# UG – A FLEXIBLE SOFTWARE TOOLBOX FOR SOLVING PARTIAL DIFFERENTIAL EQUATIONS

P. BASTIAN, K. BIRKEN, K. JOHANNSEN, S. LANG,
N. NEUSS, H. RENTZ-REICHERT, C. WIENERS

*Institut für Computeranwendungen III, Numerik für Höchstleistungsrechner*
*Universität Stuttgart, Pfaffenwaldring 27, 70569 Stuttgart*
*email:* {peter,wieners}@ica3.uni-stuttgart.de
*www:* http://www.ica3.uni-stuttgart.de

## Abstract

Over the past two decades, some very efficient techniques for the numerical solution of partial differential equations have been developed. We are especially interested in adaptive local grid refinement on unstructured meshes, multigrid solvers and parallelization techniques.

Up to now, these innovative techniques have been implemented mostly in university research codes and only very few commercial codes use them. There are two reasons for this. Firstly, the multigrid solution and adaptive refinement for many engineering applications are still a topic of active research and cannot be considered to be mature enough for routine application. Secondly, the implementation of all these techniques in a code with sufficient generality requires a lot of time and know-how in different fields.

*UG* (abbreviation for *U*nstructured *G*rids) has been designed to overcome these problems. It provides very general tools for the generation and manipulation of unstructured meshes in two and three space dimensions as well as a flexible data layout. Therefore, it can serve on the one hand as a tool for exploring new algorithms and, on the other hand, a whole range of algorithms already implemented can be applied to complex problems.

In this paper, we show the software design structure of *UG* and explore some of the subsystems in more detail. Finally, we try to illustrate the capabilities of the approach with several non–trivial examples.

# 1 Introduction

Over the past two decades, very efficient techniques for the numerical solution of partial differential equations have been developed. Most notably these are:

- use of unstructured meshes for the approximation of complex geometries;
- adaptive local grid refinement in order to minimize the number of degrees of freedom required for a certain accuracy;
- robust multigrid methods for the fast solution of systems of linear equations;
- parallelization of these algorithms on MIMD type machines.

Up to now, these innovative techniques have been implemented mostly in university research codes [2, 11, 21, 4] and only very few commercial codes use them, e. g. [22]. The reason for this is twofold. First, the construction of fast and robust iterative solvers is still a problem. Multigrid methods have been applied very successfully in the field of computational fluid dynamics [16, 20, 26] but the application to problems from nonlinear structural mechanics or multiphase–flow in porous media is still in its infancy.

The second reason is that the integration of all the above–mentioned techniques in a single code requires a major coding effort of the order of several tens of man–years. Moreover, the structure of existing codes is often not suited to incorporate all these methods since they require a strong interaction between mesh generator, error estimator, solver and load balancer.

*UG* (shorthand for *U*nstructured *G*rids) has been designed to overcome these problems by providing reusable software tools that simplify the implementation of parallel adaptive multigrid methods on unstructured meshes for complex engineering applications. The heart of *UG* is its unstructured grid data structure. It allows one to create meshes consisting of triangular, quadrilateral, tetrahedral, pyramidal, hexahedral and prism elements in two and three space dimensions. The mesh data structure is hierarchical and elements can be refined and removed locally.

The geometric data structure is complemented by the algebraic data structure used to represent sparse matrices and vectors. The degrees of freedom can be associated with nodes, edges, faces and elements of the mesh, thus also allowing the implementation of nonconforming, mixed or higher–order finite element discretizations. A large number of linear algebra subroutines, iterative kernels and multigrid components is available. For standard situations, like conforming finite elements, the user does not have to write a single line of code in order to use the multigrid method (even for systems of partial differential equations). The implementation of discretization schemes is simplified by a large collection of routines providing shape functions and their derivatives, quadrature formulas, finite volume constructions etc.

*UG* is intended primarily to be a tool to explore new discretization schemes, solvers and error estimators. A powerful graphical user interface can help to reduce development time significantly. *UG* has a built–in shell with command interpreter and allows the user to open any number of windows on his screen. Meshes, contour lines, color

Table 1: Applications implemented with *UG*.

| Name | Comment |
|---|---|
| diff2d | diffusion equation, triangular finite elements |
| scalar | convection–diffusion, 2D/3D, all element types, FE/FV |
| fem | linear elasticity, 2D/3D, all element types, many FE schemes |
| ns | incompressible Navier–Stokes equation, 2D/3D, all element types |
| df | density–driven flow in porous media, 2D/3D, FV scheme |
| pm | two–phase flow in porous media, 2D/3D, FE/FV schemes |
| kns | Euler and compressible Navier–Stokes, 2D |
| fe | linear and nonlinear structural mechanics, 2D, parallel |
| bingham | Bingham fluid flow, 2D/3D |

plots and vector plots can be displayed in two and three dimensions. Hidden line removal in 3D efficiently uses the hierarchical data representation.

Table 1 shows the applications that have been implemented so far in our group. The first six applications are compatible with the latest version, the other applications run under older versions.

A further great advantage of *UG* is its support for parallelism. The experiences from a first parallel version described in [4] have led to a new programming model *DDD* that can be used for the parallelization of applications with graph–like data structures as described in [8]. *DDD* is the basis of the parallel *UG* version but can be used independently of it. *UG* will allow a very smooth transition from sequential to parallel computation.

The rest of this paper is organized as follows. The next section gives a short introduction to local multigrid methods on unstructured meshes, since this is the mathematical basis of *UG*. Then the software tools provided by the *UG* library are explained in more detail. Finally, results obtained with some of the problem classes listed above are used to demonstrate the power of our approach.

# 2 Local multigrid methods

The standard multigrid method (see e. g. [14]) can be applied to unstructured grids and local refinement, but it must be modified in order to avoid a deterioration of the complexity of the algorithm. In particular, the smoothing process must be performed only in the refined region. The minimal requirement is the smoothing of all new points on the finer levels. This is analyzed for an additive multigrid variant by YSERENTANT [27] and for the corresponding multiplicative variant by BANK-DUPONT-YSERENTANT [3]. Here, the convergence rates depend on the number of levels. Optimal complexity can be obtained by the additive multigrid method with smoothing on all refined elements by BRAMBLE-PASCIAK-XU [10] and its multiplicative counterpart BRAMBLE-PASCIAK-WANG-XU [9] (see [5] for a discussion of the different approaches). Here, we explain in detail the concept which was realized by BASTIAN [4] and which forms the basis of

*UG*. We use a multiplicative multigrid method with smoothing in a slightly enlarged refined region. Up to now, this is the only way of obtaining optimal complexity combined with robustness for a wide range of applications.

## 2.1 Grids and Multigrids

A *grid* is a closed polygonal or polyhedral set and a decomposition into elements. We consider triangular and quadrilateral elements resp. tetrahedrons, pyramids, prisms and hexahedrons. For an element $E$ the corresponding domain in $\mathbb{R}^d$, $d = 2, 3$ is denoted by $\Omega(E)$ and the boundary is denoted by $\partial E$. A grid is *consistent*, if $\partial E \cap \partial E'$ is empty, a common node (corner), a common edge or a common side for all elements $E \neq E'$.

Let $\Omega \subset \mathbb{R}^d$ be a domain. A *multigrid* $\mathcal{T} = \{\Omega_0, \Omega_1, \Omega_2, ..., \Omega_k\}$ is a sequence of consistent grids approximating $\Omega$. The set of elements is denoted by $\mathcal{E}_l$, $l = 0, ..., k$. Elements of different grids will be distinguished. For $l > 0$, we assume that there exists a *father* element $\mathcal{F}(E) \in \mathcal{E}_{l-1}$ for all elements $E \in \mathcal{E}_l$. In general, $\Omega(E) \subset \Omega(\mathcal{F}(E))$, but this property will be violated near curved boundaries. For $F \in \mathcal{E}_{l-1}$, let

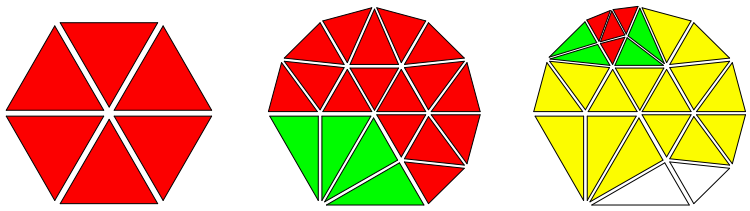
$$\mathcal{S}(F) = \{E \in \mathcal{E}_l \mid \mathcal{F}(E) = F\}$$

be the set of *son* elements of $F$. The refinement of an element $F$ to $\mathcal{S}(F)$ can be of type *regular*, *irregular* or *copy*. The application of a refinement rule results in the generation of elements of the corresponding type, e. g. application of an irregular refinement rule to $F$ results in $\mathcal{S}(F)$ being irregular elements. For copy elements, we trivially have $\Omega(F) = \Omega(\mathcal{S}(F))$.
The global refinement algorithm conforms to the following rules:

1. all elements of $\Omega_0$ are regular;
2. regular and irregular refinement may be applied only to regular elements;
3. the copy rule may be applied to any element.

For a complete discussion of the refinement rules and the refinement algorithm cf. sec. 3.3. The copy elements are needed in order to cover the domain $\Omega$ on all levels. Most of the copy elements can be omitted in the final implementation as will be explained below. In the following, we will speak of an element as *refined* if it is refined either regularly or irregularly (but not copy).

