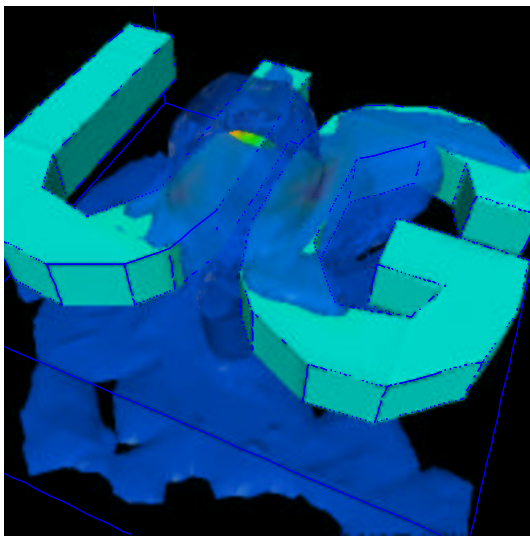


# UG Tutorial

Revision Date: October 11, 2002



P. Bastian  
K. Johannsen  
V. Reichenberger

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen  
Universität Heidelberg, Im Neuenheimer Feld 368  
D-69120 Heidelberg, Germany



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
2.1	Choosing a target directory and extracting the source code . . .	5
2.2	Setting up the UG environment . . . . .	5
2.3	Configuring UG . . . . .	6
2.3.1	Supported architectures and compilers . . . . .	7
2.3.2	More configuration . . . . .	9
	The DIM option . . . . .	9
	The IF option . . . . .	9
	The GUI option . . . . .	10
	The MODEL option . . . . .	10
	The CHACO option . . . . .	11
	The DEBUG_MODE option . . . . .	11
	The OPTIM_MODE option . . . . .	11
	The DOM_MODULE option . . . . .	11
	The GRAPE option . . . . .	11
	The COVISE option . . . . .	11
	The NETGEN option . . . . .	11
2.4	Compiling UG . . . . .	12
2.5	UG command line tools . . . . .	12
	ugclean . . . . .	12
	ugconf . . . . .	12
	uggrep, uggrepc, uggreph . . . . .	13
	ugmake, ugpmake . . . . .	13
	ugman . . . . .	13
	ugrun . . . . .	13
2.6	UG on the Macintosh . . . . .	14
2.7	How to get UG . . . . .	14
<b>3</b>	<b>Using a UG application</b>	<b>17</b>
3.1	Compiling the tutorial application . . . . .	17
3.2	A first example . . . . .	17
3.3	A sample script file . . . . .	21
3.4	2D flow and transport problem . . . . .	27
3.5	3D flow and transport problem . . . . .	28
3.6	A parallel example . . . . .	29

<b>4</b>	<b>Basic data types</b>	<b>33</b>
4.1	Unstructured mesh data structure . . . . .	33
4.1.1	MULTIGRID . . . . .	34
4.1.2	GRID . . . . .	34
4.1.3	ELEMENT . . . . .	34
4.1.4	NODE . . . . .	35
4.1.5	VERTEX . . . . .	36
4.1.6	LINK and EDGE . . . . .	36
4.2	Geometry data structure . . . . .	37
4.3	Sparse matrix vector data structure . . . . .	38
4.3.1	VECTOR . . . . .	38
4.3.2	MATRIX . . . . .	38
4.3.3	VECDATA_DESC . . . . .	39
4.3.4	MATDATA_DESC . . . . .	39
<b>5</b>	<b>The tutorial application</b>	<b>41</b>
5.1	Overview . . . . .	41
5.2	The main() function . . . . .	41
5.3	Setting up a domain . . . . .	41
5.4	Setting up a problem . . . . .	46
5.4.1	Numprocs . . . . .	46
5.4.2	An example . . . . .	48
5.4.3	Boundary conditions . . . . .	51
<b>6</b>	<b>The tutorial problem class</b>	<b>55</b>
<b>7</b>	<b>Graphics</b>	<b>57</b>
7.1	Data structures . . . . .	57
7.2	High-level methods for pictures . . . . .	66
7.3	Plotobjects . . . . .	67
	<b>Bibliography</b>	<b>79</b>

# 1

## Introduction

In this tutorial you will learn how to use the UG software to solve partial differential equations (PDEs) numerically. UG is quite large and this tutorial will not be able to explain every aspect of the code but at least it should serve as a starting point.

UG is a so-called *framework* for the numerical solution of partial differential equations. It provides tools for all parts of the numerical solution process, i. e. geometry representation, mesh generation, mesh refinement, numerical algorithms and post processing. After (Gamma et al. 1995) a framework is defined as follows:

A framework is a set of cooperating classes that make up a reusable design for a specific class of software. [...] You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework.

