Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10, D-10711 Berlin - Wilmersdorf

Rudolf Beck
Bodo Erdmann
Rainer Roitzsch

# KASKADE 3.0
# An Object-Oriented Adaptive Finite Element Code

Technical Report TR 95-4 (June 1995)

# KASKADE 3.0
# An Object-Oriented Adaptive Finite Element Code

Rudolf Beck Bodo Erdmann Rainer Roitzsch

**Abstract**

KASKADE 3.0 was developed for the solution of partial differential equations in one, two, or three space dimensions. Its object-oriented implementation concept is based on the programming language C++. Adaptive finite element techniques are employed to provide solution procedures of optimal computational complexity. This implies a posteriori error estimation, local mesh refinement and multilevel preconditioning.

The program was designed both as a platform for further developments of adaptive multilevel codes and as a tool to tackle practical problems. Up to now we have implemented scalar problem types like stationary or transient heat conduction. The latter one is solved with the Rothe method, enabling adaptivity both in space and time. Some nonlinear phenomena like obstacle problems or two-phase Stefan problems are incorporated as well. Extensions to vector-valued functions and complex arithmetic are provided.

We have implemented several iterative solvers for both symmetric and unsymmetric systems together with multiplicative and additive multilevel preconditioners. Systems arising from the nonlinear problems can be solved with lately developed monotone multigrid methods.

# Contents

# 1 Introduction

The purpose of this report is to give an overview of

- the capabilities of the software package KASKADE 3.0,
- its object-oriented implementation concept and
- the multilevel solution strategies included.

The report is not intended to serve as a user's guide or tutorial; such information will be provided elsewhere. We assume that the reader is familiar with basic finite element concepts. Furthermore he should have some knowledge about iterative solution techniques for equation systems arising from the discretization of partial differential equations.

KASKADE 3.0 was designed both as a basis for the development of adaptive multilevel codes and as a tool for the solution of practical problems. Thus we wanted to be able to handle problems for one, two, or three space dimensions within one single program. Of course, the capabilities of the predecessors, the KASKADE family version 2.0 [ELR93], had to be included as well.

The purpose of multilevel techniques is to combine reliability and numerical efficiency. Therefore we implemented – besides the routines which carry out adaptive mesh refinement – a broad framework for the solution of the arising equation systems. Linear solvers and preconditioners can be combined with high flexibility.

In the description of our program we deliberately do not want to separate mathematical and implementational issues, as the code itself should reflect natural hierarchies and dependencies of the mathematical solution concepts employed. This, in fact, was one of the main reasons why we chose the object-oriented programming language C++ . Its class hierarchies and virtual functions play an important role throughout the implementation, and often merely figures showing class hierarchies or declarations will give insight into the programmers' philosophy.

For the code development we used the GNU C++ -compiler (up to version 2.6.3), which is public domain software and one of the best compilers we encountered. We also want to mention the excellent (and witty!) book of Scott Meyers [Mey92], which tells more about the principles of object-oriented programming than many heavy tomes.

We close this section with some general remarks on the text:

- C++ -language conventions are used for code descriptions. Class names are set in `typewriter` style.
- Often expressions like ... *is represented by class* `X` ... can be found. This is a sloppy abbreviation, and the precise expression should read ... *is represented by an object of class* `X` *or one of its derived classes*. However, we consider the latter formulation to be rather ponderous and pedantic.

– We will list some pieces of C++-code or excerpts of code. In these listings dots (...) are used to indicate omitted lines. The data type `Num` represents a numerical value (`float`, `double` or `complex`). The type `Real` may be a `float` or a `double`.

# 2 The Code Organization

The main element in the code hierarchy is the base class `Problem`. Its members (see figure 1) are the 'ingredients' necessary to define and solve specific problem types by a finite element method.

The constitutive members for the problem definition are

– the finite element (class `Element`),
– the triangulation (class `Mesh`) and
– the material properties and essential boundary conditions (classes `Material` and `DirichletBCs`).

The other members of `Problem` participate in the numerical solution procedures or provide graphical output (`FEPlot`). It is worth noting that the class `ErrorEstimator` in general must have access to the complete `Problem` itself.

One major concept of the implementation was the separation of topological and algebraic entities. The former ones comprise the constitutive members quoted above. The topology of the problem, i.e. the mesh structure and node distribution, may be rather complex due to the nested refinement levels created in the adaptive solution process.

By using a global node-numbering strategy we are able to transfer the topological features into algebraic structures. The latter ones are based on special sparse matrix classes and vectors. This concept allows a rigorous separation of the constitutive classes and those which perform algebraic operations. The link between the two parts is the `Interface`.

# 3 The Problem Classes

Figure 2 gives an overview of the problem types implemented up to now. A basic distinction was made between static and time dependent problems. As both of them may contain nonlinear features, we decided to construct transient nonlinear problems by the means of multiple inheritance.

The most important members of a problem class are shown in figure 1. We think the names are self-explanatory, descriptions of these classes are given in the relevant sections.

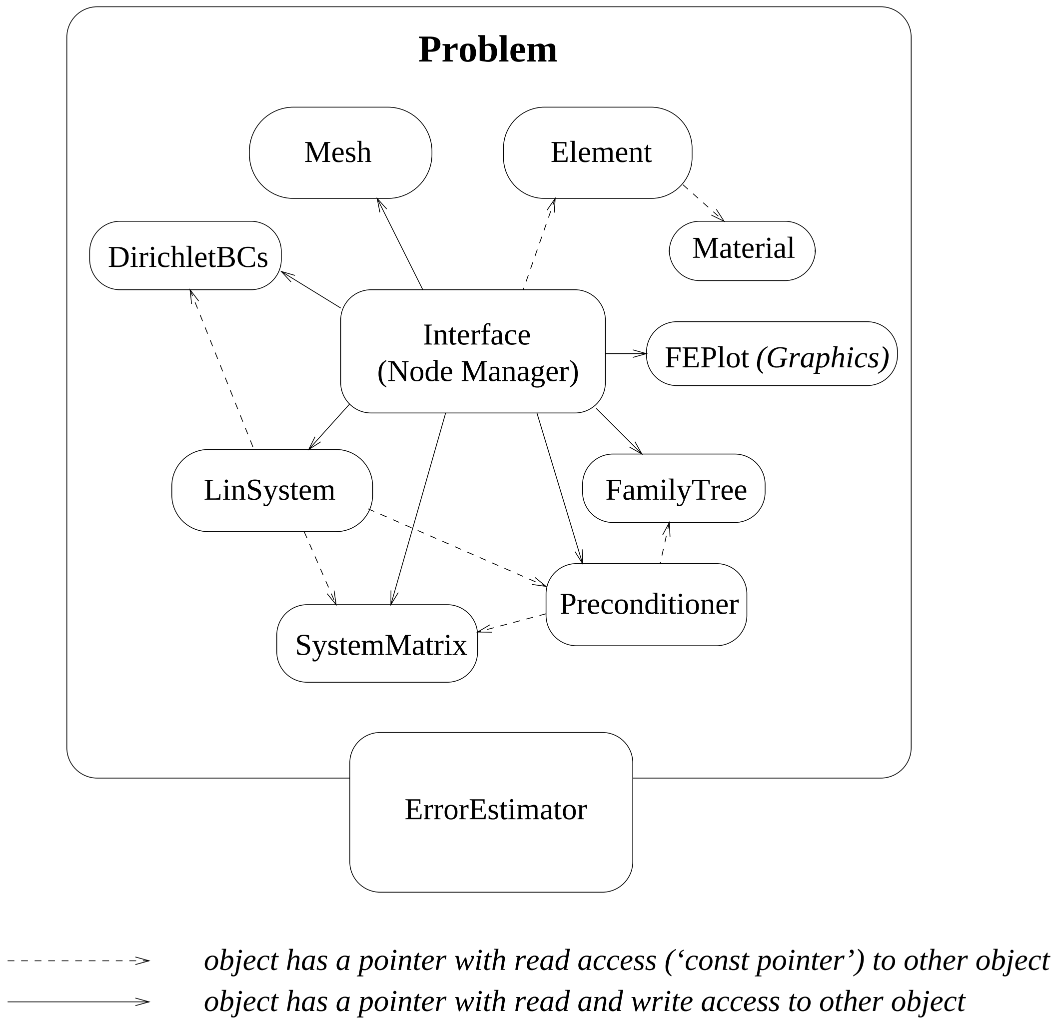


Figure 1: The problem class and its members (the error estimator has access to all of them).

