Eidgenössische Technische Hochschule Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Concepts - An Object-Oriented Software Package for Partial Differential Equations

Ph. Frauenfelder and Ch. Lage[†]

Research Report No. 2002-09
July 2002

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

[†]Funston Avenue, San Franciso, CA 94100, USA

Concepts - An Object-Oriented Software Package for Partial Differential Equations

Ph. Frauenfelder and Ch. Lage[†]

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Research Report No. 2002-09 July 2002

Abstract

Object oriented design has proven itself as a powerful tool in the field of scientific computing. Several software packages, libraries and toolkits exist, in particular in the FEM arena that follow this design methodology providing extensible, reusable, and flexible software while staying competitive to traditionally designed point tools in terms of efficiency.

However, the common approach to identify classes is to turn data structures and algorithms of traditional implementations into classes such that the level of abstraction is essentially not raised. In this paper we discuss an alternative way to approach the design challenge which we call "concept oriented design". We apply this design methodology to Petrov-Galerkin methods leading to a class library for both, boundary element methods (BEM) and finite element methods (FEM). We show as an particular example the implementation of *hp*-FEM using the library with special attention to the handling of inconsistent meshes.

Keywords: object oriented design, concept oriented design, *hp*-FEM, adaptivity

Subject Classification: 35-04, 65-04, 65N30, 65N50, 68N05

[†]Funston Avenue, San Franciso, CA 94100, USA

Introduction

Object oriented design and analysis [4] have proven themselves as a powerful tool in the field of scientific computing. Several software packages, libraries and toolkits exist, in particular in the FEM arena that follow this design methodology providing extensible, reusable, and flexible software while staying competitive to traditionally designed point tools in terms of efficiency.

However, the common approach to identify classes is to turn data structures and algorithms of traditional implementations into classes such that the level of abstraction is essentially not raised. An alternative way to identify reusable classes is *concept oriented design* proposed in [10]. This design methodology exploits the mathematical structure to find an appropriate modularisation. We explain the ideas of concept oriented design shortly in Section 1 and show its application to Finite Elements in Section 2.

For our *hp*-FEM code, it is crucial that hanging nodes can be handled. Otherwise, implementing adaptive methods is quite complicated. In the present work, a new method to deal with hanging nodes in conforming FE spaces is presented. In contrast to methods which are widely used today, our approach offers both, more flexibility and extensibility in the variety of meshes which are supported and in the spatial dimension which can be handled. We present the theory and the software design in Section 3.

We use a software package called *Concepts* developed at the Seminar for Applied Mathematics (SAM) of the Swiss Federal Institute of Technology (ETH), Zurich, following the guidelines of concept oriented design applied to Petrov-Galerkin methods. The package is based on [9].

At the SAM several projects have been realised using the library: generalised FEM in two dimensions (Ana-Maria Matache, [12]), linear elasticity (Giacomo Catenazzi, David Hoch, Andreas Rüegg), discontinuous Galerkin finite elements (Philipp Frauenfelder), boundary elements including clustering and wavelet based acceleration methods (Gregor Schmidlin, Christian Lage).

1. Software Design

Most software projects treating numerical problems have among their general aims extensibility, re-usability, flexibility and performance. With the increasing complexity of the problems which can be solved by computers, the complexity of the needed software is growing, too. Solving a complex problem is only possible if it is fragmented, i.e. the software has a modular design and the modules can be designed and coded more or less separately from each other. Using a modular design, the whole software package is built from loosely coupled modules which make exchanging a module or adding modules easy yielding extensibility and re-usability.

Flexibility can be achieved by designing generic modules which describe the properties of a whole set of modules. This is an important feature which distinguishes object oriented methods from traditional (structured) methods. These generic modules fix the interface to a set of modules which, in return, are specialisations of the generic modules.

Essentially, object oriented methods provide two methods to describe the relation of generic and specialised modules: *polymorphism* (inheritance, overloading) and *parametrised types* (templates).

In the following, we give a brief summary of [10].

In the section above, the necessity to have generic and specialised modules was described. Often, it is not evident how to identify those generic modules. In the structured programming world, a top-down or bottom-up approach was used to identify the different modules in the designing process of a software package.

The main problem of the top-down approach: since the modules are produced in the scope of their generating problem it is difficult to reuse them in a different context. On the other hand, the bottom-up approach makes it difficult to combine the resulting modules. For a new application of the software, it might be necessary to redesign several modules. This makes the design method unusable.

The above mentioned methods do not give us a satisfying recipe. Identifying the modules is one of the crucial points in object-oriented software design, though.

However, since we are interested in the development of numerical software, there is a special situation: the considered numerical methods are already formulated in an abstract way based on hierarchical structured mathematical concepts. This motivates the following approach: represent each concept by a module and

combine these modules according to the numerical algorithm to generate an implementation. This defines *concept oriented* modularisation.

Since the mathematical formulation is quite stable and reusable already, chances are quite high that these properties carry over to the modularisation in the software. Another great advantage of concept oriented design is: Mathematicians and other researchers will find the concepts familiar.

2. The Software Package Concepts

The design ideas and principles shown in the previous section are used to design the software *Concepts* [5]. Concepts is a class library nearly completely written in C++ [14]. We are working with Concepts at our institute with boundary element methods, *hp*-finite element methods and generalised finite element methods. The following subsections outline the main classes of the software by means of a very general problem. In addition, we show an application of the classes to *hp*-FEM.

The diagrams in this section follow the standard of the Unified Modelling Language UML [13].

2.1. Mathematical Concepts.

The mathematical concepts used for the design of Concepts are those of FEM and BEM or, more general, those of Petrov-Galerkin discretisation methods. In the following, we briefly recall the terminology of these methods:

In order to find a solution u of

$$Lu = f \tag{2.1}$$

where L denotes an operator and the function f a given right hand side, we rewrite (2.1) in variational form:

Find $u \in V$ such that

$$a(u, v) = l(v) \quad \forall v \in W, \tag{2.2}$$

with V, W function spaces, $a(\cdot, \cdot)$ a bilinear form and $l(\cdot)$ a linear form. A discretisation of (2.2) is obtained by replacing V and W with finite dimensional subspaces V_N and W_N, respectively.

The discretised problem may be written as a linear system of equations by choosing a suitable basis for each of the subspaces. To keep the discussion focused, we assume the same basis $\{\Phi_1, ..., \Phi_N\}$ for both spaces V_N and W_N, and obtain

$$A\,\boldsymbol{u} \;=\; \boldsymbol{l} \tag{2.3}$$

with $(A)_{ij} := a(\Phi_j, \Phi_i)$, $(\boldsymbol{l})_i := l(\Phi_i)$ and $\boldsymbol{u}$ the coefficient vector of the discrete solution.