One of the important aspects here is that in a framework-based application the framework takes over control (possibly after some initialization) and calls user specific methods at appropriate times.

In this tutorial we will start with some installation instructions and will then look at an executable that has been prepared with UG. We will learn how to use the shell language and what commands are available to all UG applications. Then we will describe how numerical algorithms are implemented in UG in a problem-independent way such that they can be reused for various PDEs to be solved. The main part of this tutorial is the description of a complete problem class with applications that solves a porous medium flow and transport model using a finite-volume method.

The UG software package is structured into the “UG library”, so-called “problem classes” and “applications”. Figure 1.1 shows this basic structure. The UG library is *independent* of the partial differential equation and is shared by all UG users. It provides a *framework* which enables efficient (in terms of man-power) construction of new simulation programs.

The features offered by the UG library are:

- Representation of two- and three-dimensional geometries,
- representation of unstructured meshes consisting of triangles, quadrilaterals, tetrahedra, pyramids, prisms and hexahedrons,

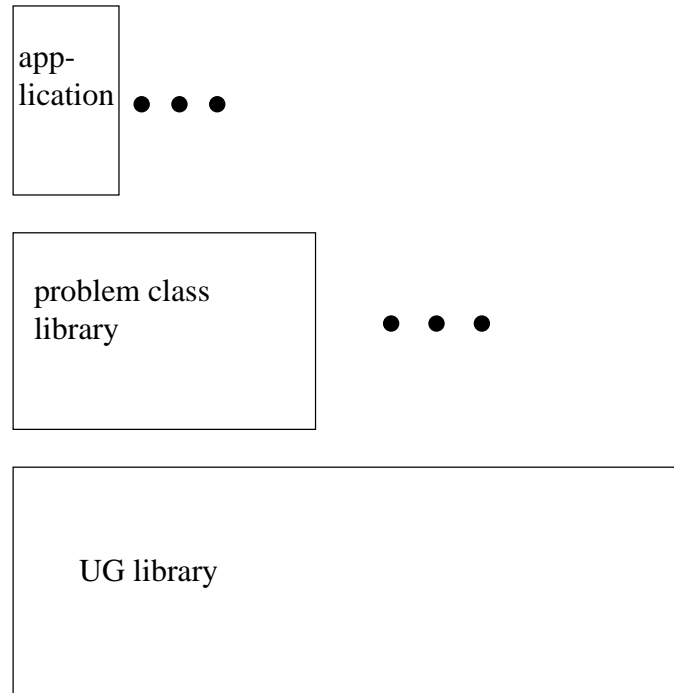


Figure 1.1: Basic Philosophy of UG applications

- built-in mesh generators for triangular and tetrahedral elements as well as interfaces to several other mesh generators,
- local refinement and derefinement of a hierarchical mesh structure,
- management of degrees of freedom in nodes, edges, faces and elements of the mesh,
- object hierarchy for numerical algorithms including multigrid solvers, krylov solvers, nonlinear solvers, time-stepping schemes, error estimators, grid transfer operators ... Most of these algorithms can be used independent of the PDE,
- graphical output of mesh structure and solutions to different devices, including X11, Macintosh, ppm, postscript etc.,
- file I/O for visualization and restart purposes,
- and everything in parallel using a SIMD-message passing programming model. All major parallel platforms are supported.

A “problem class” is built on top of the UG library and implements whatever is specific to the PDE problem to be solved. In the simplest case this is only a discretization scheme. The discretization scheme is implemented as a set of C functions with parameters and return values prescribed by the UG framework.

Once these functions are known by UG they can be used in conjunction with the already existing numerical algorithms. A problem class implements a PDE problem in a general form. The geometry, the boundary conditions and coefficient functions (e. g. density and viscosity in a flow model) are supplied by the user later in what is called the “application”. The dots in Figure 1.1 indicate that there are many problem classes built on the UG library and that there may be many applications for each problem class.

This structure is also visible in the UG source code. When you list the UG directory by typing `ls` (you might want to go to the installation chapter first if you did not unpack UG yet):

```
peter@troll:~/UG > ls
CHANGES          README.license    license.ps        tutor/
CVS/              all.proj.hqx      mufte/            ug/
README.Macintosh cm/               ns/
README.first      diff2d/           sc/
README.install    fe/              simple/
```

You find the directory `ug` containing the UG library and the directories `cm` `diff2d` `fe` `mufte` `ns` `sc` `simple` and `tutor` each containing a problem class with applications. When you enter the directory `tutor` (the problem class we will use in this tutorial) you find

```
peter@troll:~/UG/tutor > ls
CVS/  appl/  doc/  pplib/
```

where `pplib` contains the source code of the problem class library, `appl` contains the application code, `CVS` contains information from the Concurrent Versions System (a source code management system used to maintain the UG source code) and the `doc` directory containing this tutorial.





