms.

Each of these tasks is accomplished by a separate meta function. Some are rather simple to implement, while others require rather complicated compile time loops and manipulations.

The differentiation of polynomials was already discussed in Chapter 1 and is therefore already available.

Finding the anti-derivative of a power of the integration variable has to be done in two steps: First, the power of the integration variable in a multiplicative term has to be determined. This is the task of the meta function `EXPRESSION_POWER_OF_VARIABLE`:

```
1 template <typename EXPR, typename VARIABLE>
2 struct EXPRESSION_POWER_OF_VARIABLE
3 {
4   enum { ReturnValue = 0 }; //default behaviour
5 };
```

Partial specialisations lead to the desired behaviour:

```
template <typename VARIABLE>
struct EXPRESSION_POWER_OF_VARIABLE <VARIABLE, VARIABLE>
{
  enum { ReturnValue = 1 };
};

template <typename T, typename LHS, typename RHS, typename VARIABLE>
struct EXPRESSION_POWER_OF_VARIABLE < Expression<T, LHS, RHS, op_mult<T> >,
                                      VARIABLE>
{
  enum { ReturnValue =
              EXPRESSION_POWER_OF_VARIABLE<LHS, VARIABLE>::ReturnValue
            + EXPRESSION_POWER_OF_VARIABLE<RHS, VARIABLE>::ReturnValue };
};
```

Now, `EXPRESSION_POWER_OF_VARIABLE` traverses the full expression tree and counts all occurrences of the integration variable. Additional specialisations take divisions into account and force (intentional) compilation errors if plus or minus signs occur in the expression.

The expansion of the integrand is the most complicated step. Anyway, breaking this task into smaller pieces is once again the key to a successful implementation: First, we have to determine whether an expression needs to be expanded at all. This is done by `EXPRESSION_EXPANDABLE`, which recursively operates on all branches connected by operations plus and minus:

```
template < typename EXPR>
struct EXPRESSION_EXPANDABLE
{
  enum { ReturnValue = 0 }; //default behaviour
};

template < typename T, typename LHS, typename RHS>
struct EXPRESSION_EXPANDABLE < Expression<T, LHS, RHS, op_plus<T> > >
{
  enum { ReturnValue =   EXPRESSION_EXPANDABLE<LHS>::ReturnValue
                       + EXPRESSION_EXPANDABLE<RHS>::ReturnValue };
};

template < typename T, typename LHS, typename RHS>
struct EXPRESSION_EXPANDABLE < Expression<T, LHS, RHS, op_minus<T> > >
{
  enum { ReturnValue =   EXPRESSION_EXPANDABLE<LHS>::ReturnValue
                       + EXPRESSION_EXPANDABLE<RHS>::ReturnValue };
};
```

Then, as soon as two branches of the expression tree are connected by a multiplication, the expression can be further expanded if (and only if) one of the operands carries another plus or minus sign:

```
template < typename T, typename LHS, typename RHS>
struct EXPRESSION_EXPANDABLE < Expression<T, LHS, RHS, op_mult<T> > >
{
  enum { ReturnValue =
                EXPRESSION_OPERATION_COUNT<LHS, op_plus >::ReturnValue
              + EXPRESSION_OPERATION_COUNT<LHS, op_minus>::ReturnValue
              + EXPRESSION_OPERATION_COUNT<RHS, op_plus >::ReturnValue
              + EXPRESSION_OPERATION_COUNT<RHS, op_minus>::ReturnValue };
};
```

The helper meta function EXPRESSION_OPERATION_COUNT, which we are not going to discuss in further detail, counts the occurrences of plus and minus operands (specified in the second template argument) within the first template argument. Therefore, if EXPRESSION_EXPANDABLE<EXPR>::ReturnValue is positive for EXPR, the expression EXPR can be further expanded.

Next, let us consider the task of expanding a given expression by a multiplicative factor, which does not necessarily have to be a scalar. This functionality is provided by the meta function EXPRESSION_EXPAND_BY_FACTOR. Its implementation is straight-forward, but since it is the core functionality of EXPRESSION_EXPAND, the code will be given:

```
template <typename EXPR, typename FACTOR>
struct EXPRESSION_EXPAND_BY_FACTOR
{
  //default behaviour: build the product EXPR * FACTOR
  typedef Expression< ExpressionDefaultScalarType,
                      EXPR,
                      FACTOR,
                      op_mult <ExpressionDefaultScalarType>
                    >                 ResultType;
};

//specialisations for addition: multiply each term
template <typename T, typename LHS, typename RHS, typename FACTOR>
struct EXPRESSION_EXPAND_BY_FACTOR < Expression<T, LHS, RHS, op_plus<T> >,
                                     FACTOR>
{
  typedef Expression< T,
            typename EXPRESSION_EXPAND_BY_FACTOR<LHS, FACTOR>::ResultType,
            typename EXPRESSION_EXPAND_BY_FACTOR<RHS, FACTOR>::ResultType,
                      op_plus<T>
                    >                 ResultType;
};

//specialisations for subtraction in analogy to addition
```

The final implementation for EXPRESSION_EXPAND can now be given in pseudo-code:

```
while EXPRESSION_EXPANDABLE<EXPR>::ReturnValue != 0 do
  for every expandable summand 'summand' in EXPR do
    if the left-hand side 'lhs' of 'summand' contains +,- then
      for each summand 'sumlhs' of 'lhs' do
        - replace 'summand' by
          EXPRESSION_EXPAND_BY_FACTOR<'rhs', 'sumlhs'>::ResultType
          in EXPR;
      end
    else
      for each summand 'sumrhs' of 'rhs' do
        - replace 'summand' by
          EXPRESSION_EXPAND_BY_FACTOR<'lhs', 'sumrhs'>::ResultType
          in EXPR;
      end
    end
  end
end
```

An implementation in C++ is lengthy and mostly technical, therefore we skip the C++ code for EXPRESSION_EXPAND.

Now, with fully expanded expressions, it is possible to do a term-wise integration by the use of a meta function EXPRESSION_INTEGRATE:

```
template <typename L_BOUND, typename U_BOUND,
          typename EXPR, typename INTVAR>
struct EXPRESSION_INTEGRATE
{
  typedef typename EXPRESSION_INTEGRATE_IMPL
                   <  L_BOUND,
                      U_BOUND,
                      typename EXPRESSION_EXPAND<EXPR>::ResultType,
                      INTVAR
                   >::ResultType                          ResultType;
};
```

The template arguments are the lower and upper integral bounds (L_BOUND and U_BOUND), the integrand (EXPR) and the integration variable (INTVAR). A worker meta function EXPRESSION_INTEGRATE_IMPL performs integration on each summand of the integrand separately. Then, it counts and removes all occurrences of the integration variable and replaces it by the difference of the upper and lower bounds substituted for the anti-derivatives. Several partial specialisations realise this task, let us consider a specialisation that already uses that the contribution of polynomials for a lower bound zero is again zero:

```
//specialisation: a lower bound zero does not contribute
template <typename U_BOUND, typename T,
          typename LHS, typename RHS, typename INTVAR>
struct EXPRESSION_INTEGRATE_IMPL < ScalarExpression<0>,
                                   U_BOUND,
                                   Expression <T, LHS, RHS, op_mult<T> >,
                                   INTVAR
                                 >
{
  typedef Expression < T, LHS, RHS, op_mult<T> >    IntegrandType;
  enum { PowerOfIntegrationVariable =
         EXPRESSION_POWER_OF_VARIABLE<IntegrandType, INTVAR>::ReturnValue
            };

  typedef Expression < T,
                       Expression < T,
                                    typename POW< U_BOUND,
                                            PowerOfIntegrationVariable + 1
                                                >::ResultType,
                                    typename EXPRESSION_SUBSTITUTE
                                                 < IntegrandType,
                                                   INTVAR,
                                                   ScalarExpression<1>
                                                 >::ResultType,
                                    op_mult<T>
                                  >,
                       ScalarExpression< PowerOfIntegrationVariable + 1 >,
                       op_div<T>
                     >                                          ResultType;
};
```

The integer PowerOfIntegrationVariable holds the power of INTVAR, which is the result of a helper meta function EXPRESSION_POWER_OF_VARIABLE. Then, the final expression is formed by multiplying the anti-

derivate of the integration variable polynomial with a modified integrand where all occurrences of the integration variable are replaced by one. Let us demonstrate the process for the integral $\int_0^a \xi_0\xi_1\xi_0\xi_1 \,\mathrm{d}\xi_1$:

$$\int_0^a \xi_0\xi_1\xi_0\xi_1 \,\mathrm{d}\xi_1 = \underbrace{a^3/3}_{\text{Antiderivative of } \xi_1^2} \cdot \underbrace{\xi_0 \cdot 1 \cdot \xi_0 \cdot 1}_{\substack{\text{Result of} \\ \texttt{EXPRESSION\_SUBSTITUTE}}} . \tag{5.5}$$

Instead of replacing the integration variable by 1 using `EXPRESSION_SUBSTITUTE` one could also provide a separate meta function `EXPRESSION_REMOVE`.

With the meta functions developed so far, the integral (5.4) can be computed as follows:

```
typedef Expression<double,
                   ScalarExpression<1>,
                   var<0>,
                   op_minus<double>
                  >                                   OneMinusX;

typedef Expression<double,
                   var<0>,
                   var<1>,
                   op_mult<double> >                   IntegrandType;

//the type of the evaluated integral:
typedef EXPRESSION_INTEGRATE<ScalarExpression<0>,
                             ScalarExpression<1>,
                             EXPRESSION_INTEGRATE< ScalarExpression<0>,
                                                   OneMinusX,
                                                   IntegrandType,
                                                   var<1>
                                                 >::ResultType,
                             var<0>
                            >::ResultType          ResultType;
```

In the final FEM assembly procedure, `IntegrandType` is already available from plugging a pair of basis functions into the transformed weak formulation from Section 5.1.

When finally the type encoding the result of the integral is obtained, it consists of several scalar fractions, additions, subtractions and multiplications. However, compilers will typically not fully evaluate the rather complex expression tree during compile time, thus we have to simplify the scalar expression tree ourselves. This is the task of `EXPRESSION_OPTIMIZER`, which traverses the expression tree as long as simplifications can be found:

```
template <typename EXP>
struct EXPRESSION_OPTIMIZER;
```

The optimisation rules are implemented in `EXPRESSION_OPTIMIZER_IMPL`, which provides implementations for all optimisation rules to `EXPRESSION_OPTIMIZER`. The code is straight-forward, so we give an overview of the implemented rules in Tab. 5.1 only, where a, b, c and d denote scalar template parameters denoted as such, while the `ScalarExpression` wrapper is omitted. Terms in parenthesis indicate an evaluation by the compiler and thus become new scalar template arguments.

A single rule of this set is typically easy to implement, but the whole set of rules renders implementation very tedious, since for every expression only one template specialisation is allowed to match, leading to many specialisations whose only purpose is to resolve ambiguities. However, with all these optimisation rules, even complicated scalar expression trees simplify to a single fraction, as desired.

Additions:	$0+a=a$	$a+0=a$	$a+b=(a+b)$
Subtractions:	$a-0=a$	$a-b=(a-b)$	
Multiplications:	$0\cdot a=0$	$a\cdot 0=0$	$a\cdot b=(ab)$
Divisions:	$0/a=0$	$a/1=a$	
Fractionals:	$a\pm b/c=(ac\pm b)/c$	$b/c\pm a=(ac\pm b)/c$	$a\cdot b/c=(ab)/c$
	$b/c\cdot a=(ab)/c$	$a/b\pm c/d=(ad\pm bc)/(bd)$	$a/b\cdot c/d=(ac)/(bd)$

Table 5.1: Simplification rules for compile time scalar expressions a, b, c and d. Expressions in parenthesis are evaluated at compile time. Rules involving zeros and ones have been included explicitely, since they can be applied to general expressions for a and b.

There is another subtlety with the simplifications of fractionals: Assume that the term $1/120 + 30/240 + 10/1000$ has to be simplified at compile time. With the rules implemented above, we would get a denominator of $120 \cdot 240 \cdot 1000 = 28800000$. Adding another term $1/5000$ would then result in an overflow of the denominator! For this reason, common factors have to be cancelled each time two fractionals are simplified to one:

```
template <long num, long denum>
struct COMPRESS_FRACTIONAL
{
  enum{ Numerator = num / GREATEST_COMMON_DIVISOR<num, denum>::ReturnValue,
        Denominator = denum/GREATEST_COMMON_DIVISOR<num, denum>::
           ReturnValue };
};
```

The greatest common divisor of the numerator and the denominator is found using a compile time version of Euklid's algorithm:

```
template <long a, long b>
struct GREATEST_COMMON_DIVISOR
{
  enum { ReturnValue = GREATEST_COMMON_DIVISOR< b, a % b >::ReturnValue };
};

template <long a>
struct GREATEST_COMMON_DIVISOR<a, 0>
{
  enum { ReturnValue = a };
};
```

This way we obtain a stable manipulation of fractionals at compile time in the sense that now overflows of the numerator or denomiator occur as long as overflows can be prevented at all. However, even if an overflow occurs for some reason, good compilers issue a warning.

The set of manipulations to be carried out by the compiler is now complete, but very large in total. The question which naturally arises is the question of compilation times. As can be seen in Tab. 5.2, compilation for a one-dimensional problem works pretty well up to basis functions of degree

	1D	2D	3D
Linear	11s, 151MB	14s, 190MB	21s, 264MB
Quadratic	13s, 171MB	25s, 269MB	252s, 818MB
Cubic	15s, 193MB	768s, 1599MB	-
Quartic	31s, 292MB	-	-
Quintic	186s, 962MB	-	-

Table 5.2: Compilation times and memory consumption for several degrees of basis polynomials in different dimensions with brute force analytical integration. The integrals are computed by full expansions of the integrands. (Compilation times on a Pentium M, 1.5 GHz, 2 GB RAM)

five, because there are no nested integrals involved. The huge difference between quartic and quintic basis polynomials is nevertheless interesting and results from an explosion of the number of terms with higher polynomial degrees.

In two dimensions, compilation times for Laplace's equation up to degree three are within a reasonable range, but compilation times for higher order polynomials exceed the available memory of two gigabytes.

In three dimensions, however, compilation for quadratic basis functions takes a lot of time already, so that the compilation time for cubic polynomials (considering a factor of at least 30 for the transition from quadratic to cubic basis functions in two dimensions) and memory consumption (a factor of 6 in two dimensions) already become unjustifiable.

Why is there such a break-down of compilation times especially for three-dimensional problems? On the reference tetrahedron with corners at $(0,0,0)$, $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$, one quadratic basis function is $\psi = \xi_2(1-\xi_0-\xi_1-\xi_2)$. Using the product rule for differentiation, the compiler will find for the ξ_2-derivative $\partial\psi/\partial\xi_2 = (1-\xi_0-\xi_1-\xi_2) - \xi_2$. Therefore, one of the integrals that has to be evaluated at compile time could be

$$\int_0^1 \int_0^{1-\xi_0} \int_0^{1-\xi_0-\xi_1} [(1-\xi_0-\xi_1-\xi_2)-\xi_2]^2 \,\mathrm{d}\xi_2 \,\mathrm{d}\xi_1 \,\mathrm{d}\xi_0. \tag{5.6}$$

Assuming that the expression optimisation collects the two ξ_2, the integrand after a full expansion becomes

$$\begin{aligned}\int_0^1 \int_0^{1-\xi_0} \int_0^{1-\xi_0-\xi_1} (1-\xi_0-\xi_1-2\xi_2)^2 \,\mathrm{d}\xi_2 \,\mathrm{d}\xi_1 \,\mathrm{d}\xi_0 = \\ \int_0^1 \int_0^{1-\xi_0} \int_0^{1-\xi_0-\xi_1} 1-\xi_0-\xi_1-2\xi_2-\xi_0+\xi_0^2+\xi_1\xi_0+2\xi_1\xi_2-\xi_1+\xi_0\xi_1+\xi_1^2+2\xi_1\xi_2 \\ -2\xi_2+\xi_0\xi_2+\xi_1\xi_2+4\xi_2^2 \;\mathrm{d}\xi_2 \,\mathrm{d}\xi_1 \,\mathrm{d}\xi_0.\end{aligned} \tag{5.7}$$

Now, the compiler has to integrate each term separately. Let us consider the summand $4\xi_2^2$ only:

$$\int_0^1 \int_0^{1-\xi_0} \int_0^{1-\xi_0-\xi_1} 4\xi_2^2 \,\mathrm{d}\xi_2 \,\mathrm{d}\xi_1 \,\mathrm{d}\xi_0 = \frac{4}{3} \int_0^1 \int_0^{1-\xi_0} (1-\xi_0-\xi_1)^3 \,\mathrm{d}\xi_1 \,\mathrm{d}\xi_0. \tag{5.8}$$

Again, the integrand has to be fully expanded, leading to 27 terms. For each of these, a double integral has to be computed with variables of powers up to three. Anyway, there is not only one triple-integral to be computed, the computation has actually to be done for all pairs of basis functions, so for quadratic basis functions in three dimensions this results in 100 triple-integrals!

At this point, one may wonder why it is even possible to compile quadratic basis functions in three dimensions at all. The answer is that the compiler instantiates each template only once and stores the result in a type table. This prevents several evaluations of the same integral and speeds up the whole process. Anyway, going from quadratic to cubic basis functions, the number of basis functions doubles, thus the number of triple integrals increases by a factor of four. Additionally, each integrand for the stiffness matrix is a polynomial of order up to four, whose full expansion leads to a huge number of terms, thus creating even more complicated types. This slows type look-ups down, resulting in even longer compilation times. For this reasons, some other approach is necessary to get an analytic evaluation of such integrals at compile time.

5.4 Analytic Formula for Simplex Geometries

Considering the integrals arising for the mass matrix of nodal product basis functions, integrals of the form

$$\int_0^1 \int_0^{1-\xi_0} \xi_0^\alpha \xi_1^\beta (1-\xi_0-\xi_1)^\gamma \,\mathrm{d}\xi_1 \mathrm{d}\xi_0 \tag{5.9}$$

have to be computed for a triangle. For tetrahedra, triple-integrals of similar form show up. In the literature [32], formulas for the computation of monomials over triangles and tetrahedra are given. We are going to find a corresponding formula in n dimensions now.

First, we observe that (5.9) can be rewritten as ($\gamma \in \mathbb{Z}^+$)

$$\int_0^1 \int_0^{1-\xi_0} \xi_0^\alpha \xi_1^\beta (1-\xi_0-\xi_1)^\gamma \,\mathrm{d}\xi_1 \mathrm{d}\xi_0 = \gamma \int_0^1 \int_0^{1-\xi_0} \int_0^{1-\xi_0-\xi_1} \xi_0^\alpha \xi_1^\beta \xi_2^{\gamma-1} \,\mathrm{d}\xi_2 \,\mathrm{d}\xi_1 \,\mathrm{d}\xi_0 \ . \tag{5.10}$$

Let us do the calculations in arbitrary dimensions:

Definition 5. *For an $n+1$-multi-index $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_n)$ with $\alpha_i \in \mathbb{N},\ i = 0, \ldots, n$, we define*

$$I_\alpha = \int_{S_n} \left(\prod_{i=0}^{n-1} \xi_i^{\alpha_i} \right) \left(1 - \sum_{i=0}^{n-1} \xi_i \right)^{\alpha_n} \mathrm{d}\boldsymbol{\xi} \ , \tag{5.11}$$

$$\mathcal{I}_\alpha = \int_{S_{n+1}} \prod_{i=0}^{n} \xi_i^{\alpha_i} \,\mathrm{d}\boldsymbol{\xi} \ . \tag{5.12}$$

The connection (5.10) found for two and three dimensions actually holds for arbitrary dimensions:

Lemma 1. *Let $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_n)$ and $\alpha^+ = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \alpha_n + 1)$. Then*

$$\mathcal{I}_\alpha = \frac{1}{\alpha_n + 1} I_{\alpha^+} \ . \tag{5.13}$$

Proof. Since

$$\int_{S_{n+1}} \cdot \,\mathrm{d}\boldsymbol{\xi} = \int_0^1 \int_0^{1-\xi_0} \int_0^{1-\xi_0-\xi_1} \cdots \int_0^{1-\sum_{i=0}^{n-1}\xi_i} \int_0^{1-\sum_{i=0}^{n}\xi_i} \cdot \,\mathrm{d}\boldsymbol{\xi} = \int_{S_n} \int_0^{1-\sum_{i=0}^{n}\xi_i} \cdot \,\mathrm{d}\boldsymbol{\xi} \ ,$$

we obtain

$$\begin{aligned}\mathcal{I}_\alpha &= \int_{S_{n+1}} \prod_{i=0}^{n} \xi_i^{\alpha_i} \,\mathrm{d}\boldsymbol{\xi} \\ &= \int_{S_n} \int_0^{1-\sum_{i=0}^{n-1}\xi_i} \left(\prod_{i=0}^{n-1} \xi_i^{\alpha_i}\right) \xi_n^{\alpha_n} \,\mathrm{d}\xi_n \,\mathrm{d}\xi_{n-1} \,\ldots\, \mathrm{d}\xi_0 \\ &= \int_{S_n} \left(\prod_{i=0}^{n-1} \xi_i^{\alpha_i}\right) \frac{\left(1-\sum_{i=0}^{n-1}\xi_i\right)^{\alpha_n+1}}{\alpha_n+1} \,\mathrm{d}\xi_{n-1} \,\ldots\, \mathrm{d}\xi_0 \\ &= \frac{1}{\alpha_n+1} I_{\alpha^+} \ .\end{aligned}$$

□

There is a direct applicability for FEM: Integrals over products of vertex functions on a reference n-simplex S_n can be obtained from integrals over monomials in a higher dimension and vice versa. What remains is to find an explicit formula for either I_α or $\mathcal{I}_\alpha$.

Lemma 2. *For an $n+1$-multi-index $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \alpha_n)$ there holds*

1.

$$I_{(\alpha_0,\ldots,\alpha_n)} = \frac{\alpha_{n-1}}{\alpha_k+1} I_{(\alpha_0,\ldots,\alpha_k+1,\ldots,\alpha_n-1)} \ , \quad \forall k \in \{0,\ldots,n-1\} \ , \tag{5.14}$$

2.

$$I_{(\alpha_0,\ldots,\alpha_n)} = \frac{\alpha_k!\,\alpha_n!}{(\alpha_k+\alpha_n)!} I_{(\alpha_0,\ldots,\alpha_k+\alpha_n,\ldots,\alpha_{n-1},0)} \ , \quad \forall k \in \{0,\ldots,n-1\} \ . \tag{5.15}$$

Proof. 1. Partial integration with respect to ξ_{n-1} gives

$$\begin{aligned}I_{(\alpha_0,\ldots,\alpha_n)} &= \int_{S_n} \left(\prod_{i=0}^{n-1} \xi_i^{\alpha_i}\right) \left(1-\sum_{i=0}^{n-1}\xi_i\right)^{\alpha_n} \mathrm{d}\boldsymbol{\xi} \\ &= \int_{S_{n-1}} \left(\prod_{i=0}^{n-2} \xi_i^{\alpha_i}\right) \Bigg[\frac{1}{\alpha_{n-1}+1} \xi_{n-1}^{\alpha_{n-1}+1} \left(1-\sum_{i=0}^{n-1}\xi_i\right)^{\alpha_n} \Bigg|_{\xi_{n-1}=0}^{\xi_{n-1}=1-\sum_{i=0}^{n-2}\xi_i} \\ &\qquad + \frac{\alpha_n}{\alpha_{n-1}+1} \int_0^{1-\sum_{i=0}^{n-2}\xi_i} \xi_{n-1}^{\alpha_{n-1}+1} \left(1-\sum_{i=0}^{n-1}\xi_i\right)^{\alpha_n-1} \mathrm{d}\xi_{n-1} \Bigg] \,\mathrm{d}\xi_{n-2} \ldots \mathrm{d}\xi_0 \ .\end{aligned}$$

The first term in brackets vanishes, since for both bounds one of the multiplicands becomes zero. Thus,

$$\begin{aligned}I_{(\alpha_0,\ldots,\alpha_n)} &= \int_{S_{n-1}} \left(\prod_{i=0}^{n-2} \xi_i^{\alpha_i}\right) \frac{\alpha_n}{\alpha_{n-1}+1} \int_0^{1-\sum_{i=0}^{n-2}\xi_i} \xi_{n-1}^{\alpha_{n-1}+1} \left(1-\sum_{i=0}^{n-1}\xi_i\right)^{\alpha_n-1} \mathrm{d}\xi_{n-1} \,\mathrm{d}\xi_{n-2} \ldots \mathrm{d}\xi_0 \\ &= \frac{\alpha_n}{\alpha_{n-1}+1} I_{(\alpha_0,\ldots,\alpha_{n-2},\alpha_{n-1}+1,\alpha_n-1)} \ .\end{aligned}$$

This proves (5.14) for the case $k = n-1$. The more general case $k \in \{0,\ldots,n-1\}$ now follows from symmetry of the integrand and the integration domain with respect to the underlying coordinate system.