The mathematical concepts used above are easily listed: operator, function, bilinear form, linear form, (sub)space, basis function, matrix, vector. Mapping these concepts into classes, however, rises the problem to represent functions, in particular, basis functions. The standard way to approach this problem is to decompose (mesh) the domain Ω of the function into primitive sets K_i (elements). These sets themselves can be characterised by applying mappings F_{K_i} to predefined reference sets (cells), e.g. $F_{K_i} : \hat{K} \to K_i$ such that $K_i = F_{K_i}(\hat{K})$. In addition, functions N_j mounted on a reference set define so-called shape functions $\phi_j^{K_i}$ on each of the elements K_i via $\phi_j^{K_i} \circ F_{K_i} = N_j$.

Functions may now be implemented by specifying their restriction to each of the elements K_i by means of linear combinations of shape functions.

Definition 1 (T-matrix)**.** *Let m_K be the number of shape functions $\left\{\phi_j^K\right\}_{j=1}^{m_K}$ attached to element K and N the number of basis functions $\{\Phi_i\}_{i=1}^N$. The T-matrix $T_K \in \mathbb{R}^{m_K \times N}$ of element K is implicitly defined by*

$$\Phi_i|_K = \sum_{j=1}^{m_K} [T_K]_{ji}\, \phi_j^K$$

or in vector notation: $\boldsymbol{\Phi}|_K = T_K^\top \boldsymbol{\phi}^K$.

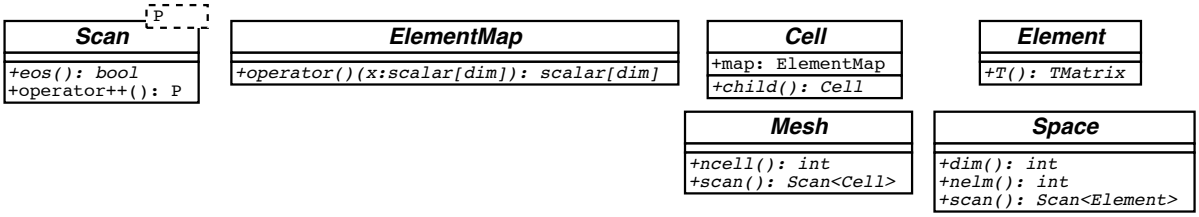

FIGURE 1. The classes in Concepts representing mesh, space and elements.

The element-by-element representation of functions carries over to the specification of linear and bilinear forms:

$$\boldsymbol{l} = l(\boldsymbol{\Phi}) = l\left(\sum_{\tilde{K}} T_{\tilde{K}}^{\top} \boldsymbol{\phi}^{\tilde{K}}\right) = \sum_{\tilde{K}} T_{\tilde{K}}^{\top} l(\boldsymbol{\phi}^{\tilde{K}}) = \sum_{\tilde{K}} T_{\tilde{K}}^{\top} \boldsymbol{l}_{\tilde{K}} \tag{2.4}$$

and

$$A = a(\boldsymbol{\Phi}, \boldsymbol{\Phi}) = a\left(\sum_{K} T_K^{\top} \boldsymbol{\phi}^K, \sum_{\tilde{K}} T_{\tilde{K}}^{\top} \boldsymbol{\phi}^{\tilde{K}}\right) = \sum_{K,\tilde{K}} T_{\tilde{K}}^{\top} a(\boldsymbol{\phi}^K, \boldsymbol{\phi}^{\tilde{K}}) T_K = \sum_{K,\tilde{K}} T_{\tilde{K}}^{\top} A_{\tilde{K}K} T_K \tag{2.5}$$

where the element vector $\boldsymbol{l}_{\tilde{K}}$ and the element matrix $A_{\tilde{K}K}$ are given by

$$(\boldsymbol{l}_{\tilde{K}})_i := l(\phi_i^{\tilde{K}}), \quad \text{and} \quad (A_{\tilde{K}K})_{ij} := a(\phi_j^K, \phi_i^{\tilde{K}}) \tag{2.6}$$

with $i = 1, \ldots, m_{\tilde{K}}$ and $j = 1, \ldots, m_K$.

2.2. **Fundamental Classes.** In this section, we show the classes in Concepts which realize the mathematical concepts presented in the previous section. An application in Concepts typically performs the following steps:

(1) Build a meshed domain of interest. A mesh is built from cells which contain the element maps.
(2) Build a space on the mesh. The space creates the elements. Typical for FEM and BEM: the elements are associated to cells in the mesh.
(3) Build the system matrix A and the system vector $\boldsymbol{l}$ from the element contributions computed by the bilinear and linear forms respectively.
(4) Solve the resulting linear system $A\boldsymbol{u} = \boldsymbol{l}$ with the solver.

In Concepts, there are classes for the space, the mesh and the elements: see Figure 1. The main member of the classes *Mesh* and *Space* is *scan()*. In both classes, it returns a *Scan<P>* with the template parameter P set accordingly. The instance of *Scan* is a scanner over the space or the mesh and makes it possible to loop over the elements of the space or the cells of the mesh. Typically, the constructor of a space loops over all cells of the mesh and creates an associated element. The element map $F_K : \hat{K} \to K$ is realized by *ElementMap*.

The most important member of the element is *T*: It returns the T-matrix of an element. The T-matrix is used to assemble the local shape functions into to global basis functions of the space whom the element belongs to. Figure 3 shows the assembly of a load vector.