**Example 1.** Multigrid for the unit circle $\Omega = \{x \in \mathbb{R}^2 \mid |x| < 1\}$.



$\Omega_0$ $\Omega_1$ $\Omega_2$

## 2.2 Geometrically based data

We assume that a linear problem is given on $\Omega$ and that we can define a finite dimensional solution approximating the continuous solution on $\Omega$ for every grid approximating $\Omega$. Therefore, vectors and matrices have to be defined on every grid. For simplicity, we explain only the case where the unknowns are assigned to the nodes. The set of nodes on level $l$ is denoted by $\mathcal{P}_l$. We call $P \in \mathcal{P}_l$ point or interpolation point and the position will be denoted by $\mathrm{pos}(P)$. The points on different levels will be distinguished, but they may have the same position. The corners of an element $E$ are denoted by $\mathcal{P}(E)$. In general, father points cannot be defined for refined elements, but for all points $P \in \mathcal{P}(E)$ on copy elements $E$ there exists a well defined father point $\mathcal{F}(P)$. Furthermore, $n[P]$ denotes the number of degrees of freedom associated with $P$. $n[E] = \sum_{P \in \mathcal{P}(E)} n[P]$ resp. $n_l = \sum_{P \in \mathcal{P}_l} n[P]$ denotes the number of degrees of freedom per element resp. in $\Omega_l$.

Let $\mathcal{T} = \{\Omega_0, ..., \Omega_k\}$ be a multigrid and let

$x_l \in \mathbf{R}^{n_l}$ be a vector for the grid on level $l$,
$x[E] \in \mathbf{R}^{n[E]}$ the restriction to an element $E \in \mathcal{E}_l$ and
$x[P] \in \mathbf{R}^{n[P]}$ the restriction to an interpolation point $P \in \mathcal{P}_l$.

On the grid of the top level $\Omega_k$, the discretized problem

$$A_k x_k = b_k$$

is given by the global stiffness matrix $A_k \in \mathbb{R}^{n_k \times n_k}$ and the global right–hand side $b_k \in \mathbb{R}^{n_k}$, the global solution vector is denoted by $x_k$. For example, a finite element discretization is constructed from element stiffness matrices $A(E)$ and element right–hand sides $b(E)$. The global stiffness matrix $A_k$ and the global right–hand side vector $b_k$ are assembled from $\sum_{E \in \mathcal{E}_k} A(E)$ resp. $\sum_{E \in \mathcal{E}_k} b(E)$ and a modification due to boundary conditions for some points $P \in \mathcal{P}(\partial\Omega_k)$. For the lower levels $l < k$, we only need the stiffness matrix $A_l$, because in a multigrid algorithm, auxiliary problems $A_l c_l = d_l$ are solved, where the right–hand side is replaced by the defect $d_l$ and the solution vector is replaced by the correction vector $c_l$.

Given two interpolation points $P$ and $Q$, the submatrix $A[P,Q] \in \mathbb{R}^{n[P] \times n[Q]}$ is constructed from all matrices $A(E)$ where $P, Q \in \mathcal{P}(E)$. Here, we only consider discretizations where $A[P,Q] = 0$ if there is no element with $P, Q \in \mathcal{P}(E)$.

## 2.3 Grid transfer

For multigrid methods, the grid transfer is essential for the coupling of values on different levels. In general, the interpolation from level $l-1$ to level $l$ is given by matrices $I_l \in \mathbb{R}^{n_l \times n_{l-1}}$. Locally, the submatrices $I[E] \in \mathbb{R}^{n[E] \times n[F]}$ and can be constructed by pointwise multilinear interpolation from a father element $F$ to a son element $E \in \mathcal{S}(F)$. Thus, the interpolation is a local operation and $I[P,Q] \neq 0$ only if there is an element $E \in \mathcal{E}_l$ such that $P \in \mathcal{P}(E)$ and $Q \in \mathcal{P}(\mathcal{F}(E))$. Usually, the grid transfer from a fine level to a coarse level will be defined by the transposed matrix $R_l = I_l^T$. Furthermore, the stiffness matrices on the coarser levels can be computed by $A_{l-1} = R_l A_l I_l$. Various concepts exist for adapting the grid transfer and the coarse grid matrix to varying diffusion constants or dominating convection.

## 2.4 Vector classes

In the implementation of multigrid methods on locally refined grids, the solution on higher levels will be computed for refined elements only. In addition, we require that the defect computation and the smoothing can be performed on every level without reference to lower levels. Therefore, we build local grids with all refined elements and some copy elements. For the precise definition, the concept of vector classes will be introduced.

The vector classes are defined recursively:

| | |
|---|---|
| set $cl(P) = 3$, | if there exists a refined element $E$ such that $P \in \mathcal{P}(E)$, |
| set $cl(P) = 2$, | if $cl(P) \neq 3$ and if there exists an element $E$ such that $P, Q \in \mathcal{P}(E)$ and $cl(Q) = 3$, |
| set $cl(P) = 1$, | if $cl(P) \neq 2$, $cl(P) \neq 3$ and if there exists an element $E$ such that $P, Q \in \mathcal{P}(E)$ and $cl(Q) = 2$, |
| set $cl(P) = 0$ | for all remaining points $P \in \mathcal{P}_l$. |

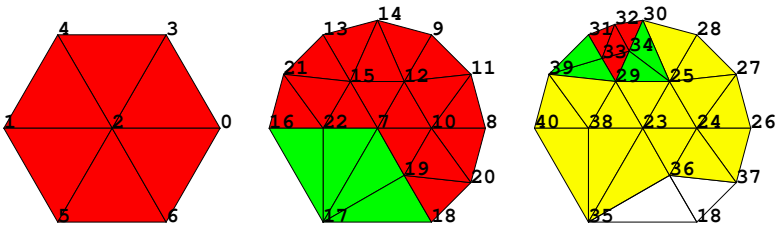
A vector $x_0, x_1, ..., x_k$ is *consistent*, if $x[P] = x[\mathcal{F}(P)]$ for all points with $cl(P) < 3$. Analogously, a matrix $A_0, A_1, ..., A_k$ is consistent, if $A[P, Q] = A[\mathcal{F}(P), \mathcal{F}(Q)]$ for all points with $cl(P) < 3$. By construction, the stiffness matrix $A$ and the right–hand side $b$ are consistent.

Let $x_0, ..., x_k$ be consistent and $d_l = b_l - A_l x_l$ the defect. If $x_l[P]$ for $cl(P) = 3$ is changed, the new defect will change for points $P$ with $cl(P) \geq 2$. In order to compute the new defect, one needs $x_l[P]$ for all points with $cl(P) \geq 1$. This motivates the following definition: the *local grids* $\Omega_l^{loc}$ are the smallest subsets of the *surface grids* $\Omega_l$, such that $P \in \mathcal{P}(\Omega_l^{loc})$ for all $P$ with $cl(P) \geq 1$, i. e.

$$\mathcal{E}(\Omega_l^{loc}) = \{E \in \mathcal{E}_l \mid cl(P) \geq 2 \text{ for some } P \in \mathcal{P}(E)\}.$$

On the base level we have $\Omega_0^{loc} = \Omega_0$.

**Example 2.** We consider a discretization where the interpolation points are built by the set of all nodes. Then, the surface grids of example 1 define the following local grids.




The points on the base level get class 3. Here, all points on level 1 get class 3 too. On the local grid $\Omega_2^{loc}$, the points $P_{25}, P_{29}, P_{30}, P_{31}, P_{32}, P_{33}, P_{34}, P_{39}$ get class 3, the points $P_{23}, P_{24}, P_{27}, P_{28}, P_{38}, P_{40}$ class 2 and the points $P_{26}, P_{35}, P_{36}, P_{37}$ class 1.

The son of point of $P_{18}$ in $\mathcal{P}_2$ is of class 0, but this point and the corresponding two elements will not be copied to $\Omega_2^{loc}$.

The local multigrid method will be defined such that the solution vector $x_0, ..., x_k$ and the correction $c_0, ..., c_k$ are always consistent. If the stiffness matrix $A_l[\Omega_l^{loc}]$ and the right–hand side $b[\Omega_l^{loc}]$ are assembled on the local grids, they are not consistent. Thus, $b[P]$ and $A[P, P]$ for $cl(P) \leq 1$ must not be used on local grids. Nevertheless, the defect $d_l = b_l - A_l x_l$ on the surface grid can be constructed recursively from the local defects $d[\Omega_l^{loc}] = b[\Omega_l^{loc}] - A[\Omega_l^{loc}]x[\Omega_l^{loc}]$:

for $l = 1, ..., k$ set $\quad d_l[P] = d[\Omega_l^{loc}][P]$ for $cl(P) \geq 2$ and
$\qquad\qquad\qquad\qquad\quad d_l[P] = d_{l-1}[\mathcal{F}(P)]$ for $cl(P) < 2$.

We do not need the values $d_l[P]$ for $cl(P) < 2$; they will be set to zero in the smoothing step. Therefore, the restriction of $d_l$ must not change $d_{l-1}[Q]$ for $Q \in \mathcal{P}_{l-1}$ such that $cl(\mathcal{S}(Q)) < 2$. For a simple notation, we set $ncl(Q) = cl(\mathcal{S}(Q))$ if $\mathcal{S}(Q)$ is a point on a copy element and $ncl(Q) = 3$ if $Q \in \mathcal{P}(\mathcal{F}(E))$ for a refined element $E$. Then, the set of points where $d_{l-1}[Q]$ can be changed get $ncl(Q) \geq 2$.

## 2.5 Smoother and solver

We formulate the multigrid method for the surface grids, but all steps will be defined such that only points on the local grids are used. The basic step on every level is the *smoothing*.

$S(c_l, d_l) = \{$ set $d_l[P] = 0$ for $cl(P) < 2$
solve $W_l w_l = d_l$,
set $w_l[P] = 0$ for $cl(P) < 3$
$d_l := d_l - A_l w_l$,
$c_l := c_l + w_l$ $\}$

The solution process of the equation $W_l w_l = d_l$ should be local, e. g. $W_l =$ blockdiag$(A_l)$ or $W_l = LU$ (incomplete LU decomposition).

A *multigrid cycle* is defined recursively and combines the grid transfer and several smoothing steps. The cycle depends on the parameters $\theta < 1$, $\nu_0 + \nu_1 \geq 1$ and $\gamma \geq 1$.