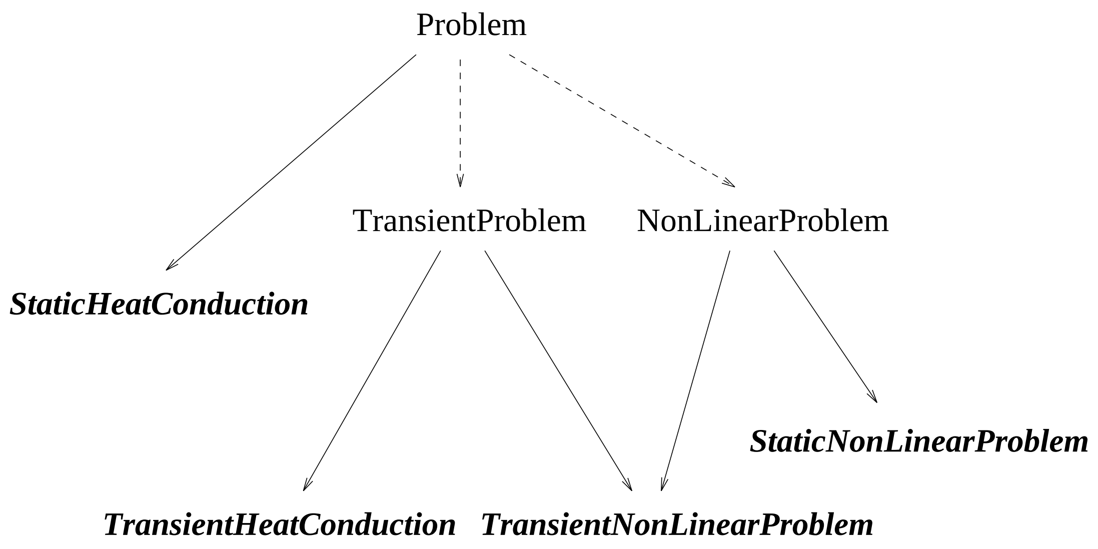


Figure 2: The main problem classes. Classes printed in bold italics can be instantiated; dashed lines originate from virtual base classes.

Basic member functions of a problem class are the adaptive solution procedure (see figure 3) and the assembling routine for the weak form of the differential equation. The former is rather general, whereas the assembling procedure, of course, depends on the specific problem.

### Vector-Valued Solutions

The problem classes `StaticHeatConduction` and `TransientHeatConduction` can be used for vector-valued functions or equation systems, too. In this case, the parameter `nComp` is provided to specify the number of components at each base node. There are derived finite element classes (see section 5) with the appropriate assembling routines. A special block matrix class can be used for the efficient handling of the linear systems (see section 7).

```
Bool Problem:: staticSolution(Real globalPrecision, Real time)
{
    ...
    newLinSystem();                          // construct and initialize
    newPreconditioner();                     //          some new members
    newInterface();
    newErrorEstimator();

    for (int step=0; step<=maxRefinementSteps; ++step)
                                            // adaptive refinement steps
    {
        assembleGlobal(time);
        linSystem->solve(...);
        ...                                  // some post-processing
                                             //    operations may follow
        errorEstimator->adapt(...);

        SolutionInfo(step);                  // print some information

        if (globalConvergenceTest(globalPrecision,step)) return True;

        interface->refine(u);
    }

    return False;
}
```

Figure 3: Structure of the basic adaptive solution procedure

## 3.1 Static Heat Conduction

This problem class involves elliptic partial differential equations of the type

$$\begin{aligned} -\nabla k \nabla u &= f && \text{in } \Omega \\ u &= u_0 && \text{on } \Gamma_D \\ k\frac{\partial u}{\partial n} &= q_N && \text{on } \Gamma_N \\ k\frac{\partial u}{\partial n} + \alpha\, u &= q_C && \text{on } \Gamma_C \end{aligned} \tag{1}$$

where $k$ and $\alpha$ denote material parameters (thermal conductivity and transfer coefficient). $\Omega$ may be a one-, two-, or three-dimensional region; boundary conditions of Dirichlet, Neumann, or Cauchy type are applied on the corresponding boundary sections $\Gamma_D$, $\Gamma_N$, and $\Gamma_C$. An extension to anisotropic materials with tensors $k_{ij}$ or $\alpha_{ij}$ is quite straightforward.

## 3.2 Transient Heat Conduction

This class comprises linear parabolic problems of the form

$$c\,\frac{\partial u}{\partial t} - \nabla k \nabla u \;=\; f \qquad \text{in } \Omega \tag{2}$$

The boundary conditions are like in (1), but may be time-dependent like the source term $f$. Additionally we have to define initial values for $t = 0$:

$$u(x,0) \;=\; u_0(x) \qquad \text{in } \Omega$$

$u_0(x)$ has to be supplied by the user or can be calculated via the solution of a static heat conduction problem.

### Adaptive Time Step Control

The adaptive Rothe-Method is employed for the time-stepping algorithm [Bor91]. This approach completely separates time and space discretization: we first discretize (2) with respect to the time variable by choosing a step width $\tau$. Then the resulting spatial elliptic subproblem may be solved adaptively like problem (1).

If the index $n$ denotes the $n$-th time step, i.e. $t_n = t_{n-1} + \tau_n$, the value $\tau_{n+1}$ for the next step is proposed by

$$\tau_{n+1} = \tau_n \sqrt{\frac{Tol}{\varepsilon_{t_n}}}$$

*Tol* is a user-specified tolerance and $\varepsilon_{t_n}$ the estimated time error of step $n$. To obtain $\varepsilon_{t_n}$ we calculate solutions of first and second order accuracy in time via a multiplicative error correction procedure:

$$u_n^{(2)} = u_n^{(1)} + \eta_n$$

An implicit Euler-step for (2) yields $u^{(1)}$ with first order accuracy:

$$(M + \tau_n A)\, u_n^{(1)} \quad = \quad \tau_n f(t_n) + M\, u_{n-1}^{(1)} \tag{3}$$

The matrices $M$ and $A$ arise from the first and second term on the left hand side in (2).

The correction $\eta_n$ is given by [Bor91]

$$(M + \tau_n A)\, \eta_n \quad = \quad \frac{\tau_n}{2}\{A(u_n^{(1)} - u_{n-1}^{(1)}) - (f(t_n) - f(t_{n-1}))\}$$

This formulation preserves the structure of the first-order system (3). The second-order correction $\eta_n$ yields an estimate of the time error $\varepsilon_{t_n}$.

For every time step both time and space error are required to remain below the tolerance *Tol*. If $\varepsilon_{t_n}$ is too large, the proposed step width $\tau_n$ is reduced. The space error is calculated by a hierarchical defect correction method (see section 11) and reduced by adaptive mesh refinement.

## 3.3 Nonlinear Problems

Nonlinear problem classes have the additional member `NonLinearity`. It defines the nonlinear characteristics of the problems, e.g. obstacle functions or a nonlinear heat capacitance. A pointer to `NonLinearity` is also handed to the nonlinear preconditioner classes to supply the appropriate updates during the solution of the equation system.

### 3.3.1 Obstacle Problems

Obstacle functions $\varphi(x)$ can be added to any differential equation:

$$\begin{array}{rcl} u(x) & < & \varphi_{up}(x) \\ u(x) & > & \varphi_{low}(x) \end{array} \tag{4}$$

Such constraints may occur in various types of applications. The corresponding variational inequality yields a nonlinear system, which is solved by a single-level Gauss-Seidel or a monotone multigrid method [Kor95] (see sections 8 and 9).

### 3.3.2 Stefan Problems and Related Nonlinear Equations

These problems may be regarded as degenerate parabolic initial value problems of the form

$$\begin{aligned} \frac{\partial}{\partial t} H(u) - \nabla k \nabla u &= f && \text{in } \Omega \\ u(x,0) &= u_0(x) && \text{in } \Omega \\ H(x,0) &= H_0(x) && \text{in } \Omega \end{aligned} \tag{5}$$

combined with boundary conditions like in (1). $H$ is the heat content or a generalized enthalpy. In KASKADE $H$ may be a continuous quadratic function of $u$ but with discontinuous derivatives $H'$. Thus $H'$ must be specified as a piecewise linear function of $u$; a discontinuity in $H'$ defines a change of phase with a certain amount of latent heat.

A characteristic feature of the nonlinear part in $H$ is that by mass-lumping its contribution can be restricted to the diagonal of the stiffness matrix. Such a nonlinearity can be handled very efficiently in the solution procedure for the equation system.

Other nonlinear problems may be formulated in a similar way, e.g. the porous media equation (see [Kor95]). Of course, we may have additional obstacle functions like in (4).

# 4 The Mesh and its Components

All our finite element meshes exclusively comprise simplexes, which are derived from the base class `PATCH` (see figure 4).