# 2

## Installation

In this chapter, the process of installing UG is explained. By the end of this chapter, you have a running copy of UG on your machine. UG is configured to run on most UNIX flavors and Apple Macintosh Computers. Most of this chapter will explain UNIX specific issues; the Mac is treated separately in section 2.6.

We assume, that you have a complete copy of the UG source code. If you don't have it, section 2.7 describes how to get it.

### 2.1 Choosing a target directory and extracting the source code

You probably received your UG sources as a single tared and gzipped file, called `UG.tar.gz` or `ug-3.8.tar.gz`. In the latter case, 3.8 refers to the version number of UG. Within this description, we assume `ug-3.8.tar.gz` to be the name of the archive.

Now choose the directory in which you want to install UG. This could be anywhere on your machine; for simplicity we assume that you choose to install UG in your home directory. Now `cd` to your home directory and put the UG archive there. Then type:

```
> gzip -dc ug-3.8.tar.gz | tar xvf -
```

This creates the directory `ug-3.8` in your current directory.

If you fetched your copy of UG from the UG ftp server, then the first step is to register; see the file `ug-3.8/README.license` which explains how to register. After registration, the file `gm.h` will be emailed to you. Put this file into the directory `ug-3.8/ug/gm/`.

### 2.2 Setting up the UG environment

The next step is to extend your environment by defining `UGROOT` and by adjusting your search path. This is absolutely necessary—without it you won't be able to compile UG.

Setting the environment variable `UGROOT` and extending your search path works like this:

If you are using a **csh** or **tcsh**:

```
setenv UGROOT $HOME/ug-3.8/ug
set path = ( $path $UGROOT/bin )
```

Insert these two lines into the file `$HOME/.cshrc` or `$HOME/.tcshrc`, depending on which shell you use.

If you are using **sh**, **ksh**, **keysh** or **bash**:

```
export UGROOT=$HOME/ug-3.8/ug
export PATH=$PATH:$UGROOT/bin
```

Insert these two lines into your `$HOME/.profile` file.

The directory `$UGROOT` contains the UG kernel and the directory `$UGROOT/bin` contains many important tools for your everyday work with UG.

## 2.3 Configuring UG

UG can be configured in a great number of ways. You have to specify at least the computer you work on, but also whether you want to use 2D or 3D code, if you want to create optimized code or code for debugging, if you want to run your UG application on a workstation or a parallel supercomputer, and many more. The recommended way to change these settings is by using the tool `ugconf`.

`ugconf` is located in `$UGROOT/bin`. If you have set up your environment like explained in the previous section, you should be able to start `ugconf` by just typing `ugconf`. When called without any arguments, `ugconf` displays the current setting:

```
> ugconf
current ug configuration is:
UGROOT = /home/dave/ug-3.8/ug
ARCH = SGI
MODEL = SEQ
DIM = 2
GRAPE = OFF
COVISE = OFF
NETGEN = OFF
REMOTE_IF = OFF
IF = S
GUI = OFF
DOM_MODULE = STD_DOMAIN
DEBUG_MODE = ON
OPTIM_MODE = OFF
CHACO = OFF
CAD = OFF
```

These settings will be explained in the following subsections. For a list of possible options type `ugconf -help`; a list of all options will be displayed.

Your UG configuration is saved in `$UGROOT/ug.conf` and `ugconf` does nothing else than changing this file. If you like, you can edit this file with a text editor instead of using `ugconf`. In some cases it can even be necessary to edit `ug.conf` directly.

The next sections guide you through the configuration of UG.

### 2.3.1 SUPPORTED ARCHITECTURES AND COMPILERS

The `ARCH` variable contains the current computer architecture and compiler. Table 2.1 contains a listing of all computer architectures and compilers supported by UG. If your architecture is among them, just set the appropriate value with `ugconf`. If for example your computer is a SGI R10000 Workstation just type

```
> ugconf SGI10
```

in your shell. If you say `ugconf` after that, the `ARCH` variable should be `SGI10`.

If your machine is not supported you can easily extend UG for your machine. Suppose that your machine has a UNIX like operating system called `HAL9000` and your compiler is `comp`. Then you have to do this:

1. Take a look at the directory `$UGROOT/arch`. There you will find a subdirectory for each machine type that is supported by UG. Inside each of these subdirectories is a file called `mk.arch`. You must set up a similar directory with a `mk.arch` file.

Create a directory with the name of your architecture in the directory `$UGROOT/arch`. In our example do the following:

```
> cd $UGROOT/arch
> mkdir HAL9000
> cd HAL9000
```

2. Create a file `mk.arch` inside the new directory. It is most sensible to copy one of the `mk.arch` files that already exist and edit it instead of creating one from scratch.