Figure 2 shows the classes of Concepts for the bilinear and linear form, the system matrix and system vector and the linear solver. The constructor of the system matrix SystemMatrix takes as arguments a space and a bilinear form. It assembles the system matrix by looping over the elements of the space and calling the application operator operator() of the bilinear form on the elements. The same is done in the constructor of the system vector Vector. By using these abstract class declarations, it is possible to explicitly implement the assembly operator: Figure 3 shows the constructor of Vector, i.e. the assembling of the global load vector—the assembly operator of a system matrix looks similar. The solver is, like SystemMatrix, derived from the general class *Operator*. *Operator* is the realization of the general concept of an operator which maps a vector from one

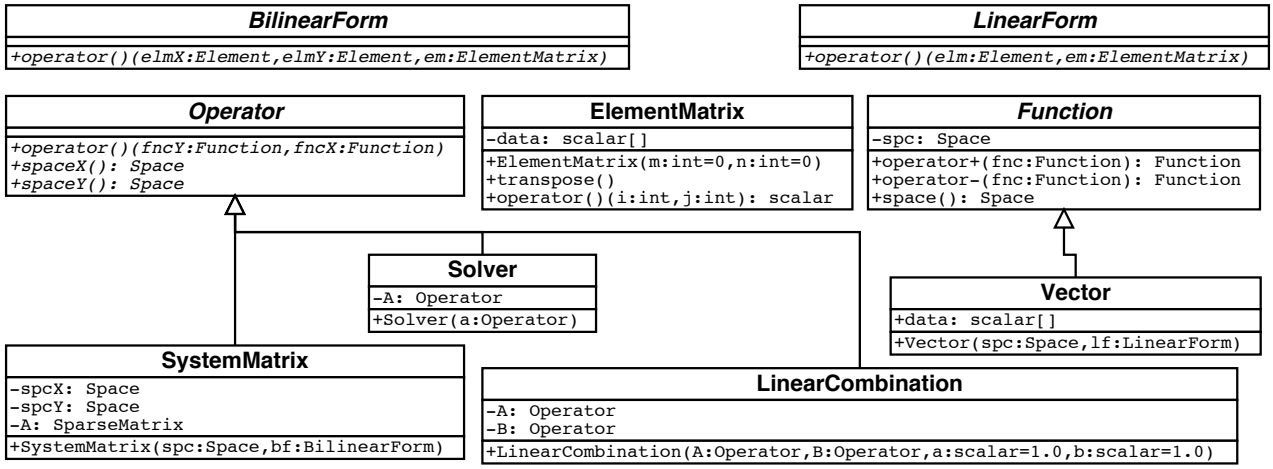

FIGURE 2. The classes in Concepts for bilinear and linear forms, system matrix and system vector and the solver.

```
Vector::Vector(const Space& spc, LinearForm& lf) : v_(new F[n_]) {
  memset(v_, 0, n_ * sizeof(v_[0]));                // set all elements of the system vector to 0

  ElementMatrix A(3, 1), B(3, 1);                   // initialize 2 element matrices

  std::auto_ptr<Space::Scan> sc(space().scan());    // get a scanner to loop over the space
  while ( *(sc.get()) ) {                           // the loop ends after all elements are treated
    Element& elm = (*sc)++;                         // get the current element
    const TMatrixBase& T = elm.T();                 // get the T-matrix of the element
    lf(elm, A);                                     // evaluate the linear form
    A.transpose(); T(A, B); B.transpose();          // apply the T-matrix
    for(int i = T.n(); i--;) {
      v_[T.index(i)] += B(i,0);                     // add the element's contribution into the system vector
    }
  }
}
```

FIGURE 3. Constructor of the class Vector which implements the assembly operator for a load vector. This does not depend on any particular implementation but only on the abstract classes.

space to another. A solver fits into this concept, too. In practice, the classes SystemMatrix and Solver are just an interface to PETSc [1, 2, 3].

2.3. **Application to *hp*-FEM: concrete classes in Concepts.** In this section, an application of the mathematical concepts and the classes shown in the Sections 2.1 and 2.2 is presented.

Consider the following scalar model problem:

$$\begin{aligned} -\Delta u + u &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega. \end{aligned}$$

Integrating over Ω and integrating by parts results in a variational formulation which is discretised with a FE space $V_N \subset H_0^1(\Omega)$, e. g. $V_N = S_0^{1,\boldsymbol{p}}(\Omega, \mathcal{T})$, the space of continuous, piecewise polynomials of degree p_K

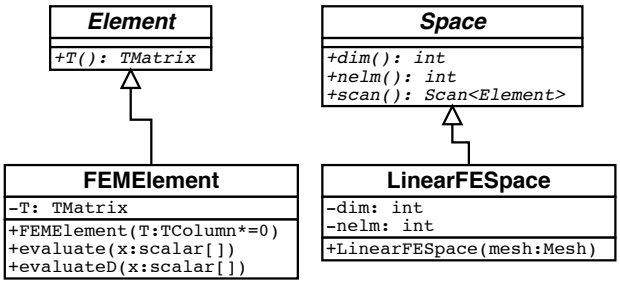

FIGURE 4. The FE space $S_0^{1,\boldsymbol{p}}(\Omega, \mathcal{T})$ implemented in LinearFESpace and the element implemented in FEMElement.

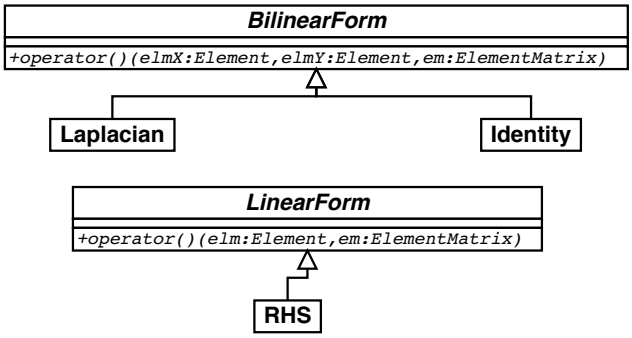

FIGURE 5. The bilinear forms for A and I implemented in Laplacian and Identity respectively. The linear form for $\boldsymbol{l}$ implemented in RHS.

functions on the mesh $\mathcal{T}$ with zero boundary values.[1] The discrete variational problem reads:
Find $u_{\mathrm{FE}} \in S_0^{1,\boldsymbol{p}}(\Omega, \mathcal{T})$ such that

$$\int_\Omega (\nabla u_{\mathrm{FE}} \cdot \nabla v_{\mathrm{FE}} + u_{\mathrm{FE}} v_{\mathrm{FE}})\, dx = \int_\Omega f v_{\mathrm{FE}}\, dx \quad \forall v_{\mathrm{FE}} \in S_0^{1,\boldsymbol{p}}(\Omega, \mathcal{T}).$$

Opposed to (2.2), $V_N = W_M$ here. Let $\{\Phi_i\}_{i=1}^N$ be a basis of $S_0^{1,\boldsymbol{p}}(\Omega, \mathcal{T})$ and $\boldsymbol{u}$ the coordinates of u_{FE} in this basis. Plugging $u_{\mathrm{FE}} = \boldsymbol{u}^\top \boldsymbol{\Phi}$ into the variational formulation yields

$$(A + I)\boldsymbol{u} = \boldsymbol{l}$$

where A is the stiffness matrix originating from the Laplacian and its bilinear form

$$a_\Omega(u, v) = \int_\Omega \nabla u \cdot \nabla v\, dx.$$

I is the mass matrix originating from the absolute term and its bilinear form

$$b_\Omega(u, v) = \int_\Omega uv\, dx.$$

Finally, define $L = A + I$ and solve the linear system $L\boldsymbol{u} = \boldsymbol{l}$ for $\boldsymbol{u}$.