In this context, the term *element* is used for the geometric elements of the mesh (tetrahedra in 3D, triangles in 2D and edges in 1D space). It should not be mixed up with the *finite element* itself, which defines the local nodes and provides assembling procedures for the weak form of the differential equations. In these procedures the finite element is mapped onto all geometric elements of the mesh by the class `Interface`. This is done via global node numbers set on all the relevant patches of the mesh. In principle, every `PATCH` may be given a global node number:

```
class PATCH               //  abstract base class for all elements
{
  protected:
    int  node;            // node number (set by Interface)
  public:
    int  getNode()  const { return node; }
    void setNode(int no)  { node = no; }
```

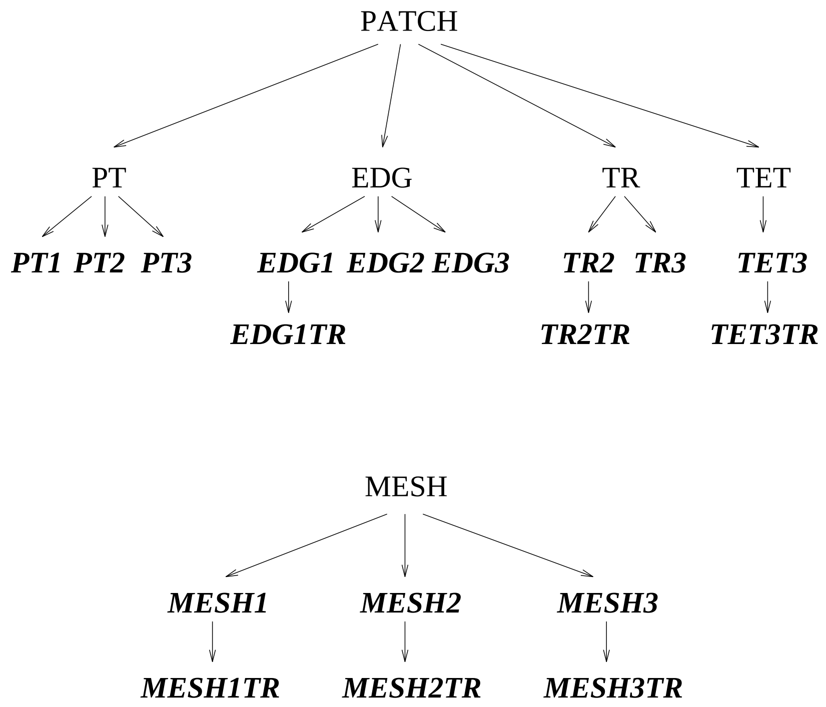


Figure 4: The classes comprising geometric entities. Each `MESH` is a non-overlapping arrangement of simplexes with the base class `PATCH`, from which points, edges, triangles and tetrahedra are derived. Numbers in class names indicate the space dimension.

```
  ...
};
```

To show the connections within a finite element mesh, we list some details of the declarations of 3D-patches:

```
class PT3 : public PT
{
  protected:
    Real  x,y,z;                            // coordinates in 3D space
    char  boundP, mark, classA, depth;      // some descriptors
    PT3   *next, *prev;             // pointers for doubly-linked list
    ...
};


class EDG3 : public EDG
{
  protected:
    PT3   *p1, *p2, *pm;                // vertices at ends and midpoint
    char  boundP, classA, type, depth;
    EDG3  *father, *firstSon;    // connections in multi-level structure
    EDG3  *next, *prev;
    ...
};


class TR3 : public TR                       // triangle for outer surface
{
  protected:
    PT3   *p1,*p2,*p3;
    EDG3  *e1,*e2,*e3;
    TET3  *t31, *t32;            // tetrahedra on both sides (0 if none)
    char  mark, type, classA, depth, boundP;
    TR3   *next, *prev, *father, *firstSon;
    ...
};


class TET3 : public TET
{
  protected:
    PT3   *p1, *p2, *p3, *p4;
    EDG3  *e1, *e2, *e3, *e4, *e5, *e6;
```

```
    PATCH *n1, *n2, *n3, *n4;   // neigbour tetra's or surface triangles
    char  type, classA, depth, mark;
    TET3  *next, *prev, *father, *firstSon;
    ...
};
```

All objects of a mesh are stored in doubly-linked lists to allow the removal or replacement of any element. The lists are stacked with respect to the refinement level (also called depth) of the elements. There are separate lists for points, edges etc.; stepping through these lists is done by iterators. The patches can be accessed by virtual functions, thus the operations of the `Interface` on the grid are independent of the actual mesh implementation and space dimension.

As an example we give an excerpt of the three-dimensional class `MESH3`:

```
class MESH3 : public MESH
{
  protected:
    Stack<DList<PT3>*>  ptList;      // points
    Stack<DList<EDG3>*> edgList;     // edges
    Stack<DList<TR3>*>  trList;      // triangles on the outer surface
    Stack<DList<TET3>*> tetList;     // tetrahedra

    int noOfPoints, noOfEdges, noOfTriangles, noOfBoundaryTriangles,
        noOfTetrahedra;
    ...
  public:
    MESH3(const char* inFileName, Bool readMesh=True);
    virtual void Refine();           // does the mesh refinement
    ...
};
```

For transient problems, the mesh and element classes are slightly modified. Due to the use of the adaptive Rothe method, the grids of the separate time steps may look quite different. Here each element has a pointer to its 'partner element' in the mesh of the previous step to facilitate the transfer of the solution $u$ between the grids. The classes in question have been given the suffix `TR` (indicating `TR`ansient).

If a mesh is to be refined adaptively, all elements with an error above a certain level are marked by the error estimator. The way of subdivision in 2D is that of PLTMG [Ban88] or the bisection strategy of Rivara [Riv84], which does not need irregular 'green closures'.

In three dimensions, an approach similar to the former one has been chosen: regular (red) subdivision of the marked tetrahedra and a green closure. The red subdivision can be carried out according to Bey [Bey91], which guarantees stability with respect to the minimum angle, or by a shortest edge strategy [Zha88, Ong89, BEK93].

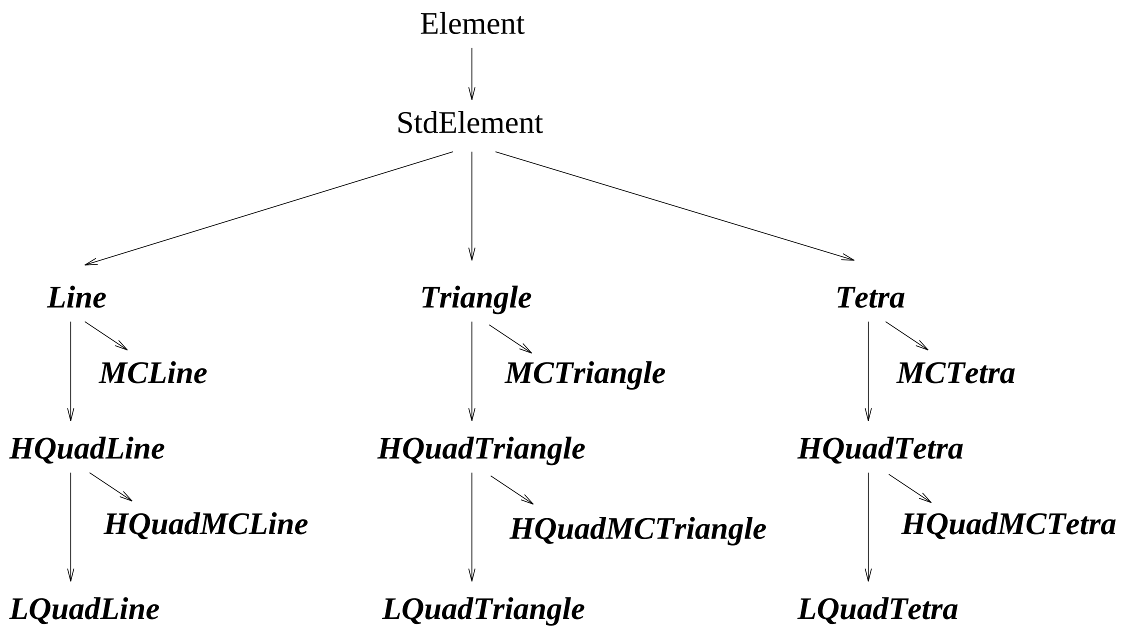


Figure 5: The finite element classes. The standard elements are nodal elements with linear or quadratic shape functions (Lagrange or hierarchical type). For vector-valued functions several degrees of freedom may be located at each node (multi-component type, indicated by the prefix `MC`).

# 5 The Finite Elements

## 5.1 Standard Elements

This family comprises simplex-type finite elements with standard nodal basis functions (Lagrange type) up to second order: line, triangle, and tetrahedron. The hierarchical second order counterparts are implemented as well and are used in error estimation procedures. The class hierarchy is shown in figure 5.