2. Applying (5.14) repeatedly until $a_n = 0$ yields

$$\begin{aligned} I_{(\alpha_0,\ldots,\alpha_n)} &= \frac{(\alpha_n)(\alpha_n-1)(\alpha_n-2)\cdots 1}{(\alpha_k+1)(\alpha_k+2)\cdots(\alpha_k+\alpha_n)} I_{(\alpha_0,\ldots,\alpha_k+\alpha_n,\ldots\alpha_{n-1},0)} \\ &= \frac{\alpha_k!\,(\alpha_n)(\alpha_n-1)(\alpha_n-2)\cdots 1}{\alpha_k!\,(\alpha_k+1)(\alpha_k+2)\cdots(\alpha_k+\alpha_n)} I_{(\alpha_0,\ldots,\alpha_k+\alpha_n,\ldots\alpha_{n-1},0)} \\ &= \frac{\alpha_k!\,\alpha_n!}{(\alpha_k+\alpha_n)!} I_{(\alpha_0,\ldots,\alpha_k+\alpha_n,\ldots,\alpha_{n-1},0)}, \quad \forall k \in \{0,\ldots,n-1\}. \end{aligned}$$

□

Now we are ready to find explicit expressions for $\mathcal{I}_\alpha$ and I_α:

Theorem 3. *For an $n+1$-multi-index $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \alpha_n)$ the following explicit formulas hold:*

1.

$$I_\alpha = \frac{\alpha_0!\,\alpha_1!\cdots\alpha_n!}{(\alpha_0+\alpha_1+\ldots+\alpha_n+n)!}\,, \tag{5.16}$$

2.

$$\mathcal{I}_\alpha = \frac{\alpha_0!\,\alpha_1!\cdots\alpha_n!}{(\alpha_0+\alpha_1+\ldots+\alpha_n+n+1)!}\,. \tag{5.17}$$

Proof. We prove both statements by simultaneous induction: For $n = 1$, we find with (5.15):

$$\begin{aligned} I_{(\alpha_0,\alpha_1)} &= \frac{\alpha_0!\,\alpha_1!}{(\alpha_0+\alpha_1)!} I_{(\alpha_0+\alpha_1,0)} \\ &= \frac{\alpha_0!\,\alpha_1!}{(\alpha_0+\alpha_1)!} \int_0^1 \xi^{\alpha_0+\alpha_1}\,\mathrm{d}\xi \\ &= \frac{\alpha_0!\,\alpha_1!}{(\alpha_0+\alpha_1)!} \frac{1}{\alpha_0+\alpha_1+1} \\ &= \frac{\alpha_0!\,\alpha_1!}{(\alpha_0+\alpha_1+1)!}\,. \end{aligned}$$

With (5.13) we get

$$\mathcal{I}_{(\alpha_0,\alpha_1)} = \frac{1}{\alpha_1+1} I_{(\alpha_0,\alpha_1+1)} = \frac{\alpha_0!\,\alpha_1!}{(\alpha_0+\alpha_1+2)!},$$

which proves the statements for $n = 1$.

Next, the induction step $n \to n+1$ needs to be done.

$$I_{(\alpha_0,\ldots,\alpha_{n+1})} = \frac{\alpha_1!\,\alpha_{n+1}!}{(\alpha_0+\alpha_{n+1})!} I_{(\alpha_0+\alpha_{n+1},\alpha_1,\ldots,\alpha_n,0)}.$$

From the definitions of I_α and $\mathcal{I}_\alpha$ we observe that

$$I_{(\alpha_0+\alpha_{n+1},\alpha_1,\ldots,\alpha_n,0)} = \mathcal{I}_{(\alpha_0+\alpha_{n+1},\alpha_1,\ldots,\alpha_n)},$$

thus

$$I_{(\alpha_0,\ldots,\alpha_{n+1})} = \frac{\alpha_0!\,\alpha_{n+1}!}{(\alpha_0+\alpha_{n+1})!} \mathcal{I}_{(\alpha_0+\alpha_{n+1},\alpha_1,\ldots,\alpha_n)}.$$

However, $\mathcal{I}_{(\alpha_0+\alpha_{n+1},\alpha_1,\ldots,\alpha_n)}$ is already known from the induction assertion for n, therefore

$$\begin{aligned} I_{(\alpha_0,\ldots,\alpha_{n+1})} &= \frac{\alpha_0!\,\alpha_{n+1}!}{(\alpha_0+\alpha_{n+1})!}\frac{(\alpha_0+\alpha_{n+1})!\,\alpha_1!\cdots\alpha_n!}{(\alpha_0+\alpha_1+\ldots+\alpha_n+\alpha_{n+1}+n+1)!} \\ &= \frac{\alpha_0!\,\alpha_1!\cdots\alpha_n!\,\alpha_{n+1}!}{(\alpha_0+\alpha_1+\ldots+\alpha_n+\alpha_{n+1}+n+1)!}, \end{aligned}$$

which is exactly (5.16) with n replaced by $n+1$. The induction is completed with

$$\begin{aligned} \mathcal{I}_{(\alpha_0,\ldots,\alpha_{n+1})} &= \frac{1}{\alpha_{n+1}+1} I_{(\alpha_0,\ldots,\alpha_{n+1}+1)} \\ &= \frac{1}{\alpha_{n+1}+1}\frac{\alpha_0!\,\alpha_1!\cdots\alpha_n!\,(\alpha_{n1}+1)!}{(\alpha_0+\alpha_1+\ldots+\alpha_n+\alpha_{n+1}+n+2)!} \\ &= \frac{\alpha_0!\,\alpha_1!\cdots\alpha_n!\,\alpha_{n+1}!}{(\alpha_0+\alpha_1+\ldots+\alpha_n+\alpha_{n+1}+n+2)!}\ . \end{aligned}$$

□

The theorem has some interesting consequences: First, it explains the poor conditioning properties of nodal product basis functions to some extent, since the resulting integrals are directly derived from monomials in a higher dimension. Such monomials, however, are known for their poor conditioning properties that can for example be observed at the Hilbert matrix.

The second consequence of the theorem is important for the implementation and will be discussed in the following section.

5.5 The Compound Expression

For sets of basis functions other than nodal product basis functions, integrals can be analytically computed at compile time based on a decomposition into vertex function products instead of monomials, which reduces the need for full polynomial expansions: For example, a basis function $\psi_{\mathbf{t}}$ on a tetrahedron $\mathbf{t}$ of the form $\psi_{\mathbf{t}} = \psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3}\phi$ for a kernel function $\phi(\xi_0,\xi_1,\xi_2) = a + b_1\xi_0 + b_2\xi_1 + b_3\xi_2$ can be decomposed into

$$\psi_{\mathbf{t}} = a\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3} + b_1\xi_0\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3} + b_2\xi_1\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3} + b_3\xi_2\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3}\ . \tag{5.18}$$

Now, since the reference tetrahedron is chosen to have vertex functions $\psi_{\mathbf{t}}^{\mathbf{v}_1} = \xi_0$, $\psi_{\mathbf{t}}^{\mathbf{v}_2} = \xi_1$ and $\psi_{\mathbf{t}}^{\mathbf{v}_3} = \xi_2$, we can further rewrite

$$\psi_{\mathbf{t}} = a\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3} + b_1\psi_{\mathbf{t}}^{\mathbf{v}_0}(\psi_{\mathbf{t}}^{\mathbf{v}_1})^2\psi_{\mathbf{t}}^{\mathbf{v}_2}\psi_{\mathbf{t}}^{\mathbf{v}_3} + b_2\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}(\psi_{\mathbf{t}}^{\mathbf{v}_2})^2\psi_{\mathbf{t}}^{\mathbf{v}_3} + b_3\psi_{\mathbf{t}}^{\mathbf{v}_0}\psi_{\mathbf{t}}^{\mathbf{v}_1}\psi_{\mathbf{t}}^{\mathbf{v}_2}(\psi_{\mathbf{t}}^{\mathbf{v}_3})^2\ . \tag{5.19}$$

Integration over the simplex then yields

$$\int_{S_3} \psi_{\mathbf{t}}\,\mathrm{d}\boldsymbol{x} = aI_{(1,1,1,1)} + b_1 I_{(1,2,1,1)} + b_2 I_{(1,1,2,1)} + b_3 I_{(1,1,1,2)}\ . \tag{5.20}$$

This way the compiler does not have to go through the expensive task of doing the integration "by hand", instead, each integral is decomposed into known ones. The benefit is a much faster compilation process.

Such simplifications still have to be made known to the compiler. A first candidate is the direct representation of the multi-index α in code, but this multi-index does not contain any information about the underlying vertex functions. They could be implicitly derived from the underlying geometry, but would unnecessarily couple geometric information with pure mathematical expressions. Therefore we encode both the underlying vertex functions and the multi-index:

```
template <long num, long denum,
          typename T1, long pow1,
          typename T2, long pow2,
          typename T3 = CompoundUnused, long pow3 = 0,
          typename T4 = CompoundUnused, long pow4 = 0
         >
struct CompoundExpression;
```

which represents the term $\frac{\text{num}}{\text{denum}} T_1^{\text{pow1}} T_2^{\text{pow2}} T_3^{\text{pow3}} T_4^{\text{pow4}}$, where T_1, T_2, T_3 and T_4 are vertex functions of the simplex. In two dimensions, T_4 is typically unused (which is indicated by a separate class `CompoundUnused`) and in one dimension also T_3 is unused.

This type also allows the specification of vertex functions on the reference element:

```
template <typename CellTag>
struct BFStock {};

//sample specialisation for a triangle:
template <>
struct BFStock<TriangleTag>
{
  typedef Expression< ExpressionDefaultScalarType,
                      var<0>,
                      var<1>,
                      op_plus<ExpressionDefaultScalarType>
                    >                              XplusY;

  typedef Expression< ExpressionDefaultScalarType,
                      ScalarExpression<1>,
                      XplusY,
                      op_minus<ExpressionDefaultScalarType> >  OneMinusXY;

  typedef CompoundExpression< 1, 1,
                              OneMinusXY, 0,
                              var<0>, 0,
                              var<1>, 0>                CompoundType;
};

 //similar for other cell tags like LineTag or TetrahedronTag
```

Since a compound expression is again a mathematical expression, it has to provide evaluation at a point just like other mathematical expressions:

```
template <long num, long denum,
          typename T1, long pow1,
          typename T2, long pow2,
          typename T3, long pow3,
          typename T4, long pow4
         >
struct CompoundExpression
{
  template <typename Point>
  double operator()(Point const & p) const
  {
    return static_cast<double>(num) / static_cast<double>(denum) *
```

```
13              pow(T1()(p), pow1) * pow(T2()(p), pow2) * pow(T3()(p), pow3) *
                   pow(T4()(p), pow4);
14    }
15  };
```

At this point a lot of optimisations can be applied for a run time evaluation: Calling `pow` can be prevented for exponents equal to zero, one or two by further partial specialisations, but since we are actually interested in computations at compile time, we will not go into further details.

Apart from an evaluation interface, some more interactions with the expression engine are defined, where the implementation is straight-forward and will not be shown:

- *Differentiation*: Using the product and chain rules for differentiation, `CompoundExpression` can make direct use of existing differentiation possibilities.
- *Overloading operators*: This allows to use the expression engine for building expressions including `CompoundExpression` types.
- *Absorption of terms*: This can be illustrated in one dimension: Let us denote `CompoundExpression <n,m,var<0>, a, var<1>, b>` with $\frac{n}{m}[x^\alpha, y^\beta]$, then simplifications like

$$x\frac{n}{m}[x^\alpha, y^\beta] \to \frac{n}{m}[x^{\alpha+1}, y^\beta] \tag{5.21}$$

$$y\frac{n}{m}[x^\alpha, y^\beta] \to \frac{n}{m}[x^\alpha, y^{\beta+1}] \tag{5.22}$$

$$\frac{n_1}{m_1}[x^{\alpha_1}, y^{\beta_1}] \cdot \frac{n_2}{m_2}[x^{\alpha_2}, y^{\beta_2}] \to \frac{n_1 n_2}{m_1 m_2}[x^{\alpha_1+\alpha_2}, y^{\beta_1+\beta_2}] \tag{5.23}$$

  have to be done by `EXPRESSION_OPTIMIZER` to ease integration as much as possible.

Let us have a look at how the integration of a `CompoundExpression` on a triangle is carried out at compile time:

```
template <long num, long denum,
          long alpha_1, long alpha_2, long alpha_3, typename T>
struct TRIANGLE_INTEGRATOR <
                //skipping the lengthy CompoundExpression<> type
                           >
{
  enum { Numerator   = FACTORIAL<alpha_1>::ReturnValue *
                       FACTORIAL<alpha_2>::ReturnValue *
                       FACTORIAL<alpha_3>::ReturnValue,
         Denominator = FACTORIAL<alpha_1+alpha_2+alpha_3+2>::ReturnValue };

  typedef COMPRESS_FRACTIONAL< Numerator, Denominator >  ResultFractional;

  typedef Expression< T,
                  ScalarExpression<   num * ResultFractional::Numerator >,
                  ScalarExpression< denum * ResultFractional::Denominator >,
                       op_div<T>
                     >                        ResultType;
};
```

The meta function `FACTORIAL` computes the factorial of its argument. `COMPRESS_FRACTIONAL` cancels common factors in the numerator and the denominator and has been introduced in the previous section. This raises the question for an overflow of template parameters of type `long` because of the factorial in the denominator: The first overflow occurs for 13!, therefore it is necessary to have $\sum_{i=0}^{n} \alpha_i \leq 12 - n$.

	1D	2D	3D
Linear	11s, 152MB	14s, 189MB	19s, 253MB
Quadratic	13s, 176MB	20s, 251MB	74s, 381MB
Cubic	14s, 179MB	36s, 310MB	819s, 1296MB
Quartic	15s, 193MB	102s, 444MB	-
Quintic	16s, 208MB	374s, 683MB	-

Table 5.3: Compilation times and memory consumption for several degrees of basis polynomials in different dimensions with analytical integration using an analytical formula.

In three dimensions, we are therefore able to integrate polynomials up to order nine without worrying about overflows. However, typically the integrand is made up of a product of two basis function polynomials, leading to a practical limitation to polynomials of degree four in three dimensions. This is nevertheless a quite natural bound arising from the size of element matrices: There are 35 quartic basis functions on a tetrahedron, leading to element matrices with 1225 entries each, so the executable becomes several megabytes in size and cannot be efficiently cached by the processor anymore. Apart from that, compilation within a reasonable amount of time is possible only for basis functions up to degree three at the moment.

5.6 Performance

Analytic integration at compile time allows a considerable run time improvement at the price of longer compilation times, which is quantified in this section.

5.6.1 Compilation Times

We consider the weak formulation

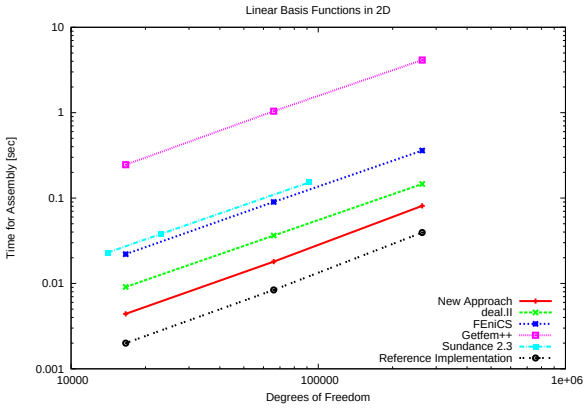
$$\text{Find } u \in H_0^1(\Omega) \text{ such that } \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = 0 \quad \text{for all } v \in H_0^1(\Omega)\ . \tag{5.24}$$

Our interest is in a first step only in compilation times and the time needed for the assembly at run time. Results for a compilation of basis functions with various degrees in one, two and three dimensions are shown in Tab. 5.3 and have been carried out using a rather aged Laptop with 1.5 GHz Pentium M CPU, 2 GB DDR SDRAM and g++ (version 4.2.1).

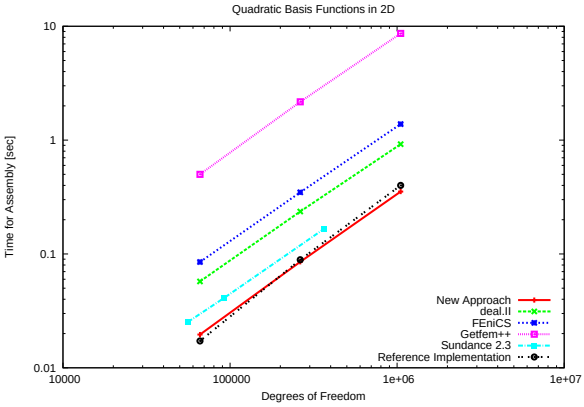
Using the analytical formula, compilation times in one dimension are very short for all polynomial degrees considered. This is indeed not surprising, because by the use of `CompoundExpression`, the number of terms of each integrands does not grow.

In two dimensions, compilation times for the considered polynomial orders are still in a reasonable range. Most likely, basis function polynomials of degree six also fit into the available memory, but an overflow in the denominator of the analytical formula may already occur. It has to be emphasized that there are already 21 basis polynomials of degree five (one per vertex, four per edge and six in the interior) defined on a triangle, so that for 441 basis polynomial pairs the bilinear form has to be evaluated during compile time. Transferring this result back to the one-dimensional case, where $p+1$ basis functions of degree p defined on a line, we can expect to have reasonable compilation times for one-dimensional problems up to degree 20.

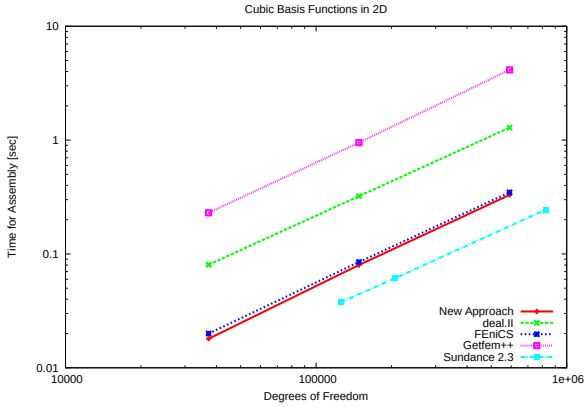
In three dimensions the compilation of cubic basis functions is the present limit for analytical integration at compile time. This is mainly due to the fact that the number of cubic basis polynomials



(a) Linear basis functions.



(b) Quadratic basis functions.



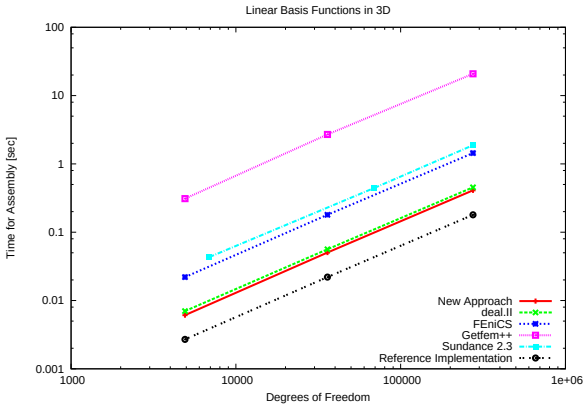
(c) Cubic basis functions.

Figure 5.2: Run time comparison for the assembly of the stiffness matrix in two dimensions.

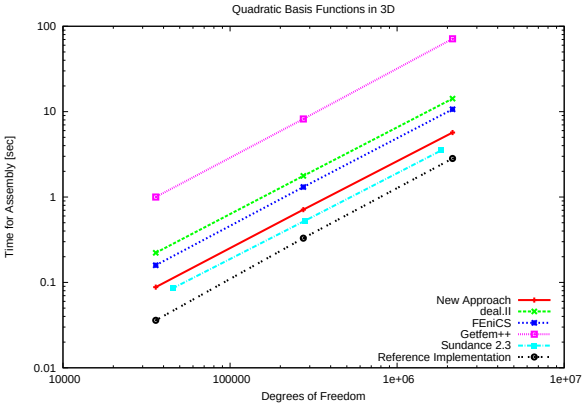
on a tetrahedron is already 20 (one per vertex, two per edge and one per facet), which is comparable with the case of quintic polynomials in two dimensions. Furthermore, the number of terms in the weak formulation increases and additional terms show up in the transformation to the reference element. Let us nevertheless estimate the compilation time and memory consumption for quartic basis functions: On average, we obtain one evaluation of the bilinear form every two seconds, using three megabytes of memory in three dimensions for cubic polynomials. For 35 quartic basis function polynomials (one per vertex, three per edge, three per facet, one bubble function), 1225 evaluations of the bilinear form are necessary, leading to a compilation time of approximately 2500 seconds and a memory consumption of slightly below four gigabytes. This amount of memory is in range of today's desktop machines and on more powerful CPUs, compilation times are reduced by a factor of approximately two to three additionally.

5.6.2 Run Time Performance

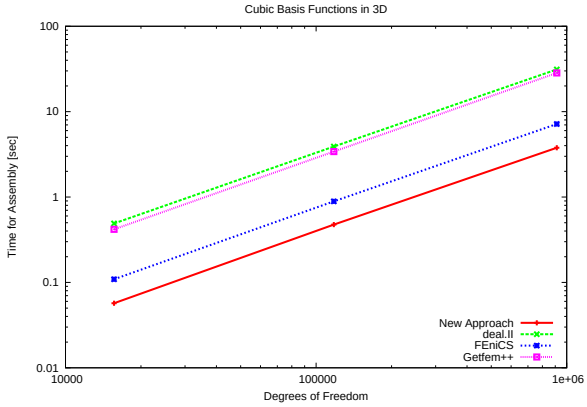
A comparison of execution times for the assembly of the stiffness matrix arising from (5.24) among the FEM packages mentioned in the introduction was carried out. Matrix access times that show up with sparse matrices have been eliminated using the same matrix type for all test candidates, which simply discards all accumulation requests, so that it emulates high speed data access. The resulting assembly times have been measured using a timer code block directly within code. The reference implementation is a hand-tuned C-code for linear and quadratic basis functions, where the element



(a) Linear basis functions.



(b) Quadratic basis functions.



(c) Cubic basis functions.

Figure 5.3: Run time comparison for the assembly of the stiffness matrix in three dimensions.

matrices summations have been unrolled by hand, which was especially very tedious in three dimensions. No further boundary handling is implemented in the reference implementation, so that it resembles a lower bound for execution times of FEM packages.

For linear basis functions in two dimensions, the framework presented in this thesis shows the shortest execution times among the FEM frameworks considered, which is mainly due to the domain management. In particular, the new approach presented here benefits from fast iterations over all cells of the domain. It is worth noting that execution times for linear basis functions differ by a factor of almost 100, which shows that a poor software design can have a huge impact on run time performance.

The test case with quadratic basis functions in two dimensions actually shows that a clever use of template metaprogramming indeed results in executables that outperforms those of hand-tuned code! The differences in execution time among the FEM packages are now smaller, because fast iteration over domain elements is now less important.

With cubic basis functions, `Sundance` is faster than the framework presented in this thesis. `Sundance` evaluates known basis function integrals in a preprocessing step during run time and simplifies the resulting expression trees as much as possible. In particular, no numerical quadrature is used, as it is for instance used by `deal.II`. The superior execution time of `Sundance` shows that the expression trees built by the new approach in this thesis still allow further simplifications and more efficient evaluations. We will come back to this soon.

In three dimensions (cf. Fig. 5.3), similar results are obtained: Linear basis functions show the highest variations in execution times. Quadratic basis functions put more emphasis on the efficient

evaluation of basis functions. `Sundance` shows shorter execution times for quadratic basis functions, but does not provide cubic basis functions in three dimensions. In the latter case the new approach again shows shortest execution. Nevertheless, more optimisations can be applied to the expression trees, as the superior execution times of `Sundance` for quadratic basis functions show.

Let us investigate the aforementioned simplifications for the expression trees: Analytical integration fully expands the integrand, so that products of the form $\alpha_{ijkl}\frac{\partial\xi_i}{\partial x_j}\frac{\partial\xi_k}{\partial x_l}$ show up. Let us consider

$$\begin{aligned}
&\alpha_{0000}\frac{\partial\xi_0}{\partial x_0}\frac{\partial\xi_0}{\partial x_0} + \alpha_{1000}\frac{\partial\xi_1}{\partial x_0}\frac{\partial\xi_0}{\partial x_0} + \alpha_{2000}\frac{\partial\xi_2}{\partial x_0}\frac{\partial\xi_0}{\partial x_0} + \\
&\alpha_{0010}\frac{\partial\xi_0}{\partial x_0}\frac{\partial\xi_1}{\partial x_0} + \alpha_{1010}\frac{\partial\xi_1}{\partial x_0}\frac{\partial\xi_1}{\partial x_0} + \alpha_{2010}\frac{\partial\xi_2}{\partial x_0}\frac{\partial\xi_1}{\partial x_0} + \\
&\alpha_{0020}\frac{\partial\xi_0}{\partial x_0}\frac{\partial\xi_2}{\partial x_0} + \alpha_{1020}\frac{\partial\xi_1}{\partial x_0}\frac{\partial\xi_2}{\partial x_0} + \alpha_{2020}\frac{\partial\xi_2}{\partial x_0}\frac{\partial\xi_2}{\partial x_0}
\end{aligned} \tag{5.25}$$

only, which shows up as one of the terms at the transformation to the reference element (compare with (5.1) for the two-dimensional case). The prefactors α_{ijkl} result from the evaluation of integrals on the reference cell then. A direct evaluation of this expression requires 18 multiplications and eight additions. However, if we rewrite (5.25) as

$$\begin{aligned}
&\left(\alpha_{0000}\frac{\partial\xi_0}{\partial x_0} + \alpha_{1000}\frac{\partial\xi_1}{\partial x_0} + \alpha_{2000}\frac{\partial\xi_2}{\partial x_0}\right)\frac{\partial\xi_0}{\partial x_0} \\
&+\left(\alpha_{0010}\frac{\partial\xi_0}{\partial x_0} + \alpha_{1010}\frac{\partial\xi_1}{\partial x_0} + \alpha_{2010}\frac{\partial\xi_2}{\partial x_0}\right)\frac{\partial\xi_1}{\partial x_0} \\
&+\left(\alpha_{0020}\frac{\partial\xi_0}{\partial x_0} + \alpha_{1020}\frac{\partial\xi_1}{\partial x_0} + \alpha_{2020}\frac{\partial\xi_2}{\partial x_0}\right)\frac{\partial\xi_2}{\partial x_0}\ ,
\end{aligned} \tag{5.26}$$

only twelve multiplications and eight additions are needed. Even faster evaluation is possible when the sums can be rewritten as a product of two sums, but this is only possible for certain symmetries of the prefactors α_{ijkl}. Such a postprocessing step after the analytical integration is possible, but not implemented yet. This should finally lead to run time efficiency more comparable to hand-tuned code again.