With this background, the necessary classes can be found easily. The specialisations of *Space* and *Element* are shown in Figure 4. The bilinear forms to compute A and I are depicted in Figure 5. The realisation of A and I are combined using LinearCombination from Figure 2—eventually, the solver is initialised and applied to $\boldsymbol{l}$ to get $\boldsymbol{u}$. An application in pseudo code looks like in Figure 6.

3. Generation of T-Matrices

Basis functions are implemented using T-matrices, i.e. the generation of T-matrices defines a set of basis functions and therefore the subspace used to discretise the variational formulation. Depending on the collection of shape functions and properties of the mesh several construction rules exist. Consistent meshes, i.e. meshes with properly aligned vertices, edges and faces, allow construction rules specifying shape functions to be "glued" together in order to form basis function. Associated T-matrices usually have at most one entry equal to 1 in each row and column which defines a simple index mapping operation (Example 2).

[1] On triangles and tetrahedra, we use $\mathcal{P}_{p_K}$ (the space of polynomials of total degree p_K) as the polynomial space for element K. On quadrilaterals and hexahedra, we use $\mathcal{Q}_{p_K}$ (the space of polynomials of maximal degree p_K) as the polynomial space for element K. On prisms, we use a tensor product of $\mathcal{P}_{p_K}$ (for the triangular base) with $\mathcal{Q}_{p_K}$ as the polynomial space for element K.

```
Domain mesh;
LinearFESpace space(mesh);  // elements and T-matrices are generated

Laplacian bfa;
SystemMatrix A(space, bfa); // computing and assembling the stiffness matrix
Identity bfi;
SystemMatrix I(space, bfi); // computing and assembling the mass matrix

RHS lf;
Vector l(space, lf);
Vector u(space);            // empty solution vector

LinearCombination L(A, I);
Solver Linv(L);
Linv(f,u);                  // solve linear system
```

FIGURE 6. An application in pseudo code.

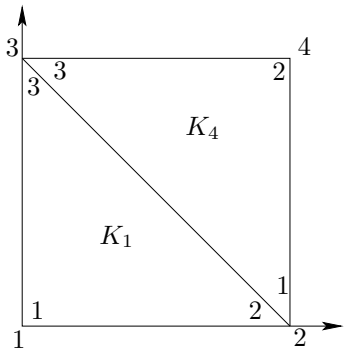

FIGURE 7. Consistent mesh with two elements with three local shape functions each and four global basis functions.

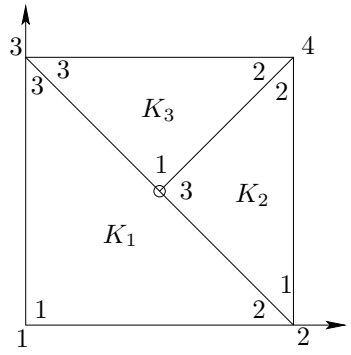

FIGURE 8. Inconsistent mesh with three elements with three local shape functions each and four global basis functions. The hanging node is marked with ∘.

Construction rules can be formally defined utilising the concept of *degrees of freedom*. However, this does not provided an efficient algorithm to generate basis functions. These algorithms depend strongly on the properties of given meshes and the applicable sets of shape functions. Therefore, a generalisation of such an algorithm is not possible.

In Concepts we deal with this problem by introducing specialisations of the abstract base class *Space*, each implementing way of generating basis functions (e.g. class LinearFESpace). Since the assembly of vectors and matrices only depends on the interface defined by the class *Space* and the general concept of a T-matrix to define basis functions (see Figure 3) new construction rules in form of concrete classes for spaces, may be added without changing existing code.

Example 2. *Consider the consistent mesh shown in Figure 7. Assuming standard linear nodal shape functions, the elements K_1 and K_4 have the T-matrices*

$$T_{K_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \textit{and} \quad T_{K_4} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.1}$$

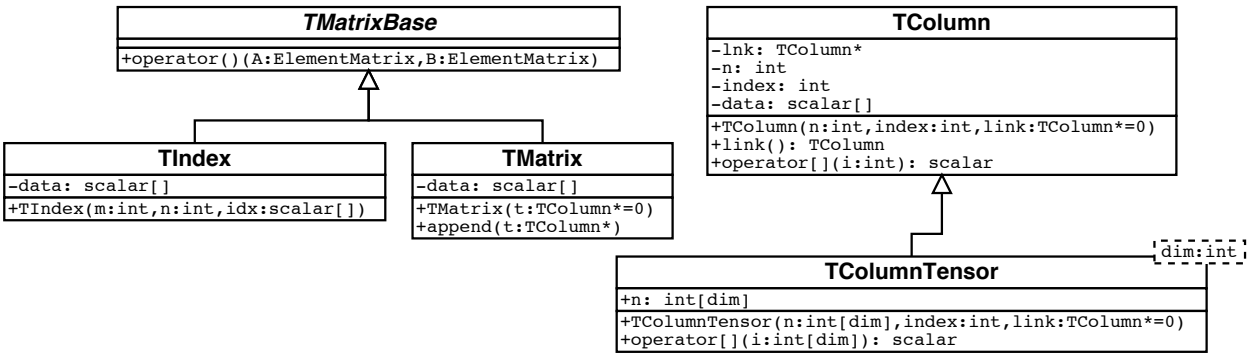

FIGURE 9. Classes for a T-matrix: T-matrices are built from T-columns. There are two specializations of *TMatrixBase*: TIndex which is for simple meshes and TMatrix which is for general meshes. TColumnTensor is a tensorized view of TColumn.

specifying four continuous basis functions. In case of the inconsistent mesh shown in Figure 8, the T-matrices of element K_2 and K_3 are:

$$T_{K_2} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1/2 & 1/2 & 0 \end{pmatrix} \quad \textit{and} \quad T_{K_3} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{3.2}$$

In case of consistent meshes the implementation of construction rules depends heavily on counting and assigning indices with respect to topological entities such as vertices, edges and faces.[2] To support this, Concepts provides the user with unique identifiers for basic topological entities. These identifiers may be used to associate information necessary for the construction with these entities using data structures like arrays and maps. Note that this kind of association is temporary, i.e. the information is no longer needed, after basis functions and T-matrices, respectively, are constructed. Moreover, with this approach it is not necessary to pollute topological and geometrical data structures with construction specific data fields which improves the encapsulation of modules.