In the `HAL9000/comp-example` we would do this:

```
> cp ../SGI/mk.arch mk.arch
```

3. Edit the file `mk.arch`. Most of the settings in this file should be quite clear; however, here is a description of them:

***ARCH\_TYPE*** The type of architecture. This string is used as a `-D` option when compiling and should have the form `__architecture__`. If your architecture is `HAL9000`, set `ARCH_TYPE=__HAL9000__`.

***ARCH\_MAKE*** Your version of `make` (e.g. `make`, `gmake`, `gnumake`, ...).

***ARCH\_CC*** The C compiler. In most cases `cc` works, but you might need or want something else like `gcc` or `mpicc`. In our example it would be `comp`.

***ARCH\_C++*** The C++ compiler. This could be `cc`, `c++`, `gcc`, ... For most parts of UG, you don't need the C++ compiler.

*ARCH\_F77* The Fortran compiler.

*ARCH\_LINK* The linker. In most cases you can use the same setting as for the compiler (*cc*, *gcc*, *mpicc*,...), but you could also call the linker directly (*ld* on many systems).

*ARCH\_AR* The library archive creation and maintenance program. Usually *ar*.

*ARCH\_SUFFIX* Some operating systems force executables to have a special extension (like *.exe*, *.px*, *texttt.app*, ...).

*ARCH\_POSTLINK* This program is run on the executable after linking. Usually used for calling *strip* to reduce the size of the executable.

*ARCH\_LIBS* The libraries which are needed for successful linking. This is usually something like *-lm -lc*.

*ARCH\_FLIBS* Libraries for Fortran programs.

*SHELL* The shell used from within *make*. Most architectures work without specifying it.

*ARCH\_CFLAGS* Compile flags for the C compiler.

*ARCH\_C++FLAGS* Compile flags for the C++ compiler.

*ARCH\_NOOPTIM* Compiler flags for the creation of non-optimized executables. If *ugconf* says that *OPTIM\_MODE* is off, then these options are used.

*ARCH\_OPTIM* Compiler flags for the creation of optimized executables. If *OPTIM\_MODE* is switched on, then these settings are used instead of the *ARCH\_NOOPTIM* settings.

*ARCH\_LFLAGS* Flags for the linker.

*ARCH\_FFLAGS* Flags for the Fortran linker.

*ARCH\_ARFLAGS* Flags for *ARCH\_AR*.

*ARCH\_XINCLUDES* Specifies where the X11 header files can be found. *-I/usr/include/X11* is a typical choice.

*ARCH\_XLIBS* Specifies the X11 libraries.

4. Edit the file *compiler.h* in *\$UGROOT/arch*. This file contains definitions for the architecture type and is included by every UG file. It has a section for each architecture where machine-dependent settings are specified. Just copy one of these sections and change it to the appropriate values of your architecture.

*ALIGNMENT* and *ALIGNMASK* are especially important.

### 2.3.2 MORE CONFIGURATION

After setting ARCH to the right value you are now able to compile UG. The result of this compilation is however determined by some other settings, so before creating your first UG application, you should consider these switches.

#### *The DIM option*

By setting DIM to 2 or 3 you choose whether you want to run 2 dimensional or 3 dimensional calculations. The decision for which dimension you want to use has to be done here—there is no other switch for choosing the dimension (especially not at run time).

#### *The IF option*

The IF options decides which (graphical or non-graphical) interface you want to use. The options are:

*SIF* The standard interface. This means that no graphic output to the screen will be done. Instead, UG will use the text capabilities of the shell for input/output. While you are still able to draw pictures into files, you will not be able to monitor your results on screen.

*XIF* The X11 interface. UG will use its own shell window and you will be able to use the full interactivity possibilities of UG. You will be able to open graphic windows and modify the display and even some data with your mouse.

By default, the X11 interface uses the Athena widget set which is installed with most X11 implementations. Should your installation lack the Athena Widget set, you can turn its usage off by changing the line

```
#define USE_XAW

in file $UGROOT/ug/dev/xif/xmain.c to

#undef USE_XAW
```

However, the Athena widget set adds some nice capabilities and we recommend installing it.

*MIF* The MacOS X Server interface. This uses the Display PostScript window server for its graphical interface. As soon as MacOS X is released, the displaying model will be changed to use the Quartz window server.

*RIF, XRIF, NORIF* The remote interface. This interface is intended for computers that don't have X11. Using the remote interface, it is possible to communicate with a computer running X11 and thereby to redirect the graphical output.

*The GUI option*

Don't use it.