$MC(c_l, d_l) = \{$ $c_l = 0$
if $l = 0$
$\varepsilon = \theta \|d_0\|$
call $S(c_0, d_0)$ until $\|d_0\| < \varepsilon$ (coarse grid solver)
else
for $j = 0, ..., \nu_0$ call $S(c_l, d_l)$ (pre-smoothing)
$w_{l-1} = R_l d_l$ (restriction)
$d_{l-1}[P] = w_{l-1}[P]$ for $ncl(P) \geq 2$
for $j = 1, ..., \gamma$ call $MC(c_{l-1}, d_{l-1})$ (coarse grid corr.)
$w_l = I_l c_{l-1}$ (prolongation)
$d_l := d_l - A_l w_l$
$c_l := c_l + w_l$
for $j = 0, ..., \nu_1$ call $S(c_l, d_l)$ $\}$ (post-smoothing)

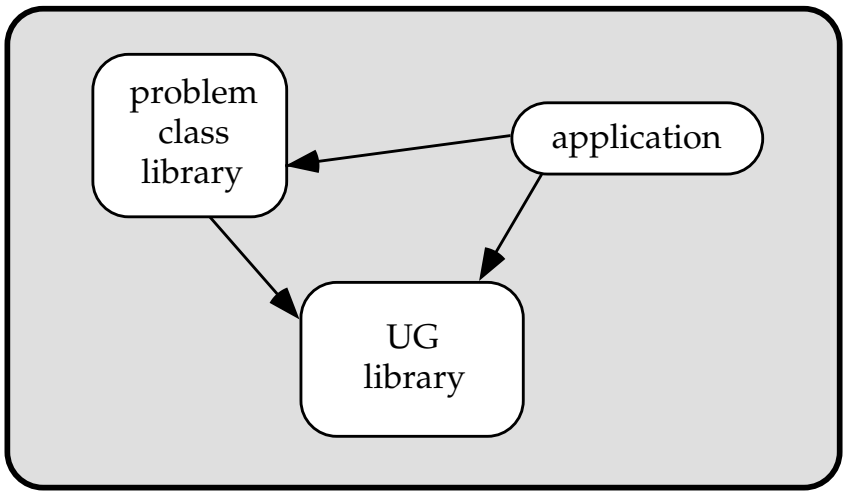



Figure 1: The overall *UG* architecture design.

Now, we can define a *multigrid solver*.

$$
\begin{aligned}
\mathrm{MS}(x_k, b_k) = \{ \quad & d_k = b_k - A_k x_k,\ \varepsilon = \theta \|d_k\| \\
& \text{repeat} \\
& \quad \text{call } \mathrm{MC}(c_k, d_k) \\
& \quad x_k := x_k + c_k \\
& \text{until } \|d_k\| < \varepsilon \quad \}
\end{aligned}
$$

Here, no update of the defect is necessary because after every multigrid cycle the equality $d_k = b_k - A_k(x_k + c_k)$ holds. The defect computation and the update of the solution are global operations on the surface grid which can be constructed of local operations using vector classes.

# 3 The UG library

A large software system like *UG* is usually described at a number of different levels of abstraction. In this chapter, we move through this hierarchy from top to bottom where, on the lower levels, we will focus on a few essential parts due to space limitations. *UG* knows three design levels which are called *architectural design*, *subsystem design* and *component design*.

At least on the architecture and subsystem level, *UG* is a modular design and the information hiding principle is used extensively. All state information is distributed among the subsystems. *UG* is implemented in the C programming language.

## 3.1 Architecture Design

The highest level of abstraction in *UG* is the architecture design level shown in Fig. 1. This decomposition is motivated as follows:

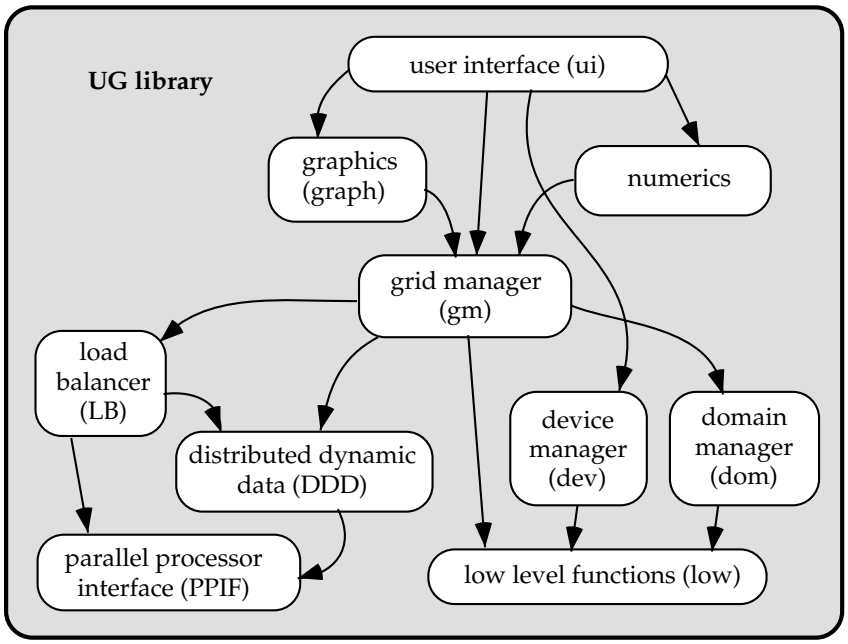



Figure 2: Subsystem design of the *UG* library.

UG LIBRARY
: The *UG* library is *completely independent* of the partial differential equation to be solved. It provides the geometric and algebraic data structures and a huge number of mesh manipulation options, numerical algorithms, visualization techniques and the user interface.

PROBLEM CLASS LIBRARIES
: This part provides discretization, error estimator and, if required, non–standard solvers for a particular set of partial differential equations.

APPLICATIONS
: The application finally provides the domain description, boundary conditions and coefficient functions in order to complete the problem description. A simulation run is typically controlled by a script file that is interpreted by *UG*'s user interface.

## 3.2 UG Library Subsystem Design

Each of the building blocks of the architectural design is decomposed into several subsystems. The subsystems of the *UG* library are shown in Fig. 2. We now give an informal specification of the services provided by each subsystem. Some of the subsystems will then be explained in more detail in the next subsections.

USER INTERFACE
: The user interface provides the user with a "shell–like" command language. All operations of the *UG* library can usually be executed either via a command typed into the shell or by calling a C function within the code. A scripting language is available to control complex simulation runs. Multiple graphics windows can be opened to visualize simulation results. In two space dimensions the mesh can be manipulated interactively. The user interface is based on the portable device interface described below.

GRAPHICS
: The graphics subsystem provides some elementary visualization methods like mesh plots, contour plots, color plots or vector fields. In three dimensions planar cuts and hidden line removal have been implemented. The advantages of an integrated graphics package are that no intermediate data files have to be written and also that information like matrix structure and entries can be displayed easily. The graphics subsystem is described in more detail below.

NUMERICS
: The numerics subsystem provides numerical algorithms in a modular form ranging from basic linear algebra (level 1 and 2) up to methods for the solution of nonlinear time–dependent partial differential equations. In addition, it provides support for the discretization process, e. g. quadrature rules. The numerics subsystem is described in more detail below.

GRID MANAGER
: The grid manager subsystem provides the unstructured mesh and sparse matrix data structures together with functionality for their manipulation. This includes the generation of two– and three–dimensional simplicial triangulations. The grid manager is explained in more detail below.

DOMAIN MANAGER
: The purpose of the domain manager is to provide functionality for the description of general two– and three–dimensional domains as well as functions on the surface (boundary conditions) and the interior (coefficients) of a domain. The general approach is that a $d$–dimensional domain $\Omega$ is described by its boundary $\partial\Omega$ which is a $d-1$–dimensional hypersurface. This is very natural in the context of partial differential equations since boundary conditions have to be provided anyway. The standard way of describing the boundary is through local maps $f_i : \mathbb{R}^{d-1} \to \mathbb{R}^d$ with $i = 1 \ldots n$ and $n$ the number of patches. Another approach consists of decomposing the boundary in a number of patches where each patch is given by a simplicial surface mesh. In that case, no easy mapping $f_i$ exists for a patch. The domain interface will also be used to access CAD data.

DEVICE MANAGER
: The device manager provides a default device called "screen" that allows at least basic character input/output. Optionally, the screen device also has interactive graphics capabilities. The screen device has been implemented for the standard C library, X11, remote X11 (uses socket communication and an X11 capable daemon on a remote machine), and Apple Macintosh. Write only graphical output is available in postscript and a binary format ("metafile").

LOW
: This subsystem provides some basic functionality like memory management, a simple database tool and portable file input/output. Furthermore, some debugging tools are included.

LOAD BALANCER
: The load balancer subsystem is intended to solve graph partitioning and scheduling problems that arise when mapping data must be mapped to processors in a parallel environment. The current implementation uses CHACO [15, 17] for that purpose.

DYNAMIC DISTRIBUTED DATA (DDD)
: The *DDD* subsystem implements a new parallel programming model that is especially suited for managing distributed "graph–like" data structures. Data objects can be created, deleted and transferred between processes easily. Communication among distributed objects is supported in a flexible and efficient way. This subsystem is explained in more detail below.

PARALLEL PROCESSOR INTERFACE (PPIF)
: PPIF is a portable message passing interface used by DDD. It has been implemented for PVM, MPI, PARIX, NX and the T3D/T3E. PPIF has very little overhead when used with fast native communication (e. g. shared memory get/put on the T3D).

## 3.3 The Grid Manager Subsystem

Each subsystem is decomposed into components. Components are typically implemented in a single file or a few files. Fig. 3 shows the component design of the grid manager subsystem. The main components are coarse–grid generation, refinement and derefinement of a hierarchical multigrid structure and provision of standard shape functions for various element types. Automatic coarse–grid generation is currently only possible for simplicial grids. For the three-dimensional case, the grid generator of J. Schöberl [23] has been connected to *UG*.

In the following, we concentrate on two aspects of the grid manager, the unstructured grid data structure and the refinement algorithm.

#### Unstructured Grid Data Structure

The heart of *UG* is its unstructured grid data structure which consists of a "geometric" and an "algebraic" part. The geometric part is element–based, i. e. all information is locally accessible from the element. The `ELEMENT` data type is given in Table 2.