The main member functions serve for assembling the weak representation of the differential equations or for interpolating a discrete solution vector on the element. For numerical integration on elements with varying material coefficients Gaussian quadrature formulae are provided (up to degree 15 in two and three space dimensions).

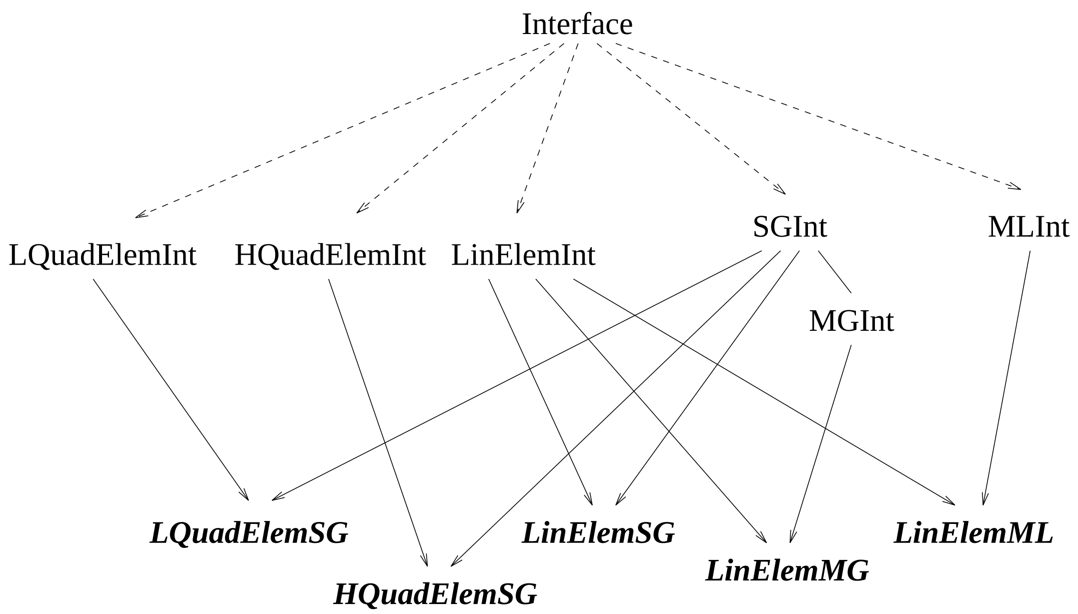


Figure 6: The interface classes. The interface carries out the 'node management' and is constructed with respect to the specific finite element and the preconditioner type employed for the problem solution.

## 5.2 Multi-Component Elements

If several degrees of freedom are located at each node, e.g. in the case of vector-valued functions or systems of equations, these classes form an appropriate extension of the standard scalar elements. A set of assembling routines is provided, which can be adapted to the specific problem.

# 6 The Node Interfaces

As we mentioned above, we attempted a thorough separation of topological and algebraic code structures. This is achieved by a global node numbering concept. The interface classes are responsible for the node management, including update and refinement functions, which modify the data in various members of the `Problem` class.

We list some details of the base class `Interface`:

```
class Interface
{
  protected:      // pointers to objects supplied by problem

    const Element*      element;
    MESH*               mesh;
    DirichletBCs*       dirichletBCs;
    LinSystem*          Ab;
    Preconditioner*     precond;
    ...

  public:               // some member functions
    ...
    virtual void setNodeNumbers() = 0;        // set global nodes
    virtual void getGlobalNodes(const PATCH* t,
                                Vector<int>& globalNodes) const = 0;
    virtual void refine(Vector<Num>& u) = 0;   // mesh refinement and
                                               //  and update functions
    ...
};
```

A finite element defines local nodes, which are mapped onto the global nodes of the mesh. This is carried out by the function `getGlobalNodes(...)`, for example during the global assembling procedure. The function `setNodeNumbers(...)` distributes the global node numbers after a mesh refinement.

The patches of the mesh (points, edges, triangles etc.) are accessed via list iterators and virtual functions. Thus nearly all operations of the interface can be implemented without regarding the specific mesh class and space dimension.

A basic distinction is made between interfaces for single- and multi-level preconditioners. For the latter, of course, the node management is more complicated, as the family tree (see section 10) and the system matrices of the different refinement levels have to be maintained.

# 7 The Sparse Matrices

Sparse matrix structures play an important role in the implementation of finite element codes. They are a natural consequence of locally defined basis functions, allowing the storage allocation for system matrices to be of order $N$, $N$ being the total number of degrees of freedom on the mesh. Combined with an 'order-1 solver' for the equation system this yields a solution strategy of optimal complexity with respect to memory space and execution time.

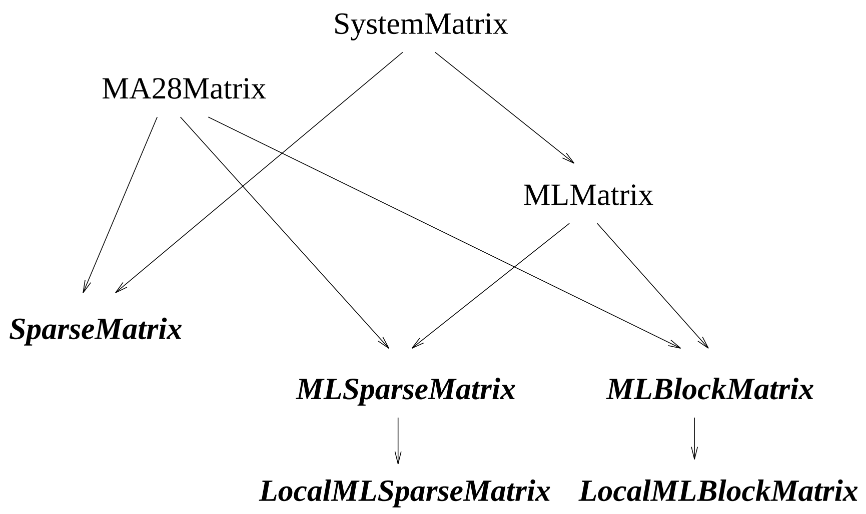


Figure 7: The system matrix classes. Special types are provided for single- and multi-level preconditioning. The prefix `Local` in the class names indicates the possibility of local smoothing operations in multilevel preconditioners. An extension for the Harwell MA28 sparse matrix solver has been included.

Figure 7 shows the implemented system matrix classes. Various types are available for different problem classes and solution techniques. All of them have a sparse structure and can be used for both symmetric and unsymmetric systems (in the latter case they are assumed to be symmetrically populated). The class `SparseMatrix` stores all data entries in vectors, whereas in all the other classes the off-diagonal entries are collected in linked lists, thus allowing more flexibility (see figure 8).

The letters `ML` indicate that a matrix is of 'multilevel-type', i.e. it can be used in conjunction with a multilevel-preconditioner. These matrices are supplied with a Galerkin procedure for the computation of coarse grid matrices.

The most important member functions of the sparse matrix classes carry out the following operations:

- multiplication with a vector
- forward and backward Gauss-Seidel and SOR smoothing
- LU-decomposition
- forward-backward substitution of a decomposed matrix

For LU-decomposition and forward-backward substitution we have added the Harwell-MA28 sparse matrix solver [Duf80]. A drop tolerance parameter may be set to get