The concept of T-matrices is captured in an abstract base class, which provides among others a multiplication operation with element matrices. Specialisation of this interface include the implementation as index mapping operation in order to keep the application of T-matrices efficient in case of consistent mesh discretisations and the implementation of T-matrices as general matrix (class *TMatrix* shown in Figure 9) to cover non-consistent or nonstandard meshes. However, T-matrices are in general highly sparse matrices with only a few nonzero columns. Therefore, instances of *TMatrix* are created column-wise as TColumns and TColumnTensors, respectively.

The handling of inconsistent meshes involves more complex construction rules for basis functions compared to the rules for consistent meshes. In addition, in many cases one has to distinguish several situations such as single constraint hanging nodes and double constraint nodes.

In the following we present an approach that extends simple construction rules for consistent meshes to the inconsistent case. The assumption made for this approach is that the inconsistency (hanging nodes) is introduced due to recursive refinements of an initially consistent mesh. For example, the mesh in Figure 8 is a

[2]The user who implements a concrete FE space can choose what should be done if two neighbouring elements have different polynomial degrees. There are two possibilities:

- the set of local shape functions on the element with the lower polynomial degree is enriched on the shared interface or
- the set of shape function on the other element (the one with the higher polynomial degree) is diluted, i.e. the shape functions on the shared interface which do not have a matching partner on the neighbouring element are dropped.

It is possible to implement both variants in Concepts.

refined version of the mesh in Figure 7 and the meshes in Figure 11 result from a mesh consisting of a single quadrilateral. This assumption is met for example by methods offering local adaptivity.

Consider a mesh $\mathcal{M}$ for which all elements and associated T-matrices have been generated. Suppose the mesh $\mathcal{M}'$ is the result of splitting several elements of $\mathcal{M}$. The basis functions $B := \{\Phi_1, \ldots, \Phi_N\}$ defined for $\mathcal{M}$ may be partitioned into two sets – one denoted by B_{replace} containing all basis functions that can be described solely by elements of $\mathcal{M}'$ that are not part of $\mathcal{M}$ and another one denoted by B_{keep} representing the rest:

$$B = B_{\text{replace}} \cup B_{\text{keep}}.$$

Note that B_{replace} is easily determined; the support of basis functions in B_{replace} consists entirely of newly inserted elements.

The set of basis functions B' related to mesh $\mathcal{M}'$ contains all basis functions in B_{keep} plus an additional set B_{insert} of basis functions generated by consistent components of mesh $\mathcal{M}'$ formed by elements not part of $\mathcal{M}$:

$$B' = B_{\text{insert}} \cup B_{\text{keep}}.$$

In Example 2 we find $B = B'$ since $B_{\text{insert}} = \emptyset$ whereas for the mesh shown on the left in Figure 11, basis functions related to newly created vertices on the boundary as well as the basis function related to the center vertex form B_{insert} in each refinement step. In addition, the basis function associated with the lower left corner will be replaced.

The construction of columns of T-matrices for basis functions in B_{insert} follows the rules for consistent meshes and may be implemented using standard FEM techniques. For functions in B_{keep}, however, there already exists a T-matrix column with respect to mesh $\mathcal{M}$. These columns are easily converted into T-matrix columns with respect to mesh $\mathcal{M}'$ by means of so-called S-matrices (see Proposition 4).

3.1. **S-Matrices.**

Definition 3 (S-matrix)**.** *Let $K' \subset K$ be the result of a refinement of element K. The S-matrix $S_{K'K} \in \mathbb{R}^{m_{K'} \times m_K}$ is defined by*

$$\phi_j^K\big|_{K'} = \sum_{l=1}^{m_{K'}} [S_{K'K}]_{lj}\, \phi_l^{K'}$$

and in vector notation: $\boldsymbol{\phi}^K\Big|_{K'} = S_{K'K}^\top \boldsymbol{\phi}^{K'}$, respectively. It represents the restriction of the shape functions ϕ_j^K onto K' as linear combination of the shape functions $\left\{\phi_l^{K'}\right\}_{l=1}^{m_{K'}}$ of K'. For $K = K'$, i.e. no refinement, the S-matrix S_{KK} is equal to the identity matrix.

Proposition 4. *Let $K' \subset K$ be the result of a refinement of an element K. Then, the T-matrix of K' can be computed as*

$$T_{K'} = S_{K'K}\tilde{T}_K + \tilde{T}_{K'}$$

where $\tilde{T}_K$ denotes the T-matrix of element K with columns not related to functions in B_{keep} set to zero and $\tilde{T}_{K'}$ the T-matrix for functions in B_{insert} with respect to K'.

Proof. Let $K' \subset K$ and $\Phi_i \in B_{\text{keep}}$. Then

$$\Phi_i|_{K'} = \Phi_i|_K|_{K'} = \sum_{j=1}^{m_K} [T_K]_{ji}\, \phi_j^K\big|_{K'} = \sum_{j=1}^{m_K} [T_K]_{ji} \sum_{l=1}^{m_{K'}} [S_{K'K}]_{lj}\, \phi_l^{K'},$$

i.e. $[T_{K'}]_i = S_{K'K}\,[T_K]_i = S_{K'K}\left[\tilde{T}_K\right]_i$. For $\Phi_i \in B_{\text{insert}}$ the assertion holds by definition. □

S-matrices do not depend on the exact geometry of elements but only on topology and subdivision ratio of the refinement.

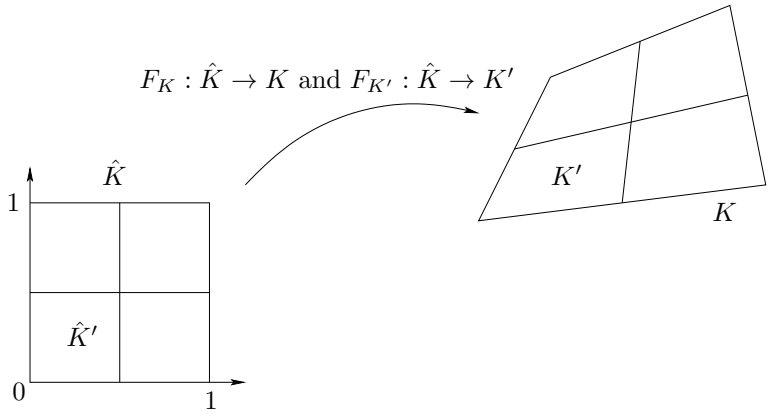

FIGURE 10. Element maps for K and K'.