The unsigned integer component `control` is used bitwise to store various pieces of information such as the element type (triangle, quadrilateral, tetrahedron, etc.), how the element is refined, the number of son elements, etc. The `id` component gives a globally unique identification for the element which is mainly used by the user interface. All elements of one grid level `l`, i. e. the set $\mathcal{E}_l$, are contained in a double linked list implemented by the `pred` and `succ` pointers. The `n[]` array contains references to

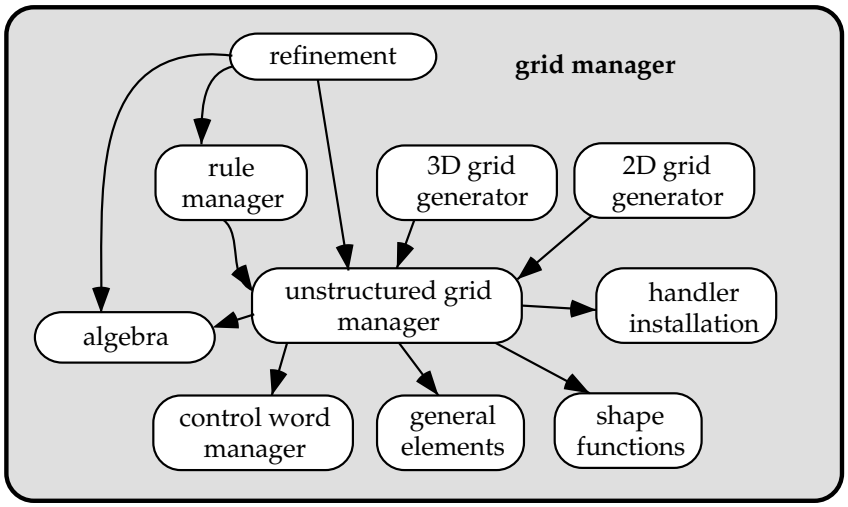



Figure 3: Component design of the grid manager subsystem.

Table 2: The ELEMENT data type.

| ELEMENT | | |
|---|---|---|
| UINT | control | used bitwise |
| INT | id | unique id |
| ELEMENT | *pred, *succ | double linked list |
| NODE | *n[nc] | reference to corner nodes |
| ELEMENT | *father, *son | element hierarchy |
| ELEMENT | *nb[ns] | neighbors over each face |
| VECTOR | *vector | dof's associated with element |
| VECTOR | *sidevector[ns] | dof's associated with faces |
| BNDS | *bnds[ns] | boundary information |

the corners of the element, which are of type NODE. The father and son pointers provide access to the father and son elements on the lower and higher grid levels. Note that only one reference to a son element is stored. The remaining son elements can be easily accessed by knowledge of the refinement rule. The nb[] array provides access to all neighboring elements over each of the faces (edges in 2D). All algebraic information associated with a single geometric object is stored in a data structure of type VECTOR. The vector component of an element refers to the algebraic information associated with this element. Since *UG* does not have a data structure for faces in 3D, the algebraic information associated with a face in 3D is also accessible from the element via the sidevector[] array. Finally, the bnds[] array provides access to additional information if one or more faces are part of the domain boundary. The BNDS data type is exported by the domain subsystem. The ELEMENT data structure is generic,

Table 3: The NODE and VERTEX data types.

| NODE | | |
|---|---|---|
| UINT | control | used bitwise |
| INT | id | unique id |
| NODE | *pred, *succ | double linked list |
| LINK | *start | neighbor node list |
| NODE | *father, *son | node hierarchy |
| VERTEX | *myvertex | position information |
| VECTOR | *vector | dof's associated with node |

| VERTEX | | |
|---|---|---|
| UINT | control | used bitwise |
| INT | id | unique id |
| COORD | x[dim] | global coordinates |
| COORD | xi[dim] | local coordinates in father |
| ELEMENT | *father | position in coarser grid |
| BNDP | *bndp | position on domain boundary |

i. e. the actual number of pointers depends on the context in which the element is used. The numbers nc and ns are given by the element type. In addition, certain pointers may not exist at all, e. g. bnds[] is omitted for interior elements, sidevector[] is omitted if no degrees of freedom in faces are required, etc.

The next data types to be considered in detail are the NODE and VERTEX data types given in Table 3. The nodes on different grid levels of the multigrid hierarchy are distinguished but, since they often share the same position with nodes on coarser grid levels, the position information is stored in the VERTEX data type which is not duplicated over the levels. The essential information of the VERTEX data type is the global position in its x component, the position with respect to the next coarser grid level given by its xi and father components and the boundary information in the bndp component. Interpolation from a coarser grid level is done by using a reference to a coarse grid element father that contains the global position and local coordinates xi within that element. This allows one to move new nodes away from their hierarchical position, e. g. to resolve internal boundaries.

The NODE data structure again has a control component holding status information, a globally unique id and pred and succ components implementing a double linked list. The start component points to a single linked list of LINK structures which implements a list of all neighboring nodes in the grid. The father and son components give access to the nodes sharing the same VERTEX structure, given by myvertex on the next coarser and finer grid levels. Finally, vector refers to the degrees of freedom associated with the node.

A linked list of neighbors of a node is realized with the LINK data type shown in Table 4. Each list element contains a control word, a reference to the next list element and a reference to the neighboring node. Since node–neighborship is a symmetric

Table 4: The LINK and EDGE data types.

| LINK | | |
|---|---|---|
| UINT | control | used bitwise |
| LINK | *next | neighbor node list |
| NODE | *nbnode | neighbor node |

| EDGE | | |
|---|---|---|
| LINK | links[2] | bi–directional links |
| NODE | *midnode | new node on finer grid |
| VECTOR | *vector | dof's associated with edge |

Table 5: The VECTOR and MATRIX data types.

| VECTOR | | |
|---|---|---|
| UINT | control | used bitwise |
| GEOM_OBJ | *object | associated geometric object |
| VECTOR | *pred, *succ | double linked list |
| INT | index | block number |
| INT | skip | component skip info |
| MATRIX | *start | corresponding block row |
| DOUBLE | value[$n\nu$] | block components |

| MATRIX | | |
|---|---|---|
| UINT | control | used bitwise |
| MATRIX | *next | single linked row list |
| VECTOR | *vect | column block |
| DOUBLE | value[$nm$] | block components |

relation, we can combine two LINK structures into an EDGE structure also given in Table 4. In addition, the EDGE data type contains references to the degrees of freedom associated with the edge and to the new node on the finer mesh obtained by a subdivision of the edge. Note that there is fast access from a given LINK structure to its EDGE and the LINK for the opposite direction.

This completes the description of the geometric part of the data structure. The algebraic part uses a (modified) block compressed row storage format to store matrices and vectors. The vector and matrix blocks are implemented by the VECTOR and MATRIX data types shown in Table 5.

The VECTOR data type provides access to the geometric object it is associated with, i. e. an object of type NODE, EDGE or ELEMENT. Additional information in the control word indicates if the vector block is associated with a face of an element. All vector blocks on a grid level are connected by a double linked list allowing easy re-

Table 6: The GRID and MULTIGRID data types.

| GRID | | |
|---|---|---|
| UINT | control | used bitwise |
| INT | level | grid level |
| GRID | *coarser,*finer | grid hierarchy |
| ELEMENT | *firstel,*lastel | element list |
| NODE | *firstnd,*lastnd | node list |
| VECTOR | *firstvc,*lastvc | vector list |
| MULTIGRID | *mg | associated multigrid |

| MULTIGRID | | |
|---|---|---|
| UINT | status | status information |
| INT | top, current | levels |
| GRID | *grids[ng] | access to grid levels |
| FORMAT | *theFormat | block components |
| HEAP | *theHeap | memory management |
| BVP | *theBVP | boundary value problem |

ordering. The index component contains the block number and the skip component indicates whether a row of the matrix corresponds to Dirichlet boundary conditions and has to be skipped in the grid transfer. The matrix block row associated with a given vector block is accessed via the start component. Finally, the algebraic components are stored in the value[] array. Note that the VECTOR data type is generic, i. e. the number of components $n\nu$ may vary.

The MATRIX data type realizes a block row of the matrix with a single linked list. The diagonal block is always the first block in this list. The number of values $nm$ stored in a single block depends on the type of geometric objects associated with the row and column blocks, i. e. a node–node coupling may contain a different number of components than a node–edge coupling.

Note that there is only one object of type VECTOR per geometric object. The number $n\nu$ of entries in the value field not only contains the degrees of freedom but also the right–hand side and additional storage needed by the numerical algorithms (defect, correction, last time step, etc.). The same is true also for the MATRIX objects which may actually contain several matrices, e. g. the stiffness matrix and an incomplete decomposition of the stiffness matrix. The numerics subsystem provides a symbolic mechanism for describing vectors and matrices to be used in actual computations. The components of these vectors and matrices are in general subsets of the components stored in the VECTOR and MATRIX data structures.

The mesh data structure is completed by providing access to the various linked lists of elements, nodes, vertices and vectors. The form of the numerical algorithms suggests a level–wise access to the data structure. The GRID data type shown in Table 6 provides this access. Since all the lists are double linked, access to the first and last elements is possible.

All grid levels are then combined in the MULTIGRID data type. It provides access to all grid levels, some status information and the context of the multigrid hierarchy. The component theFormat refers to a structure of type FORMAT which describes the number of DOUBLE values to be reserved in the VECTOR and MATRIX data structures per geometric object. In addition, one can control the interconnection pattern of the VECTOR objects. All memory needed by a multigrid data structure is allocated from a special HEAP implemented in the low subsystem. This memory manager is especially suited to manage many small objects efficiently. Finally, each MULTIGRID data structure refers to a description of a boundary value problem (domain, boundary condition, coefficient functions) of type BVP which is exported by the domain subsystem.

One may argue that this unstructured grid data structure requires a lot of memory. This is certainly true if one considers scalar model problems. However, if we consider complex three–dimensional applications, most of the memory is occupied by numerical data. To illustrate this, consider the two–phase flow application presented in the next section. The three–dimensional simulation on a 32 by 32 by 32 hexahedral mesh requires 102 MB of memory. 66 MB are occupied by the entries of the stiffness matrix, the ILU matrix and the unknown and auxiliary vectors.