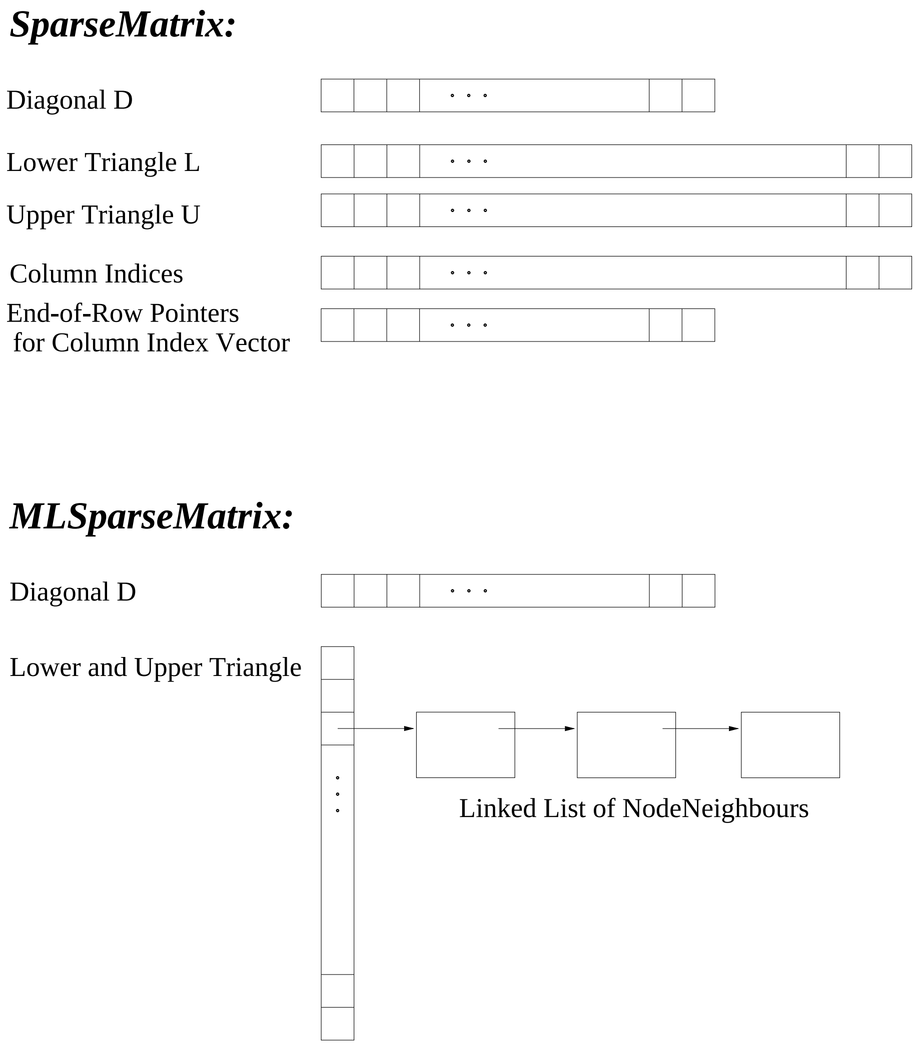


Figure 8: Data arrangement in sparse matrix classes. In `SparseMatrix` all data are stored in vectors. For a symmetric matrix the upper triangle vector U is identified with L and does not need extra space. L is stored row-wise, whereas U is stored column-wise. Thus for symmetrically populated matrices the column vector contains the column indices for the entries of L and the row indices for those of U. This structure is similar to the one used in PLTMG [Ban88].
The type `MLSparseMatrix` maintains lower and upper triangle by linear lists. The class `NodeNeighbour` contains the column index of the entry and one or two off-diagonal terms (one in the symmetric case).
In an `MLBlockMatrix` the diagonal entries and NodeNeighbours are substituted by special block-entries.

incomplete LU-decompositions, which are used in the `ILU`-preconditioner (see section 9).

The block matrices (`MLBlockMatrix`) are useful for problems with vector-valued functions. If there are several degrees of freedom at each node, these are collected in block entries. For smoothing operations all blocks of the matrix diagonal are inverted by LU-decomposition.

# 8 The Equation Systems and Solvers

We have implemented the class `LinSystem` for the storage and solution of equation systems arising from finite element discretizations. Its main members are the system matrix $A$ and the right-hand side vector $b$. $A$ is always stored as a sparse matrix.

The equation system can be solved via LU-Decomposition of $A$, which is carried out by a sparse matrix routine [Duf80], or by an iterative procedure.

The class `LinSystem` is also used in the context of nonlinear problems. Here the relevant preconditioners supply the updates for nonlinear terms (see section 9).

## 8.1 The Direct Sparse Matrix Solver

We employ the Harwell-MA28 sparse matrix solver [Duf80] for an LU-decomposition. It is based on a nested dissection approach and thus achieves optimal complexity for a direct solution procedure. It is also possible to generate incomplete LU-factorizations, which may be used as preconditioners for an iterative solver. The incomplete factorization works with a user-specified drop-tolerance.

For technical reasons the factorization and forward-backward substitution algorithms are included in the sparse matrix classes (see section 7).

## 8.2 Iterative Solvers

We have implemented several iterative solvers for systems arising from linear and nonlinear problems.

### Linear Equation Systems

- Systems with symmetric positive definite (SPD) matrix:
  - Conjugate Gradients
- Systems with symmetric indefinite matrix:
  - Conjugate Residuals

- Unsymmetric systems:
  - Bi-Conjugate Gradients
  - the CGS algorithm of Sonneveld [Son89]
  - BiCGStab of van der Vorst [vdV92]
  - GMRES of Saad and Schultz [SS86]
  - Conjugate Gradients for the normal equations (CGNR)
  - Relaxation routines (Jacobi, SSOR etc.; for technical reasons the type is determined by the choice of a preconditioner applied to the Richardson iteration)

### Nonlinear Equation Systems

Here we apply relaxation routines, which work efficiently in conjunction with a nonlinear multigrid preconditioner (see section 9).

### Structure of the Iterative Solvers

All preconditioned iterative solvers that are implemented in KASKADE have the following structure (here `iterSolve` is a fictitious name):

```
Bool LinSystem:: iterSolve(Vector<Num>& u, int maxIter)    // solve Au=b
{
    ...
    preCond->initialize(A,u,b);
    ...
    for (iter=1; iter<=maxIter; ++iter)
    {
        ...
        preCond->invert(e,A,r);      // solve defect equation Ae=r
                                     // approximately (r is the residual)
        ...
        if (convergenceTest(...))  break;
        ...
    }
    preCond->close(A,u,b);
    return iter <= maxIter;
}
```

The calls `preCond->initialize(A,u,b)` and `preCond->close(A,u,b)` are required for preconditioners which implicitely transform the system (like `TrSSOR`, see section 9).

The function `preCond->invert(e,A,r)` involves the actual preconditioning operation. Here the defect equation

$$A\,e = r$$

is solved approximately (see section 9).

## 8.3 Convergence Tests for Iterative Solvers

### The Residual Norm

If we consider the equation system $A\,u = b$, the 'standard' stopping criterion is

$$\frac{\|\,r\,\|}{\|\,b\,\|} < \varepsilon\ ,$$

with $\|\,r\,\|$ denoting the Euclidian norm of the residual $r = b - A\,\tilde{u}$. Here $\tilde{u}$ is the approximate solution of the current iteration step. By default the limit $\varepsilon$ is chosen to be

$$\varepsilon \quad = \quad 0.1\ \varepsilon_{disc}\ ,$$

where $\varepsilon_{disc}$ is the maximal allowed relative discretization error for the solution and is specified by the user. Usually this choice is quite reliable but $\varepsilon$ may be changed to any other value, of course.

### Cascadic Iterations

For positive definite systems and single-level preconditioning this stopping criterion may be quite efficient if the user is merely interested in the energy norm of the solution.

The convergence test arises from the lately developed 'cascadic iteration' concept [Bor94, Deu94]. Instead of monitoring the residual, here the iterative process is stopped if the estimated energy norm of the iteration error $u - \tilde{u}$ is below a certain limit. The choice of this limit is accomplished by to a clever matching of the estimated discretization error of the previous refinement level (calculated in the energy norm) and the iteration error of the current iteration step.

# 9 The Preconditioners

Throughout the text we use the term 'preconditioner' in a rather extensive sense. Here every device allowing the (approximate) solution of a defect equation is called a preconditioner. In KASKADE even classical iteration schemes like Jacobi or SSOR are implemented in the form of a linear relaxation solver (Richardson iteration) combined with the respective preconditioner (`Jacobi` or `SSOR`).

Let us have a brief look onto the characteristics of preconditioning operations. We consider the solution of the linear system

$$A\,u \;=\; b \tag{6}$$

by one of the iterative procedures of section 8. The usual way to introduce preconditioning is to define a transformation matrix $C$ with a suitable splitting

$$C \;=\; F\,G$$

to transform (6) into

$$F^{-1}AG^{-1}\,Gu \;=\; F^{-1}b \tag{7}$$

with improved condition number for $F^{-1}AG^{-1}$. For symmetric $A$, a preconditioner with symmetric splitting should be chosen:

$$G \;=\; F^T\,,$$

where $F^T$ denotes the transpose of $F$. Thus the transformation preserves the symmetry of the original system.

The iterative algorithms of section 8 are arranged in a way that does not form (7) explicitely, but rather incorporates the transformation via a defect equation. If we have an approximation $\tilde{u}$ of the exact solution $u$, we define the error or defect

$$e \;=\; u - \tilde{u}$$

and its residual

$$r \;=\; b - A\,\tilde{u}$$

The preconditioning operation can be included formally as the approximate solution of the defect equation

$$A\,e \;=\; r\,,$$

in which $A$ is replaced by $C$:

$$C\,e \;=\; r$$

The formalism is equivalent to (7) and preserves the symmetry of the original system if $C$ is symmetric.

The preconditioner $C$ may be a matrix derived from $A$ (e.g. an incomplete factorization) or may represent a more complex operation (like a multigrid cycle), which nevertheless can be written algebraically in matrix form. If $C$ involves operations on different refinement levels of our finite element mesh, we call it a multigrid or multilevel preconditioner.

Figure 9 shows the preconditioners implemented in KASKADE. A more detailed description is given in the following sections.