Proposition 5. *Let $\hat{K}' \subset \hat{K}$ be the result of a refinement of the reference element $\hat{K}$ with $H : \hat{K} \to \hat{K}'$ the subdivision map (see Figure 10). The element maps are $F_K : \hat{K} \to K$ and $F_{K'} : \hat{K} \to K'$ and*

$$F_{K'} \circ H^{-1} = F_K \tag{3.3}$$

holds.

Then, $S_{\hat{K}'\hat{K}} = S_{K'K}$.

Proof. The local element shape functions and the reference element shape functions are connected by the element map:

$$\begin{aligned} \phi_j^K \circ F_K &= N_j, \\ \phi_j^{K'} \circ F_{K'} &= N_j. \end{aligned} \tag{3.4}$$

Using Definition 3:

$$\phi_j^K\big|_{K'} = \sum_{l=1}^{m_{K'}} [S_{K'K}]_{lj}\, \phi_l^{K'}.$$

Taking the local element shape functions back to the reference element $\hat{K}$ by F_K yields

$$\phi_j^K \circ F_K\big|_{\hat{K}'} = \sum_{l=1}^{m_{K'}} [S_{K'K}]_{lj}\, \phi_l^{K'} \circ F_K.$$

Using (3.3) and (3.4), it follows:

$$N_j|_{\hat{K}'} = \sum_{l=1}^{m_{K'}} [S_{K'K}]_{lj}\, N_l \circ H^{-1}. \tag{3.5}$$

Comparing (3.5) and the definition of the S matrix $S_{\hat{K}'\hat{K}}$

$$N_j|_{\hat{K}'} = \sum_{l=1}^{m_{\hat{K}'}} \left[S_{\hat{K}'\hat{K}}\right]_{lj} N_l \circ H^{-1}$$

concludes the proof. □

Example 6 (Examples of Meshes)**.** *The meshes in Figure 11 can be handled by Concepts using quadrilaterals and a subdivision strategy which creates four children out of an element. The subdivision ratio used here is 1/2, i.e. the children all have the same size.*

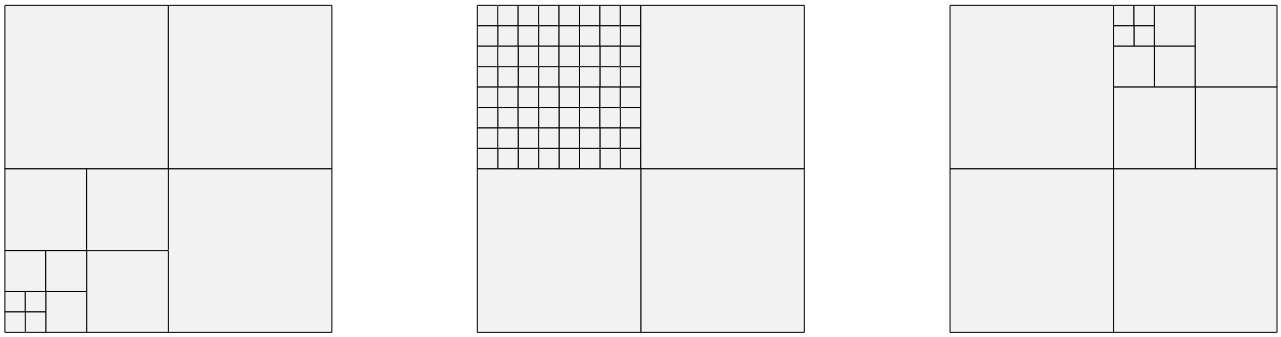

FIGURE 11. Meshes which can be handled by Concepts. The mesh in the middle is not handled by some sort of Mortar method. Instead, a continuous FE space is created on the whole computational domain, as it is the case for the other two meshes as well.

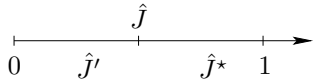

FIGURE 12. One dimensional reference element with left and right child.

The mesh on the left shows a geometrical refinement towards the lower left corner. This mesh has only single constrained nodes and could also be handled by an algorithm which is able to eliminate only single constrained nodes.

The mesh in the middle shows a quadrilateral (the top left one) which is refined three times recursively, i.e. it is divided into $2^3 \cdot 2^3 = 64$ *small quadrilaterals. This can no longer be handled by an algorithm which is able to eliminate only single constrained nodes. An S-matrix can be applied recursively as it was done for the subdivision algorithm, though.*

The right hand mesh does no look very different to the left hand mesh but the constrained nodes are not just single constrained. Again, the S-matrix has to be applied several times.

3.2. **Generation of S-Matrices.** If, in higher dimensions, the reference element shape functions are tensorised one dimensional reference element shape functions, the S matrices also have a tensor product structure. In the following, we restrict the discussion to reference elements which allow tensorised element shape functions: quadrilaterals and hexahedrons in two and three dimensions, respectively.

3.2.1. *S-Matrix in One Dimension.* In one dimension, the S-matrices can be computed by solving a linear system. See Figure 12.

$$\boldsymbol{N}|_{\hat{J}'} = S^{\top}_{\hat{J}'\hat{J}} \boldsymbol{N} \circ G^{-1}, \tag{3.6}$$

where $G : \hat{J} \to \hat{J}'$, $\xi \mapsto \xi/2$. Evaluating (3.6) in $m_{\hat{J}}$ distinct points in the interval $[0, 1/2]$ results in a linear system which can be solved for $S_{\hat{J}'\hat{J}}$. The same holds for $S_{\hat{J}^\star\hat{J}}$.