**The Refinement Algorithm**

From the discussion in section 2 it is clear that a multigrid sequence $\mathcal{T} = \{ \Omega_0, \Omega_1^{loc}, \Omega_2^{loc}, ..., \Omega_k^{loc} \}$ should satisfy the following properties:

*nestedness:* Each element $E$ on a grid level $l > 0$ has a unique father element $F = \mathcal{F}(E)$ with $\Omega(E) \subset \Omega(F)$. As far as the code is concerned this inclusion is not necessary but then additional conditions for stability are needed;

*consistency:* Each (local) grid level $\Omega_l^{loc}$ is a consistent grid (see definition in subsection 2.1);

*stability:* All interior angles must be uniformely bounded away from zero.

The refinement algorithm works with a variety of element types and refinement rules. In general, a refinement rule applied to an element $E$ allows $\Omega(E)$ to be retriangulated by using the corners, edge centers, face centers and the element center of $E$. All element types can be used for this purpose. Regular refinement rules are those that are stable when applied successively. Fig. 4 shows regular refinement rules for various element types. For tetrahedrons, the regular refinement strategy of BEY [6] is used. In contrast to the other element types, the four–sided pyramid can not be subdivided by using pyramids only. It is subdivided into 6 pyramids and 4 tetrahedra.

If only regular refinement rules are applied, the resulting grids are automatically consistent. This is in contrast to bisection type refinement strategies in three dimensions (cf. [19]). If only some elements are refined regularly, the consistency of the grids must be ensured by matching edge and face refinement patterns in the transition region. This is achieved by introducing irregular refinement rules which may decrease interior angles. The refinement algorithm in *UG* works with a full set of refinement rules, i. e. *there always exists a refinement rule matching any edge and face refinement pattern*.

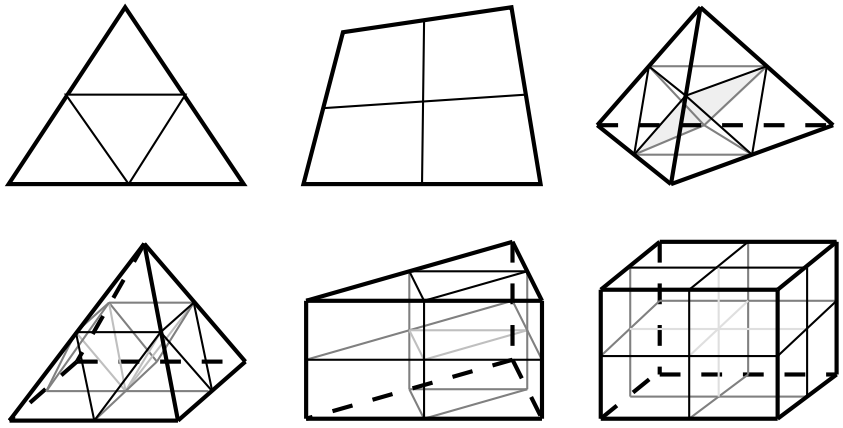


Figure 4: Regular refinement rules.

For the two–dimensional case and the simplicial case in three dimensions, the number of rules is so small that they can be stored in a data base. In the other cases, an appropriate rule is generated on demand by connecting the element center with quadrilaterals and triangles on the faces, resulting in pyramidal and tetrahedral elements [18]. Due to the full set of refinement rules, no iteration is required to find the consistent closure which is especially useful in the parallel version.

Overall stability is ensured by allowing only regular elements to be refined. If irregular elements must be refined, they have to be replaced by regular elements first which in turn must be consistent with neighboring elements. It is the task of the global refinement algorithm to match the refinement requested by the error estimator with consistency and stability. Below we give a high–level version of the global refinement algorithm. As input this algorithm receives a multigrid hierarchy $\mathcal{T}$ where the leaf elements (those without sons) have been tagged with a desired refinement rule or a coarsening tag. If all sons of an element are tagged for coarsening, they are deleted from the data structure. Of course, only leaf elements can be removed, since coarsening should be an inverse operation to refinement.

```
RefineMultiGrid(T, k) = {   for l = k, k − 1, ..., 2, 1          (1)
                              MakeConsistent(Ω_l^loc)            (2)
                              RestrictTags(Ω_l^loc, Ω_{l−1}^loc) (3)
                            for l = 0, 1, ..., k                 (4)
                              MakeConsistent(Ω_l^loc)            (5)
                              DetermineCopies(Ω_l^loc)           (6)
                              RefineGrid(Ω_l^loc)    }           (7)
```

The overall structure of algorithm RefineMultiGrid resembles a multigrid V–cycle. The first for–loop (1) proceeds from the top level $k$ to level 1. *No* manipulation of the data structure is done in this first loop, only the refinement tags are manipulated! The function MakeConsistent computes a consistent grid for level $l+1$ by tagging elements of level $l$ with a consistent set of refinement rules. Function RestrictTags computes the

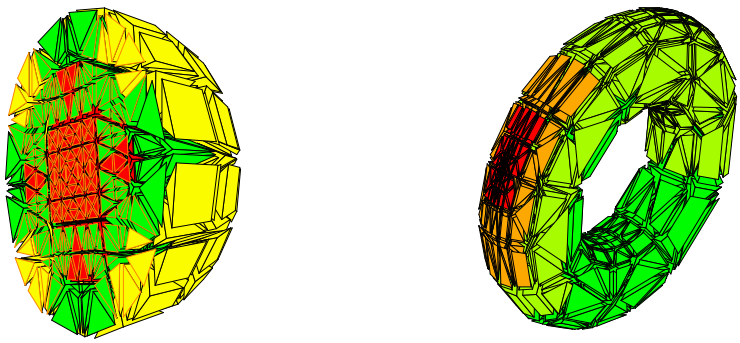


Figure 5: Examples for locally refined grids.

influence of level–l–tags on level–l–1–tags. This is where irregular refinement rules are replaced by regular refinement rules if necessary. In the second for–loop starting in line (4), the data structure is actually modified proceeding from coarse to fine levels. When the second loop is entered in line (5), the grid on level $l$ has already been modified and level $l+1$ is constructed from the tags stored on level $l$. Again MakeConsistent is called to compute consistent tags in line (5); then DetermineCopies computes the algebraic classes which in turn determine the copy elements required on level $l+1$. The function RefineGrid in line (7) actually modifies the data structure, i. e. appropriate objects are created/deleted on level $l+1$ and pointers are set correctly. After completion the funtion RefineMultiGrid has constructed a new multigrid hierarchy. The top level of this hierarchy may be either $k-1$, $k$ or $k+1$. The level–wise structure of the algorithm is very well suited to parallelization as is shown in [4]. Fig. 5 shows a cut through a locally refined ball triangulated with pyramidal elements and a torus.

## 3.4 The DDD Subsystem

In order to exploit the power of parallel computers for *UG* applications, the grid manager subsystem is extended to handle grids which are distributed across the processor's local memories. Given the complexity of the data structures involved and their dynamic changes because of grid refinement and load balancing, straightforward usage of message-passing programming models will certainly be doomed to failure due to the lack of abstraction of communication and distribution methods. Therefore, an abstract programming model supporting high-level operations on the distributed grid data has been designed and integrated into the *UG* library. From the architecture's viewpoint this *Dynamic Distributed Data* (*DDD*) model is implemented as a *UG* subsystem; a standalone version is nevertheless available in order to allow parallelization of other applications as well.

The basic abstraction *DDD* supports is the notion of *distributed graphs*. The graph nodes correspond to those application's data structures, which should be distributable on different processors (*DDD objects*); the graph edges represent references between

Table 7: Overview of the components in the DDD library.

| | |
|---|---|
| INTERFACES | Supports communication operations on existing static data topologies. Interfaces are subsets of distributed objects at inter-processor boundaries and can be used in a transparent manner after their initial definition. They are kept consistent despite all dynamic data changes by the other components. |
| TRANSFER | Provides procedures for creating object copies on remote processors or deleting local object copies. This enables dynamic changes of the data topology at runtime. The most important application is the transparent transfer of DDD objects across local memory boundaries without troublesome address space conversions. This component allows easy implementation of dynamic load balancing techniques with arbitrary grid overlapping strategies. |
| IDENTIFICATION | Creation of distributed objects via identification of local objects. This is possible during the complete program run making dynamic grid changes possible (e. g. for adaptive grid refinement). |

the data structures (cf. formal specification document [8]). Distributed objects are used to couple several local graphs (one for each processor) in order to build the distributed graph. The DDD library implementation contains three components with different functionality described in Table 7, which can be used to manipulate the distributed graph structure and to invoke efficient communication procedures (see the *DDD Reference Manual* [7] for a detailed description).

Concerning the *UG* parallel grid manager, the following data types are regarded as DDD objects:

`ELEMENT`: Elements are the central data structures for distributed grids. As a result of dynamic load balancing, the elements are marked with a destination processor. A grid migration algorithm executes the redistribution of the grid by calling DDD commands from the Transfer component. The redistribution of all other data types depends directly on the element migration. The grid overlapping strategy is formulated in terms of *Master* and *Ghost* elements.

`NODE`: The overlapping of nodes in a distributed grid depends on the element overlapping scheme. Three kinds of nodes exist: *Master*, *Border* and *Ghost* nodes. For the identification of newly created nodes (during adaptive refinement), the DDD Identification component is used.

`VERTEX`: The replication of vertices is directly linked with the replication of the corresponding nodes. DDD provides mechanisms which allow the expression of this dependency in a simple way.