Analytic integration performs even better for mass matrices when compared to traditional numerical integration approaches: On the one hand, there is typically much less manipulation required at compile time, and on the other hand an integration rule with a higher degree of exactness has to be chosen for numerical quadrature than for the stiffness matrix. In particular, a quadrature rule exact for quadratic polynomials has to be chosen for linear basis functions. This typically requires two quadrature points in each spatial direction, leading to eight times more evaluations in three dimensions than for the integrals in the stiffness matrix, where only constant integrands occur.

On the whole, the new framework presented in this thesis shows the shortest overall assembly times. Further optimisations may even lead to the fastest assembly times in all test cases considered.

Chapter 6

Multigrid

With the analytic integration introduced in the previous chapter, assembly times can be reduced considerably. The resulting sparse system matrix still has to be solved, which can take considerably longer than the assembly procedure. For large systems of equations, *multigrid methods* are optimal in the sense that the computational effort grows linearly with the problem size only. This chapter illustrates the implemented domain handling concepts, which can be seamlessly incorporated into existing algorithms. In particular, the implemented multigrid capabilites allow finite element methods with basis functions of arbitrary order on unstructured grids, even though such cases are not fully investigated from the mathematical point of view. Nevertheless, the main motivation for the implementation of multigrid features has been due to the future support of self-adaptive refinement schemes (*p*-, *h*- and *hp*-refinement).

6.1 Basic Ideas of Multigrid

A brief introduction to the ideas of the multigrid method is given to motivate the domain handling strategies presented in the following sections. We follow the introduction given by Trottenberg et al. [29].

Given the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$, an iterative scheme for the true solution $\boldsymbol{x}$ with approximate solutions $\boldsymbol{x}^0$, $\boldsymbol{x}^1$, ... can be constructed by an iterative solution process as follows: Consider

$$\boldsymbol{d}^m = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^m \ , \tag{6.1}$$

where $\boldsymbol{d}^m$ is the *defect* of $\boldsymbol{x}^m$. By solving the *defect equation*

$$\boldsymbol{A}\boldsymbol{y}^m = \boldsymbol{d}^m \tag{6.2}$$

for the *correction* $\boldsymbol{y}^m$, the solution can be obtained as $\boldsymbol{x} = \boldsymbol{x}^m + \boldsymbol{y}^m$. Therefore, (6.1) is equivalent to (6.2). However, if we use an approximation $\tilde{\mathbf{A}}$ of $\mathbf{A}$ such that

$$\tilde{\boldsymbol{A}}\boldsymbol{y}^m = \boldsymbol{d}^m \tag{6.3}$$

can be solved more easily, an iterative process defined by successive application of

$$\boldsymbol{d}^m = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^m, \tag{6.4}$$

$$\tilde{\boldsymbol{A}}\boldsymbol{y}^m = \boldsymbol{d}^m \tag{6.5}$$

$$\boldsymbol{x}^{m+1} = \boldsymbol{x}^m + \boldsymbol{y}^m \tag{6.6}$$

for $m = 0, 1, \ldots$ is obtained. The update process can thus be rewritten as

$$\boldsymbol{x}^{m+1} = \left(\boldsymbol{I} - \tilde{\boldsymbol{A}}^{-1}\boldsymbol{A}\right)\boldsymbol{x}^m + \tilde{\boldsymbol{A}}^{-1}\boldsymbol{b}^m \ , \tag{6.7}$$

where $\boldsymbol{I}$ is the identity matrix in $\mathbb{R}^{n\times n}$. Hence, the task is to find a good approximation $\tilde{\boldsymbol{A}}$ of $\boldsymbol{A}$.

Let us now assume that $\boldsymbol{A}$ arises from a FEM assembly on a mesh Ω_h of a given PDE. We refer to the underlying mesh with characteristic measure h by writing $\boldsymbol{A}_h$ instead of $\boldsymbol{A}$. Assume now that the same PDE is assembled into a system matrix $\boldsymbol{A}_H$ on a coarser mesh Ω_H with a different characteristic measure H, for instance such that $H = 2h$. Similar subscripts are used for the solution and right-hand side vectors.

We approximate the defect equation (6.2) over Ω_h by the corresponding equation on Ω_H:

$$\boldsymbol{A}_H\boldsymbol{y}_H^m = \boldsymbol{d}_H^m \ . \tag{6.8}$$

This requires the transfer of functions (vectors) defined on Ω_h to Ω_H and vice versa. First, the *restriction operator*

$$\boldsymbol{I}_h^H : \mathcal{F}(\Omega_h) \to \mathcal{F}(\Omega_H) \tag{6.9}$$

is used to restrict the defect $\boldsymbol{d}_h^m$ to Ω_H:

$$\boldsymbol{d}_H^m := \boldsymbol{I}_h^H\boldsymbol{d}_h^m \ . \tag{6.10}$$

After solving the defect equation on Ω_H, the *prolongation operator*

$$\boldsymbol{I}_H^h : \mathcal{F}(\Omega_H) \to \mathcal{F}(\Omega_h) \tag{6.11}$$

prolongates the correction $\boldsymbol{y}_H^m$ to Ω_h:

$$\boldsymbol{y}_h^m := \boldsymbol{I}_H^h\boldsymbol{d}_H^m \ . \tag{6.12}$$

Summing up, one *coarse grid correction* step for two grids Ω_h (fine) and Ω_H (coarse) reads as follows:

1. Compute the defect on Ω_h: $\boldsymbol{d}_h^m = \boldsymbol{b}_h - \boldsymbol{A}_h\boldsymbol{x}_h^m$
2. Restrict the defect to Ω_H: $\boldsymbol{d}_H^m = \boldsymbol{I}_h^H\boldsymbol{d}_h^m$
3. Solve the defect equation on Ω_H: $\boldsymbol{A}_H\boldsymbol{y}_H^m = \boldsymbol{d}_H^m$.
4. Prolongate the correction to Ω_h: $\boldsymbol{y}_h^m = \boldsymbol{I}_H^h\boldsymbol{y}_H^m$
5. Update the approximation: $\boldsymbol{x}_h^{m+1} = \boldsymbol{x}_h^m + \boldsymbol{y}_h^m$

Taken on its own, a coarse grid correction process is of no use, since the kernel of $\boldsymbol{I}_h^H$ is non-empty for dimensional reasons, thus the defect $\boldsymbol{d}_h^m$ will not converge to the zero-vector in general. Typically, the kernel of $\boldsymbol{I}_h^H$ consist of high-frequency components on Ω_h that cannot be tracked by the coarse grid Ω_H. However, if we provide an additional smoothing mechanism to eliminate such high frequency errors in $\boldsymbol{d}_h^m$ before and/or after each coarse grid correction, the solution process becomes convergent.

Relaxation methods for the iterative solution of systems of linear equations are known to have such a smoothing behaviour. Widely used in practice are ω-Jacobi relaxation and Gauss-Seidel type relaxations, where the former has weaker smoothing properties (which further depends on the relaxation parameter ω), but is very attractive for parallelisation. Gauss-Seidel type relaxation methods have much better smoothing properties, but smoothing properties depend on the ordering of the underlying equations and a full parallelisation is not possible.

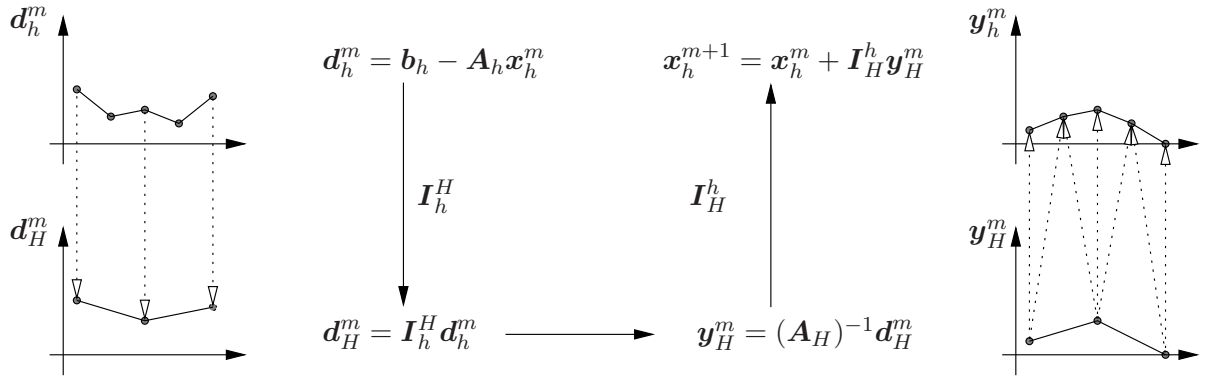


Figure 6.1: Graphical illustration of a multigrid correction step.

The multigrid idea is now to apply the ideas presented so far to the defect equation on the coarse grid Ω_H again. This leads to a defect equation for example on Ω_{2H}, which can be solved using even coarser grids on Ω_{4H} and so on.

Let us now denote a sequence of meshes $(\Omega_k)_{k=0}^K$ as Ω_{h_k} with mesh sizes h_k. Let the meshes be nested such that

$$\Omega_{h_0} \subset \Omega_{h_1} \subset \Omega_{h_2} \subset \dots \tag{6.13}$$

With more than two grids, the number γ_k of coarse grid corrections on a level k, $0 < k < K$, before prolongation is an additional parameter. Typically, all γ_k are chosen to equal to a global parameter γ. The case $\gamma = 1$ is often called *V-cycle* and the choice $\gamma = 2$ is known as *W-cycle*. The names result from the resulting graphical structures shown in Fig. 6.2.

6.2 Parents and Children

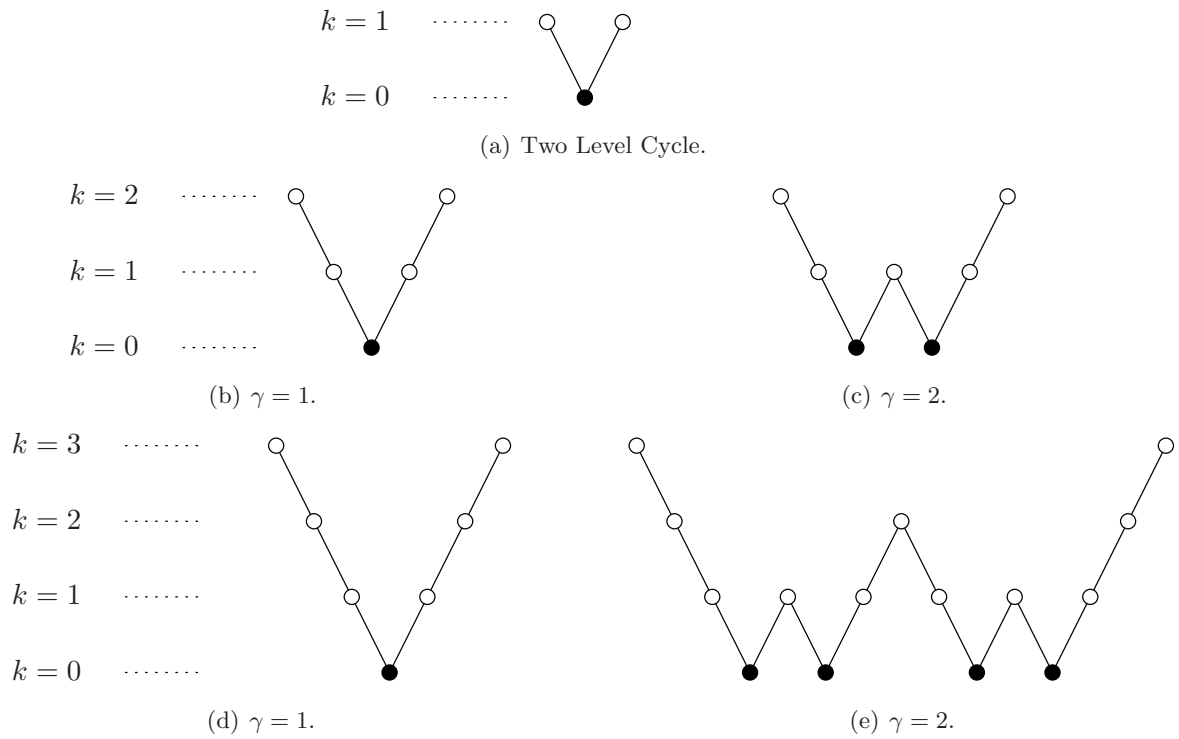
The requirement of nested meshes as in (6.13) raises the question of a software design that allows flexible transitions between these nested meshes. Typically, the restriction and prolongation operators work locally on a per-cell basis, so that for a coarse cell the access to refined cells has to be directly available. Let us define suitable vocabulary for more preciseness:

Definition 6. *Let $\mathcal{T}_1 = \mathcal{T}_2(\Omega)$ and $\mathcal{T}_2 = \mathcal{T}_2(\Omega)$ be two meshes.*

1. *$\mathcal{T}_1$ and $\mathcal{T}_2$ are equal ($\mathcal{T}_1 = \mathcal{T}_2$), if $T \in \mathcal{T}_1 \Leftrightarrow T \in \mathcal{T}_2$.*
2. *$\mathcal{T}_2$ is called* child *of $\mathcal{T}_1$, $\mathcal{T}_1 \neq \mathcal{T}_2$, if for every $T_1 \in \mathcal{T}_1$ there exists a finite decomposition $\mathcal{T}(T_1)$ of T_1 with elements from $\mathcal{T}_2$, i.e. $T \in \mathcal{T}_2$ for all $T \in \mathcal{T}(T_1)$. Consequently, the elements of $\mathcal{T}(T_1)$ are called* children *of T_1.*
3. *Vice versa, T_1 is called* parent *for all $T \in \mathcal{T}(T_1)$. $\mathcal{T}_1$ is called* parent *of $\mathcal{T}_2$ if $\mathcal{T}_2$ is a child of $\mathcal{T}_1$.*

We are now going to apply these parent-child-relations to the domain decomposition model from Chapter 2. A single-grid domain handling should still be provided, thus we introduce multigrid configuration tags:

```
1 struct FullMultigridTag {};
2 struct NoMultigridTag {};
```



(a) Two Level Cycle.

(b) $\gamma = 1$.

(c) $\gamma = 2$.

(d) $\gamma = 1$.

(e) $\gamma = 2$.

Figure 6.2: Graphical illustration of a multigrid correction step. Empty circles denote smoothing, filled circles denote an exact solution.

We extend the template class `segment` by one additional parameter, namely the multigrid configuration tag. For `NoMultigridTag`, the class specialisation equals the single grid implementation, while for `FullMultigridTag`, nested meshes are realised as linked lists at cell-level:

```
1  emplate <typename T_Configuration, unsigned long levelnum,
2           typename HandlingTag>
3  class segment <T_Configuration, levelnum, HandlingTag, FullMultigridTag>
4    : public segment <T_Configuration, T_Configuration::CellTag::TopoLevel-1>
5  {
6      typedef segment <T_Configuration,
7                       T_Configuration::CellTag::TopoLevel - 1>     Base;
8      typedef typename DomainTypes<T_Configuration>::CellType
          LevelElementType;
9
10     //some more type definitions as for the single grid case
11
12   public:
13     //several member functions as for the single grid case
14
15     segment & getRefinedSegment(long level)
16     {
17       if (level > 0 && child_seg != 0)
18         return child_seg->getRefinedSegment(level-1);
19       else
20         return *this;
21     };
22
```

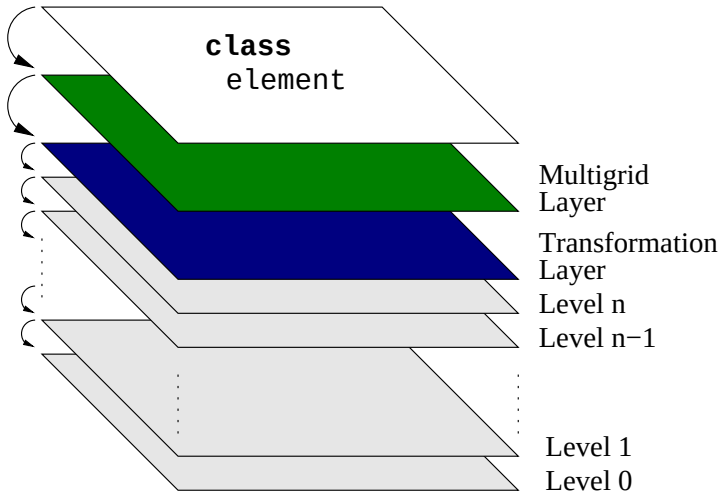


Figure 6.3: A domain element directly inherits from the multigrid layer. Since there is no dependency between the transformation layer and the multigrid layer, these two layers permute.

```
23    private:
24      std::vector< LevelElementType > elements;
25      segment * child_seg;
26  };
```

We postpone the discussion of setting up a refined segment to Section 6.3. For a given segment `seg`, the child segment can be obtained by

```
1   SegmentType & child_segment = seg.getRefinedSegment(1);
```

The child of a child of a segment is therefore obtained from `seg.getRefinedSegment(2)`, so the function parameter returns the `level`-th descendant of `seg`.

Next, multigrid capabilities have to be provided by domain elements as well. Following the layer model introduced in Chapter 2, we introduce a multigrid layer between `element` and `dtdx_handler`:

```
1   template <typename T_Configuration,
2             typename ElementTag,
3             typename MultigridTag = typename T_Configuration::MultigridTag>
4   class multigrid_layer;
```

Here, multigrid capabilities are controlled by the domain-wide configuration class via a type definition `MultigridTag`, but this could also be provided by the element tag for a finer control or multigrid capabilities at different layers.

The specialisation for `NoMultigridTag` is just an almost empty class which passes constructor calls to lower layers. A specialisation for `FullMultigridTag` adds the necessary multigrid functionality:

```
1   template <typename T_Configuration, typename ElTag>
2   class multigrid_layer<T_Configuration, ElTag, FullMultigridTag>
3     : public dt_dx_handler < T_Configuration, ElTag >
4   {
5       typedef dt_dx_handler < T_Configuration, ElTag >        Base;
6       typedef element<T_Configuration, ElTag>                 ElementType;
7
8     public:
9       typedef childit< element<T_Configuration, ElTag> >       ChildIterator;
10
```

```
11      multigrid_layer() : Base() {}
12      multigrid_layer(const multigrid_layer & mgl) : Base(mgl) {}
13
14      ChildIterator getChildrenBegin()
15      {
16        return ChildIterator(children.begin());
17      }
18      ChildIterator getChildrenEnd() { return ChildIterator(children.end());
           }
19
20      long getChildNum() { return children.size(); }
21
22      //member functions for the refinement of the element postponed
23
24    private:
25      std::list<ElementType *>  children;     //pointers to children
26  };
```

The `ChildIterator` is a wrapper for the iterator provided by `std::list`; its task is to dereference the pointers stored in the list.

There is only one single line that has to be changed in the existing class hierarchy introduced in Chapter 2: `element` has to inherit from `multigrid_layer` instead of `dt_dx_handler`:

```
1  template <typename T_Configuration, typename ElTag>
2  class element
3    : public QuantityManager< T_Configuration, ElTag >,
4      //inherit from multigrid_layer instead of dt_dx_handler:
5      public multigrid_layer < T_Configuration, ElTag >
6  {
7      typedef multigrid_layer < T_Configuration, ElTag >      Base;
8
9      //remaining implementation stays unchanged
10 };
```

Please note that now all elements within the domain provide multigrid capabilities! However, there are some more subtleties we have to care about, which will be the topics of the following sections.

6.3 Refinement of Segments

With the extension of the domain model for multigrid capabilities, the crucial step is to set up a new segment resulting from the refinement of a coarse segment. However, from the new (refined) segment's point of view, there is no difference in whether it is set up from a mesh file or whether it the child of some coarser mesh. For this reason we can make use of the existing functionality for a segment setup. The extra bit of information for nested segments, i.e. the parent-child-relation, is consequently stored in the multigrid layer of elements from the parent mesh.

Since the refinement of a segment boils down to the refinement of each cell of that segment, it is sufficient to consider the refinement of a single cell. The refinement of a cell depends on its geometric shape, thus the element tag has to provide the geometrical information of all children of a cell. In one dimension, a line is typically refined into two equidistant lines, so that a refinement routine in `LineTag` reads

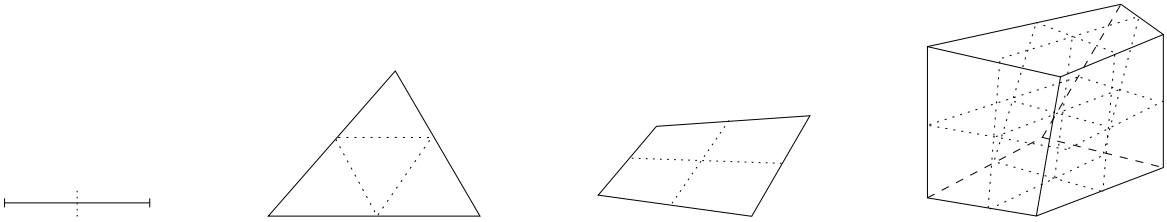
```
1  struct LineTag
2  {
3    //many other members here
```

```
4
5   template <typename VertexType,
6             typename ChildrenList,
7             typename SegmentType>
8   static void refineUniformly(VertexType ** vertices,
9                               ChildrenList & children,
10                              long & cell_id,
11                              SegmentType & seg)
12  {
13    typedef typename SegmentType::Configuration          DomainConfig;
14    typedef typename DomainTypes<DomainConfig>::CellType CellType;
15    typedef typename CellType::ElementTag                CellTag;
16
17    //set up new vertices:
18    VertexType * newvertices[3];
19    VertexType vertex;
20
21    newvertices[0] = seg.template addElement<0>(0, *(vertices[0]));
22    vertex.getPoint() = (vertices[0]->getPoint() + vertices[1]->getPoint())
          ;
23    vertex.getPoint() /= 2.0;
24    newvertices[1] = seg.template addElement<0>(0, vertex);
25    newvertices[2] = seg.template addElement<0>(0, *(vertices[1]));
26
27    //set up new children of the cell:
28    CellType cell;
29    VertexType * cellvertices[2];
30
31    cellvertices[0] = newvertices[0];
32    cellvertices[1] = newvertices[1];
33    cell.setVertices(&(cellvertices[0]));
34    cell.setID(cell_id);
35    children.push_back(
36               seg.template addElement<CellTag::TopoLevel>(cell_id, cell) );
37    ++cell_id;
38
39    //similar for second child
40  }
41 };
```

The arguments of this function are the (pointers to) vertices of the parent cell, a list where the new children are stored in, the cell ID for the first child (consequent children of a cell posses ID in ascending order) and the child segment.

For a triangle, a uniform refinement into triangles with similar shape is possible, while a tetrahedron does not allow for such a decomposition. After intersection of the six edges of a tetrahedron, four tetrahedra of similar shape occur in the parent tetrahedron's corners. However, the interior is a octahedron that can be split into tetrahedra by using one of the three diagonals as a common edge, but the resulting tetrahedra are not similar to the parent tetrahedron. The current strategy is to pick the shortest of these three diagonals in order to maintain a certain degree of shape regularity [8] (cf. Fig. 6.5).

After hiding the shape-dependent implementation of the refinement procedure behind the element tag, we are able to finish the implementation of `multigrid_layer` from the previous section:



(a) Line. (b) Triangle. (c) Quadrilateral. (d) Hexahedron.