For the reference element shape functions

$$N_j(\xi) = \begin{cases} 1-\xi & j = 1 \\ \xi & j = 2 \\ \xi(1-\xi)P^{1,1}_{j-3}(2\xi-1) & j = 3, \dots, J \end{cases}$$

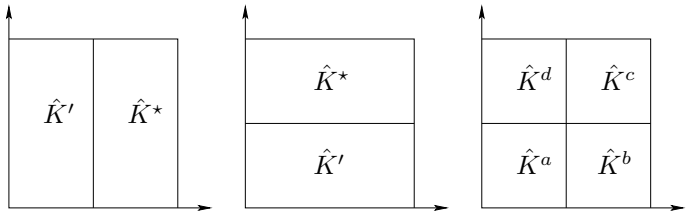

FIGURE 13. Variants of subdividing a quadrilateral.

the S-matrices are ($J = 4$):

$$S_{\hat{J}'\hat{J}} = \begin{pmatrix} 1/2 & 1/2 & 1/4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/4 & 3/4 \\ 0 & 0 & 0 & 1/8 \end{pmatrix} \text{ and } S_{\hat{J}^\star\hat{J}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1/2 & 1/2 & 1/4 & 0 \\ 0 & 0 & 1/4 & -3/4 \\ 0 & 0 & 0 & 1/8 \end{pmatrix}.$$

3.2.2. *S-Matrix in Two Dimensions.* The three different subdivisions shown in Figure 13 shall be considered. The reference element shape functions are tensorised versions of the one dimensional shape functions:

$$N_{i,j} = N_i \otimes N_j. \tag{3.7}$$

Consider the left subdivision variant in Figure 13 with the subdivision map

$$H : \hat{K} \to \hat{K}', \boldsymbol{\xi} \mapsto \begin{pmatrix} \xi_1/2 \\ \xi_2 \end{pmatrix}$$

By Definition 3, the S-matrix $S_{\hat{K}'\hat{K}}$ is defined as

$$N_{i,j}|_{\hat{K}'} = \sum_{k,l} \left[S_{\hat{K}'\hat{K}}\right]_{(k,l),(i,j)} N_{k,l} \circ H^{-1}.$$

Plugging (3.7) into this yields

$$(N_i \otimes N_j)|_{\hat{K}'} = \sum_{k,l} \left[S_{\hat{K}'\hat{K}}\right]_{(k,l),(i,j)} (N_k \otimes N_l) \circ H^{-1}. \tag{3.8}$$

The S-matrices for the one dimensional reference element shape functions used in (3.8) are

$$N_i|_{\hat{J}'} = \sum_m \left[S_{\hat{J}'\hat{J}}\right]_{mi} N_m \circ G^{-1} \qquad \text{for the } \xi_1 \text{ part and}$$

$$N_j = \sum_n [E]_{nj} N_n \qquad \text{for the } \xi_2 \text{ part.}$$

Plugging this into the left hand side of (3.8) gives:

$$\begin{aligned}(N_i \otimes N_j)|_{\hat{K}'} = N_i|_{\hat{J}'} \otimes N_j &= \sum_{m,n} \left(\left[S_{\hat{J}'\hat{J}}\right]_{mi} N_m \circ G^{-1}\right) \otimes \left([E]_{nj} N_n\right) \\ &= \sum_{m,n} \left[S_{\hat{J}'\hat{J}}\right]_{mi} \cdot [E]_{nj} N_m \circ G^{-1} \otimes N_n\end{aligned}$$

comparing with the right hand side of (3.8)

$$= \sum_{k,l} \left[S_{\hat{K}'\hat{K}}\right]_{(k,l),(i,j)} N_k \circ G^{-1} \otimes N_l$$

gives (using the definition of matrix tensor products of Fiedler [6]):

$$S_{\hat{K}'\hat{K}} = S_{\hat{J}'\hat{J}} \otimes E \qquad \text{for the left quadrilateral } \hat{K}'.$$

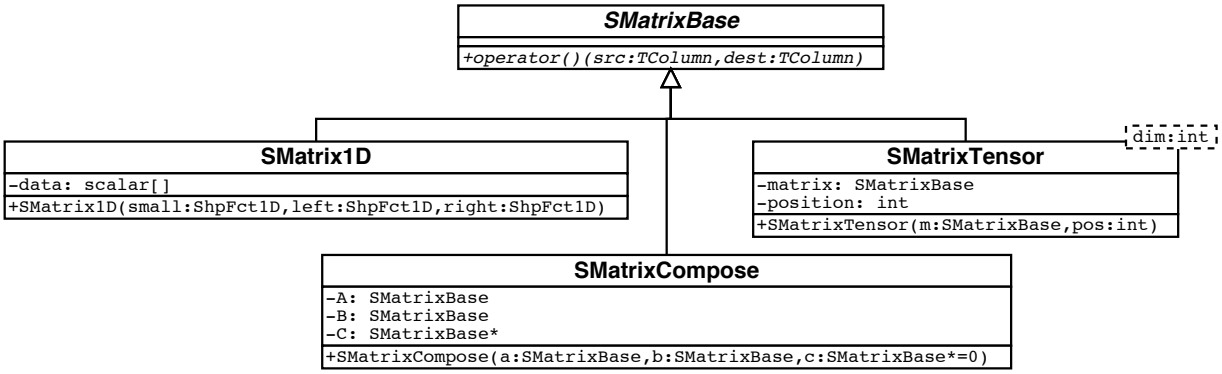

FIGURE 14. Classes for S-matrices in one, two and three dimensions.

The right child $\hat{K}^\star$ in the left subdivision variant of Figure 13 has the S-matrix

$$S_{\hat{K}^\star \hat{K}} = S_{\hat{\jmath}^\star \hat{\jmath}} \otimes E.$$

The children of the middle subdivision variant of Figure 13 have

$$S_{\hat{K}' \hat{K}} = E \otimes S_{\hat{\jmath}' \hat{\jmath}} \qquad \text{for the bottom quadrilateral } \hat{K}' \text{ and}$$
$$S_{\hat{K}^\star \hat{K}} = E \otimes S_{\hat{\jmath}^\star \hat{\jmath}} \qquad \text{for the top quadrilateral } \hat{K}^\star.$$

It remains to compute the S-matrices for the right subdivision variant (four children) of Figure 13. Since only the topology and the subdivision ratio influences the S-matrix, the subdivision of $\hat{K}$ into the four children $\hat{K}^a$, $\hat{K}^b$, $\hat{K}^c$ and $\hat{K}^d$ can be reached by subdividing $\hat{K}$ horizontally into two children and subdividing both children vertically into two children. This is reflected by concatenating the S matrices of the two subdivision processes above:

$$\begin{aligned}
S_{\hat{K}^a \hat{K}} &= \left(S_{\hat{\jmath}' \hat{\jmath}} \otimes E\right) \cdot \left(E \otimes S_{\hat{\jmath}' \hat{\jmath}}\right), \\
S_{\hat{K}^b \hat{K}} &= \left(S_{\hat{\jmath}^\star \hat{\jmath}} \otimes E\right) \cdot \left(E \otimes S_{\hat{\jmath}' \hat{\jmath}}\right), \\
S_{\hat{K}^c \hat{K}} &= \left(S_{\hat{\jmath}^\star \hat{\jmath}} \otimes E\right) \cdot \left(E \otimes S_{\hat{\jmath}^\star \hat{\jmath}}\right), \\
S_{\hat{K}^d \hat{K}} &= \left(S_{\hat{\jmath}' \hat{\jmath}} \otimes E\right) \cdot \left(E \otimes S_{\hat{\jmath}^\star \hat{\jmath}}\right).
\end{aligned}$$

3.2.3. *S-Matrix in Higher Dimensions.* With the same idea which was used to derive the two dimensional S matrices, the S-matrices in three and higher dimensions can be derived from the one dimensional S-matrices with matrix tensor products.