`VECTOR`: The distribution of this algebraic data type also depends on the element distribution. As each vector is coupled with its copies on other processors, the DDD Interface component can be used in order to make vectors consistent which

Table 8: Plotobjects available in the graphics subsystem.

| Name | dimension | description |
|---|---|---|
| Matrix | 2D/3D | standard representation of a sparse-matrix |
| VecMat | 2D | graph-like representation of the sparse-matrix-structure |
| Grid | 2D/3D | mesh (3D: optional with an arbitrary cut) |
| EScalar | 2D/3D | contour/color plot of a scalar-field (3D: evaluated on an arbitrary cut of the mesh) |
| EVector | 2D/3D | plot of a vector-field (3D: evaluated on an arbitrary cut of the mesh) |
| Line | 2D | two-dimensional graph of a scalar-field along an arbitrary line within the domain |

have been stored in an inconsistent manner before. Typically this is done from the numerics subsystem by invoking a DDD Interface communication with three parameters: the size of data which should be communicated for each `VECTOR` data structure, a *gather*-procedure which copies the data from the `VECTOR` structure into the message buffer, a *scatter*-procedure which integrates the data from the message buffer into the `VECTOR`.

By using DDD the parallel grid manager is the same as the sequential one, both in terms of its usage interface and its program code. The former is advantageous for a clear algorithmic and hierarchical abstraction, the latter is necessary with respect to ongoing code development and maintenance. Apart from the grid manager subsystem, noticeable code adaptions have been necessary only for the numerics subsystem; hence, any sequential *UG* application using standard numerical techniques can use all the advantages of state-of-the-art parallel computing without additional effort for parallelization.

## 3.5 The Graphics Subsystem

The graphics subsystem in *UG* provides some elementary visualization methods as well as some interactive tools. Depending on the capabilities of the device manager, the graphical output is displayed on an X11 window, written to a binary metafile format or to a postscript file ("devices"). An arbitrary number of windows can be opened on different devices simultaneously.

The basic objects in the graphics subsystem are the *plotobjects* (represented by the structure *PLOTOBJ*) and the *work* (represented by the structure *WORK*). *Plotobjects* are representations of (abstract) objects, for example meshes (2D/3D) or the graph of a sparse matrix (2D). They contain a reference to the *multigrid* structure serving as a database. Different *methods* can be applied to plotobjects. The simplest is the *draw method*, which displays a *plotobject*. Others are able to change the *plotobject*'s configuration like the *findrange method*, changing, for example the color range of a *plotobject* containg the evaluation of a scalar-field. Again, other *methods* can change the state of the underlying database (i.e. *multigrid* structure) like the *movenode method*.

Table 9: Methods implemented in the graphics subsystem.

| Name | dimension | changes | description |
|---|---|---|---|
| draw | 2D/3D | no | display |
| findrange | 2D/3D | *plotobject* | find the range of a scalar- or vector-field |
| selectnode | 2D | no | select a node |
| selectelement | 2D/3D | no | select an element |
| markelement | 2D | no | mark an element for refinement |
| insertnode | 2D | *multigrid* | insert a node |
| movenode | 2D | *multigrid* | move a node |
| insertbndnode | 2D | *multigrid* | insert a boundary node |
| selectvector | 2D | no | select a vector |

Table 10: Overview of the components in the graphics subsystem.

| | |
|---|---|
| WINDOW PICTURE MANAGER | The window picture manager provides the management of *windows* and *pictures*. A *window* represents a kind of wall to which *pictures* can be attached. A *picture* displays a single *plotobject*. The functionality includes the opening and closing of *windows* and *pictures* as well as the configuration of the *plotobjects*. |
| GRAPH | The graph component provides a set of low level drawing routines based on the device manager. Typical low level drawing routines are Move, Draw, Line, PolyLine, etc. |
| WORK ON PICTURE | The work–on–picture component implements the *methods* applicable to a *plotobject*. |

Table 8 lists the *plotobjects* available and gives a short description of them. Table 9 lists the *methods* implemented in *UG* together with a short description. Note that not every *method* can be applied to every *plotobject*. Finally, Table 10 describes the three components of the graphics subsystem in more detail.

## 3.6 The Numerics Subsystem

The numerics subsystem in *UG* offers a wide range of numerical tools for computations with vectors and matrices in the geometrical based data structure of the grid manager (cf. sec. 2.2). All algebraic objects can be described and referenced by abstract symbols in *UG*-scripts.

Using scripts the numerical components can be freely configured, e. g. the cg method can use a multigrid cycle as a preconditioner, the multigrid cycle can use all types of smoothers and grid transfers. This linear solver can be called in a Newton method, which can be applied in every time step. Thus, for a typical time-dependent nonlinear problem, the user only provides the assembling routines for the defect computation and the assembling of the stiffness matrix for the linearized problem on element

Table 11: Overview of the components in the numerics subsystem.

| | |
|---|---|
| BASICS | clear, copy, scale, axpy, dot for vectors (level 1)<br>clear, copy, diagonal scaling for matrices,<br>matrix-vector product (level 2) |
| GRID TRANSFER | standard interpolation and restriction for conforming multilinear approximations based on local coordinates, matrix based grid transfer, assembling routines for the interpolation matrix based on the stiffness matrix or a Schur complement |
| DISCRETIZATION TOOLS | finite volume computations, quadrature formulas for finite elements, modifications of the stiffness matrix for Dirichlet boundaries, assembling tools, Galerkin assembling of the stiffness matrix |
| SMOOTHER | Jacobi, Gauß-Seidel, symmetric Gauß-Seidel, SOR, incomplete LU decomposition, ILU with extension of the sparsity pattern of the stiffness matrix, ILU with $\beta$-modification or with spectral shift |
| SOLVER | multigrid solver, cg method, newton method with step length control, fixed point iteration, inverse iteration for eigenvalue computations, one- and two step implicit time solver |

level. A general–purpose hierachical error indicator is provided for time–dependent grid adaption, but to obtain the optimal grid, a problem dependent error estimator is usually required. For the flexible combination of the components on script level, we developed the concept of numerical procedures.

In many cases, the problem class and the application do not need any parallel extensions, since all global numerical operations are included in the *UG* library. In particular, gather and scatter routines are called in the BLAS routines and in the smoothing procedures.

# 4 Applications

## 4.1 The finite element library

For many applications, the multilinear conforming approximation is not the optimal choice. Particularly, if the solution is smooth and the geometry of the domain is simple and can be resolved on the coarsest grid, the convergence of higher–order elements is far better. Nonconforming and mixed elements can be used in order to reduce locking phenomena in linear elasticity, effective error bounds can be obtained by higher–order mixed elements and fourth–order problems can be approximated by $C^1$-elements.

In *UG*, a general finite element library is realized [25]. Therefore, we consider interpolation points $\mathcal{P}$ on nodes, edges, faces and elements.

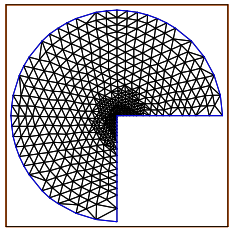
Table 12: Finite Element Discretizations implemented with *UG*.

| | |
|---|---|
| DIFFUSION EQUATION | |
| linear conforming Lagrange elements (P1) | 1 d.o.f. per node |
| quadratic conforming Lagrange elements (P2) | 1 per node, 1 per edge |
| linear nonconforming elements (Crouzeix-Raviart) | 1 per side |
| Raviart-Thomas elements RT0 | 1 per side |
| Raviart-Thomas elements RT1 | $d$ per side |
| Brezzi-Douglas-Martini elements (BDM) | $d$ per side |
| LINEAR ELASTICITY | |
| linear conforming Lagrange elements | $d$ per node |
| quadratic conforming Lagrange elements | $d$ per node, $d$ per edge |
| Stabilization of linear nonconforming elements (Falk) | $d$ per side |
| Stabilizition of the BDM elements (Stenberg) | $d^2$ per side |
| PLATE EQUATION | |
| nonconforming quadratic elements (Morley) | 1 per node, 1 per edge |
| Argyris elements | 6 per node, 1 per edge |
| STOKES EQUATION | |
| P2 – P1 elements | $d+1$ per node, $d$ per edge |

This list can be easily extended: essentially, the assembling of the local stiffness matrices $A(E)$ and the local interpolation matrices $I(E)$ must be implemented for every discretization, then all other tools can be applied.

First, we compare the discretizations for a scalar test problem in a reentrant corner domain.

$$-\Delta u = 0 \text{ in}$$
$$\Omega = \{z \in \mathbf{C} \mid |z| \le 1,\ 0 \le \arg z \le 3\pi/2\},$$
$$\text{exact solution } u(z) = \operatorname{Im} z^{2/3}$$



Here, the refinement strategy ensures an optimal order of convergence in all cases (cf. Table 13).

The next example demonstrates the approximation of a diffusion problem with varying coefficients.

$$\begin{aligned} -\operatorname{div} a(x) \operatorname{grad} u(x) &= 1 && |x| < 1 \\ u(x) &= 0 && |x| = 1 \\ a(x) &= 1 && |x| > 0.5 \\ a(x) &= 0.001 && |x| \le 0.5 \end{aligned}$$

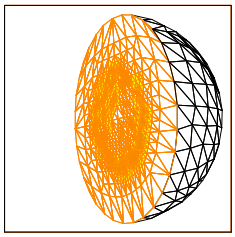


Table 13: Examples for different finite element discretizations

| P1 | | | P2 | | | CR | | |
|---|---|---|---|---|---|---|---|---|
| n | it | err | n | it | err | n | it | err |
| 12 | 1 | 0.29 | 12 | 1 | 0.17 | 12 | 1 | 0.28 |
| 48 | 2 | 0.19 | 44 | 2 | 0.087 | 44 | 3 | 0.19 |
| 174 | 2 | 0.12 | 110 | 2 | 0.056 | 172 | 3 | 0.12 |
| 312 | 2 | 0.086 | 218 | 2 | 0.023 | 434 | 4 | 0.082 |
| 738 | 2 | 0.057 | 372 | 2 | 0.011 | 860 | 4 | 0.055 |
| 1230 | 2 | 0.040 | 842 | 2 | 0.0044 | 2132 | 5 | 0.029 |
| 2858 | 2 | 0.027 | 1650 | 2 | 0.0022 | 5522 | 5 | 0.016 |
| 4547 | 2 | 0.019 | 2807 | 2 | 0.0013 | 8757 | 4 | 0.013 |
| 9814 | 2 | 0.013 | 3673 | 2 | 0.0010 | 15481 | 5 | 0.0097 |
| 16714 | 2 | 0.010 | 6262 | 2 | 0.00061 | 22614 | 4 | 0.0078 |
| 34147 | 2 | 0.0071 | 8686 | 2 | 0.00042 | 35765 | 4 | 0.0062 |
| 63768 | 2 | 0.0051 | 18790 | 2 | 0.00019 | 61348 | 4 | 0.0047 |
| 121046 | 2 | 0.0036 | 30951 | 2 | 0.00012 | 91244 | 4 | 0.0038 |
| $O(n^{-1/2})$ | | | $O(n^{-1})$ | | | $O(n^{-1/2})$ | | |