The GUI is an implementation of a graphical user interface for UG that doesn't use the shell. While this might sound like a reasonable option at first, it is not—the shell is by far the superior way to run UG. The GUI option was created on special demand for a project partner and only works with one special application. The source code for the GUI is not part of the standard distribution of UG.

*The MODEL option*

The MODEL option decides whether UG is run in sequential or parallel mode. On a single processor workstation, set MODEL = SEQ. If you are using a parallel computer or want to use PVM or MPI for developing a parallel application, then choose your option according to these parameters:

**MPI** The Message Passing Interface. MPI is installed on many parallel computers and can be used on workstations or clusters of workstations.

If you are using MPI, we recommend that you reflect its usage in adapting your `mk.arch` file. Setting the compiler to `mpicc` will usually be the easiest way to get UG with MPI running, because `mpicc` knows about where MPI header files and libraries are located. Should this not be sufficient, edit `$(UGROOT)/ug.conf`, uncomment the lines

```
#MODEL_ENV_CFLAGS = -I$(MPIHOME)/include
#MODEL_ENV_LFLAGS = -L$(MPIHOME)/lib/IRIX/ch_p4 -lmpi
```

and set the flags to the appropriate values for your machine.

**PVM** The Parallel Virtual Machine. It is present on many parallel computers and can be used on most workstations. In order to use PVM, uncomment the lines

```
#MODEL_ENV_CFLAGS = -I$(PVM_ROOT)/include
#MODEL_ENV_LFLAGS = -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lpvm3
```

in `$(UGROOT)/ug.conf` and set the flags to the values that are appropriate for your machine.

**NX** Intel's message passing library (e.g. Intel Paragon).

**NXLIB** An NX simulation for workstation clusters.

**SHMEM** The shared memory model of the Cray T3E.

**SHMEMT3D** The shared memory model of the Cray T3D.

**PARIX** The Parsytec message passing library.

*The CHACO option*

If you are using UG for parallel computing, you will certainly know about the issue of load balancing. Although UG has some simple load balancing strategies, it uses the CHACO package for more advanced load balancing. Using CHACO is highly recommended.

If you are not using the parallel capabilities of UG keep CHACO turned off.

*The DEBUG\_MODE option*

Setting `DEBUG_MODE` to `ON` will enable the debugging flags for the compiler that have been set in `mk.arch`. Use `ugconf NODEBUG` to turn off debugging mode.

*The OPTIM\_MODE option*

Setting `OPTIM_MODE` to `ON` will enable the optimization flags for the compiler that have been set in `mk.arch`.

*The DOM\_MODULE option*

This option determines the domain module that is used for the representation of the domains you are using for your calculations. Currently the `STD_DOMAIN` and the `LGM_DOMAIN` are available.

*The GRAPE option*

For postprocessing of your results, there is an interface to the GRAPE visualization software. GRAPE is developed and distributed by the SFB 256, Institut für Angewandte Mathematik (Rheinische Friedrich-Wilhelms-Universität Bonn) in cooperation with the Institut für Angewandte Mathematik (University of Freiburg). It can be obtained from

<http://www.iam.uni-bonn.de/sfb256/grape/main.html> .

*The COVISE option*

UG has an interface to COVISE, which stands for Collaborative Visualization and Simulation Environment. It is an extendable distributed software environment to integrate simulations, postprocessing and visualization functionalities. See

<http://www.hlrs.de/structure/organisation/vis/covise/>  
for further details.

*The NETGEN option*

NETGEN is a 3D grid generator developed by Joachim Schöberl that can be used from within UG. Please see

<http://www.sfb013.uni-linz.ac.at/~joachim/netgen/> .

You will need an additional license and a C++ compiler.

## 2.4 Compiling UG

Please make sure that you have placed the file `gm.h` in the directory `ug-3.8/ug/gm` and that the environment variable `UGROOT` is set.

To make sure that everything is working you can start compilation of all UG libraries and examples by typing `ugproject`. This can take some time on slower machines, since it compiles all libraries and example applications for 2D and 3D.

If you only want to compile the UG libraries and have completed your setup using `ugconf`, then it is sufficient to say `ugmake ug`. This will create the UG libraries (which will be placed in `$UGROOT/ug/lib`).

To compile the tutor problem class and applications which are used in the subsequent chapters, change into the directory `$UGROOT/tutor/appl` and type

```
> ugconf 2
> ugmake ug
> make
> ugclean
> ugconf 3
> ugmake ug
> make
```

This will create the programs `tutor2d` and `tutor3d`. For their usage see chapter 3.

## 2.5 UG command line tools

This section describes some of the tools located in `ug/bin`. Most of them are shell scripts which should run in almost any UNIX environment. These tools help you developing and debugging applications.

Please make sure that your `$UGROOT` environment variable is set and that your path has been adjusted as explained in section 2.2.