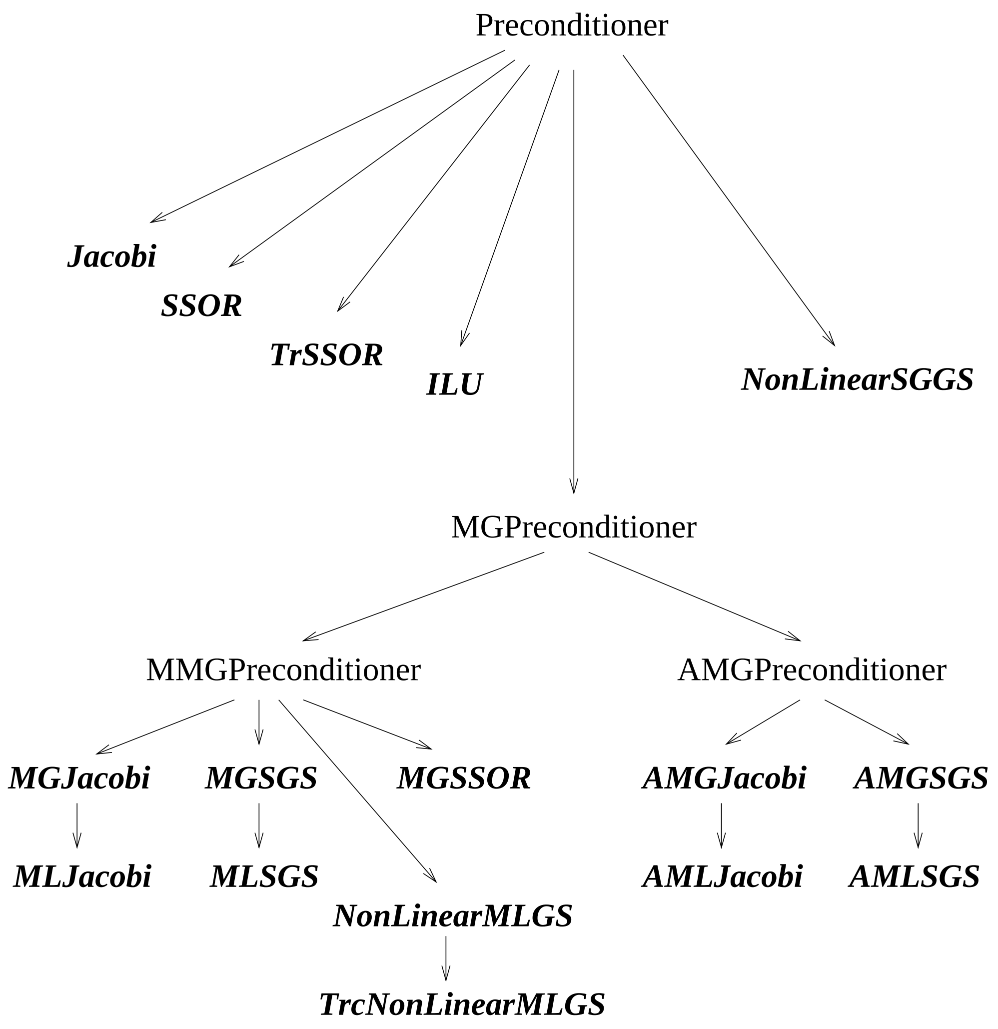


Figure 9: The preconditioners. Several types of single- and multi-level preconditioners have been implemented.

## 9.1 The Single-Level Preconditioners

These variants derive the preconditioner $C$ exclusively from the stiffness matrix $A$ of the current refinement step.

### Jacobi Preconditioning

Here the diagonal $D$ of the system matrix $A$ is chosen for preconditioning:

$$C = D$$

### SSOR and TrSSOR Preconditioning

In this variant the preconditioner $C$ is formed by an SSOR-splitting of $A$ (with relaxation parameter $\omega$):

$$\begin{aligned} C &= F\,G \\ &= \frac{1}{\omega(2-\omega)}(D+\omega L)D^{-1}(D+\omega U) \end{aligned}$$

$L$ and $U$ denote the lower and upper triangle of $A$, $D$ its diagonal.

The TrSSOR ('Transforming SSOR') directly uses the transformed system in (7). Of course, the matrix $F^{-1}AG^{-1}$ is never formed explicitely by matrix multiplication, which would cause heavy fill-in. But for an SSOR-splitting the multiplication of $F^{-1}AG^{-1}$ with a vector can be carried out at the cost of one matrix-vector multiplication and three multiplications of vectors with a scalar [AB84]. Usually this is much more efficient than one matrix-vector multiplication plus one inversion of the preconditioner, as required in the non-transforming variant.

### ILU Preconditioning

An incomplete factorization of $A$ can be carried out by using the sparse matrix solver MA28 (see section 8). Fill-in is generated with respect to a user-specified drop tolerance.

## 9.2 The Multilevel Preconditioners

For the implementation of these preconditioners we made an arbitrary technical distinction between multigrid and multilevel types.

Here multigrid preconditioners quite generally are all those ones which use smoothing operations on different grids. The multilevel preconditioners use smoothers which are arranged with respect to the refinement levels of the nodal basis functions. In the class names the latter ones are marked with the prefix `ML`, the former ones with `MG`.

The `ML`-type preconditioners compute their coarse grid matrices via a Galerkin procedure. Local smoothing is possible, the smoothing pattern can be determined by the user. By default, the set of nodes to be smoothed on a specific level contains all nodes which are new on this level and their neighbour nodes.

On the other hand, our `MG`-type preconditioners simply store the system matrix of selected refinement steps and use these ones for the smoothing operations. No Galerkin procedure is necessary for the computation of coarse grid matrices, thus the implementation is simplified considerably. These variants are often very efficient concerning memory space and execution time. In the case of uniform grid refinement, `ML`- and `MG`-type preconditioners operate equivalently.

The following sketch shows the most important data and functions within the class MGPreconditioner:

```
class MGPreconditioner : public Preconditioner
{
  public:
    ...
    virtual void invert(Vector<Num>& e, SystemMatrix* A,
                        Vector<Num>& r);
  protected:

    FamilyTree*          familyTree;      // used for grid transfer
    Stack<SystemMatrix*>  Al;             // the coarse grid matrices
    Stack<Vector<Num>*>   rl;             // residuals on coarse grids
    Stack<Vector<Num>*>   el;             // defects on coarse grids
    Stack<Real>           omega;          // relaxation parameters

    int  nPreSmooth, nPostSmooth;    // number of smoothing operations
    ...

    virtual void MGCycle(Vector<Num>& e, SystemMatrix& A,
                         Vector<Num>& r, int level);
                                           // the multigrid V-Cycle
};
```

### 9.2.1 Multiplicative Versions

These preconditioners are derived from the class `MMGPreconditioner`. Below we list the simplified code of the V-Cycle. The parameters passed to the function `MGCycle` are the defect correction `e`, the residual `r` and the system matrix `A` (all of the same level).

```
void MMGPreconditioner:: MGCycle(Vector<Num>& e, SystemMatrix& A,
                                 Vector<Num>& r, int level)
{
    if (A.DirectSolution()) A.FBSubst(e,r);    // matrix is decomposed
    else
    {
        preSmooth(level, e, A, r);
        residual(rNew, r, A, e);          // compute new residual rNew
        restrict(rNew, *rl[level-1], level);

        MGCycle(*el[level-1], *Al[level-1],
                *rl[level-1], level-1);   // recursive call of MGCycle

        prolong(*el[level-1], e, level);
        postSmooth(level, e, A, r);
    }
}
```

We give a list of some of the derived classes:

- `MGJacobi`: multigrid preconditioner with Jacobi-type smoothing. The default relaxation parameter $\omega$ is $2/3$.
- `MGSGS`: multigrid preconditioner with symmetric Gauss-Seidel smoothing. Forward Gauss-Seidel is applied for pre-smoothing, the backward operation for post-smoothing.
- `NonLinearMLGS`: multilevel preconditioner for nonlinear systems with Gauss-Seidel smoothing (see below).