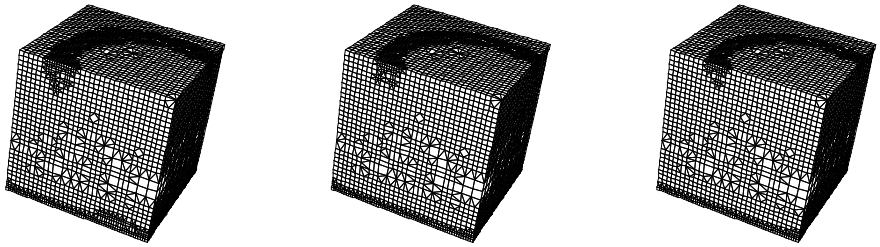
| RT1 | | | RT2 | | | BDM | | |
|---|---|---|---|---|---|---|---|---|
| n | it | err | n | it | err | n | it | err |
| 12 | 1 | 0.28 | 12 | 1 | 0.13 | 12 | 1 | 0.13 |
| 46 | 3 | 0.19 | 44 | 3 | 0.078 | 44 | 4 | 0.078 |
| 148 | 3 | 0.12 | 90 | 7 | 0.053 | 124 | 4 | 0.032 |
| 246 | 3 | 0.092 | 124 | 8 | 0.033 | 284 | 4 | 0.0145 |
| 420 | 3 | 0.065 | 162 | 7 | 0.024 | 466 | 4 | 0.0066 |
| 690 | 3 | 0.049 | 296 | 8 | 0.017 | 948 | 4 | 0.0032 |
| 1155 | 3 | 0.037 | 506 | 8 | 0.0077 | 1886 | 4 | 0.0016 |
| 1740 | 3 | 0.028 | 778 | 7 | 0.0057 | 3132 | 4 | 0.00084 |
| 2658 | 3 | 0.023 | 1022 | 7 | 0.0039 | 5980 | 4 | 0.00045 |
| 4642 | 3 | 0.017 | 1320 | 7 | 0.0026 | 7803 | 4 | 0.00032 |
| 6931 | 3 | 0.014 | 1963 | 8 | 0.0021 | 11038 | 4 | 0.00024 |
| 10633 | 3 | 0.011 | 2580 | 8 | 0.0014 | 18207 | 4 | 0.00013 |
| 18547 | 4 | 0.0085 | 3347 | 8 | 0.0010 | 35270 | 4 | 0.000071 |
| $O(n^{-1/2})$ | | | $O(n^{-1})$ | | | $O(n^{-1})$ | | |

n number of elements

it number of iterations for reducing the discrete defect of the interpolated solution of the previous step by the factor $10^{-3}$

err error of the approximation in energy norm

Finally, we give an example in linear elasticity with time-dependent surface load. Here, a grid consisting of hexahedrons, pyramids and tetrahedrons is refined and coarsened simultaneously. It is plotted for three subsequent time steps.



## 4.2 Density–driven flow in porous media

In many cases, groundwater flow in porous media involves the transport of solutes that affect liquid density. If density variations exceed 20%, which occurs near salt domes or bedded salt formations, flow and transport are strongly coupled. The primary coupling arises in the equations through the body–force term of the fluid equation and the advection term of the transport equation. A second coupling arises from the velocity–dependent hydrodynamic dispersion in the transport equation. These couplings cause nonlinearities in the equations that preclude analytical solutions and are a challenge for numerical simulations.

Density–driven flow problems can be described by two nonlinear, coupled, time–dependent differential equations, a continuity equation for the fluid and a continuity equation for the solute transport. The fluid continuity equation is written in terms of pressure, assuming that Darcy's law is valid. Both equations are discretized on vertex-centered finite volumes using different constructions for the control volumes. In cases of dominant convection, an aligned finite volume method, where the finite volumes are aligned to a given velocity, is preferable to the standard finite volume method. Furthermore, a consistent velocity approximation of terms involved in the fluid velocity calculation is implemented. The transient equations are solved with a fully implicit time-stepping scheme with time step control. The nonlinear equations are solved in a fully coupled mode using an approximative Newton multigrid method where the linearized system is solved with a linear multigrid method.

For the problem of seawater intrusion in a coastal aquifer, known as the Henry problem, a semi-analytic steady–state solution by SEGOL [24] exists and is therefore widely used as a benchmark. However, this problem is not very sensitive with respect to the consistent velocity and description of small transition zones. Figure 6 shows the steady–state 20%, 50% and 80% concentration isolines in comparison with the analytic solution and other numerical results [12][13]. The numerical solutions obtained all match the analytical solution very closely.

Another test problem is the so–called Elder problem. A closed rectangular box, modelled in cross section, with a source of solute on the top boundary is considered. We have modified this problem, dividing the domain in five subdomains. In four of the subdomains, the permeability value is reduced by a factor of $10^3$ compared to the fifth

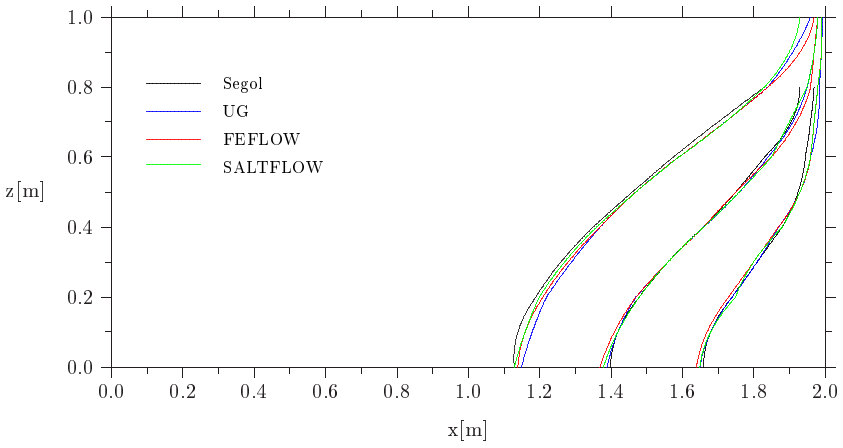
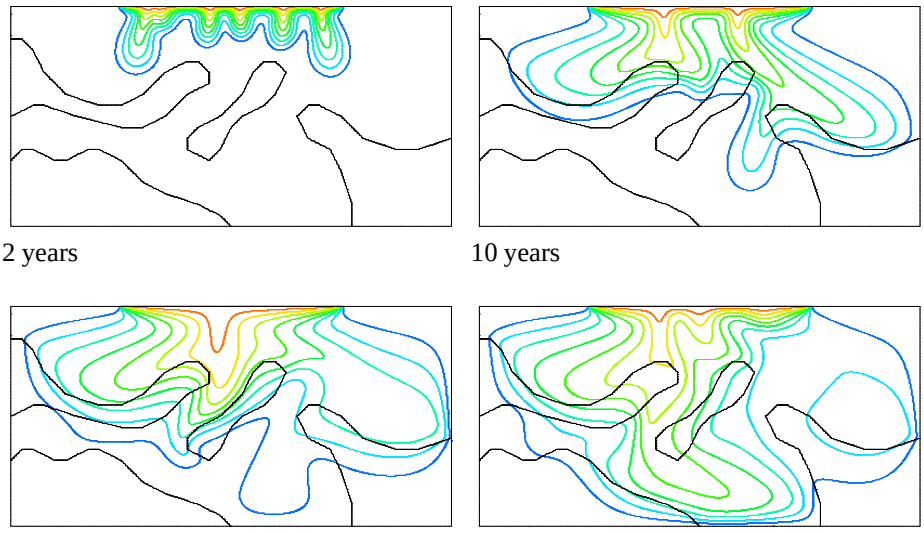



Figure 6: Comparison of calculated isochlors with the analytical solution of the Henry problem; freshwater inflow from the left and fixed seawater concentration at the right. The isochlor values are (from left to right) 20%, 50% and 80% of seawater concentration.



2 years 10 years

20 years 30 years

Figure 7: Salt concentration isolines of the modified Elder problem at different times.

subdomain. The calculations are done on an unstructered grid with 50849 nodes on the finest grid using the aligned finite volumes. Figure 7 shows the temporal evolution of the salt distribution.

Table 14: Simulation data for 2D infiltration problem

| Elements | Elapsed [s] | Newtons | MG/Newton | MaxMG |
|---|---|---|---|---|
| 288 | 512 | 495 | 1.7 | 4 |
| 1152 | 2580 | 613 | 2.0 | 5 |
| 4608 | 14132 | 741 | 2.8 | 8 |
| 18432 | 79478 | 953 | 3.2 | 12 |
| 73728 | 435178 | 1561 | 2.9 | 10 |

### 4.3 Two–phase Flow in Porous Media

The flow of two immiscible fluids in a porous medium is described by two coupled highly non–linear time–dependent partial differential equations, see [1] for an introduction. These equations play an important role in the development of effective in–situ remediation techniques. Due to the hyperbolic/parabolic character of the equations, strong heterogeneities and high non–linearity, they pose a challenging problem for multigrid solution.