Figure 6.4: Uniform refinement of several geometries.

```
template <typename T_Configuration, typename ElTag>
class multigrid_layer<T_Configuration, ElTag, FullMultigridTag>
  : public dt_dx_handler < T_Configuration, ElTag >
{
    typedef dt_dx_handler < T_Configuration, ElTag >  Base;
    typedef element<T_Configuration, ElTag>           ElementType;

  public:
    template <typename SegmentType>
    void refineUniformly(long & cell_id, SegmentType & seg)
    {
      //delegate the work to ElementTag:
      ElTag::refineUniformly( &(Base::vertices_[0]), children, cell_id, seg
          );
    }

    //other members and member functions as before
};
```

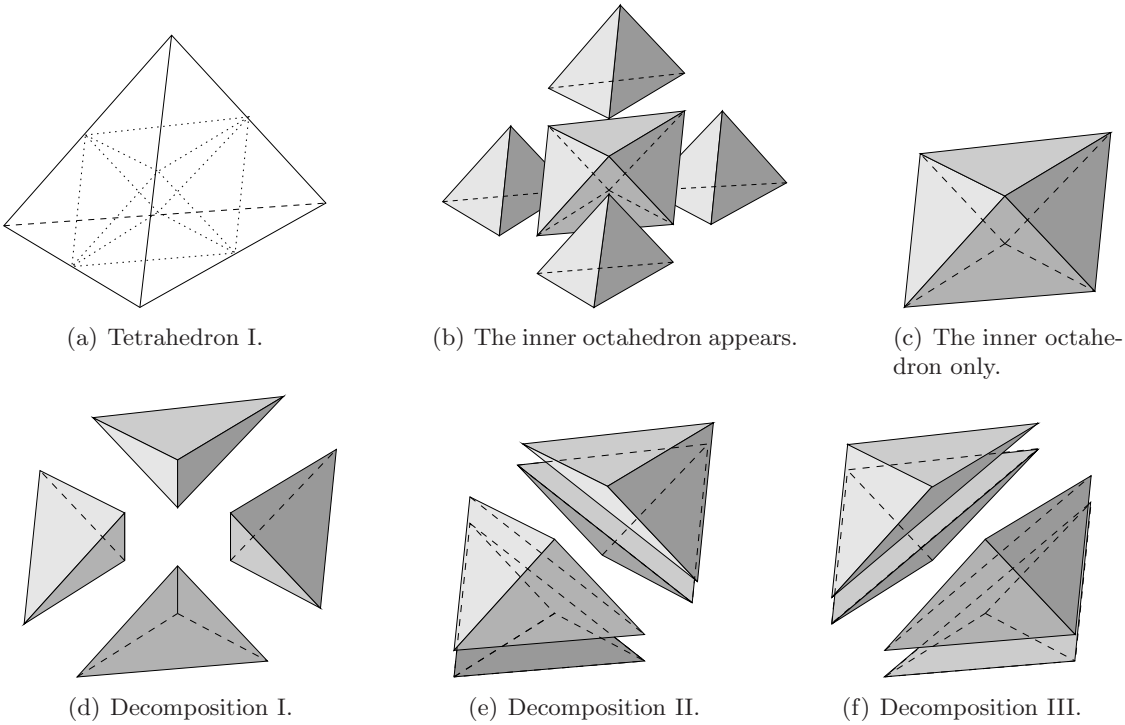
The refinement interface `refineUniformly()` for a segment consists of an iteration over all cells on the finest level with a call to `refineUniformly()` for all cells. The code is straight-forward, so there is no need for a closer inspection.

We are not going into further details of non-uniform refinement here, since this topic could fill a master's thesis on its own. However, with the abstraction of a separate multigrid-layer and element tags, all refinement strategies working on a per-cell basis can be integrated seamlessly and many different refinement strategies can be employed.

Finally, one additional detail should be mentioned: When reading a mesh from an external mesh file, vertices are read one after another and no duplicates occur. However, in a refinement is carried out by refining each cell separately, the refinement routines call the vertex insertion routines of a segment several times. Therefore, extra care has to be taken to ignore duplicates at segment setup during refinement.

6.4 Quantity Management for Multigrid

The general quantity management available at the starting point of this thesis (cf. Chapter 1) also supports multigrid-enriched domain handling. However, for maximum run time efficiency, several improvements have been introduced in Section 2.4, which do not naturally extend to a multigrid domain handling anymore. In particular, mapping numbers of global basis functions are stored in an array instead of a map, if the underlying element provides an ID. However, if a segment is refined, the number of elements at every topological level grows. This forces `QuantityManager` to resize the underlying



(a) Tetrahedron I. (b) The inner octahedron appears. (c) The inner octahedron only.

(d) Decomposition I. (e) Decomposition II. (f) Decomposition III.

Figure 6.5: A uniform refinement of a tetrahedron is not possible. The inner octahedron can be split into four tetrahedra in three ways, where taking the shortest diagonal as common edge is usually the best choice.

vector holding the data, typically resulting in the need for copying a lot of data to the new data array during this setup period.

For a uniform refinement, only vertices of the coarse mesh show up in the refined segment as well, while higher level domain elements from the refined segment are different from those of the coarse level. This lead to the decision of providing a `QuantityManager` for each segment, i.e. an element's ID is not globally unique, but unique within the segment it is located in. However, when accessing data for a particular element, the data set for the element's segment has to be used, otherwise there is an undefined behaviour of the framework. However, storing a link for each element to the segment it is located in is a waste of memory. Similarly, providing the segment as another argument of `storeQuantity` and `retrieveQuantity` is not at all neither desired nor comprehensible.

Let us think about a segment-wise implementation of `QuantityManager` first: The address of the current segment has to be stored in a place accessible for all objects. For this we use a class holding the address of the current segment as a member variable only:

```
1  template <typename DomainConfiguration>
2  struct CurrentSegmentHolder
3  {
4    typedef segment<DomainConfiguration>                SegmentType;
5
6    static SegmentType * p_seg;
7  };
8
9  template <typename DomainConfiguration>
10 segment<DomainConfiguration> * CurrentSegmentHolder<DomainConfiguration>::
     p_seg = 0;
```

This implementation is not thread-safe at present, but after an enumeration of all worker threads, one can remove this restriction by the use of an dispatcher to find the current segment for each thread.

A public member function in `QuantityManager` sets the current segment:

```
template <typename DomainConfiguration, typename ElementTag>
class QuantityManager : public ElementTag::IDHandler
{
  public:
    void setCurrentSegment(SegmentType & segment)
    {
      CurrentSegmentHolder<DomainConfiguration>::p_seg = &segment;
    }
    //other members untouched
};
```

The appropriate data set for a particular element and a specific key type can then be obtained from another class:

```
template <typename DomainConfiguration, typename ElementTag,
          typename KeyType, typename T>
class QuantityManagerSelector
{
  public:
    //type definition for QuanManType here

    QuanManType & getQuantityManager(SegmentType * segment)
    {
      return segment_to_qm[segment];
    }

  private:
    std::map<SegmentType *, QuanManType> segment_to_qm;
};
```

Now, the necessary changes in the implementation of `QuantityManager` in order to use this new storage scheme is a modification of its private member function `getManager`:

```
template <typename DomainConfiguration, typename ElementTag>
class QuantityManager : public ElementTag::IDHandler
{
    template <typename T, typename KeyType>
    typename QuantityManagerSelector<DomainConfiguration,
                                     ElementTag,
                                     KeyType,
                                     T>::QuanManType &
    getManager() const
    {
      static QuantityManagerSelector<DomainConfiguration,
                                     ElementTag, KeyType, T>     qms;

      return qms.getQuantityManager(
                      CurrentSegmentHolder<DomainConfiguration>::p_seg);
    };

    //other member functions untouched
};
```

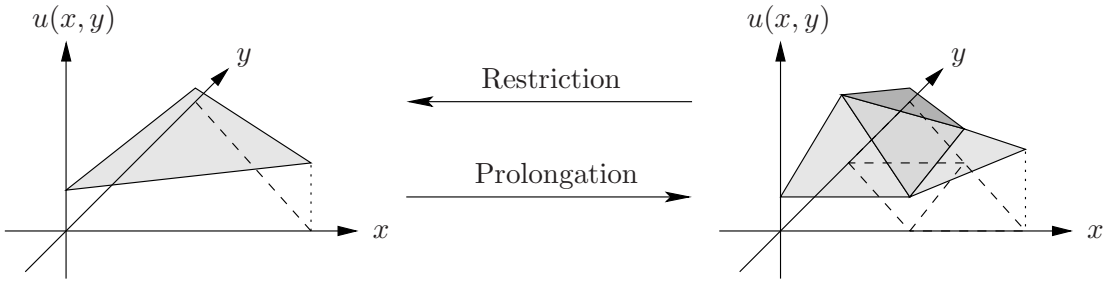


Figure 6.6: Restriction and prolongation of a piecewise linear function defined over a coarse (left) and over a refined (right) triangle.

Recall that `QuantityManagerSelector` is unique for each element type, key type and value type, while `CurrentSegmentHolder` is globally unique, therefore they cannot be merged into a single class.

The price we have to pay for the introduction of this per-segment operation of `QuantityManager` is that we have to set the current segment prior to data storage or retrieval. Since the implementation of FEM is designed to work on a per-segment basis anyway, this does not pose any restrictions. There is, however, one case where some more care has to be taken: The coupling of segments using interface conditions. In this case the segment pointers for the coupled segment are stored on the interface elements so that a switching between the segments at some point deep inside the implementation is possible without changing any function parameters of existing implementations (cf. Section 3.5).

6.5 Implementation of Transfer Operators

Multigrid methods require the translation of functions between nested meshes. Such a transfer is typically done locally on one cell or on a localized cell patch, because global operations (like a projection onto the smaller function space) are usually too costly.

We aim at a construction of transfer operators that work with basis functions of arbitrary degree. However, for unstructured meshes it is much harder to obtain good transfer operators than for structured meshes, where a lot of geometric regularity and information is implicitly available.

6.5.1 The Restriction Operator

Let us motivate the procedure in two dimensions on the reference triangle $\mathbf{T}^{\text{ref}}$: As for restriction, we have to transfer a function given on the children of $\mathbf{T}^{\text{ref}}$ (Fig. 6.6). We do not restrict ourselves to uniform refinement, therefore we seek for a procedure that also works for non-uniform refinement. Furthermore, the method should be flexible in a way that it can be reused for higher order basis functions and higher spatial dimensions. We consider a common type of transfer operators, namely interpolation of the function given on the fine grid by a function defined on the coarse grid. To do so, we have to determine one interpolation point for every basis function. The choice could be arbitrary, but it is advantageous to select the interpolation point in such a way that it accounts for the specific shape of the associated basis function. For linear basis functions we therefore select the corners of the cell for the interpolation points. Quadratic basis functions consequently have one interpolation point on each vertex and one interpolation point on each edge. Cubic basis functions on a triangle have one interpolation point on each vertex and on the barycenter (to account for one bubble function), and two interpolation points on each edge. Thinking back to compressed basis functions in Section 4.5, the idea for an implementation is to reuse this facility for the construction of interpolation points. For a given compressed basis function, we can therefore write

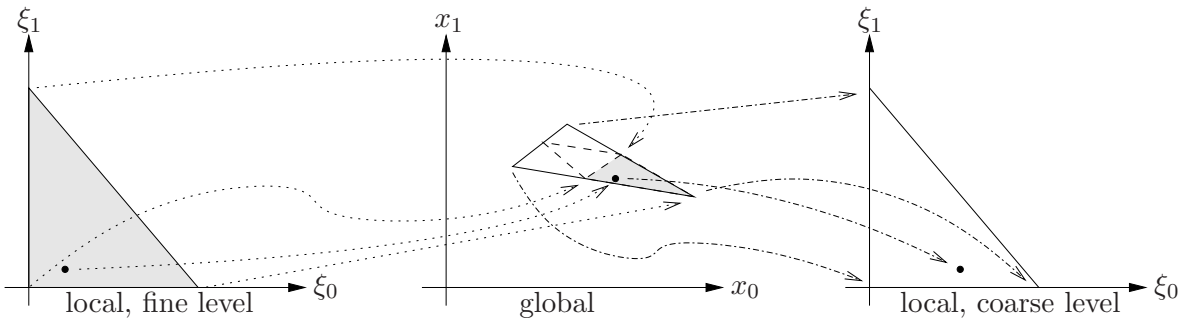


Figure 6.7: Mapping of an interpolation point from the fine to the coarse level.

```
template <typename PointType, typename CompressedBfID>
struct makePointFromBfID;

//construct interpolation point on the reference interval:
template <typename PointType, long a0, long a1>
struct makePointFromBfID<PointType, compressed_bf_2<a0, a1> >
{
  typedef compressed_bf_2<a0, a1>           CompressedBfType;

  static void apply(std::vector<PointType> & refpoints, long index)
  {
    refpoints[index] =
      PointType( a1 / static_cast<double>(CompressedBfType::Degree) );
  }
};

//interpolation point for the reference triangle:
template <typename PointType, long a0, long a1, long a2>
struct makePointFromBfID<PointType, compressed_bf_3<a0, a1, a2> >
{
  typedef compressed_bf_3<a0, a1, a2>           CompressedBfType;

  static void apply(std::vector<PointType> & refpoints, long index)
  {
    refpoints[index] =
        PointType( a1 / static_cast<double>(CompressedBfType::Degree),
                   a2 / static_cast<double>(CompressedBfType::Degree) );
  }
};

//similarly for longer compressed_bf-types
```

By iteration over all basis functions using a `BasisFunctionIterator`, a list of all interpolation points can be obtained. After that, the interpolation matrix $\boldsymbol{A} = (a_{ij})_{i,j=1}^{m}$ with $a_{ij} = \varphi_j(x_i)$ can be set up, where x_i is the i-th interpolation point and φ_j is the j-th basis function defined on the element. The right-hand side of this system of equations is made up from the evaluations $d_{k+1}(x_i)$ of the fine-grid function at the interpolation points. The coefficient vector $\boldsymbol{s}$ for the coarse grid is then obtained from the solution of this system.

The evaluations $d_{k+1}(x_i)$ have to be carried out in a way that the type of refinement does not matter. Therefore, the reference point x_i is mapped back to the location $\tilde{x}_i$ within the untransformed

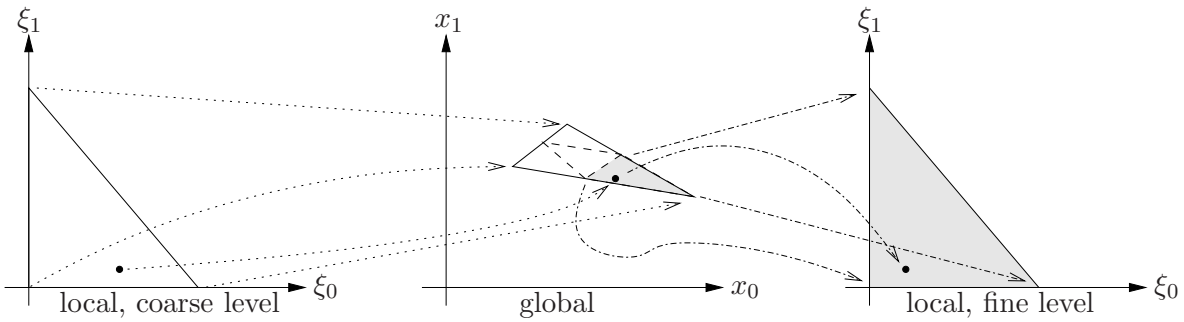


Figure 6.8: Mapping of an interpolation point from the coarse to the fine level.

cell. Then, in an iteration over all children of the cell, it is checked whether $\tilde{x}_i$ lies in the interior or on the boundary of each child. If so, $\tilde{x}_i$ is transformed to local coordinates of the child and is then locally evaluated there. This way any decomposition of the cell into smaller cells can be handled, although there is a run time penalty for uniformly refined meshes because additional information is not used. Nevertheless, the solution of the resulting system of interpolation equations typically requires most of the overall computational effort, so that a double transformation of reference points can be justified.

6.5.2 The Prolongation Operator

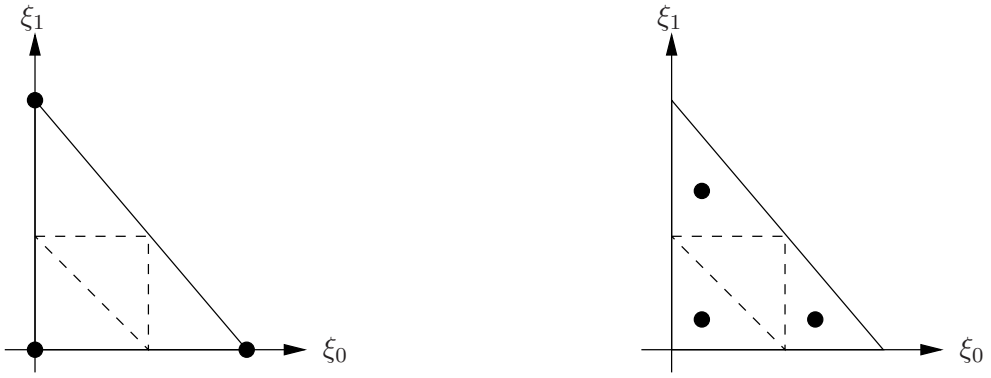
The transfer of a function defined on the coarse grid to a function defined on the fine grid can also be done by means of interpolation points. In fact, the whole process is dual to the restriction operator: This time, we map the reference point given by in the local coordinates of a child to local coordinates of the coarse cell by going the long way round via global coordinates. For reasons of this duality, the prolongation operator is often chosen to be the transpose of the restriction operator [9].

6.5.3 Modifications for Linear Basis Functions

A restriction operator constructed from interpolation points does not lead to satisfactory results in case of linear basis functions. As mentioned by Hackbusch [18], the defect vanishes in the vertices of the coarse grid, provided some prerequisites are met. Therefore, the multigrid-correction reduces to adding the zero-vector only, rendering any multigrid support useless. The situation can be improved if averaging around each vertex is performed. This can be done in several ways: One can average over all vertices on the fine grid that are connected to the vertex of interest. This requires additional topological information that is not used otherwise, therefore we choose different interpolation points: We shift the interpolation points half way towards the barycenter of the simplex in both two and three dimensions. In higher dimensions, a similar modification can be applied.

6.6 A Full Multigrid Solver

The multigrid cycles presented in Section 6.1 can be used as an iterative solver and like all of their kind, they start with an initial guess of the solution. It is possible to use a random function as initial guess on the finest level, but a much better choice is to solve the problem on the coarsest level (where a solution is computationally cheap) and prolongate the obtained solution to the finest level. Each prolongation step, however, is lossy in a sense that the defect increases. This motivates a multigrid cycle after each prolongation step on intermediate levels, leading to so-called *Full Multigrid*, whose structure is depicted in Fig. 6.10.



(a) Interpolation points obtained from the basis function ID.

(b) Modified interpolation points.

Figure 6.9: For piecewise linear basis functions it is of advantage to use interpolations points located closer to the barycenter for the restriction operator.

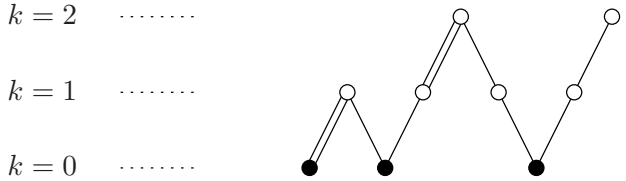


Figure 6.10: Structure of full multigrid for three levels with one correction step after each prolongation. The doubled lines denote a prolongation of the *solution*, while the single lines denote restriction of the defect and prolongation of the correction respectively. The filled circles denote a direct solution, while open circles denote smoothing operations.

A full multigrid solver does not solve the discretised problem with full accuracy. Instead, it is considered to be sufficient to solve the discretised problem up to discretisation accuracy. For the true solution u, the discretised solution u_h and the full multigrid solution $u_{h,\mathrm{FMG}}$, we require

$$\|u_h - u_{h,\mathrm{FMG}}\| \leq \beta \|u - u_h\| \ , \tag{6.14}$$

where we consider $\beta \approx 1$ to be a sufficiently good value. In such a case, we obtain

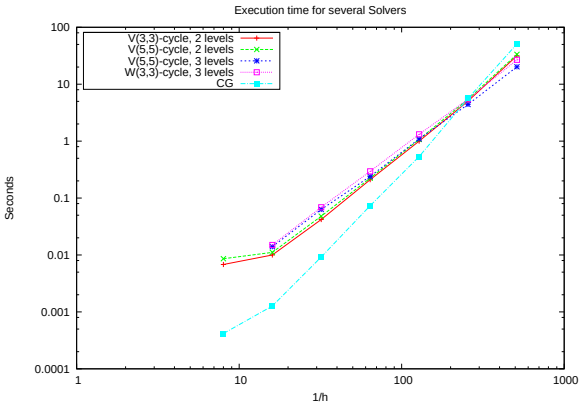
$$\|u - u_{h,\mathrm{FMG}}\| \leq \|u - u_h\| \ + \|u_h - u_{h,\mathrm{FMG}}\| \leq (1 + \beta)\|u - u_h\| \ . \tag{6.15}$$

Thus, for $\beta \approx 1$ we obtain a solution with an error comparable to the discretisation error, while the computational costs grow linearly with the problem size only. Instead of spending a lot of effort on finding a more accurate solution of the discretised problem, where we have to live with the discretisation error anyway, one better steps to a finer level and solves the problem there (again up to discretisation accuracy only).

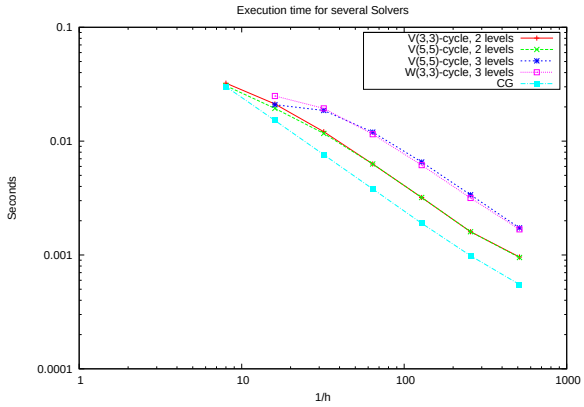
There are several parameters for a full multigrid solver, we summarise them briefly:

- The number r of multigrid cycles after each prolongation of the solution is typically one or two, but can also depend on the current level k.
- The multigrid cycle parameter γ can depend on k and for full multigrid also on r.
- The number of smoothing steps can also be chosen in dependence of k, r and γ.

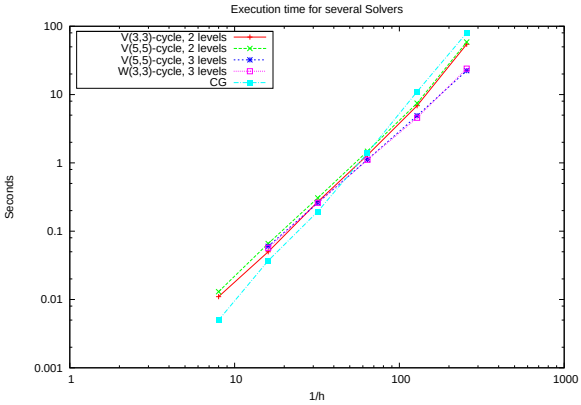
This high number of free parameters can be controlled by so-called *self controlling algorithms*, but we do not go into further details here.



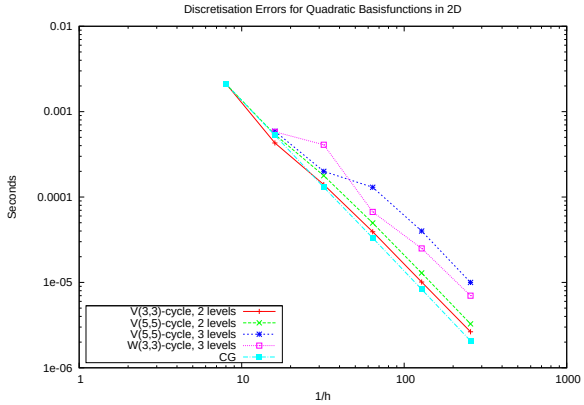
(a) Linear basis functions in 2D: Execution times.



(b) Linear basis functions in 2D: Errors in H^1.



(c) Quadratic basis functions in 2D: Execution times.



(d) Quadratic basis functions in 2D: Errors in H^1.

Figure 6.11: Run time comparison for linear and quadratic basis functions in two dimensions in (a) and (c), while the errors in the H^1-norm are given in (b) and (d). After each prolongation step, one multigrid-cycle was applied ($r = 1$). At finest level, a second multigrid-cycle was carried out additionally.

6.7 Performance of Full Multigrid

The full multigrid solver presented in the previous section is applied to the test case

$$-\Delta u = f \quad \text{on } \Omega = [0,1]^2 \tag{6.16}$$

$$u|_{\partial\Omega} = 0\ , \tag{6.17}$$

where f is chosen such that the $u = x(1-x)y(1-y)$. In three dimensions, the boundary conditions for $z = 0$ and $z = 1$ are of homogeneous Neumann type, so that the solution remains the same. The H^1-norm of the solution is (after same calculations) found as $1/45$. Since $u \in C^\infty(\Omega)$, the convergence rate in the H^1-norm is equal to the polynomial degree of the selected basis functions.

At present, the code for the assembly of matrices and right-hand side vectors has to be provided for each level by hand. In the future, all top level functions can be applied recursively to each child segment. For example, the single-grid function `assemble<...>(...)` is called recursively by `multigrid_assemble <...>(...)`, so that the number of code lines from the end-user's point of view remains (almost) unchanged no matter how many grid are actually used for the solution of the problem.

Full multigrid solvers are compared to a conjugate gradient (CG) solver without preconditioning. We are mainly interested in the asymptotic behaviour, therefore the absolute execution times are of

minor interest. A preconditioned CG-solver would benefit from (usually) shorter execution times, while on the other specialisations of transfer operators for uniform refinement would accelerate full multigrid solvers, leading to the same qualitative results. The termination of the CG-solver is such that the discrete solution on the finest level is sufficiently accurate, but not of higher accuracy than actually needed.

Unlike for structured grids, a Gauss-Seidel-smoother has poorer smoothing properties on unstructured grids. To account for this reduced smoothing, several smoothing steps are performed: In two dimensions, we use three and five pre- and post-smoothing steps, while in three dimensions their number is five and eight.

The results for the two-dimensional problem are shown in Fig. 6.11. Looking at the execution time only, one can clearly see that the full multigrid solver asymptotically outperforms the standard CG-solver. However, the convergence in the H^1-norm is not very satisfactory for linear basis functions, which is most likely because of the simple structure of the prolongation operator. This can be seen from the errors in H^1, where the solutions are comparable to the discretisation error of the coarsest grids for both two and three grid solvers. Consequently, a more complex prolongation operator is required, that also takes discontinuities of the first derivatives between triangles into account.