The characteristic functions of all these preconditioners are pre- and post-smoothing operations. As an example we list the declaration of the symmetric Gauss-Seidel multigrid version:

```
class MGSGS : public MMGPreconditioner
{
  public:
    ...
    virtual void initialize(SystemMatrix* A, Vector<Num>& x,
                            Vector<Num>& b);
    virtual void close (SystemMatrix* A, Vector<Num>& x,
                        Vector<Num>& b);
  protected:
    virtual void preSmooth (int level, Vector<Num>& e,
                            SystemMatrix& A, Vector<Num>& r);
```

```
    virtual void postSmooth(int level, Vector<Num>& e,
                            SystemMatrix& A, Vector<Num>& r);
};
```

### 9.2.2 Additive Versions

Additive multilevel preconditioners are prefixed with an 'A'. We give the simplified code of the V-Cycle for these preconditioners:

```
void AMGPreconditioner:: MGCycle(Vector<Num>& e, SystemMatrix& A,
                                 Vector<Num>& r, int level)
{
    if (A.DirectSolution())  A.FBSubst(e,r);   // matrix is decomposed
    else
    {
        smooth(level, e, A, r);
        restrict(r, *rl[level-1], level);

        MGCycle(*el[level-1], *Al[level-1], *rl[level-1], level-1);
        prolong(*el[level-1], e, level);
    }
}
```

Additive preconditioning involves only one smoothing operation on each level. Here is the declaration of the additive multigrid preconditioner with symmetric Gauss-Seidel smoothing:

```
class AMGSGS : public AMGPreconditioner
{
  public:
    virtual void initialize(SystemMatrix* A, Vector<Num>& x,
                            Vector<Num>& b);
  protected:
    virtual void smooth(int level, Vector<Num>& e, SystemMatrix& A,
                        Vector<Num>& r);
};
```

The `AMLJacobi` type is also called BPX-preconditioner [BPX90]. If the smoothing pattern is restricted to the new nodes on each level, it is equivalent to the hierarchical basis preconditioner in [DLY89].

## 9.3 Preconditioners for Nonlinear Problems

These preconditioners are applied to the nonlinear relaxation-type solvers. They are equipped with a pointer to the class `NonLinearity` (see section 3.3), which yields the appropriate updates for every iteration step.

### Nonlinear Single-Grid Gauss-Seidel Preconditioning

The preconditioner `NonLinearSGGS` realizes the classical nonlinear Gauss-Seidel iteration.

### Nonlinear Multilevel Gauss-Seidel Preconditioning

Here the coarse grid problems are substituted by linear approximations, which are constructed such that the monotone decrease of the energy functional is guaranteed. Thus the procedure is globally convergent [Kor93].

In every iteration the coarse grid problems are set up due to the result of one nonlinear Gauss-Seidel step on the fine grid. It determines the free boundary and the obstacle functions for the coarse-grid corrections.

### Truncated Nonlinear Multilevel Gauss-Seidel Preconditioning

In this variant the coarse grid basis functions are truncated appropriately in the neighbourhood of the free boundary. This is realized by modified restriction procedures for the residuals and the coarse grid matrices. The approach creates an optimal support for the coarse grid problem and may yield drastically improved convergence rates.

# 10 The Grid Transfer

The grid transfer routines play an essential part in multilevel algorithms. Their main tasks are restriction of residuals, prolongation of solutions and transformations between nodal and hierarchical basis representations of such objects.

In a general way these operations can be considered as matrix-vector multiplications, thus revealing their rather simple algebraic nature. If we define a prolongation matrix $P_l$, the related grid transfer is given by

$$u_l \;=\; P_l\, u_{l-1} \;,$$

where $u_l$ is the solution on level $l$. For all our multilevel preconditioners we use the transpose of $P_l$ for restriction (sometimes called 'full weighting'), i.e.

$$r_{l-1} \;=\; P_l^T\, r_l \;,$$

with $r_l$ denoting the residual on level $l$. For symmetric smoothers this choice results in symmetric multilevel cycles and thus preserves the symmetry of the solution algorithm for the linear system (if, for example, a conjugate gradient solver is used). Other choices, like injection, are possible but usually not to be recommended.

Every transfer matrix $P_l$ is sparse and coded in a data structure called `Generation`. All generations of a mesh are collected on a stack within the class `FamilyTree`. There is one generation for every refinement level, in which each node of this level is represented by the class `Son`. A son stores the node numbers of his father nodes and their weights.

We briefly sketch the classes `FamilyTree` and `Generation`:

```
class FamilyTree
{
  protected:
    Stack<Generation*>  generation;

  public:
    virtual void prolong (const Vector<Num>& eLow, Vector<Num>& eHigh,
                          int highLevel) const
                 { generation[highLevel]->prolong(eLow, eHigh); }
    virtual void restrict(const Vector<Num>& rHigh, Vector<Num>& rLow,
                          int highLevel) const
                 { generation[highLevel]->restrict(rHigh, rLow); }
    virtual void solToNB(Vector<Num>& e, int level) const;
                  // hierarchical basis transformation of a
                  //                                  solution vector
    virtual void rhsToHB(Vector<Num>& r, int level) const;
                  // hierarchical basis transformation of a
                  //          right-hand-side vector (e.g. a residual)
    ...
};

class Generation
{
  protected:
    Vector<Son*> son;

  public:
    virtual void prolong (const Vector<Num>& eLow, Vector<Num>& eHigh)
                                                                 const;
    virtual void restrict(const Vector<Num>& rHigh, Vector<Num>& rLow)
                                                                 const;
    ...
};
```

The `FamilyTree` is maintained by the class `Interface`. The latter extracts the topological structure of the mesh and combines it with the node arrangement of the finite element to establish the multilevel connections of all nodes. Thus the numerical routines can operate efficiently on a simple algebraic structure even in the case of rather intricate nodal arrangements.

# 11 The Error Estimators and Adaptive Strategies

## 11.1 Error Estimation by Hierarchical Defect Correction

Here we employ a finite element basis which is a hierarchical extension of the one used in the previous solution procedure. Error Estimators of this type are described in [DLY89] and [ZdSRGK83].

In the simplest case we extend a linear finite element with quadratic hierarchical shape functions. Then the hierarchical equation system reads in block form

$$\begin{pmatrix} A_{LL} & A_{LQ} \\ A_{QL} & A_{QQ} \end{pmatrix} \begin{pmatrix} u_L \\ u_Q \end{pmatrix} = \begin{pmatrix} b_L \\ b_Q \end{pmatrix} \tag{8}$$

For $u_L$ we use the solution of our original low-order problem, which we assume to be exact. Neglecting the influence of $u_Q$ on $u_L$ we rewrite the quadratic part (second block row) of (8) in the form of a defect equation

$$A_{QQ}\, u_Q = b_Q - A_{QL}\, u_L = r_Q$$

For the solution of this system we carry out one step of a Jacobi iteration with the initial solution $u_Q = 0$:

$$u_Q = D_{QQ}^{-1}\, r_Q \tag{9}$$

$D_{QQ}$ denotes the diagonal of $A_{QQ}$. Loosely speaking, $u_Q$ reflects the high-frequency components of the extended solution $(u_L, u_Q)$ and this is exactly the quantity we are interested in for local error estimation. As the Jacobi iteration may be regarded as a smoother (reducing predominantly the high-frequency error in $(u_L, u_Q)$), such an approximate solution step is quite efficient in the positive definite case. Alternatively we could say that the coupling among the quadratic shape functions is neglected.

We obtain an estimate for the energy norm $\|u\|_E$ of the global error by

$$\|u\|_E = u_Q\, D_{QQ}\, u_Q \tag{10}$$

The local components $u_{Q_j}\, D_{QQ_j}\, u_{Q_j}$ of the error are distributed appropriately to the elements of the mesh.

**Nonlinear Problems**

In this case we use a nodal basis instead of a hierarchical one to formulate a higher-order system like (8). A modification of (9) takes into account the additional nonlinear terms appearing on the diagonal. As these terms depend on the values of the solution, we cannot employ a hierarchical basis for the extended vector $(u_L, u_Q))$. However, a transformation of the result $u_Q$ into the hierarchical basis allows an error estimation by (10).

In an alternative approach we solve the complete extended system in the nodal basis by one nonlinear Gauss-Seidel step including a multigrid V-Cycle (see section 9). Usually this costly procedure does not yield significantly better results [Kor95].

## 11.2 Residual-Based Error Estimators

These estimators directly evaluate the residuals in the interior of the elements and on their boundaries [BM81, ZKGB82]. Here we only consider estimators for (1).

If $u$ is the solution within the discrete finite element space, the inner residual of an element is – with respect to (1) – given by

$$r_I \;=\; \nabla k \nabla u + f$$

The residuals on the element boundaries result from the jump of the flux normal $k\, \partial u / \partial n$ across the boundaries:

$$r_B \;=\; k_{out} \left.\frac{\partial u}{\partial n}\right|_{out} \;-\; k_{in} \left.\frac{\partial u}{\partial n}\right|_{in}$$

The indices *out* and *in* refer to the element under consideration and to its neighbours. If $h_T$ is the maximal edge length of an element $T$ and $\partial T$ its boundary, the energy norm $\eta_T$ of the error is given by

$$\eta_T^2 \;=\; \frac{h_T^2}{k} \int_T r_I^2 \,\mathrm{d}\Omega \;+\; \frac{h_T}{k} \oint_{\partial T} r_B^2 \,\mathrm{d}\Gamma$$

This error estimator may require less computational effort than the defect correction scheme of section 11.1. However, it may not be guaranteed to be correct in the asymptotic limit.