### *ugclean*

This is a global make `clean` on the UG libraries—it mainly removes object files. If you say `ugclean all`, then even the libraries and the contents of the `ug/include` directory are removed (the `ug/include` directory contains only links so removing it doesn't do any harm).

Switching from 2D to 3D is a typical situation in which you would want to apply `ugclean`, but also if you changed an important header file.

### *ugconf*

The `ugconf` command is explained in section 2.3.



*uggrep, uggrep.c, uggreph*

The `uggrep` tools offer a convenient way to perform greps on the UG source code. The command

```
> uggrep NODE
```

will search for all occurrences of `NODE` in UG source and header files. You can limit the search to source files (files with the suffix `.c`) by using `uggrep.c` and to header files (files with the suffix `.h`) by using `uggreph`.

*ugmake, ugpmake*

Using `ugmake` can be used instead of `make` and offers the advantage that it knows where and how the UG libraries are to be build. You can type `ugmake ug` anywhere in your shell, resulting in a complete make of UG. Its general syntax is:

```
ugmake [<ugmodule>] [makeoptions]
```

Some common `<ugmodules>` other than `ug` are `dev`, `meta`, `xif`, `gm`, `graph`, `low`, `numerics`, `ui` and `dom`. You can pass additional options to `make` with the `makeoptions`; the command `ugmake np -k` will compile the `np` module with the `-k` option.

`ugpmake` is used like `ugmake` but tries to speed up the compilation by using parallel versions of `make`. On sequential machines, this can be achieved by using versions of `make` (like GNU `make`) that can start several compilation processes with the `-j n` option; `n` specifies the number of compilation processes. Some parallel computers or multiprocessor workstations also offer the possibility to use several processors for compilation. Since this is very machine dependent, you might have to adjust `ugpmake` to your local architecture.

*ugman*

The standard UG distribution comes with man pages for almost all commands and functions of UG. To get the man page for the UG shell command `drawtext` type `ugman drawtext`. In the same way the man page for the UG function `dcopy` can be displayed with `ugman dcopy`.

*ugrun*

The way a parallel program has to be started is different on each type of computer. Once you are working on a couple of machines it can get tedious to remember the exact syntax of every such command. `ugrun` can be used to start parallel programs by simply saying

```
> ugrun tutor2d 32
```

This will start the program `tutor2d` on 32 processors of your machine. Since `ugrun` knows the environment variables that the application was compiled with, it will use the appropriate command for starting the application (which could be `mpirun` when using MPI or `mpprun` on a Cray T3E with SHMEM).

## 2.6 UG on the Macintosh

UG can be used on the Macintosh too. You will need the Metrowerks Codewarrior Professional Release 4 to compile the sources.

The process of getting a working version of UG onto your Mac is similar to the UNIX procedure. First of all, follow the instructions in section 2.7 on how to get the UG source code. You will probably use Fetch or some other Macintosh FTP tool. You can unpack the UG archive with StuffitExpander or with MacGzip and `suntar` or `tar` (the infomac ftp archives should have the appropriate software). Register your UG version. When you receive the file `gm.h`, place it in the Folder `ug-3.8:ug:gm` (your version number might be different). Now unpack the file `ug-3.8:ug:lib:MWCW:ug.proj.hqx` (e.g. with StuffitExpander). The resulting CodeWarrior Project has a target called `ug`. Make it active and choose *make* from the *Project* menu. This will create all UG libraries.

There are several binhexed application project files in the application folders `diff2d`, `fe`, `ns` and `sc`. Unpack them and build the applications.

## 2.7 How to get UG

In order to get a working copy of UG, you will have to follow these steps which will be explained in greater detail below.

1. Get the UG distribution from `ftp.ica3.uni-stuttgart.de`.
2. Register. The license agreement is part of the UG distribution.
3. After registration, the file `gm.h` will be emailed to you.
4. Put `gm.h` in its place. Now you are ready to compile (see section 2.2).

As you will have guessed, the file `gm.h` is absolutely crucial for UG and compiling and running UG without it is hardly possible.<sup>1</sup> Since we would like to keep track of who is using UG we thought that this might be a good way to enforce registration.

The UG source code is available from `ftp.ica3.uni-stuttgart.de`. As of this writing, the current UG version is 3.8; should you find a UG distribution with a higher version number on our server please use it. Only stable versions are put on the server (unless explicitly stated otherwise) and new versions usually contain a great number of new features and bug fixes. However, older versions are still kept on the server.