Full multigrid works quite well for quadratic basis functions in two dimensions. The additional degrees of freedom on the edges are able to track the defect after restriction much better than linear basis functions do. Deviations from an asymptotic optimality of execution times is due to the fact that the coarsest level is not fixed and execution time for the CG solver at the coarsest level thus become dominant especially in the case of two grids. While the errors of two grid cycles follow (6.16) with $\beta \approx 1$, three grid cycles only yield $\beta \approx 3$, so that the error is of similar magnitude than the true discretisation error of the intermediate grid. Most likely, a second multigrid cycle after each prolongation step eliminates this deterioration.

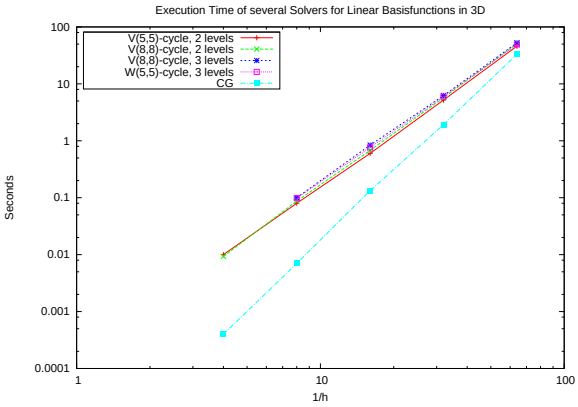
On the whole, the full multigrid solver made up from components presented in this chapter works well with quadratic basis functions, while linear basis functions do not yield satisfactory results due to the simple structure of the transfer operators.

In three dimensions, similar results are obtained and illustrated in Fig. 6.12: Again, linear basis functions do not show a satisfactory convergence and mainly keep the discretisation error of the coarsest grid. Again, a more complex prolongation operator is likely to yield better results.

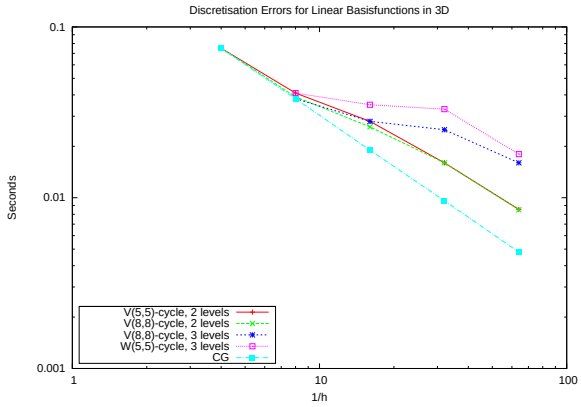
Quadratic basis functions in three dimensions lead to slightly better execution times compared to a CG solver. For a two-grid solver, the errors of the computed (discrete) solution fully match up with the true discretisation error. It is interesting to note that $V(5,5)$-cycle appears to find a "better" approximation. However, this is only due to round-off errors and approximations in the computation of the errors and should not lead to confusion: The conclusion is that the $V(5,5)$-cycle leads to numerical solutions of similar accuracy as the solution obtained by a CG solver.

It can be observed that for cubic basis functions around 20 pre- and post-smoothing steps are needed to obtain a convergence of multigrid cycles. Presumably, the reason is that the condition number of the global matrix is higher than for the linear and quadratic case. This is backed up by the observation that the condition number turns up in convergence estimates for Jacobi- and Gauss-Seidel-methods, thus leading to slower convergence and poorer smoothing properties. This confirms the recommendation to use algebraic multigrid methods for higher order finite element methods [29].

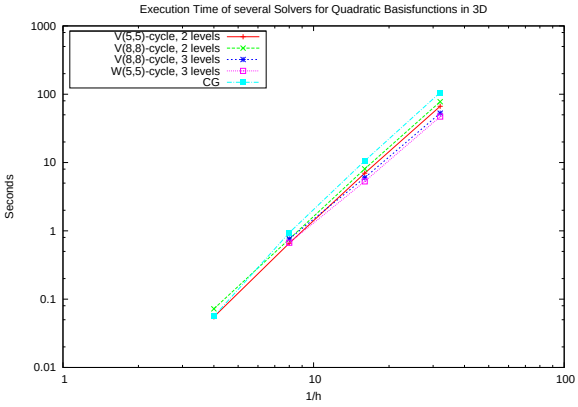
The conclusion for this chapter is that (geometric) multigrid methods do have potential for quadratic basis functions on unstructured grids. For linear basis functions, more complex prolongation operators than those presented here are needed. Basis functions of higher degree do not allow for effective smoothing operations, therefore algebraic multigrid methods should then be used. Nevertheless, the rationale for the introduction of multigrid capabilities in the presented FEM framework is the future support of adaptive schemes, where transfer operations between meshes are needed as well.



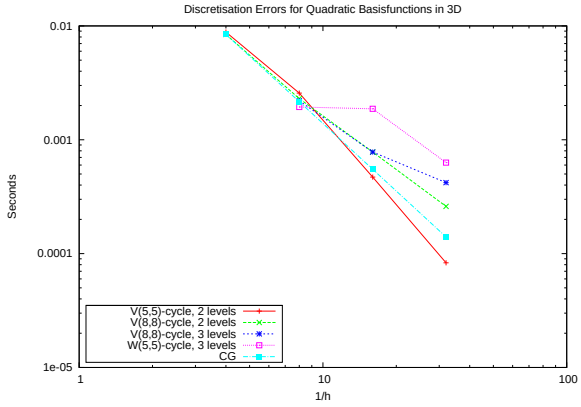
(a) Linear basis functions in 3D: Execution times.



(b) Linear basis functions in 3D: Errors in H^1.



(c) Quadratic basis functions in 3D: Execution times.



(d) Quadratic basis functions in 3D: Errors in H^1.

Figure 6.12: Run time comparison for linear and quadratic basis functions in three dimensions in (a) and (c), while the errors in the H^1-norm are given in (b) and (d). After each prolongation step, one multigrid-cycle was applied ($r = 1$). At finest level, a second multigrid-cycle was carried out additionally.

Chapter 7

Results for Selected Applications

A generic multi-physics framework is of no use if it cannot be applied to realistic problems with reasonable effort. For this reason, we avoid standard examples like the Laplace equation on the unit square or unit cube. Instead, we consider three applications that are deduced from questions arising in practice within this chapter. All three examples originate from microelectronics, the first two from the modelling of electromigration, the third one from the design of one of the basic MEMS structures, a cantilever.

7.1 The Segregation Model applied to a Diffusive Hourglass

Consider two segments Ω_1 and Ω_2 that are connected by a common boundary Γ. In each of these segments a particle concentration $u_{i,0}$, $i = 1, 2$ is given. We assume that the connection between these two segments is not ideal: The exchange of particles depends on the concentration difference in the vicinity of the interface and is not symmetric: Particles from, say Ω_1, do not flow as easily to Ω_2 as particles from Ω_2 flow to Ω_1 (cf. Fig. 7.1). Such a setting is common in the modelling of segregation on material interfaces.

In more mathematical terms, in the interior of each of the segments Ω_1 and Ω_2 we assume the diffusion equation to hold:

$$\frac{\partial u_i}{\partial t} = -\nabla \cdot \boldsymbol{q}_i \ , \quad i = 1, 2 \ , \tag{7.1}$$

with

$$\boldsymbol{q}_i = -D_i \nabla u \ , \quad i = 1, 2 \ , \tag{7.2}$$

where u_i denotes the particle concentration in Ω_i with scalar diffusion coefficient $D_i > 0$. At the interface the total species flux J between Ω_1 and Ω_2 (normal to the interface) is given by [22]

$$J = \boldsymbol{q}_1 \cdot \boldsymbol{n}_1 = -\boldsymbol{q}_2 \cdot \boldsymbol{n}_2 = h\,(u_1 - m u_2) \tag{7.3}$$

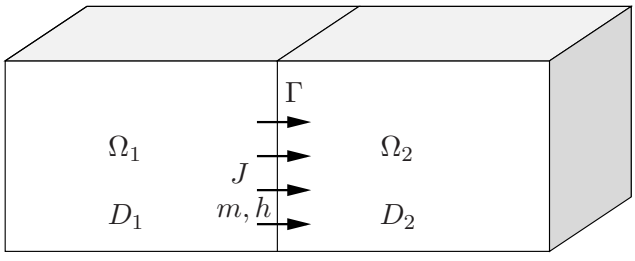


Figure 7.1: Setting for the segregation model.

where h is the *transport coefficient*, m is the *segregation coefficient* and $\boldsymbol{n}_i$ is the outer normal vector on Ω_i, $i = 1, 2$. For the following considerations it is sufficient to interpret (7.3) only in a weak sense.

An integral formulation of (7.1) together with (7.3) for zero outflux on $\partial\Omega_1 \setminus \Gamma$ and $\partial\Omega_2 \setminus \Gamma$ is

$$\int_{\Omega_1} \frac{\partial u_1}{\partial t} v_1 + D_1 \nabla u_1 \nabla v_1 \,\mathrm{d}\boldsymbol{x} + h \int_\Gamma (u_1 - m u_2)\, v_1 \,\mathrm{d}A = 0 \quad \forall v_1 \in H^1(\Omega_1) \;, \tag{7.4}$$

$$\int_{\Omega_2} \frac{\partial u_2}{\partial t} v_2 + D_2 \nabla u_2 \nabla v_2 \,\mathrm{d}\boldsymbol{x} - h \int_\Gamma (u_1 - m u_2)\, v_2 \,\mathrm{d}A = 0 \quad \forall v_2 \in H^1(\Omega_2) \;, \tag{7.5}$$

where we seek for a solution (u_1, u_2) in an appropriate solution space. Writing $u = (u_1, u_2)$, $v = (v_1, v_2)$ with $u, v \in V := H^1(\Omega_1) \times H^1(\Omega_2)$, we define the bilinear form

$$a(\cdot,\cdot) : \left\{ \begin{array}{rcl} V \times V & \to & \mathbb{R}, \\ (u, v) & \mapsto & D_1 \int_{\Omega_1} \nabla u_1 \nabla v_1 \,\mathrm{d}\boldsymbol{x} + D_2 \int_{\Omega_2} \nabla u_2 \nabla v_2 \,\mathrm{d}\boldsymbol{x} \\ & & \quad + h \int_\Gamma \big(u_1 - m u_2\big)\big(v_1 - v_2\big) \,\mathrm{d}A \end{array} \right. \tag{7.6}$$

and equip V with the norm

$$\|u\|_V := \|u_1\|_{H^1(\Omega_1)} + \|u_2\|_{H^1(\Omega_2)} \;. \tag{7.7}$$

The dual space of V is $V^* = H^{-1}(\Omega_1) \times H^{-1}(\Omega_2)$. We denote $W := L^2(\Omega_1) \times L^2(\Omega_2)$ and observe that $V \subset W \subset V^*$ is an evolution triple, since this is true for each component of the product spaces.

With this mathematical foundation we are able to state and prove the following theorem:

Theorem 4. *For $u(0) = u_0 \in V$ there exists a unique solution $u \in C([0, T], W)$ of (7.1) with flux term (7.3) on Γ and homogeneous Neumann boundary conditions on $\Omega_1 \setminus \Gamma$ and $\Omega_2 \setminus \Gamma$.*

Proof. We have to prove that the bilinear form $a(\cdot,\cdot)$ is bounded and coercive, then the existence and uniqueness follows from a (sufficiently general) standard result for linear parabolic partial differential equations.

To show that $a(u, v)$ is bounded, we apply the Cauchy-Schwarz inequality to the integrals over Ω_1 and Ω_2:

$$\begin{aligned} |a(u, v)| \le{}& \|u_1\|_{H^1(\Omega_1)} \|v_1\|_{H^1(\Omega_1)} + \|u_2\|_{H^1(\Omega_1)} \|v_2\|_{H^1(\Omega_2)} \\ & + h \int_\Gamma |u_1 v_1| + m|u_2 v_1| + |u_1 v_2| + m|u_2 v_2| \,\mathrm{d}A \;. \end{aligned}$$

Next, we apply Young's inequality to the integrands of the boundary integral and obtain

$$\begin{aligned} |a(u, v)| \le{}& \|u_1\|_{H^1(\Omega_1)} \|v_1\|_{H^1(\Omega_1)} + \|u_2\|_{H^1(\Omega_2)} \|v_2\|_{H^1(\Omega_2)} \\ & + C_1 \left[\|u_1\|^2_{L^2(\Gamma)} + \|v_1\|^2_{L^2(\Gamma)} + \|u_2\|^2_{L^2(\Gamma)} + \|v_2\|^2_{L^2(\Gamma)} \right] \end{aligned}$$

with a constant $C_1 = C_1(h, m)$. Application of the trace theorem for u_1, v_1 with respect to $H^1(\Omega_1)$ and for u_2, v_2 with respect to $H^2(\Omega_2)$ yields

$$\begin{aligned} |a(u, v)| \le{}& \|u_1\|_{H^1(\Omega_1)} \|v_1\|_{H^1(\Omega_1)} + \|u_2\|_{H^1(\Omega_2)} \|v_2\|_{H^1(\Omega_2)} \\ & + C_2 \left[\|u_1\|^2_{H^1(\Omega_1)} + \|v_1\|^2_{H^1(\Omega_1)} + \|u_2\|^2_{H^1(\Omega_2)} + \|v_2\|^2_{H^1(\Omega_2)} \right] \end{aligned}$$

for a constant C_2 that also takes the norms of the trace operators into account. Since $\|\cdot\|_{H^1(\Omega_i)} \le \|\cdot\|_V$ for $i = 1, 2$, we conclude that $a(u, v)$ is bounded.

To show coercivity of the bilinear form $a(\cdot,\cdot)$, i.e. $a(u,u) \geq \alpha\|u\|_V - \beta\|u\|_W$ with $\alpha > 0$, we have to be careful with the terms carrying a minus in front. In the following we use $|u_i|_{H^1(\Omega_i)} = \int_{\Omega_i} (\nabla u_i)^2 \,\mathrm{d}x$ for the semi-norm on $H^1(\Omega_i)$, $i = 1, 2$:

$$a(u,u) = D_1|u_1|^2_{H^1(\Omega_1)} + D_2|u_2|^2_{H^1(\Omega_2)} + h\int_\Gamma u_1^2 \,\mathrm{d}A + h\int_\Gamma u_2^2 \,\mathrm{d}A - mh\int_\Gamma u_1 u_2 \,\mathrm{d}A \;.$$

Unfortunately we do not have any information for the sign of u_1u_2 available, therefore $\int_\Gamma u_1u_2 \,\mathrm{d}A$ can also be negative. With Young's inequality we find

$$\begin{aligned} a(u,u) &\geq D_1|u_1|^2_{H^1(\Omega_1)} + D_2|u_2|^2_{H^1(\Omega_2)} + h\|u_1\|^2_{L^2(\Gamma)} + h\|u_2\|^2_{L^2(\Gamma)} \\ &\quad - \frac{mh}{2}\left(\|u_1\|^2_{L^2(\Gamma)} + \|u_2\|^2_{L^2(\Gamma)}\right) \\ &= D_1|u_1|^2_{H^1(\Omega_1)} + D_2|u_2|^2_{H^1(\Omega_2)} \\ &\quad - \frac{h}{2}(2m-2)\|u_1\|^2_{L^2(\Gamma)} - \frac{h}{2}(2m-2)\|u_2\|^2_{L^2(\Gamma)} \;. \end{aligned}$$

For brevity, let us introduce $\tilde{h} := \frac{h}{2}(2m-2)$. Using the trace theorem, we can bound L^2-norms on Γ by $H^{1/2+\nu}(\Omega_i)$ norms on Ω_i, $i = 1, 2$, with $\nu > 0$ on the appropriate domain:

$$a(u,u) \geq D_1|u_1|^2_{H^1(\Omega_1)} + D_2|u_2|^2_{H^1(\Omega_2)} - \tilde{h}C_1\|u_1\|^2_{H^{1/2+\nu}(\Omega_1)} - \tilde{h}C_2\|u_2\|^2_{H^{1/2+\nu}(\Omega_2)} \;,$$

where C_1 and C_2 are the squared norms of the trace operator over Ω_1 and Ω_2 respectively. The norms in $H^{1/2+\nu}$ are now estimated using an interpolation inequality for Sobolev spaces [2]. We define $\theta := 1/2 + \nu$, select ν sufficiently small and make use of

$$\|u_i\|_{H^\theta(\Omega_1)} \leq K_i\|u_i\|^\theta_{H^1(\Omega_i)}\|u_i\|^{1-\theta}_{L^2(\Omega_i)} \;, \quad \text{for } i = 1,2 \text{ and } 0 < \theta < 1 \;.$$

This leads to

$$a(u,u) \geq D_1|u_1|^2_{H^1(\Omega_1)} + D_2|u_2|^2_{H^1(\Omega_2)} - \tilde{h}C_1K_1^2\|u_1\|^{2\theta}_{H^1(\Omega_1)}\|u_1\|^{2-2\theta}_{L^2(\Omega_1)} - \tilde{h}C_2K_2^2\|u_2\|^{2\theta}_{H^1(\Omega_2)}\|u_2\|^{2-2\theta}_{L^2(\Omega_2)}$$

and application of Young's inequality with $p = 1/\theta$ and $q = 1/(1-\theta)$ yields

$$\begin{aligned} a(u,u) &\geq D_1|u_1|^2_{H^1(\Omega_1)} + D_2|u_2|^2_{H^1(\Omega_2)} \\ &\quad - \tilde{h}C_1K_1^2\left(\frac{\varepsilon\|u_1\|^2_{H^1(\Omega_1)}}{1/\theta} + \frac{\|u_1\|^2_{L^2(\Omega_1)}}{\varepsilon^{\theta/(1-\theta)}/(1-\theta)}\right) - \tilde{h}C_2K_2^2\left(\frac{\|u_2\|^2_{H^1(\Omega_2)}}{1/\theta} + \frac{\|u_2\|^2_{L^2(\Omega_2)}}{1/(1-\theta)}\right) \;, \end{aligned}$$

with a free parameter $\varepsilon > 0$. Using the definition of the H^1-norm, $\|\cdot\|^2_{H^1(\Omega_i)} = |\cdot|^2_{H^1(\Omega_i)} + \|\cdot\|^2_{L^2(\Omega_i)}$ for $i = 1, 2$, we can collect terms:

$$\begin{aligned} a(u,u) &\geq \big(D_1 - \tilde{h}C_1K_1^2\theta\varepsilon\big)|u_1|^2_{H^1(\Omega_1)} + \big(D_2 - \tilde{h}C_2K_2^2\theta\varepsilon\big)|u_2|^2_{H^1(\Omega_2)} \\ &\quad - \tilde{h}C_1K_1^2\big(\theta\varepsilon + (1-\theta)\varepsilon^{-\theta/(1-\theta)}\big)\|u_1\|^2_{L^2(\Omega_1)} \\ &\quad - \tilde{h}C_2K_2^2\big(\theta\varepsilon + (1-\theta)\varepsilon^{-\theta/(1-\theta)}\big)\|u_2\|^2_{L^2(\Omega_2)} \;. \end{aligned}$$

We select $\varepsilon > 0$ sufficiently small such that $\tilde{D}_1 := D_1 - \tilde{h}C_1K_1^2\theta\varepsilon > D_1/2$ and $\tilde{D}_2 := D_2 - \tilde{h}C_2K_2^2\theta\varepsilon > D_2/2$ and use shorthand notations $\tilde{C}_1 := \tilde{h}C_1K_1^2\big(\theta\varepsilon + (1-\theta)\varepsilon^{-\theta/(1-\theta)}\big)$ and $\tilde{C}_2 := \tilde{h}C_2K_2^2\big(\theta\varepsilon + (1-$

$\theta)\varepsilon^{-\theta/(1-\theta)}\big)$ to obtain

$$\begin{aligned}
a(u,u) &\geq \tilde{D}_1|u_1|^2_{H^1(\Omega_1)} + \tilde{D}_2|u_2|^2_{H^1(\Omega_2)} - \tilde{C}_1\|u_1\|^2_{L^2(\Omega_1)} - \tilde{C}_2\|u_2\|^2_{L^2(\Omega_2)} \\
&= \tilde{D}_1|u_1|^2_{H^1(\Omega_1)} + \tilde{D}_1\|u_1\|^2_{L^2(\Omega_1)} + \tilde{D}_2|u_2|^2_{H^1(\Omega_2)} + \tilde{D}_2\|u_2\|^2_{L^2(\Omega_2)} \\
&\quad - \big(\tilde{C}_1 + \tilde{D}_1\big)\|u_1\|^2_{L^2(\Omega_1)} - \big(\tilde{C}_2 + \tilde{D}_2\big)\|u_2\|^2_{L^2(\Omega_2)} \\
&= \tilde{D}_1\|u_1\|^2_{H^1(\Omega_1)} + \tilde{D}_2\|u_2\|^2_{H^1(\Omega_2)} - \big(\tilde{C}_1 + \tilde{D}_1\big)\|u_1\|^2_{L^2(\Omega_1)} - \big(\tilde{C}_2 + \tilde{D}_2\big)\|u_2\|^2_{L^2(\Omega_2)} \\
&\geq \min\big\{\tilde{D}_1, \tilde{D}_2\big\}\|u_1\|_V - \max\big\{\tilde{C}_1 + \tilde{D}_1, \tilde{C}_2 + \tilde{D}_2\big\}\|u_2\|_W \ .
\end{aligned}$$

This proves coercivity of the bilinear form $a(\cdot,\cdot)$.

By a standard-result for parabolic PDEs (cf. Theorem 11.3 in the book of Renardy and Rogers [25], also given in Appendix B) there exists a unique solution $u \in L^2((0,T),V) \cap H^1((0,T),V^*)$ and a subsequent lemma (cf. Renardy and Rogers) finally shows $u \in C([0,T],W)$. □

This theorem can be extended to more general boundary conditions and space dependent coefficients as well as for a coupling of several segments. For example, the given proof also holds for diffusion coefficients that fulfill $0 < D^- \leq D_i(\boldsymbol{x}) \leq D^+$, but we will stop our theoretical investigations at this point.

Let us continue with a physical interpretation of the segregation parameter m: In steady state, there is no net flux through the boundary Γ, thus $u_1 = mu_2 = \text{const.}$ is the solution for homogeneous Neumann boundary conditions on $\Omega_1 \setminus \Gamma$ and $\Omega_2 \setminus \Gamma$. The constant depends on the initial conditions and assures mass conservation over time (which is an implication of the homogeneous Neumann boundary conditions on the outer boundaries).

For the implementation, equations (7.4) and (7.5) translate almost directly into code. The missing time discretisation is done by means of a backward Euler scheme, which leads to the system

$$\int_{\Omega_1} u_1^{n+1}v_1^{n+1} + \Delta t\left[D_1\nabla u_1^{n+1}\nabla v_1^{n+1}\,\mathrm{d}\boldsymbol{x} + h\int_\Gamma \left(u_1^{n+1} - mu_2^{n+1}\right)v_1^{n+1}\,\mathrm{d}A\right] = \int_{\Omega_1} u_1^n v_1\,\mathrm{d}\boldsymbol{x}\ , \tag{7.8}$$

$$\int_{\Omega_2} u_1^{n+1}v_2^{n+1} + \Delta t\left[D_2\nabla u_2^{n+1}\nabla v_2^{n+1}\,\mathrm{d}\boldsymbol{x} - h\int_\Gamma \left(u_1^{n+1} - mu_2^{n+1}\right)v_2^{n+1}\,\mathrm{d}A\right] = \int_{\Omega_2} u_2^n v_2\,\mathrm{d}\boldsymbol{x}\ , \tag{7.9}$$

where the superscripts n and $n+1$ denote the discrete solution at times $n\Delta t$ and $(n+1)\Delta t$ respectively. The mass matrices arising from $\int_{\Omega_i} u_i v_i\,\mathrm{d}\boldsymbol{x}$, $i = 1,2$ for the time discretisation are set up in the usual way (using the default integration rule):

```
1 //assembling of mass matrix on Omega_1:
2 assemble<FEMConfig>( segment1, mass_matrix, rhs,
3     integral< Omega > ( basisfun<1>() * basisfun<2>() )
4     = ScalarExpression<0>() );
5
6 //assembling of mass matrix on Omega_2:
7 assemble<FEMConfig>( segment2, mass_matrix, rhs,
8     integral< Omega > ( basisfun<1>() * basisfun<2>() )
9     = ScalarExpression<0>() );
```

The assembly requires to set up the interface first. Assuming that it is located at $x = 0$, this can be done as

```
//for Gamma<1>:
setBoundaryArc<FEMConfig, 1>(seg1, (x_ == 0.0));
setBoundaryArc<FEMConfig, 1>(seg2, (x_ == 0.0));

//for Interface<1>:
setInterface<1>(seg1, seg2, (x_ == 0.0));
```

Note that we have to define two different types of boundaries: First, a standard Neumann boundary is set for the terms $\int_\Gamma u_i v_i \, \mathrm{d}x$, $i = 1, 2$. For the coupling of segments, the boundary of type `Interface<1>` is specified in the last line (for an explanation of this implementation choice, see Section 3.5).