## 11.3 Mesh Refinement Strategies

Once we have calculated an estimated error for each element of the mesh, we have to determine the set of elements to be marked for refinement. The straightforward approach is to define a limit $\eta$ and to select all elements with an estimated error above this threshold.

A simple, but usually quite satisfying choice is to set $\eta$ to a fixed percentage of the maximum error encountered on the mesh:

$$\eta = \frac{1}{4} \max_T \varepsilon_T \tag{11}$$

The factor $\frac{1}{4}$ is empirical.

An attractive alternative is to use extrapolated local errors for the determination of the limit [BR78]. For the element error $\varepsilon_T$ we assume a local behaviour of the form

$$\varepsilon_T = c_T \, h_T^{P_T}$$

If element $T$ was created by a subdivision of its 'father' $T_0$, we can predict a value $\varepsilon_t$ for the sons of element $T$

$$\varepsilon_t = \frac{\varepsilon_T^2}{\varepsilon_{T_0}}$$

Then we set the limit $\eta$ to

$$\eta = \frac{1}{2} \max_t \varepsilon_t \tag{12}$$

The factor $\frac{1}{2}$ is empirical, too.

One important additional point is to refine at least a certain percentage of all elements to ensure a proper convergence behaviour even in cases where our choice for $\eta$ is too large. A minimum ratio of five per cent is the default and usually quite satisfying.

# 12 Some Utilities: Template Classes

## 12.1 Vectors

We have implemented vectors as a rather simple template class. It is possible to construct vectors of all data types; we deliberately did not incorporate numerical functions (like a dot product). The main goal was to provide index-checking on the vector bounds, which can be turned on and off by a flag defined in the declaration file `vector.h`. We give a brief overview of the class `Vector`:

```
#define CheckBoundsFlag 1         // index-checking on
```

```
template<class T>  class Vector
{
  protected:
    T*  v;                          // pointer to the actual data array
    int l, h;                       // lower and upper bound

    void allocate(int l, int h);
    void checkBounds(int i) const;

  public:
    Vector(int l, int h) { allocate(l, h); }
    Vector(int h)        { allocate(1, h); }
    virtual ~Vector()    { v+=l; delete[] v; }

    int low()  const { return l; }
    int high() const { return h; }
    virtual void resize(int newl, int newh);

    T& operator[] (int i)
    {
        if (CheckBoundsFlag) checkBounds(i);
        return v[i];
    }
    const T& operator[] (int i) const
    {
        if (CheckBoundsFlag) checkBounds(i);
        return v[i];
    }
    ...
};
```

## 12.2 Stacks

The class `Stack` is derived from `Vector`. A stack behaves like a vector of variable size; the functions `push` and `pop` are provided to extend and shrink the stack:

```
template<class T> class Stack : public Vector<T>
{
    ...
    void push(const T a);
    T pop();
    ...
};
```

Stacks are often preferable to linked lists, as they allow random access via an index.

## 12.3 Matrices

Most that was said about vectors holds for the matrix class as well. The data of a matrix are allocated as one contiguous memory block. Additionally there are pointers to the first element of each row:

```
template<class T> class Matrix
{
  protected:
    T**  row;                  // pointer to array of row pointers
    int  rl, rh, cl, ch;       // row and column bounds

    void allocate (int rl, int rh, int cl, int ch);
    void checkBounds(int i, int j) const;

    T& operator() (int i, int j)
    {
        if (CheckBoundsFlag) checkBounds(i,j);
        return row[i][j];
    }
    const T& operator() (int i, int j) const
    {
        if (CheckBoundsFlag) checkBounds(i,j);
        return row[i][j];
    }
    ...
};
```

## 12.4 Lists

We have implemented linear and doubly-linked lists: the template classes `SList` and `DList`. The latter one is used for the objects in the finite element mesh, as the removal or replacement of any element must be possible. The off-diagonal entries in the classes `MLSparseMatrix` and `MLBlockMatrix` are organized in linear lists.

## 12.5 Memory Allocators

For certain classes large numbers of objects have to be allocated and deleted during run-time. We have created two template classes of allocators. A `StaticAllocator` can be incorporated as a static member of a class. Thus the operators `new` and `delete` may be overloaded, like in the class `TR3`:

```
class TR3 : public TR
{
  protected:
    static StaticAllocator<TR3> alloc;
    ...
  public:
    void* operator new(size_t size) { return alloc.Get(); }
    void  operator delete(void* tr) { alloc.Return((TR3*) tr); }
    ...
};
```

The derived class `Allocator` cannot be incorporated as a static member and thus does not allow the overloading of `new` and `delete`. But for this reason any `Alligator` can be deleted at run-time, whereas its static counterpart is always removed at the end of program execution.

# References


[AB84] O. Axelsson and V. A. Barker. *Finite Element Solution of Boundary Value Problems. Theory and Computation.* Academic Press, Orlando, 1984.

[Ban88] R. E. Bank. PLTMG Users' Guide - Edition 5.0. Technical report, Department of Mathematics, Univ. of Calif., San Diego, 1988.

[BEK93] F. Bornemann, B. Erdmann, and R. Kornhuber. Adaptive multilevel-methods in three space dimensions. *Int. J. Numer. Meths. in Eng.*, 36, 1993.

[Bey91] J. Bey. Analyse und Simulation eines Konjugierte-Gradienten-Verfahrens mit einem Multilevel-Präkonditionierer zur Lösung dreidimensionaler, elliptischer Randwertprobleme für massiv parallele Rechner. Diplomarbeit, RWTH Aachen, 1991.

[BM81] I. Babuška and A. Miller. A posteriori error estimates and adaptive techniques for the finite element method. Technical report, BN-968, Institute for Physical Sciences and Technology, Univ. of Maryland, 1981.

[Bor91] F. Bornemann. *An Adaptive Multilevel Approach to Parabolic Equations in Two Space Dimensions.* Dissertation, Freie Universität Berlin, 1991.

[Bor94] F. Bornemann. On the convergence of cascadic iterations for elliptic problems. *Preprint SC 94-8*, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1994.

[BPX90] J.H. Bramble, J.E. Pasciak, and J. Xhu. Parallel multilevel preconditioners. *Math. Comp.*, 55, 1990.

[BR78] I. Babuška and W. D. Rheinboldt. Error estimates for adaptive finite element computation. *SIAM J. Numer. Anal.*, 15, 1978.

[Deu94] P. Deuflhard. Cascadic Conjugate Gradients for Elliptic Partial Differential Equations. Algorithm and Numerical Results. In D. Keyes and J. Xhu, editors, *Proceedings of the 7th International Conference on Domain Decomposition Methods 1993*, AMS, Providence, 1994.

[DLY89] P. Deuflhard, P. Leinen, and H. Yserentant. Concepts of an adaptive hierarchical finite element code. *IMPACT*, 1, 1989.

[Duf80] I. S. Duff. MA28 – A Set of FORTRAN Subroutines for Sparse Unsymmetric Linear Equations. Technical report, AERE–R.8730, Harwell, 1980.

[ELR93] B. Erdmann, J. Lang, and R. Roitzsch. KASKADE Manual Version 2.0: FEM for 2 and 3 Space Dimensions. Technical report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1993.

[Kor93] R. Kornhuber. Monotone multigrid methods for elliptic variational inequalities II. *Preprint SC 93-19*, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1993.

[Kor95] R. Kornhuber. *Monotone Multigrid Methods for Nonlinear Variational Problems*. Habilitationsschrift, Freie Universität Berlin, 1995.

[Mey92] Scott Meyers. *Effective C++. 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, Massachusetts, 1992.

[Ong89] M. E. Go Ong. Hierarchical Basis Preconditioners for Second Order Elliptic Problems in Three Dimensions. Ph.D. Thesis, University of California, Los Angeles, 1989.

[Riv84] M. C. Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *Int. J. Numer. Meths. in Eng.*, 20, 1984.

[Son89] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 10(1), 1989.

[SS86] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7(3), July 1986.

[vdV92] Henk A. van der Vorst. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 13(3), March 1992.

[ZdSRGK83] O. C. Zienkiewicz, J. P. de S. R. Gago, and D. W. Kelly. The hierarchical concept in finite element analysis. *Computers and Structures*, 16, 1983.

[Zha88] S. Zhang. Multilevel Iterative Techniques. Ph.D. Thesis, Pennsylvania State University, 1988.

[ZKGB82] O. C. Zienkiewicz, D. W. Kelly, J. Gago, and I. Babuška. Hierarchical finite element approaches, error estimates and adaptive refinement. In J. R. Whiteman, editor, *Mathematics of Finite Elements and Applications*. Academic Press, 1982.