In Concepts, the classes shown in Figure 14 implement the concept of the S-matrices which was presented in this section. *SMatrixBase* just prescribes the interface: an S matrix is applied to a column of a T-matrix (i.e. a TColumn). SMatrix1D is used to compute the one dimensional S-matrices for the set of shape functions given on a set of points (see Section 3.2.1). SMatrixTensor implements a tensor product in dim dimensions with the given one dimensional matrix at the given position. SMatrixCompose is used to concatenate S-matrices.

SUMMARY

In this paper, we have shown how mathematical concepts can be used to identify and characterize the modules which can then be used to implement a mathematical method in an object oriented programming language. We used this approach called *concept oriented design* to show how we implemented *hp*-FEM. It has been applied to implement codes for BEM and (g-)FEM in the same framework, too.

A more in depth analysis of the treatment of hanging nodes in irregular meshes is given: T-matrices are used to formulate a general and flexible assembly operator. S-matrices are used to handle the hanging nodes of irregular

meshes. A dimension independent and flexible approach to compute the S-matrices is shown mathematically and algorithmically.

Acknowledgements

We would like to thank the following projects of the open source community:

- The GNU project [8] for the GNU Compiler Collection [7] from which we use the C++ compiler as our main development platform.
- Dia [11] for the diagram creation program. Dia was used to create the UML diagrams in this paper.
- Xfig [15] for the drawing program which was used to create the other figures in this paper.

References

[1] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. `http://www.mcs.anl.gov/petsc`, 2001.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 2001.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Object Technology Series. Addison Wesley Longman, Inc., second edition, 1994.

[5] Concepts Development Team. Concepts. Internet, 2001. `http://www.math.ethz.ch/~concepts/`.

[6] Miroslav Fiedler. *Special matrices and their applications in numerical mathematics*, chapter Tensor Product of Matrices. Compound Matrices, page 136ff. Martinus Nijhoff Publishers, Dordrecht, The Netherlands and SNTL—Publishers of Technical Literature, Prague, Czechoslovakia, 1986.

[7] GNU Project. GNU Compiler Collection. Internet, 2001. `http://www.gnu.org/software/gcc/`.

[8] GNU Project. GNU is Not Unix and the Free Software Foundation. Internet, 2001. `http://www.gnu.org/`.

[9] Christian Lage. *Softwareentwicklung zur Randelementmehtode: Analyse und Entwurf effizienter Techniken*. PhD thesis, Christian-Albrechts-Universität, Kiel, 1995.

[10] Christian Lage. Concept oriented design of numerical software. Technical Report 98-07, Seminar for Applied Mathematics, Swiss Federal Institute of Technology, Zürich, 1998.

[11] Alexander Larsson, James Henstridge, et al. Dia. Internet, 2001. `http://www.lysator.liu.se/~alla/dia/`.

[12] Ana-Maria Matache. *Spectral and p-Finite Elements for problems with microstructure*. PhD thesis, Swiss Federal Instititute of Technology, Zürich, 2000.

[13] Object Management Group, Inc., Framingham, USA. *OMG Unified Modeling Language Specification*, 1.3 edition, 1999. `http://www.rational.com/uml/resources/documentation/`.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, Inc., third edition, 1997.

[15] Supoj Sutanthavibul, Brian V. Smith, and Paul King. Xfig. Internet, 2001. `http://epb.lbl.gov/xfig/`.

Research Reports

No.	Authors	Title
02-09	Ph. Frauenfelder, Ch. Lage	Concepts - An object-oriented software package for partial differential equations
02-08	A.-M. Matache, J.M. Melenk	Two-Scale Regularity for Homogenization Problems with Non-Smooth Fine Scale Geometry
02-07	G. Schmidlin, C. Lage, C. Schwab	Rapid solution of first kind boundary integral equations in $\mathbb{R}^3$
02-06	M. Torrilhon	Exact Solver and Uniqueness Conditions for Riemann Problems of Ideal Magnetohydrodynamics
02-05	C. Schwab, R.-A. Todor	Sparse Finite Elements for Elliptic Problems with Stochastic Data
02-04	R. Jeltsch, K. Nipp	CSE Program at ETH Zurich: Are we doing the right thing?
02-03	L. Diening, A. Prohl, M. Ruzicka	On Time-Discretizations for Generalized Newtonian Fluids
02-02	A. Toselli	*hp* Discontinuous Galerkin Approximation for the Stokes Problem
02-01	F.M. Buchmann, W.P. Petersen	Solving Dirichlet problems numerically using the Feynman-Kac representation
01-09	A.-M. Matache	Sparse Two-Scale FEM for Homogenization Problems
01-08	C. Lasser, A. Toselli	Convergence of some two-level overlapping domain decomposition preconditioners with smoothed aggregation coarse space
01-07	T. von Petersdorff, C. Schwab	Wavelet-discretizations of parabolic integro-differential equations
01-06	A.-M. Matache, C. Schwab	Two-Scale FEM for Homogenization Problems
01-05	A. Buffa, R. Hiptmair, T. von Petersdorff, C. Schwab	Boundary Element Methods for Maxwell Equations in Lipschitz Domains
01-04	R. Hiptmair, C. Schwab	Natural BEM for the Electric Field Integral Equation on polyhedra
01-03	C. Schwab, A.-M. Matache	Generalized FEM for Homogenization Problems
01-02	A. Toselli, C. Schwab	Mixed *hp*-finite element approximations on geometric edge and boundary layer meshes in three dimensions
01-01	A. Buffa, M. Costabel, C. Schwab	Boundary element methods for Maxwell's equations on non-smooth domains