Now the remaining terms are assembled with the lines (assuming the parameters D_1, D_2, h and m to be unity and the use of a default integration rule)

```
//assembling of Omega_1:
assemble<FEMConfig>(segment1, system_matrix, rhs,
      integral< Omega >         ( Gradient_1() * Gradient_2() )
    + integral< Gamma<1> >      ( basisfun<1>() * basisfun<2>() )
    - integral< Interface<1> > ( basisfun<1>() * basisfun<2>() )
    = ScalarExpression<0>() );

//assembling of Omega_2:
assemble<FEMConfig>(segment2, system_matrix, rhs,
      integral<Omega>           ( Gradient_1() * Gradient_2() )
    + integral< Gamma<1> >      ( basisfun<1>() * basisfun<2>() )
    - integral< Interface<1> > ( basisfun<1>() * basisfun<2>() )
    = ScalarExpression<0>() );
```

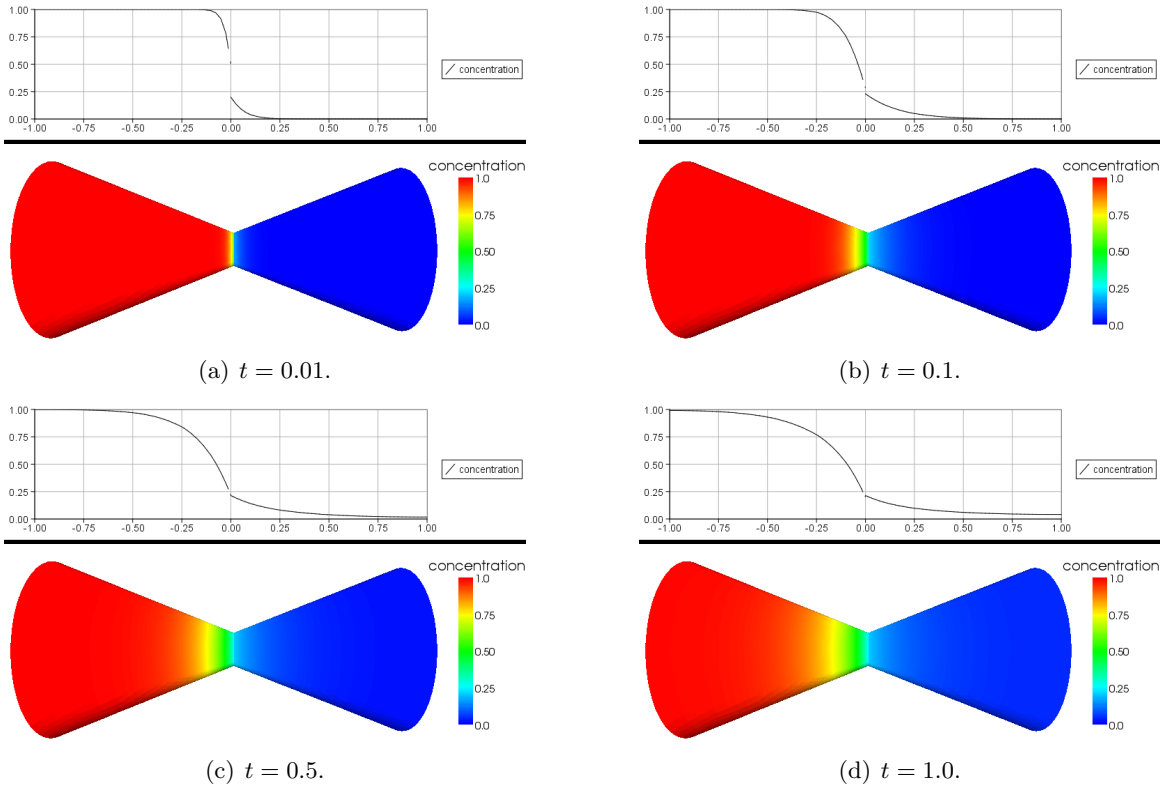
With this, we obtain the solution at time step $n+1$ from the solution at time n (assumed to be stored in a vector named `solution`) as

```
double delta_t = 0.1;   //time step length

//assemble matrices here

system_matrix = system_matrix * delta_t + mass_matrix;

//compute one timestep:
solution = cg_solve(system_matrix, mass_matrix * solution);
```

Instead of computing only one time step, one typically loops over the solver routine and obtains the solution for each time step one after another.

Fig. 7.2 and Fig. 7.3 depict the numerical results of a diffusive hourglass: Two truncated cones Ω_1 and Ω_2 are connected by means of the segregation model. Initially, the particle concentration of the left segment Ω_1 equals one, while the particle concentration in Ω_2 equals zero. The diffusion coefficients are $D_1 = 0.1$ and $D_2 = 0.5$ in Ω_1 and Ω_2 respectively. At $t = 0$, the two segments are connected by means of the segregation model with transport coefficient $h = 2.0$ and segregation coefficient $m = 3.0$. Since the volumes of Ω_1 and Ω_2 are equal, we expect equilibrium concentrations of 0.25 and 0.75 in Ω_1 and Ω_2.

At $t = 0.01$, particles flow from Ω_1 to Ω_2. Due to the larger diffusion coefficient in Ω_2, particles diffuse into the interior of Ω_2. At $t = 0.1$, most of the particles near the boundary have been absorbed to Ω_2, so that the particle exchange slows down. Due to the smaller diffusion coefficient in Ω_1 and the small interface, particle transfer slows down. At time $t = 0.5$ and $t = 1.0$, the particle depletion near the interface of Ω_1 spreads into the interior. At $t = 5$, segregation can already be observed: The interface behaves like a vacuum pump, transferring particles from Ω_1 to Ω_2, even if the concentration in Ω_2 is



(a) $t = 0.01$. (b) $t = 0.1$.

(c) $t = 0.5$. (d) $t = 1.0$.

Figure 7.2: Results for a diffusive hourglass for small times t, where initially all particles have been located on the left. The xy-plot on top shows the particle concentration along the x-axis at each time step.

already larger then the concentration in Ω_1. Finally, at $t = 60$ the concentrations are already close to their equilibrium; particle interchange slows down and diffusion leads to almost constant concentrations in each segment at this time step.

7.2 Electromigration with Grain Boundaries

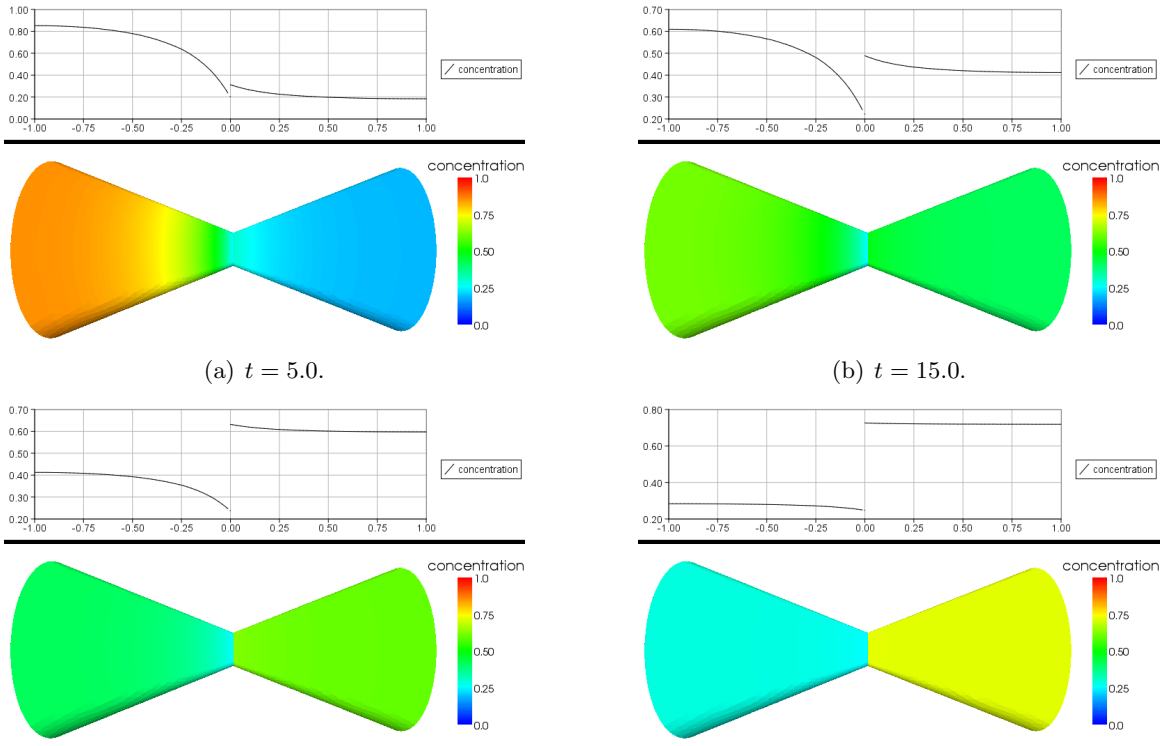
The physical effect of segregation shows up in several contexts: While Lau et al. [22] applied the segregation model to phosphorus segregation at the Si-SiO_2 interface, we consider segregation at the grain boundaries of interconnects from integrated circuits (ICs). In modern ICs, such interconnects are typically made of copper which forms small grains resulting from the growth process. In particular, individual copper grains are formed from small seeds, which initially grow freely and after some time they touch each other, leading to a formation of these characteristic grain interfaces. In presence of a (sufficiently strong) electric field, there is accumulation and depletion of atoms on either side of a grain boundary (see Fig. 7.4). Vacancies behave in the opposite way, i.e. they accumulate at locations where atoms deplete and vice versa.

The governing equation for the concentration u of vacancies is

$$\frac{\partial u}{\partial t} = -\nabla \cdot \boldsymbol{q} \ , \quad u(t=0) = u_0 \tag{7.10}$$

where

$$\boldsymbol{q} = -D\left(\nabla u + Au\nabla\Psi\right) \ , \tag{7.11}$$



(a) $t = 5.0$. (b) $t = 15.0$.

(c) $t = 30.0$. (d) $t = 60.0$.

Figure 7.3: Results for a diffusive hourglass for large times t.

with parameters A and D and initial (equilibrium) concentration u_0. The electrostatic potential Ψ is obtained from solution of the Laplace equation

$$\Delta \Psi = 0 \tag{7.12}$$

subject to appropriate boundary conditions. Additionally, the interconnect simulation domain is split up into several segments that represent one copper grain each, so the domains are coupled by segregation interface conditions with transport coefficient $h = 0.01$ and segregation coefficient $m = 1$.

Since we do not have experimental data available where parameters can be extracted, we choose A and D such that the physical effects can clearly be observed, but typically these effects show up in different scales in practice. Consequently, length and time scales have to be understood in abstract units instead of, say, micrometer and seconds.

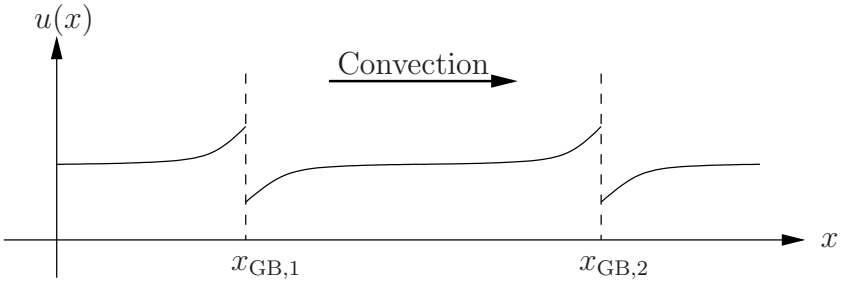


Figure 7.4: Effect of material accumulation and depletion at grain boundaries located at $x_{\mathrm{GB},1}$ and $x_{\mathrm{GB},2}$ for particles with concentration u.

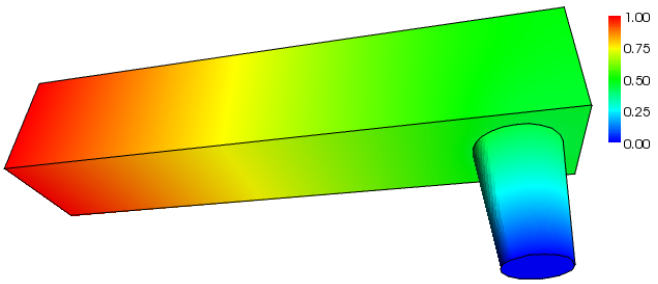


Figure 7.5: Potential distribution in the interconnect.

From the algorithmic point of view, this electromigration model leads to some additional requirements compared to the plain segregation model described in the previous section. In particular, the gradient of the electrostatic potential $\nabla\Psi$ in (7.11) turns up as additional parameter. Since the potential is known from the solution of Laplace's equation in a preprocessing step, it can be seen as a space dependent parameter in the formulation.

In the fundamental work for this thesis [26], such a placeholder was already introduced. In the meantime, its interface was slightly modified to take the new FEM configuration into account:

```
template <typename FEMConfig, typename VectorType,
          typename diff_tag, long resultnum,
          long component = 0, typename EvalClass = IdentityEval >
class evalResult;
```

The template parameters `FEMConfig` and `VectorType` represent the FEM configuration and the vector type of the result vector (where Ψ is stored in), `diff_tag` specifies derivatives of the underlying function discretised by the result vector and `resultnum` specifies which of the two vectors passed to the assembly call is intended. The template parameter `component` allows to evaluate a particular component of a vector-valued solution, while `EvalClass` allows an additional post-processing, like taking the exponential of the underlying function.

Since the whole simulation domain is split up into several segments and grain boundaries are assumed to be fully transparent with respect to the electrostatic potential, interface elements have to be coupled appropriately for the the numerical solution of Laplace's equation. This is done with the call

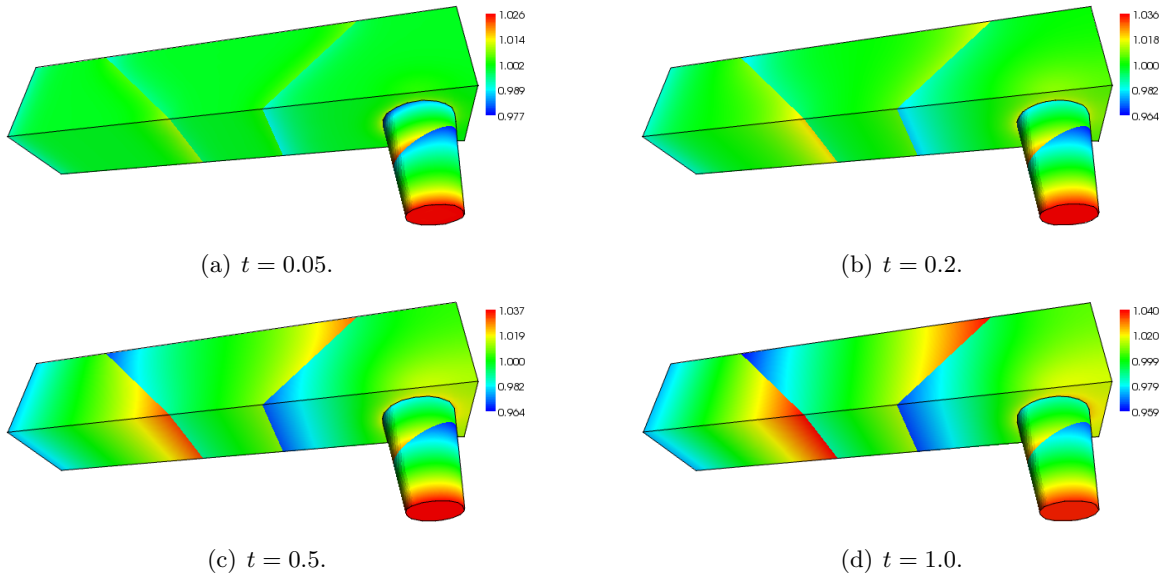
```
setSegmentConnection<FEMConfig_Laplace>(seg1, seg2);
```

for two segments `seg1` and `seg2`. Elements on the interface are detected automatically and interface elements are tagged to have common degrees of freedom (cf. Section 3.5). Repeating this step for all segment pairs, one can proceed with the assembly of Laplace's equation. The required code has showed up several times in this thesis already, thus there is no need for another replication. After a compilation and an execution of the program one obtains a potential as depicted in Fig. 7.5.

Let us proceed with the assembly of the electromigration equation (7.10). The weak form on the i-th segment Ω_i (coupled with Ω_{i-1} and Ω_{i+1} at interfaces $\Gamma_{i,i-1}$ and $\Gamma_{i,i+1}$ respectively) is given as

$$\int_{\Omega_1} \frac{\partial u_i}{\partial t} v_i + D\nabla u_i \nabla v_i + DAu_i \nabla\Psi\nabla v_i \,\mathrm{d}\boldsymbol{x} + h\int_{\Gamma_{i,i-1}} (u_i - u_{i-1})\, v_i \,\mathrm{d}A + h\int_{\Gamma_{i,i+1}} (u_i - u_{i+1})\, v_i \,\mathrm{d}A = 0 \quad \forall v_i \in H^1(\Omega_i)\ . \tag{7.13}$$

First, the interfaces on `seg1` and `seg2` are set using (cf. Section 3.5)



(a) $t = 0.05$. (b) $t = 0.2$.

(c) $t = 0.5$. (d) $t = 1.0$.

Figure 7.6: Simulation of electromigration in an interconnect for small times t. Initially, the concentration in the whole interconnect is equal to one.

```
setInterface<12>(seg1, seg2);
setBoundaryArcAtInterface<12, 12>(seg1);
setBoundaryArcAtInterface<12, 12>(seg2);
```

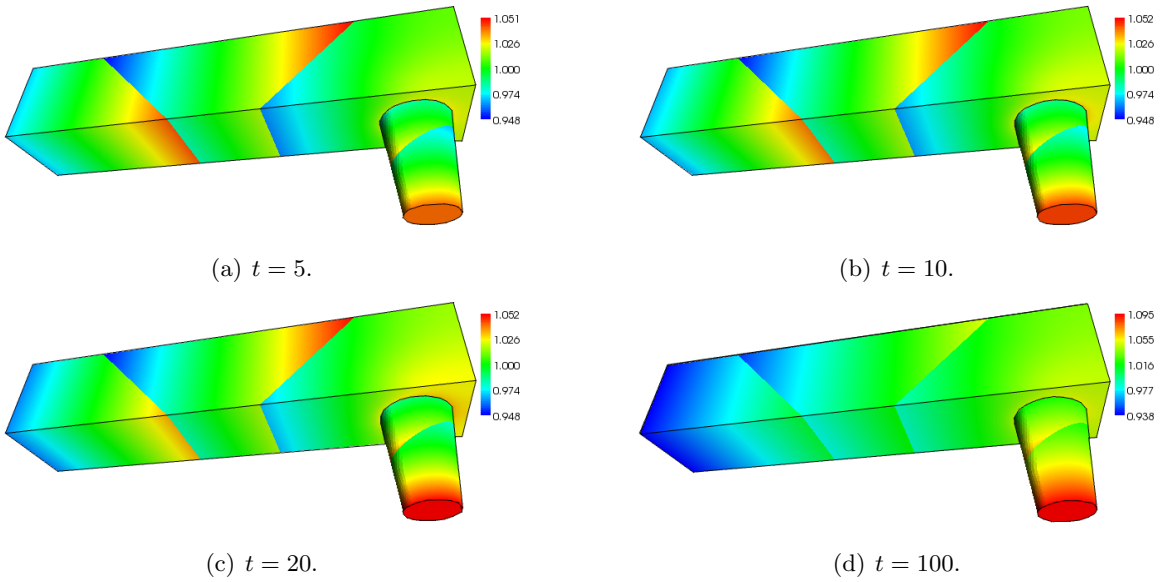
and similarly for other interfaces. At the interface $\Gamma_{1,2}$, contributions are then assembled via

```
assemble<FEMConfig>(seg1, interface_matrix12, rhs1,
           integral< Gamma<12> >( basisfun<1>() * basisfun<2>() ) -
           integral< Interface<12> > ( basisfun<1>() * basisfun<2>() )
            = _0_ );
assemble<FEMConfig>(seg2, interface_matrix12, rhs1,
           integral< Gamma<12> >( basisfun<1>() * basisfun<2>() ) -
           integral< Interface<12> > ( basisfun<1>() * basisfun<2>() )
            = _0_ );
```

and similarly for other interfaces. Mass matrices and stiffness matrices are assembled in the usual way, so let us proceed with the drift term. Using the placeholder `evalResult`, the assembly reads

```
evalResult<FEMConfig_Laplace, VectorType, diff<0>, 1>   psi_x;
evalResult<FEMConfig_Laplace, VectorType, diff<1>, 1>   psi_y;
evalResult<FEMConfig_Laplace, VectorType, diff<2>, 1>   psi_z;
basisfun<1, diff<0> > v_x;
basisfun<1, diff<1> > v_y;
basisfun<1, diff<2> > v_z;

assemble<FEMConfig_EMigration>(seg1, convection_matrix, rhs1,
            integral<Omega>(
                basisfun<2>() * (psi_x * v_x + psi_y * v_y + psi_z * v_z)
                              ) = _0_ ,
            result_laplace);
```

Note the result vector `result_laplace`, which is the fifth parameter passed to `assemble`, and the two different FEM configurations `FEMConfig_Laplace` and `FEMConfig_EMigration`, which both work on the



(a) $t = 5$. (b) $t = 10$.

(c) $t = 20$. (d) $t = 100$.

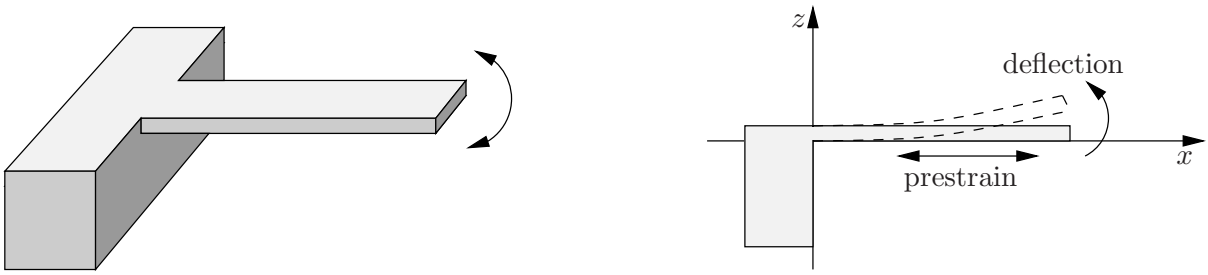
Figure 7.7: Simulation of electromigration in an interconnect for medium and large times t. The evolution in time is determined by the flux through the grain boundaries.

same geometric domain, but the test and trial spaces result in different degrees of freedoms.

A backward Euler time discretisation finally leads to numerical results shown in Fig. 7.6 and Fig. 7.7. Several effects can be observed: Immediately after applying a voltage to the contacts, the vacancies are driven towards the ground electrode located at the bottom of the via (Fig. 7.6(a)). Since the electric field in the via is stronger than in the copper line because of the smaller dimensions, a concentration gradient develops which is largest close to the ground electrode. Due to the segregation model at the grain boundaries (interfaces), vacancies are blocked and accumulate or deplete there. The concentration gradients in the copper lines are smaller due to their larger volumes: It takes more time to reach the grain boundaries there. At $t = 0.2$ (Fig. 7.6(b)) and $t = 0.5$ (Fig. 7.6(c)) the accumulation and depletion processes in the copper lines continue, while in the via a quasi-equilibrium is already reached. At time $t = 1.0$ (Fig. 7.6(d)) accumulation and depletion concentrations in the copper lines are of similar magnitude than in the via[1]. For these short times, vacancies fluxes through the grain boundaries are very small, thus similar results would have been obtained if we had assumed blocking grain boundaries.

At larger times t, however, fluxes through the grain boundaries play a role: Starting from a quasi-equilibrium state at $t = 5$ (Fig. 7.7(a)), where initial transients due to the application of the electric field have vanished and the peak concentration is located in the copper line, we see that the concentration differences at the grain boundaries lead to net fluxes towards the ground electrode. For longer times we therefore expect that the peak concentration is again located at the via (ground electrode) and concentration differences at the grain boundaries decay. This is indeed the case: Comparing $t = 10$ (Fig. 7.7(b)) and $t = 20$ (Fig. 7.7(c)) the concentration in the via increases and exceeds the peak concentration in the copper line, which decreases due to the flux towards the ground electrode. Finally, at $t = 100$ it can be seen that the result asymptotically converges to the case that no grain boundaries are considered at all. This is not surprising: In equilibrium state there is no net flux through the grain boundaries, which can (at least in this test case) only be fulfilled if there is no concentration difference at the boundaries.

[1] In real devices, the main concentration gradients are located in the via. To emphasize the physical processes, parameter A in (7.11) was chosen to be larger in the copper line than in the via, so that vacancies "feel" a stronger electric field there.



(a) Simplified layout of a MEMS cantilever.

(b) Prestrain on the bottom of the cantilever results in a deflection.

Figure 7.8: During the fabrication of a MEMS cantilever, prestrain on the bottom of the cantilever develops. Therefore, increasing deflection with decreasing thickness can be observed.

From the point of electromigration, several other effects can be observed. The most prominent one is the formation of so-called *voids*, which can be seen as holes in the metal lattice. Such voids have the unpleasant property of propagation and movement towards the via. It narrows the paths of current flow down and finally leads to device failure. For the algorithmic treatment of void evolution several numerical methods exist. For example the *Level Set Method* is frequently used, but this is beyond the scope of this thesis.