A problem class has been developed that solves the two–phase flow equations in a fully–coupled manner using wetting–phase pressure and non–wetting phase saturation as unknowns. A finite volume and a control–volume–finite–element discretization with first–order upwinding have been implemented. Both discretizations support all element types in two and three dimensions. Time discretization is fully implicit, resulting in a large set of nonlinear algebraic equations per time step. The nonlinear equations are then solved iteratively by a Newton–Multigrid technique. A line search method is used to achieve global convergence. Several multigrid techniques have been implemented in *UG* to handle coefficient jumps induced by saturation fronts and absolute permeability variations. These jumps are in general *not* aligned with coarse grid lines. In the simulations below, a multigrid method with diagonal scaling, point–block ILU smoother ($\nu_1 = \nu_2 = 2$) and a W-cycle ($\gamma = 2$) has been used.

Figure 8 shows a saturation plot for the infiltration of a dense non–aqueous phase liquid (DNAPL, a fluid with density higher than water and immiscible with it) into a randomly generated porous medium. The simulation history is shown in Table 14. The table shows that the average and maximum number of multigrid iterations stays bounded with increasing mesh size (and fixed time–step size).

Figure 9 shows saturation iso–surfaces for a three–dimensional DNAPL infiltration into a heterogeneous porous medium. A block of low permeability has been inserted into the center of the reservoir. Entry pressure effects prevent the DNAPL from infiltrating the low permeability lense. The multigrid performance for this problem is shown in Table 15.

## 5 Conclusions

We described in detail the basic ideas and the software design structure of *UG*. The concepts and the implementation of most of the subsystems in the *UG* library are now

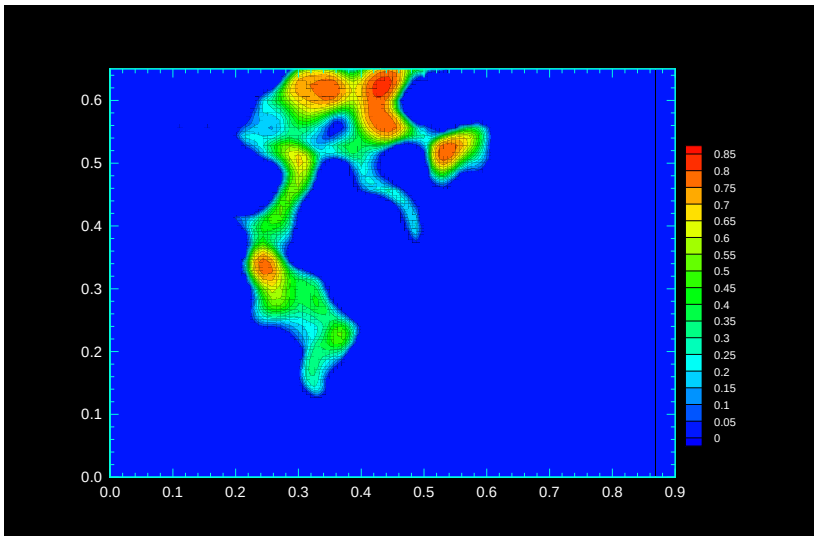



Figure 8: 2D DNAPL infiltration into a randomly generated porous medium after 240 time steps.

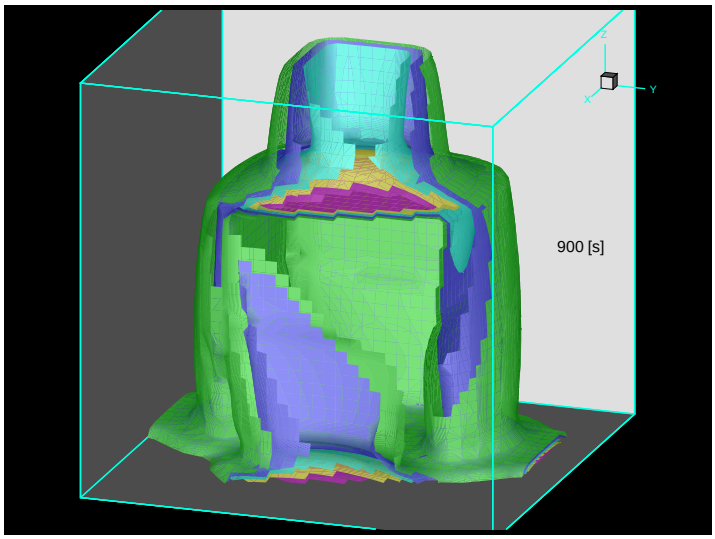



Figure 9: 3D DNAPL infiltration into a heterogeneous aquifer after 90 time steps.

essentially finished, but the components will be extended further. In future work, we will focus on the different applications and large simulations of complex problems with several million unknowns. Although software design was the main emphasis in this paper, one should have in mind that the development of efficient and robust numerical methods is at least as important. Our work in this direction is illustrated by the

Table 15: Simulation data for 3D infiltration problem

| Elements | Elapsed [s] | Newtons | MG/Newton | MaxMG |
|---|---|---|---|---|
| 512 | 898 | 196 | 1.8 | 3 |
| 4096 | 10507 | 251 | 2.9 | 5 |
| 32768 | 100914 | 285 | 3.0 | 6 |

numerical examples.

From the point of view of the user concerned with software applicability, we can state that not only a basic code documentation but further a description of concepts and explanations of the theoretical background (both connected with examples) have to be available. Up to now, only basic documentation is provided for *UG*.

We want to emphasize that the development of this software involves a large group of coworkers and a period of several years. Our experiences during development have shown that it is imperative to include a version control system and debugging tools for efficient software design and coding. A clear definition of internal interfaces between the subsystems and external interfaces (e. g. for domain description, coarse grid generation, visual postprocessing) is essential for maintenance and extendability.

# References


[1] K. AZIZ AND A. SETTARI, *Petroleum Reservoir Simulation*, Elsevier, 1979.

[2] R. BANK, *PLTMG Users Guide Version 7.0*, SIAM, 1994.

[3] R. BANK, T. DUPONT, AND H. YSERENTANT, *The hierarchical basis multigrid method*, Numer. Math., 52 (1988), pp. 427–458.

[4] P. BASTIAN, *Parallele adaptive Mehrgitterverfahren*, Teubner Skripten zur Numerik, Teubner-Verlag, 1996.

[5] P. BASTIAN AND G. WITTUM, *On robust and adaptive multigrid methods*, in Proc. of the $4^{\text{th}}$ European Multigrid Conference, P. W. P. Hemker, ed., Birkhäuser, 1994.

[6] J. BEY, *Tetrahedral grid refinement*, Computing, 55 (1995), pp. 355–378.

[7] K. BIRKEN, *Dynamic Distributed Data in a parallel programming environment – DDD Reference Manual*, Forschungs- und Entwicklungsberichte RUS–23, Rechenzentrum der Universität Stuttgart, Germany, September 1994.

[8] K. BIRKEN AND P. BASTIAN, *Dynamic Distributed Data (DDD) in a parallel programming environment – specification and functionality*, Forschungs- und Entwicklungsberichte RUS–22, Rechenzentrum der Universität Stuttgart, Germany, September 1994.

[9] J. BRAMBLE, J. PASCIAK, J. WANG, AND J. XU, *Convergence estimates for multgrid algorithms without regularity assumptions*, Math. Comp., 57 (1991), pp. 23–45.

[10] J. H. BRAMBLE, J. E. PASCIAK, AND J. XU, *Parallel multilevel preconditioners*, Math. Comp., 55 (1990), pp. 1–22.

[11] P. DEUFLHARD, P. LEINEN, AND H. YSERENTANT, *Concepts of an adaptive hierarchical finite element code*, IMPACT of Computing in Science and Engineering, 1 (1989), pp. 3–35.

[12] H.-J. DIERSCH, *Feflow user's manual version 4.2*, tech. rep., WASY GmbH, Berlin, Germany, 1994.

[13] E. FRIND AND J. MOLSON, *Saltflow 2.0 user's guide*, tech. rep., Waterloo Centre for Groundwater Research, Canada, 1994.

[14] W. HACKBUSCH, *Multi-Grid Methods and Applications*, Springer, 1985.

[15] HENDRICKSON AND R. LELAND, *The chaco user's guide version 1.0*, Tech. Rep. SAND93-2339, Sandia National Laboratory, October 1993.

[16] E. H. HIRSCHEL, ed., *Flow Simulation with High-Performance Compters II*, Vieweg Verlag, Braunschweig, 1996.

[17] S. LANG, *Lastverteilung für paralle adaptive Mehrgitterberechnungen*, Master's thesis, Universität Erlangen-Nürnberg, IMMD III, 1994.

[18] ———, *Adaptive refinement and derefinement of unstructured grid hierarchies*, tech. rep., Universität Stuttgart, Institute for Computer Applications, 1997. In preparation.

[19] J. M. MAUBACH, *Local bisection refinement for n–simplicial grids generated by reflection*, SIAM J. Sci. Comput., 16 (1995), pp. 210–227.

[20] D. J. MAVRIPILIS, *Three–dimensional multigrid reynolds–averaged navier–stokes solver for unstructured meshes*, AIAA Journal, 33 (1995).

[21] L. C. MCINNES AND B. SMITH, *Petsc2.0: A case study of using mpi to develop numerical software libraries*, in Proc. of the MPI Developers Conference, Notre Dame, IN, 1995.

[22] M. RAW, *A coupled algebraic multigrid solver for the 3d navier–stokes equations*, in Proc. of the $10^{\text{th}}$ GAMM Seminar Kiel, Notes on Numerical Fluid Mechanics, G. W. W. Hackbusch, ed., vol. 49, Vieweg–Verlag, 1995.

[23] J. SCHÖBERL, *Analysis and realisation of mixed finite element models in computational mechanics*, Master's thesis, Universität Linz, 1995.

[24] G. SÉGOL, *Classic Groundwater Simulations*, Prentice Hall, 1994.

[25] C. WIENERS, *The implementation of adaptive multigrid methods for finite elements*, tech. rep., Universität Stuttgart, SFB 404 Preprint 97/12, 1997.

[26] G. WITTUM, *Multigrid methods for stokes- and navier-stokes equations*, Numer. Math., 54 (1989), pp. 543–563.

[27] H. YSERENTANT, *Über die Aufspaltung von Finite–Element–Räumen in Teilräume verschiedener Verfeinerungsstufen*, 1984. Habilitationsschrift.