A sample ftp session is shown here:

---

<sup>1</sup>If you should be able to create your own `gm.h` that works with UG please let us know—we are very much interested in hiring you.

```
% ftp ftp.ica3.uni-stuttgart.de
Connected to dom.ica3.uni-stuttgart.de.
220 dom FTP server ready.
Name (ftp.ica3.uni-stuttgart.de:clinton): ftp
331 Guest login ok, send e-mail address as password.
Password:
230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub
ftp> cd ug
ftp> cd ug-3.8
ftp> bin
ftp> get ug-3.8.tar.gz
ftp> bye
```

You should now unpack UG and check the file `README.license` in the UG root directory. If you don't know how to unpack `.tar.gz`-files then look up section 2.1. After fulfilling the license agreement, you will receive the file `gm.h`.

ugconf	Architecture	Compiler
AIX	IBM AIX	cc
AIXGCC	IBM AIX	gcc
C90	Cray 90	
CC	XPLORER (Power PC, Parsytec)	
DEC	DEC Workstations	
HP	HP Workstations (HPUX 10)	cc
HP20	HP Workstations (HPUX 10.20)	cc
HPUX9	HP Workstations (HPUX 9)	cc
LINUXPPC	LinuxPPC	gcc
MACOSXSERVER	MacOS X Server	cc
MKLINUX	Microkernel Linux (PowerMacintosh)	gcc
NECSX4	NEC SX4	
NEXTSTEP	NEXTSTEP 3.3	
OPENSTEP	OPENSTEP 4.2	
ORIGIN	SGI Workstations	
PARAGON	Intel Paragon	
PC	Linux-PC	
POWERGC	XPLORER (Power PC, Parsytec)	
SOLARIS	SUN Sparc 5 Workstations (SOLARIS 5.3)	cc
SOLARISGCC	SUN Sparc 5 Workstations (SOLARIS 5.3)	gcc
SGI	SGI Workstations	cc
SGI10	SGI R10000 Workstations	cc
SP2	IBM AIX SP2	
SR2201	Hitachi SR2201	
SUN4GCC	SUN Workstations (SunOS 5.3)	gcc
T3D	Cray T3D	
T3E	Cray T3E	
YMP	Cray YMP	

Table 2.1: Options for the ARCH variable of UG

# 3

## Using a UG application

In this chapter you will learn how to use an existing UG application. This assumes that you successfully installed UG and compiled the tutor problem class with the graphical user interface turned on (either X11 or Macintosh).

### 3.1 Compiling the tutorial application

Having compiled the UG library for two and three space dimensions with the appropriate switches (if not return to chapter 2) you can now compile the tutorial application. To do that, select the appropriate space dimension with `ugconf`, change to the `UG/tutor/appl` directory and type `make`. If all goes right you should have applications `tutor2d` and `tutor3d`.

### 3.2 A first example

Let us start by playing around with our first example. Change to the directory `UG/tutor/appl` and start the application `tutor2d`. You should see a window like the one shown in Figure 3.1. This is the UG shell window where you can enter commands. Like the UNIX shell this shell can also execute text files containing shell commands. Some example script files are provided in the `UG/tutor/appl/scripts` directory. We will now execute our first script file. For that type

```
> ex first
```

in the UG shell window. Some output is printed to the shell window and two graphics windows will open. These two windows are shown in the upper two pictures in Figure 3.2. The window labelled “W1” shows the current mesh consisting of 8 triangular elements approximating a circle domain. The solution of the equation

$$-\Delta p = 0, \quad u = x^3 + y^2 \text{ on } \partial\Omega$$

is shown in the window labelled “W2” in the form of contour lines. As you can see the approximate solution is linear on each triangle.

Now enter the following commands (at the end of each line press the RETURN or ENTER key):

```
> mark $i 1
    using rule red (no side given)
```

```

1 elements marked for refinement
> refine
circle refined
>

```

The first command tags element number 1 for refinement (note the little numbers in each triangle). The second command refines the mesh. Note that options to a command are preceded by the \$ sign. As you can see in the window W1 (depicted in the middle picture in the left row in Figure 3.2) element 1 has been refined into four smaller triangles and the two green elements have been inserted to make the mesh “consistent” (which means that the intersection of any two elements is either empty, a node or an edge of both elements). You can still see the previous mesh by entering

```

> level 0
current level is 0 (bottom level 0, top level 1)
>

```

and switch back to the finest level by entering

```

> level 1
current level is 1 (bottom level 0, top level 1)
>