7.3 MEMS Cantilever with Prestrain

The third application considered in this chapter is the deflection of cantilevers due to intrinsic strain in micro-electro-mechanical systems (MEMS). Such intrinsic strain acts as prestrain and originates from the deposition of a thin film at high temperatures on a sacrificial SiO_2 layer. During deposition, a complex growth process of microstructures ultimately forms the cantilever, where mechanical stresses due to non-uniform growth close to the sacrificial layer develop. As long as the thin cantilever film is fixed to the sacrificial SiO_2 layer, no deflection occurs. After the removal of the sacrificial layer, the interface region of the thin film expands and a deflection of the cantilever can be observed. The mechanical stresses in the microstructures build up slowly; the time constant of this process is typically orders of magnitude larger than the duration of the fabrication process and additionally shows a strong temperature dependence. In this section we investigate the quasi-equilibrium state in which the prestrain is supposed to be in equilibrium, i.e. all initial transients have decayed. Such a state is called a quasi-equilibrium state, because it is only valid as long as the device is not exposed to large temperature changes. More details about the physical processes that lead to prestrain in a cantilever can be found in recent publications [19].

From a physics point of view, the cantilever deflects such that the energy Π of the system without external forces is minimised:

$$\Pi := \frac{1}{2} \int_\Omega \boldsymbol{\varepsilon} : \boldsymbol{\sigma} \, \mathrm{d}x \to \min. \tag{7.14}$$

Here, ε is the strain tensor, σ is the stress tensor and $\boldsymbol{\varepsilon} : \boldsymbol{\sigma} = \sum_{i,k} \boldsymbol{\varepsilon}_{i,k} \boldsymbol{\sigma}_{i,k}$ denotes the tensor product. We suppose small deflections, hence the strain tensor is related to the displacement $\boldsymbol{u} = (u_i)_{i=1}^n$ by

$$\varepsilon_{i,j} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) , \tag{7.15}$$

with shorthand notation $\boldsymbol{\varepsilon} = D\boldsymbol{u}$. For linear materials, the stress-strain relationship

$$\boldsymbol{\sigma} - \boldsymbol{\sigma}_0 = \frac{E}{1+\nu}\big(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_0 + \frac{\nu}{1-2\nu}\mathrm{tr}(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_0)I\big) \tag{7.16}$$

holds, where $\boldsymbol{\sigma}_0$ denotes prestress, E is Young's modulus, ν is the Poisson ratio of the material, $\boldsymbol{\varepsilon}_0$ is the prestrain tensor, $\mathrm{tr}(\cdot)$ denotes the trace operator and I is the identity matrix. As shorthand notation for (7.16) we write $\boldsymbol{\sigma} - \boldsymbol{\sigma}_0 = \mathcal{C}(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_0)$.

We assume that the deflection of the MEMS cantilever is caused by prestrain only, thus $\boldsymbol{\sigma}_0 = 0$. Substituting (7.16) into (7.14) and using the identity

$$\frac{1}{2}\,\boldsymbol{\varepsilon} : \boldsymbol{\sigma} = \frac{\lambda}{2}\big(\mathrm{tr}(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_0)\big)^2 + \mu\,\varepsilon : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_0)\ , \quad \text{with } \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)}\ , \tag{7.17}$$

the energy in the system can be written as

$$\Pi = \int_\Omega \mu\,\boldsymbol{\varepsilon} : \boldsymbol{\varepsilon} + \frac{\lambda}{2}\big(\mathrm{tr}(\boldsymbol{\varepsilon})\big)^2 - \mu\,\boldsymbol{\varepsilon} : \mathcal{C}\boldsymbol{\varepsilon}_0\,\mathrm{d}x \to \min\,. \tag{7.18}$$

For the incorporation of homogeneous Dirichlet boundary conditions on parts of the boundary $\Gamma \subseteq \Omega$, we set

$$H^1_\Gamma := \big\{\boldsymbol{v} \in H^1(\Omega)^n \;\big|\; \boldsymbol{v}(x) = 0 \text{ for } x \in \Gamma\big\}\ . \tag{7.19}$$

The weak formulation of (7.18) is to find $\boldsymbol{u} \in H^1_\Gamma$ such that

$$\int_\Omega \mu\, D\boldsymbol{u} : D\boldsymbol{v} + \frac{\lambda}{2}\,\mathrm{div}(\boldsymbol{u})\,\mathrm{div}(\boldsymbol{v})\,\mathrm{d}x = \mu\int_\Omega \mathcal{C}\boldsymbol{\varepsilon}_0 : D\boldsymbol{v}\,\mathrm{d}x \quad \forall \boldsymbol{v} \in H^1_\Gamma\ . \tag{7.20}$$

Let us now consider the three-dimensional case: We choose the coordinate system such that the cantilever is located directly above the plane $z = 0$. Therefore, the prestrain tensor $\boldsymbol{\varepsilon}_0$ is given by

$$\boldsymbol{\varepsilon}_0 = \begin{pmatrix} s(z) & 0 & 0 \\ 0 & s(z) & 0 \\ 0 & 0 & 0 \end{pmatrix}\ , \tag{7.21}$$

where we assumed tensile strains in x- and y-direction only. The function $s(z)$ has to be specified such that directly at the interface at $z = 0$, strains are very high, whereas strains are considerably smaller away from the interface.

The prestrain was chosen such that measured depth dependence given in [19] is approximated by a parabola close to the interface and by a linear dependence away from the interface. This approximation is illustrated in Fig. 7.9 and is given by

$$s(z) = \begin{cases} 0.057 - x/3.9 + x^2/3.3 & z < 0.5 \\ 0.001 + 0.0008x & z \geq 0.5 \end{cases}\ . \tag{7.22}$$

With the prestrain tensor (7.21) the weak formulation reads

$$\int_\Omega \mu\, D\boldsymbol{u} : D\boldsymbol{v} + \frac{\lambda}{2}\,\mathrm{div}(\boldsymbol{u})\,\mathrm{div}(\boldsymbol{v})\,\mathrm{d}x = \mu\int_\Omega \frac{\lambda s(z)}{\nu}\big(\frac{\partial v_1}{\partial x} + \frac{\partial v_2}{\partial y} + 2\nu\frac{\partial v_3}{\partial z}\big)\,\mathrm{d}x \quad \forall v \in H^1_\Gamma\ . \tag{7.23}$$

Here, v_1, v_2 and v_3 denote the x-, y- and z-component of the vector-valued test function $\boldsymbol{v}$.

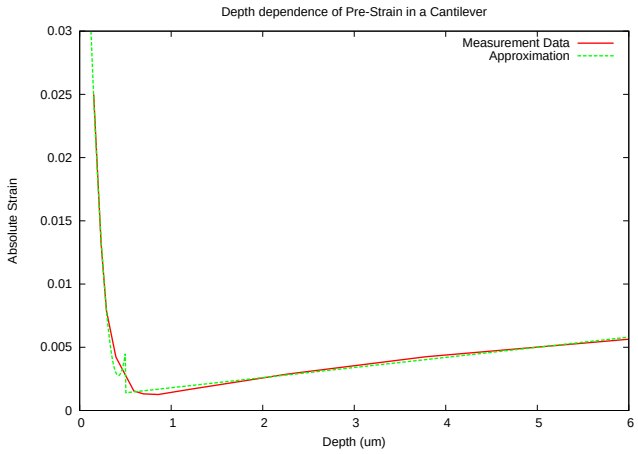


Figure 7.9: Measured and approximated depth-dependence of prestrain in a MEMS cantilever.

The vector-valued test functions are directly deduced from scalar-valued basis functions: For a scalar-valued test function φ_i, the vector-valued test functions in three dimensions are

$$\boldsymbol{\varphi}_{i,1} = \begin{pmatrix} \varphi_i \\ 0 \\ 0 \end{pmatrix} , \quad \boldsymbol{\varphi}_{i,2} = \begin{pmatrix} 0 \\ \varphi_i \\ 0 \end{pmatrix} , \quad \boldsymbol{\varphi}_{i,3} = \begin{pmatrix} 0 \\ 0 \\ \varphi_i \end{pmatrix} .$$

An analogous construction for the trial functions holds. For a given pair of scalar-valued test and trial functions (φ_i, φ_j), testing with all combinations of vector-valued test and trial functions leads to the system

$$\begin{aligned} &\int_\Omega \mu \begin{pmatrix} \frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial x} + \frac{1}{2}\frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial y} + \frac{1}{2}\frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial z} & \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial x} & \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial x} \\ \frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial y} & \frac{1}{2}\frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial x} + \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial y} + \frac{1}{2}\frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial z} & \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial y} \\ \frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial z} & \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial z} & \frac{1}{2}\frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial x} + \frac{1}{2}\frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial y} + \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial z} \end{pmatrix} \mathrm{d}x \\ &+ \int_\Omega \frac{\lambda}{2} \begin{pmatrix} \frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial x} & \frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial y} & \frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial z} \\ \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial x} & \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial y} & \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial z} \\ \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial x} & \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial y} & \frac{\partial\varphi_i}{\partial z}\frac{\partial\varphi_j}{\partial z} \end{pmatrix} \mathrm{d}x \\ &= \frac{\mu\lambda}{2} \int_\Omega s(z) \begin{pmatrix} \frac{\partial\varphi_i}{\partial x} \\ \frac{\partial\varphi_i}{\partial y} \\ 2\nu\frac{\partial\varphi_i}{\partial z} \end{pmatrix} \mathrm{d}x \ . \end{aligned} \tag{7.24}$$

This system is directly translated into code. To keep the code snippets short, we split (7.24) into two parts: First, the second matrix on the left-hand side, which originates from $\mathrm{div}(\boldsymbol{u})\mathrm{div}(\boldsymbol{v})$, is assembled with the lines

```
1 basisfun<1, diff<0> > v_x;   basisfun<2, diff<0> > u_x;
2 basisfun<1, diff<1> > v_y;   basisfun<2, diff<1> > u_y;
3 basisfun<1, diff<2> > v_z;   basisfun<2, diff<2> > u_z;
4
5 // div(v) * div(u)
6 assemble<FEMConfig>
7  (seg1, matrix, rhs,
8   makeExpressionList3x3(
```

```
        integral<Omega>(v_x * u_x),
                        integral<Omega>(v_x * u_y),
                                        integral<Omega>(v_x * u_z),
        integral<Omega>(v_y * u_x),
                        integral<Omega>(v_y * u_y),
                                        integral<Omega>(v_y * u_z),
        integral<Omega>(v_z * u_x),
                        integral<Omega>(v_z * u_y),
                                        integral<Omega>(v_z * u_z)
                        ) = _0_
);
```

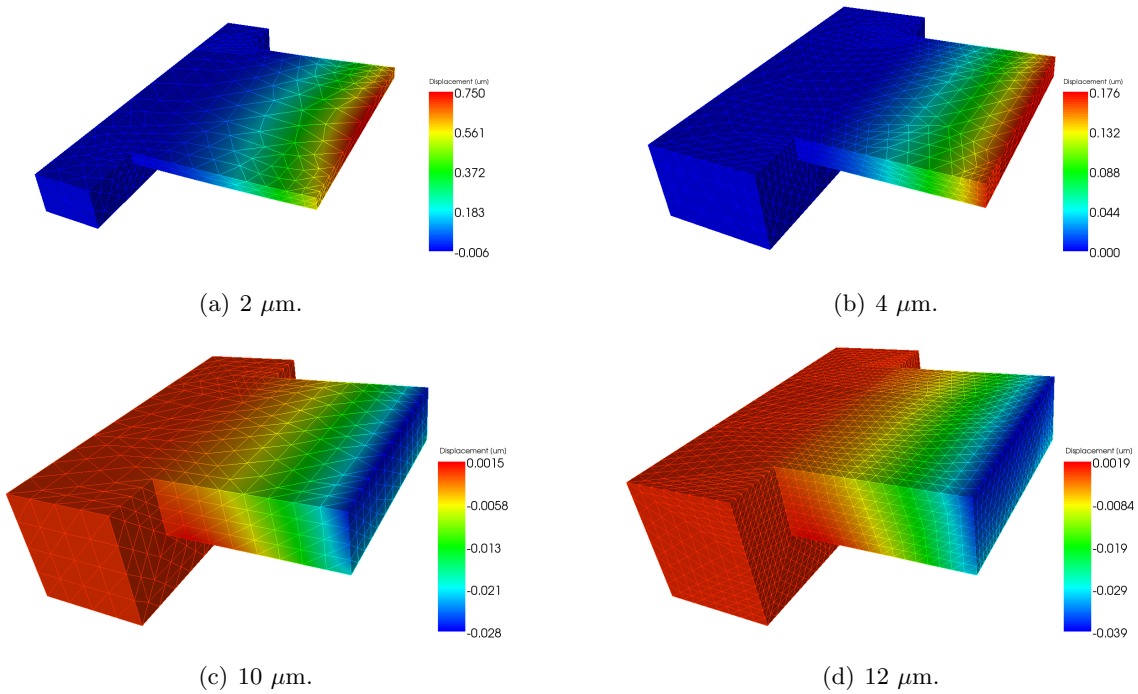
The remaining terms are then assembled similarly[2]. Since the resulting code lines are quite lengthy as it is the case for the mathematical formulation in (7.24), only the first row of the vector-valued equation is shown:

```
// eps (v) : eps (u)
assemble<FEMConfig>
  (seg1, matrix, rhs,
   makeExpressionList3x3(
     //first row:
     integral<Omega>( v_x * u_x + v_y * u_y / _2_ + v_z * u_z / _2_ ),
     integral<Omega>( v_y * u_x / _2_ ),
     integral<Omega>( v_z * u_x / _2_ ),

     //second row and third row similarly
   )
   =
   makeExpressionList3(
     integral<Omega>( ( x_ > _0_) * (
               (z_ > _5_ / _10_) * (_10_ + _8_ * z_) / _10000_ +
               (z_ < _5_ / _10_) * (_57_/_1000_ - z_ * _10_/_39_ + z_ * z_
                  * _10_/_33_)
                                    ) * basisfun<1, diff<0> >()
                  ),

     //second row and third component similarly
                     )
        );
```

Note the use of logical expressions to specify the inhomogeneous strain on the right-hand side: The bulk region of the cantilever is located at $x < 0$, thus there is no prestrain. The piece-wise approximation of the prestrain depth dependence from (7.22) is directly transferred to code using compile time expressions and is directly accessible for manipulation by the end-user of the framework. The use of fractional expressions for the specification of the depth dependence is certainly a little bit unhandy compared to a direct use of floating points values, but the flexibility is nevertheless considerably larger than in many other FEM packages, where space dependent coefficients often face poor support.

For the simulation the Lamé coefficients $2\mu = \lambda = 200$ GPa were used. The deflection of a cantilever is an ill-conditioned problem [9], which complicates the numerical simulation considerably. To achieve reasonably accurate results, cantilevers with a length of 30 μm, a width of 40 μm and thicknesses between 1 μm and 12 μm have been investigated. For the finite element approximation quadratic basis functions have been used, which are more robust with respect to the so-called *locking effect* compared to linear basis functions.

[2]Note that the assembly routine only updates the matrix and vector entries, but does not reset them.



(a) 2 μm. (b) 4 μm.

(c) 10 μm. (d) 12 μm.

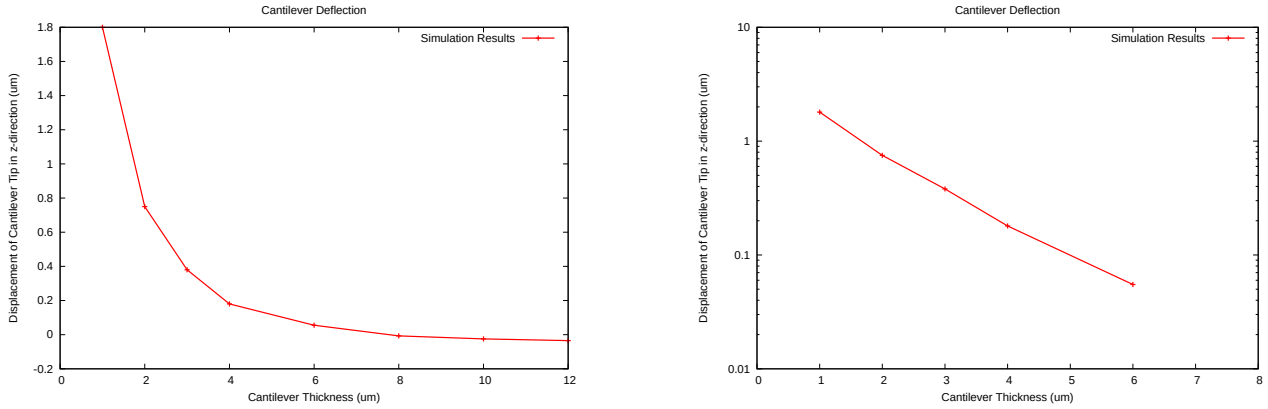
Figure 7.10: Vertical displacements of MEMS cantilevers with (30 μm length, 40 μm width) for several thicknesses.

Simulation results are shown in Fig. 7.10. It can be observed that for cantilever thicknesses larger than six microns the cantilever is deflected towards the negative values of z, i.e. the cantilever is bended towards the substrate as if it were due to gravity (which was neglected in the simulations). The reason for this behavior is the positive slope in the approximation of the prestrain at higher values of z in (7.22) and Fig. 7.9. A better approximation would take a saturation of the prestrain at a depth z_0 into account, so that cantilevers thicker than z_0 show a negligible deflection.

For cantilevers thinner than approximately 7 μm, a deflection in positive z-direction (i.e. away from the substrate) can be observed. This is due to the high prestrain at the bottom of the cantilever, which now becomes dominant. In Fig. 7.10 (a) it can even be observed that there is an additional bending along the y-coordinate. Nevertheless, the length of real cantilevers is typically by one order of magnitude larger than their width, therefore such a bending along the y-axis is typically negligible.

The dependence of the displacement at the tip of the cantilever on the thickness is illustrated in Fig. 7.11. From the semi-logarithmic plot an exponential dependence of the displacement on the cantilever thickness can clearly be observed. A similar behaviour can be observed from measurement data of cantilevers with thicknesses between 4 and 10 microns given in [19]. Such an exponential dependence is at first sight surprising, because the depth-dependence of the prestrain curve is approximated by a parabola close to zero and by a straight line away from zero.

For cantilevers with much larger aspect ratios, the uniform prestrain distribution should allow for a direct scaling of the results. In particular, a square-law dependence of the displacements on the cantilever length is expected, provided that the prestrain serves as a volume load constant along the length axis and the prestrain depth dependence does not change. The numerical results from Fig. 7.11 can then be scaled to the geometry of interest and a similar qualitative behaviour will be observed. In particular, prestrain in cantilevers has to be minimised if cantilevers with high aspect



(a) Linear plot. (b) Semi logarithmic plot.

Figure 7.11: Deflection at the tip for several cantilever thicknesses.

ratios (length:thickness) and a thickness below ten microns are fabricated.

A crude analytical justification for the exponential dependence of the deflection on the cantilever thickness can be found as follows: Consider the one-dimensional Euler-Bernoulli equation for a beam:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\left(EI\frac{\mathrm{d}^2u}{\mathrm{d}x^2}\right) = w \ , \tag{7.25}$$

where E is the elastic modulus, I is the second moment of inertia, $u(x)$ is the displacement at point x and w is a distributed load. In our simulation setting, both E and I are constant. The tensile stress in the beam can be computed as

$$\sigma = Ec\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \ , \tag{7.26}$$

with the distance c of a point of interest from the neutral axis. Since the stresses and strains in one dimension are proportional up to a constant factor (cf. (7.16)), we claim that

$$w \sim \sigma \sim \varepsilon_0 \sim g(z)\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \ , \tag{7.27}$$

where $g(z)$ is a depth dependent parameter which represents the average slope of the prestrain relation $s(z)$ for a cantilever with thickness z from (7.22).

Substituting (7.27) into (7.25) gives

$$\frac{\mathrm{d}^4u}{\mathrm{d}x^4} = Cg(z)\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \ , \tag{7.28}$$

where C is a numerical constant that depends on the material considered. Setting $v = \mathrm{d}^2u/\mathrm{d}x^2$, we obtain

$$\frac{\mathrm{d}^2v}{\mathrm{d}x^2} = Cg(z)v \ . \tag{7.29}$$

The solution of this ordinary differential equation is $v(x) = A\exp(Cg(z)x) + B\exp(-Cg(z)x)$, where A and B are parameters. From this, the displacement u is obtained from integration with respect to x:

$$u = \frac{A}{\big(Cg(z)\big)^2}\exp(Cg(z)x) + \frac{B}{\big(Cg(z)\big)^2}\exp(-Cg(z)x) + Sx + R \ , \quad S, R \in \mathbb{R} \ . \tag{7.30}$$

If we consider the displacement of a cantilever with given length L, $g(z)$ takes large negative values at small thicknesses z, hence one of the exponentials dominates in (7.30). Therefore, the exponential behaviour observed numerically is also backed up by analytical computations.

Chapter 8

Outlook and Conclusion

A highly generic and generative programming framework for the finite element method was presented in this work. Since its introduction in the middle of the 20th century, a vast amount of knowledge about FEM has been accumulated all over the world by many scientists, therefore it would be overconfident to say that the presented framework covers most aspect of FEM. Frankly, it seems as if the converse is true: A huge number of possible extensions to the existing capabilities is available. The consequence is that the development of the presented programming framework will not stop with this thesis.

Before a conclusion is drawn, some topics for future development are discussed. They are all to some extent obvious extensions of existing features, but due to time constraints an implementation has not been done yet. The outlines given here only scratch the surface and are only a small selection of many possible ways how the framework can (and hopefully will) be extended, but the intention is to give a brief overview of the development in the near future.

8.1 Automatic Linearisation of Nonlinear Problems

Although the mathematical problem formulation is now fully reflected within code in terms of the weak formulation, the only manipulation of this formulation done so far is a rearrangement of terms. Differentiation is used for basis function polynomials only, but it could also be applied for the integrands in the weak formulation: Let us consider the weak formulation

$$\int_\Omega \nabla u \nabla v \,\mathrm{d}x + \int_\Omega u^2 v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in V \ , \tag{8.1}$$

with sufficiently regular source term f, suitable test space V and appropriate boundary conditions. This nonlinear problem cannot be assembled directly, because the form

$$a(u, v) := \int_\Omega \nabla u \nabla v \,\mathrm{d}x + \int_\Omega u^2 v \,\mathrm{d}x \tag{8.2}$$

is linear in its second argument only. For using Newton's method on the residual R of (8.1), a Fréchet-derivative of the bilinear form with respect to u is necessary:

$$\langle R(u), v\rangle = \langle f, v\rangle - a(u, v) \tag{8.3}$$

The update δu can be obtained from a linearisation of the residual:

$$\begin{aligned} 0 &\overset{!}{=} \langle R(u + \delta u), v\rangle \\ &\approx \langle R(u), v\rangle + \langle \frac{\partial R(u)}{\partial u}\delta u, v\rangle \\ &= \langle R(u), v\rangle - \frac{\partial a(u, v)}{\partial u}(\delta u) \end{aligned}$$

A linearisation of $a(u, v)$ can in many cases, where the finite element method is applied, be found by a differentiation of the integrand:

$$\begin{aligned}\frac{\partial a(u,v)}{\partial u}(\delta u) &= \lim_{\varepsilon \to 0} \frac{a(u+\varepsilon\delta u, v) - a(u,v)}{\varepsilon} \\ &= \int_\Omega \nabla \delta u \nabla v \, \mathrm{d}x + \lim_{\varepsilon \to 0} \frac{\int_\Omega \left[(u+\varepsilon\delta u)^2 - u^2\right] v \, \mathrm{d}x}{\varepsilon} \\ &= \int_\Omega \nabla \delta u \nabla v \, \mathrm{d}x + 2 \int_\Omega \delta u \, v \, \mathrm{d}x \ ,\end{aligned}$$

which is just what we would have obtained if we differentiated all integrands with respect to u. Translated into code, a linearisation requires a differentiation with respect to `basisfun<2>`. Thus, existing differentiation capabilities could be used for a differentiation of the weak form. However, an extra detail has to be considered: `basisfun<2>` (i.e. the code representation for the unknown function u) and `basisfun<2, diff<0> >` (for $\frac{\partial u}{\partial x_0}$) are two distinct types for the compiler, but both represent the unknown u. Nevertheless, remedies for this subtlety exist.

The possibility of automatic linearisation is very appealing from the point of usability and could be provided even for vector-valued unknowns. However, such a powerful tool has to work in a controlled way, i.e. the implementation has to be rather pessimistic: If a linearisation is carried out by the framework automatically, it must be a valid linearisation. If the framework cannot guarantee a proper linearisation on the other hand, appropriate feedback has to be given to the user.

8.2 Detection of Symmetry in the Bilinear Form

Similar to an automatic linearisation of nonlinear PDEs, the weak formulation can be checked for symmetry: If an underlying weak formulation is symmetric with respect to the unknown and the test function, for example

$$a(u, v) = \int_\Omega \nabla u \nabla v \, \mathrm{d}x = a(v, u) \ , \tag{8.4}$$

the resulting system matrix is also symmetric. This means that the assembly can be accelerated by a factor of almost two! Even if the final system matrix is not symmetric anymore, it typically consists of contributions from a mass matrix or a stiffness matrix, which are both symmetric and can be assembled much faster in a preprocessing step. It is also possible (and in fact a good idea) to use a symmetry of the weak formulation for the matrix storage scheme. In this case, matrix memory requirements can also be reduced by a factor of almost two.

8.3 Automatic Construction of a Time Discretisation

For parabolic PDEs, a finite element scheme is typically applied to the spatial domain only, while a discretisation in time is done by means of a backward Euler method. There are several other methods for a discretisation in time, for example Crank-Nicholson's method or a discontinuous Galerkin method, which all originate from the time-derivate of the unknown in the weak formulation.

At present, the time discretisation has to be done "by hand", which means that the loop over all time steps has to be implemented by the end-user. It would be more appealing to supply several schemes for the time discretisation, so that the desired method is defined by a type definition just as it is done for instance for basis functions now. Such assistance will be optional only, since a potential user of the framework may also want to implement a sophisticated time discretisation herself.

8.4 Error Estimation and Error Indication

For most problems in practice, a given problem needs to be solved up to a certain allowed error, thus the knowledge of the error of a numerical approximation is necessary. A-priori error estimates including the true solution are important for theoretical investigations such as proving convergence of certain schemes. However, if the true solution is already known, there is no need to compute a numerical approximation. Consequently, so-called *a-posteriori* estimates for the error $e_h = u - u_h$ of a numerical approximation u_h to the unknown true solution u have to be derived from u_h and the underlying variational formulation only.

Typically, an estimate for the global error is obtained from error indicators η_T for each cell T of a mesh $\mathcal{T}$:

$$\|e_h\| \approx \sqrt{\sum_{T\in\mathcal{T}} \eta_T^2} \tag{8.5}$$

Therefore, it is sufficient to find good error indicators η_T for each $T \in \mathcal{T}$. Clearly, the optimal error indicators are

$$\eta_{T,\text{opt}} = \|u - u_h\|_T \ , \tag{8.6}$$

where $\|\cdot\|_T$ is the restriction of $\|\cdot\|$ to $T \in \mathcal{T}$. Since we do not know u, one possibility is to derive a function $u_{T,\text{ref}}$ on each $T \in \mathcal{T}$, such that $u_{T,\text{ref}}$ is a sufficiently better approximation of the true solution u than u_h locally on T. With such a better approximation we can write

$$\eta_T \approx \|u_{T,\text{ref}} - u_h\|_T \tag{8.7}$$

and the error estimator (8.5) can be computed. Such a better approximation $u_{T,\text{ref}}$ can for example be obtained from Babuška's *extraction formulae* (see, e.g. [6] or [24]). A different possibility is to compute $u_{T,\text{ref}}$ on an either h-, p- or hp-refined grid. This immediately leads us to the topics of multigrid discussed in Chapter 6.

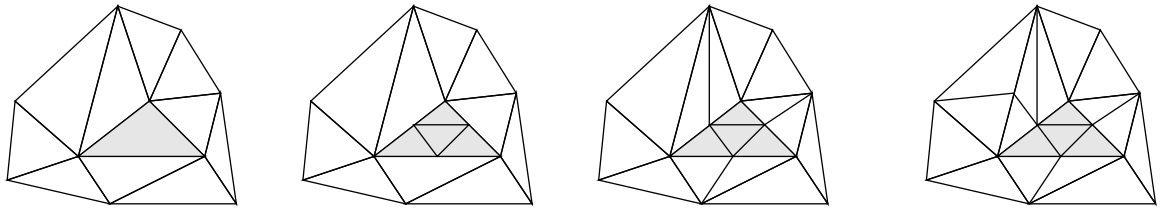
A different approach is to extract error estimators and indicators directly from the given variational problem in terms of the residual. For the Poisson equation $-\Delta u = f$, one can show [21] that ($\|\cdot\|_a$ is the energy norm)

$$\|e_h\|_a^2 = \sum_{T\in\mathcal{T}} \left(\alpha \int_T h^2 R(u_h)^2 \mathrm{d}x + \int_{\partial T} h[\vec{n}\cdot\nabla u_n]^2 \mathrm{d}s \right) \ , \tag{8.8}$$

where $R(u_h) = |f + \Delta u|$ is the residual and the notation $[\cdot]$ denotes the jump of the quantity enclosed in brackets. The constants α and β depend on the element type and the integral over ∂T is omitted on boundaries with Dirichlet values.

A closer look at the structure and the proof of (8.8) leads to the idea to automatically construct error estimators for PDEs with similar structure as the Poisson equation: The variational formulation has to be integrated by parts to find the strong formulation of the problem, which leads to an integral over the whole domain similar to the one in (8.8). The boundary integral in (8.8) is a consequence of the integration by parts.

An automatic construction of residual error estimators from the variational problem cannot be better than a derivation of a reliable error estimator by hand. Thus, things have to kept in perspective, because the mathematical construction of a good error estimator is a highly nontrivial task. The challenge for the future development is thus to allow for custom error estimators specified by a mathematically experienced end-user, while providing general error estimators with good reliability for users inexperienced in the field of error estimation.



(a) The initial grid with one triangle tagged for refinement.

(b) Hanging nodes have to be eliminated.

(c) A simple bisection algorithm leads to degenerated angles.

(d) One possible additional requirement: Always refine the longest edge of a triangle where bisection is required.

Figure 8.1: Hanging nodes lead to the need for regularisation algorithms, resulting in additional non-uniform refinements in the vicinity of the refined element.

8.5 Adaptive Refinement

The presented framework does not natively support adaptive refinement driven by error indicators η_T from the previous section *yet*, which is certainly a disadvantage for its applicability in practice for the moment. Most prerequisites for adaptive refinement are, however, already met:

- For h-refinement, one can either rely on external tools, or reuse the available multigrid-tools. Instead of a globally uniform refinement, a refinement of selected cells is already possible. So-called *hanging nodes* have to be eliminated by an appropriate refinement of neighbouring cells, which is so far missing. Nevertheless, details of implementation are already hidden behind the element tag, therefore h-refinement can be integrated rather easily and will find full support from existing multigrid capabilities.

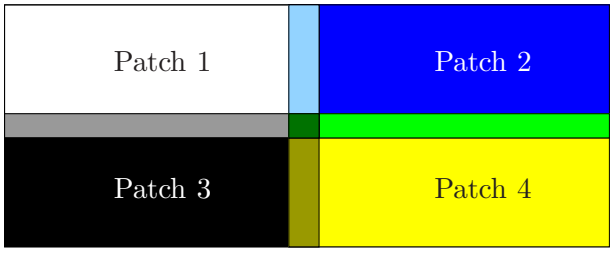
- Adaptive p-refinement can be realised by a run time variant of the basis function identifier `BasisFunctionID`, which for the moment exists during compile time only. Since the number of basis functions is defined for each topological element, continuity of basis functions at adjacent cells, which is the tricky issue in adaptive p-refinement, is assured by construction.

- Since hp-refinement is a consequence of the availability of h- and p-refinement, the framework can incorporate hp-refinement as soon as above refinement strategies are implemented. Unlike h- or p-refinement, hp-refinement leads to exponential convergence rates for many problems also relevant in practice.

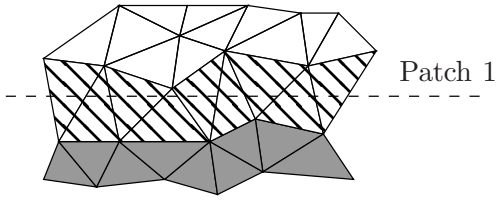
Without going into details, Fig. 8.1 illustrates some of the troubles that show up with hanging nodes. In brief, a mesh used for FEM has to fulfill certain regularity conditions. In particular, the mesh is not allowed to have k-elements located on the interior of m-elements whenever $m > k$, otherwise additional effort is required to assure continuity of basis functions from one element to another. In two dimensions this means that vertices must not lie in the interior of edges.

8.6 Parallelisation

Nowadays, multi-core CPUs are common in average workstations while most programs are ab initio designed for a single CPU core. Especially in the area of scientific computing, computational resources must not be left unused. The finite element method ends, in some sense, once the system matrix is assembled. The solution of the system of linear equations is then a general task, so we will see the solution process in a first step as a black-box process from an external library, where parallelisation



(a) Splitting a rectangular domain into four patches. The overlap is indicated with appropriate coloring.



(b) Overlapping patches: Striped cells belong to both patches.

Figure 8.2: Domain decomposition techniques for parallel systems with distributed memory.

properties are available. Indeed, many solver libraries (like AztecOO [5]) provide parallel solution algorithms for systems of linear equations and mathematical results are readily available [14].

The FEM assembly process iterates over all cells of a problem domain one after another, therefore a parallelisation for n threads can be obtained immediately by splitting the total number c of cells into sets of size c/n, so that each set is assembled by one thread. Such an approach is expected to scale well for CPUs with shared memory, as it is the case for desktops and notebooks today.

Systems with separate memory for each CPU, as it is common for large server farms, do not fully benefit from a decomposition of the assembly into threads, because the distributed memory requires the transfer of data. The amount of this slow communication can be reduced considerably by so-called *domain decomposition methods*: The problem domain is decomposed into patches, one for each CPU. Such patches typically overlap by the size of a few elements at the boundaries and each patch is treated separately. The given problem is then solved for each patch. After that the solution coefficients of the overlapping cells are compared and new boundary conditions for each patch are deduced. Then, a new iteration is started. Provided that the method converges, one arrives at the final solution. Domain decomposition scales very well with the number of CPUs, as already reported for example by [10]. Several important mathematical results are also available [23].

The current framework does not provide any parallelisation at present, but the existing design is sufficiently modular: For a shared memory system, one can assign one thread to each segment. In case there is only one segment or the number of segments is smaller than the number of CPU cores, the assembly of a segment allows parallelisation by the use of a modified `CellIterator`: At present, the `for`-loop for iteration over cells of a segment `seg` is

```
1 for (CellIterator cell_it = seg.getLevelIteratorBegin<CellLevel>();
2       cell_it != seg.getLevelIteratorEnd<CellLevel>();
3       ++cell_it)
```

This can be split into smaller pieces, one for each thread, by providing the thread-id as argument to `getLevelIteratorBegin` and `getLevelIteratorEnd`, so that iteration is carried out over all cells relevant for the current thread only. Please note that the design of the `QuantityManager` as presented in Section 2.4 is already such that thread-safety can be achieved easily.

A domain decomposition poses less requirements on the software design: In fact, each process solves the problem independently from others on the cell patch provided. The only communication between the processes is at the end of each iteration, when solutions on the overlapping patch boundaries are compared.

Finally, let us have a look at the parallelisation possibilities of multigrid: A closer inspection is only necessary for a shared memory system, since for distributed memory a domain decomposition is more effective. On a shared memory system, a parallelisation for shared memory can again be achieved by a modified `CellIterator` as outlined above, provided that the projection and injection operators

work on a per-cell basis. If more sophisticated transfer operators working on cell patches are used, the situation gets more complicated, but a concurrent processing should still be feasible. The smoothing operations have varying parallelisation capabilities: A Jacobi smoother has poor smoothing properties, but can be fully parallelised, while a Gauss-Seidel smoother has better smoothing properties, but a full parallelisation is not possible. Thus, using transfer operators working on a per-cell basis and a Jacobi smoother, the whole multigrid-process scales very well with the number of of available CPU cores.

8.7 The new C++ Standard

By the time of writing these lines, the new C++ standard, termed C++0x, is still in development. The latest working draft is dated Oct. 2008 [7], therefore the new standard is not expected to be available before the beginning of 2010. Support for selected new language features is already available in some compilers, the GNU Compiler Collection (GCC) for example already supports variadic templates [11], but many other features like the `auto`-keyword are by the time of writing still missing. Let us take a look at how these two language features can be used in the following.

Variadic Templates allow for an arbitrary number of template arguments. The present solution is to use a predefined set of template classes, whose maximum number of templates parameters is created by the (ab)use of the preprocessor. Another possibility is to use type lists, but both remedies have their disadvantages. With variadic templates, a `tuple` class, which serves as a list of an arbitrary number of types, can be written as

```
1 template<typename... Types> class tuple;
```

just like for a variable number of function arguments like for `printf`. A `tuple`-type will be provided by the new standard library, allowing convenient access to each parameter of the `tuple`. Unfortunately there is no simple mechanism to iterate over the type arguments, but recursive evaluation is possible. We can think of using variadic templates for summands of an compile time expression: For instance, an evaluation of an expression at a particular point is carried out by recursive evaluation of each summand. This allows a reduction of the average template depth of the expression type tree, which will hopefully lead to faster compilation times. Similarly, the implementation for `compressed_bf` can be reduced to one single variadic template declaration. Anyway, much more evaluation has to be done on this fairly new feature in order to improve, simplify or generalise existing implementations.

The second extension we address is the `auto`-keyword, which is a placeholder for a type that is well known to the compiler, but typically hard to specify by the user. This will especially allow short-hand names for expressions at compile time:

```
1 //hold the integral formulation in a separate variable:
2 auto integral_formulation =
3    (integral<Omega>(basisfun<1>() * basisfun<2>())
4        = ScalarExpression<0>());
5
6 //a call to assemble(...) is much shorter now:
7 assemble<FEMConfig>(segment, matrix, rhs, integral_formulation);
```

Without the `auto`-keyword, we would have to specify the exact type of `integral_formulation`, which requires the full knowledge of all implementation internals and is thus unreasonable for an end-user. Thus, with the new keyword it is then possible to split initially large code blocks arising from the specification of complex systems into much smaller bits.

8.8 Conclusion

The concepts of generic and generative programming applied to the finite element method allow to formulate the underlying weak formulation of the mathematical problem directly as source code. In this way, the full mathematical information is preserved at code level, which allows for the selection of the implementations suited best for a given problem. With this additional information, the compiler can build executables that achieve optimal performance at run time. Furthermore, the full decoupling of the domain handling and the finite element algorithms allows a flexible choice of the underlying cell geometry and the desired basis function degree. Consequently, a weak formulation that is mathematically independent of the underlying spatial dimension can automatically be adapted to the spatial dimension of interest by a single type definition.

It turns out that preserving the mathematical abstraction at code level leads to both high flexibility and run time performance comparable to that of hand-tuned code. The price to pay is an increase in compilation times, but there are means for trading compile time for run time efficiency available, so that an end-user can benefit from faster compilation times during the prototyping phase and get best run time performance once the implementation is verified.

Appendix A

Domain Terminology

The problem domain is denoted as $\Omega \subset \mathbb{R}^n$ and is assumed to be bounded, but not necessarily connected. Let $\mathcal{T}$ be a decomposition of Ω such that all $T \in \mathcal{T}$ belong to a particular class of polytopes and $|\mathcal{T}| < \infty$.

Definition 7. *An element T of $\mathcal{T}$ is called* cell[1].

Definition 8. *A point $v \in \Omega$ located at the boundary and defining the characteristic shape of a cell $T \in \mathcal{T}$ is called* vertex. *Typically, a vertex lies in the corner of a cell.*

Definition 9. *A connection between two vertices lying on the boundary of a cell $T \in \mathcal{T}$ and defining the characteristic shape is called* edge.

Definition 10. *A* facet *is an element of the characteristic decomposition of the boundary of a cell $T \in \mathcal{T}$ that determines its characteristic shape.*

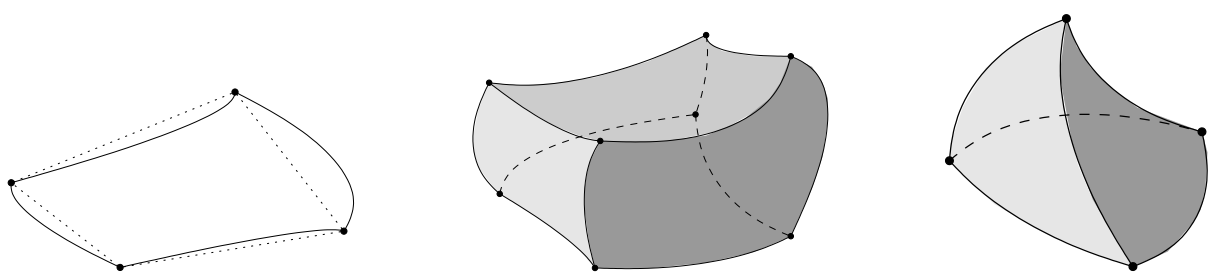
We do not necessarily restrict ourselves to polytopes, because we also want to allow curvi-linear elements in above definitions. Since the underlying reference element for all the FEM calculation will almost always be a polytope, we will speak about *quasi-polytopes*:

Definition 11. *A* quasi-polytope *is a geometrical object that is diffeomorph to a polytope and the mapping from a reference polytope to the quasi-polytope is an isoparametric polynomial mapping. The dimension of this quasi-polytope is the same as the dimension of the underlying polytope.*

In practice, however, only quasi-polytopes with mappings of low polynomial order are used and they are closely related to the geometric shape of underlying polytope.

Next, we need to label subsets of a decomposition $\mathcal{T}$ of Ω:

[1] In the literature also denoted as *body*. On the other hand, some literature defines a cell as a polytope of dimension three, which is not the case here.



(a) Curvilinear Quadrilateral. (b) Curvilinear Hexahedron. (c) Curvilinear Tetrahedron.

Figure A.1: Three different types of quasi-polytopes.

	Dimension	Codimension
Vertex	1	-
Edge	2	-
k-element	k	-
Facet	-	1
Cell	-	0

Table A.1: Dimensional characterisation of domain elements.

Definition 12. *A set $S \subseteq \mathcal{T}$ is called a* segment*. Note that we do not require a segment to be in any sense connected, although this is usually the case.*

Definition 13. *A set of pairwise disjoint segments S_i such that $\bigcup_{i \in I} S_i = \Omega$ is called a domain.*

Similar to polytopes, we have to label elements of dimension $k \leq n$ within our domain:

Definition 14. *A* k-element *of a domain D is a quasi-polytope of dimension k that is member of at least one cell in the domain.*

A rule of thumb for the characterisation of k-elements is the following: A k-element is a geometric object that can be parameterised using k parameters of finite range each.

Appendix B

Mathematical Tools

The proof of theorem 4 requires rather involved mathematical tools. For a full coverage of all mathematical backgrounds we refer the interested reader to the literature ([4],[9],[25]). The most important tools are summarised in the following.

Definition 15. *Let $\Omega \subset \mathbb{R}^n$ or $\Omega \subset \mathbb{C}^n$, m be a non-negative integer, $1 \leq p \leq \infty$. The* Sobolev-space *$H^{m,p}(\Omega)$ is then defined as*

$$H^{m,p}(\Omega) := \{\, u \in L^p(\Omega) \mid D^\alpha \in L^p(\Omega) \text{ for } |\alpha| \leq m \} \ , \tag{B.1}$$

where D^α denotes the differentiation operator with respect to a multi-index α.

In case $p = 2$, it can be shown that $H^{m,2}(\Omega)$ is a Hilbert space with scalar product

$$(u, v)_m = \sum_{|\alpha| \leq m} \int_\Omega D^\alpha \overline{u(\boldsymbol{x})} D^\alpha v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \ . \tag{B.2}$$

We also write $H^m(\Omega)$ instead of $H^{m,2}(\Omega)$.

Theorem 5 (Young's Inequality)**.** *For $1 < p, q < \infty$ with $\frac{1}{p} + \frac{1}{q} = 1$ and non-negative functions f and g there holds*

$$fg \leq \frac{f^p}{p} + \frac{g^q}{q} \ . \tag{B.3}$$

Young's inequality is frequently used with $p = q = 2$. Additionally, a free parameter $\varepsilon > 0$ is often used, such that Young's inequality becomes

$$fg = (\varepsilon^{1/p} f)(\varepsilon^{-1/p} g) \leq \varepsilon \frac{f^p}{p} + \varepsilon^{-q/p} \frac{g^q}{q} \ . \tag{B.4}$$

Theorem 6 (Hölder's Inequality)**.** *Let $1 < p, q < \infty$ with $\frac{1}{p} + \frac{1}{q} = 1$, $f \in L^p$ and $g \in L^q$. Then*

$$\|fg\|_{L^1} \leq \|f\|_{L^p} \|g\|_{L^q} \tag{B.5}$$

In the special case $p = q = 2$, one obtains the *Cauchy-Schwarz inequality* from Hölder's inequality.

Theorem 7 (Trace Theorem)**.** *Let $\Omega \subset \mathbb{R}^n$ be a bounded Lipschitz-domain, $1 < p < \infty$ and $s > 1/p$. The trace operator γ defined as*

$$\gamma : u \mapsto u|_\Gamma \ , \qquad u \in C^\infty(\overline{\Omega}) \ , \tag{B.6}$$

on $C^\infty(\overline{\Omega})$ can be uniquely extended to a continuous operator from $H^{s,p}(\Omega)$ to $H^{s-1/p,p}(\Gamma)$, thus

$$\|\gamma u\|_{H^{s-1/p,p}(\Gamma)} \leq C \|u\|_{H^{s,p}(\Omega)} \ , \qquad u \in H^{s,p}(\Omega) \ , \tag{B.7}$$

with a constant $C = C(\Omega, p)$ independent of u.

For reference, we give the existence and uniqueness result given for example in the book of Renardy and Rogers [25] adapted to the requirements in Theorem 4

Theorem 8. *Let $V \subset H \subset V^*$, $f \in L^2((0,T),V^*)$, $u_0 \in H$, $A \in \mathcal{L}(V,V^*)$ and the quadratic form associated with A,*

$$a(u,v) = -(Au,v) \tag{B.8}$$

defined on $V \times V$ be coercive, i.e.

$$a(u,u) \geq a\|u\|_V^2 - b\|u\|_H^2 \ . \tag{B.9}$$

Then, the evolution problem

$$\frac{\partial u}{\partial t} = Au + f(t), \quad u(0) = u_0 \tag{B.10}$$

has a unique solution $u \in L^2((0,T),V) \cap H^1((0,T),V^*)$ (interpreted in the sense of V^*-valued distributions).

The following lemma from the same textbook [25] is also of interest:

Lemma 3. *Suppose that $u \in L^2((0,T),V) \cap H^1((0,T),V^*)$. Then, in fact, $u \in C([0,T],H)$.*

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] R. A. Adams. *Sobolev Spaces*. Academic Press, New York, 1975.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[4] O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems: Theory and Computation*. Academic Press, Orlando, Fla., 1984.

[5] AztecOO - Object-Oriented Aztec Linear Solver Package . Internet: `http://trilinos.sandia.gov/packages/aztecoo/`.

[6] I. Babuška and T. Strouboulis. *The Finite Element Method and its Reliability*. Numerical Mathematics and Scientific Computation. Oxford University Press, 2001.

[7] P. Becker. Working Draft, Standard for Programming Language C++ (N2798). Technical report, C++ Standards Committee, 2008. Internet: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf`.

[8] J. Bey. *Finite-Volumen- und Mehrgitter-Verfahren für elliptische Randwertprobleme*. B. G. Teubner, 1998.

[9] D. Braess. *Finite Elements. Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, Cambridge, 1997.

[10] F. Cirak and J. C. Cummings. Generic Programming Techniques for Parallelizing and Extending Procedural Finite Element Programs. *Eng. with Comput.*, 24(1):1–16, 2008.

[11] C++0x Language Support in GCC . Internet: `http://gcc.gnu.org/projects/cxx0x.html`.

[12] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston et al., 2000.

[13] Deal.ii. Internet: `http://www.dealii.org/`.

[14] C. C. Douglas, G. Haase, and U. Langer. *Tutorial on Elliptic PDE Solvers and Their Parallelization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[15] Fenics. Internet: `http://www.fenics.org/`.

[16] Free Finite Element Package . Internet: `http://ffep.sourceforge.net/`.

[17] Getfem++. Internet: `http://home.gna.org/getfem/`.

[18] W. Hackbusch. *Multigrid Methods and Applications*. Springer-Verlag, Berlin, 1985.

[19] Ch. Hollauer, H. Ceric, G. van Barel, A. Witvrouw, and S. Selberherr. Investigation of Intrinsic Stress Effects in Cantilever Structures. *Nano/Micro Engineered and Molecular Systems, 2007. NEMS '07. 2nd IEEE International Conference on*, pages 151–154, Jan. 2007.

[20] Kaskade. Internet: `http://www.zib.de/Numerik/numsoft/kaskade/index.en.html`.

[21] H. P. Langtangen. *Computational Partial Differential Equations*. Number 2 in Lecture Notes in Computational Science and Engineering. Springer, Berlin, 1999.

[22] F. Lau, L. Mader, C. Mazure, Ch. Werner, and M. Orlowski. A model for phosphorus segregation at the silicon-silicon dioxide interface. *Applied Physics A: Materials Science & Processing*, 49(6):671–675, 1989.

[23] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, 1999.

[24] W. Rachowicz, J. T. Oden, and L. Demkowicz. Towards a Universal hp Adaptive Finite Element Strategy. *Comput. Methods Appl. Mech. Engrg.*, 77:181–212, 1989.

[25] M. Renardy and R. C. Rogers. *An Introduction to Partial Differential Equations*, volume 13 of *Texts in Applied Mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2004.

[26] K. Rupp. Solving PDEs in Electromigration using a Generic Template-Metaprogramming Framework. Master's thesis, Brunel University, 2007.

[27] P. Ŝolín, K. Segeth, and I. Doleẑel. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC, 2004.

[28] Sundance 2.3. Internet: `http://www.math.ttu.edu/~klong/Sundance/html/`.

[29] U. Trottenberg, C. W. Oosterlee, and A. Schuller. *Multigrid*. Academic Press, December 2000.

[30] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[31] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[32] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method - Volume 1: The Basis*. Butterworth-Heinemann, 5th edition, 2000.

Index