```

We continue refining two times by giving the following commands

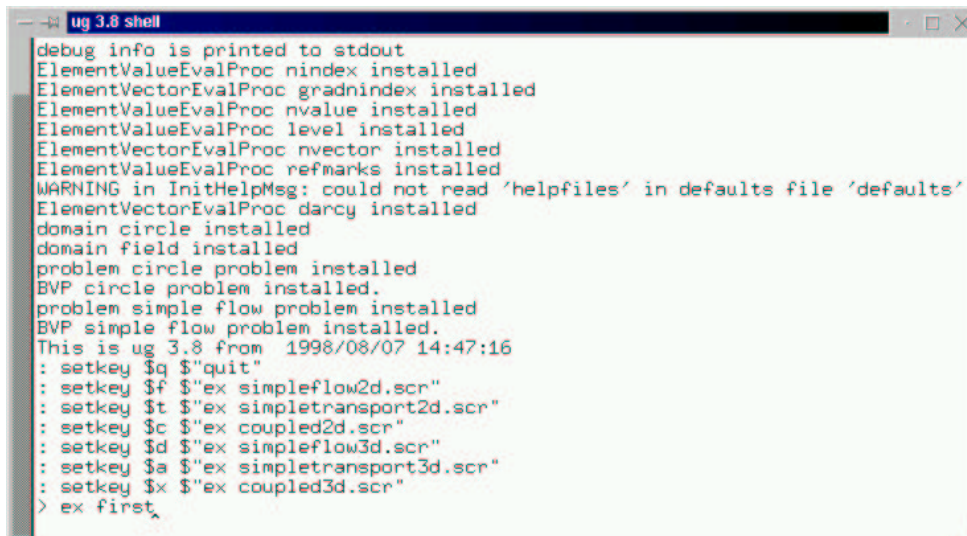
```

> mark $i 12
using rule red (no side given)
1 elements marked for refinement
> refine
circle refined
> mark $i 32
using rule red (no side given)
1 elements marked for refinement
> refine
circle refined
>

```

and end up with the mesh looking like the middle picture in the right row of Figure 3.2. Note that the circle has been approximated better and better as the mesh has been refined. The geometry is really entered as a circle and is not defined by the coarsest mesh.

The graphical user interface has also some interactive capabilities. Move the mouse into mesh window (window “W1”) and click. Notice that the window gets an orange border indicating that this is the “active picture”. Now move the mouse pointer to the lower part right part of the window where the little icons are and place it over the arrow icon. The text to the left of the arrow now says pointer [1/4]. When you press the left mouse button once it will say pan [2/4] and when you press again it will say zoom [3/4]. This means that the



```

ug 3.8 shell
debug info is printed to stdout
ElementValueEvalProc nindex installed
ElementVectorEvalProc gradnindex installed
ElementValueEvalProc nvalue installed
ElementValueEvalProc level installed
ElementVectorEvalProc nvector installed
ElementValueEvalProc remarks installed
WARNING in InitHelpMsg: could not read 'helpfiles' in defaults file 'defaults'
ElementVectorEvalProc darcy installed
domain circle installed
domain field installed
problem circle problem installed
BVP circle problem installed.
problem simple flow problem installed
BVP simple flow problem installed.
This is ug 3.8 from 1998/08/07 14:47:16
: setkey $q $"quit"
: setkey $f $"ex simpleflow2d.scr"
: setkey $t $"ex simpletransport2d.scr"
: setkey $c $"ex coupled2d.scr"
: setkey $d $"ex simpleflow3d.scr"
: setkey $a $"ex simpletransport3d.scr"
: setkey $x $"ex coupled3d.scr"
> ex first

```

Figure 3.1: The UG shell.

zoom tool has been enabled. You can use it to zoom into the part of the mesh that has been refined. Just move the mouse pointer near the refined part. This will be the first point of the rectangle indicating the new viewport. Press the left mouse button, hold it and move to the second point. A rectangle will be shown. When you release the mouse button the window will be redrawn and looks something like the lower left one in Figure 3.2.

Now you can recompute the solution by entering (the @ is important):

```

> @solve
BOX: [0:a] [1:a] [2:a] [3:a]

***** linear_solver.ls.mgs *****
0   u: 3.3939336e+00  ---
1   u: 8.4281017e-03  2.4832842e-03
2   u: 5.6300590e-05  6.6801033e-03
3   u: 3.7552681e-07  6.6700332e-03

3   avg:  u: 3.3939336e+00  3.7552681e-07  4.8007881e-03

LS  : L= 3 N= 3 TSOLVE=  -0.0673 TIT=  -0.02243
>

```

The first line of output starting with BOX is from assembling the system of linear equations on each grid level and the following output is from the multigrid solver showing convergence rate and summary.

Another useful command is

```

> glist
grids of 'circle':
level maxlevel #vert #node #edge #elem #side #vect #conn #imat minedge maxedge
0       3       9       9       16       8       8       9       25       0 7.654e-01 1.000e+00
1       3       5      14      29      16      10      14      43       0 3.827e-01 1.000e+00
2       3       5      19      42      24      12      19      61       0 1.951e-01 1.000e+00

```

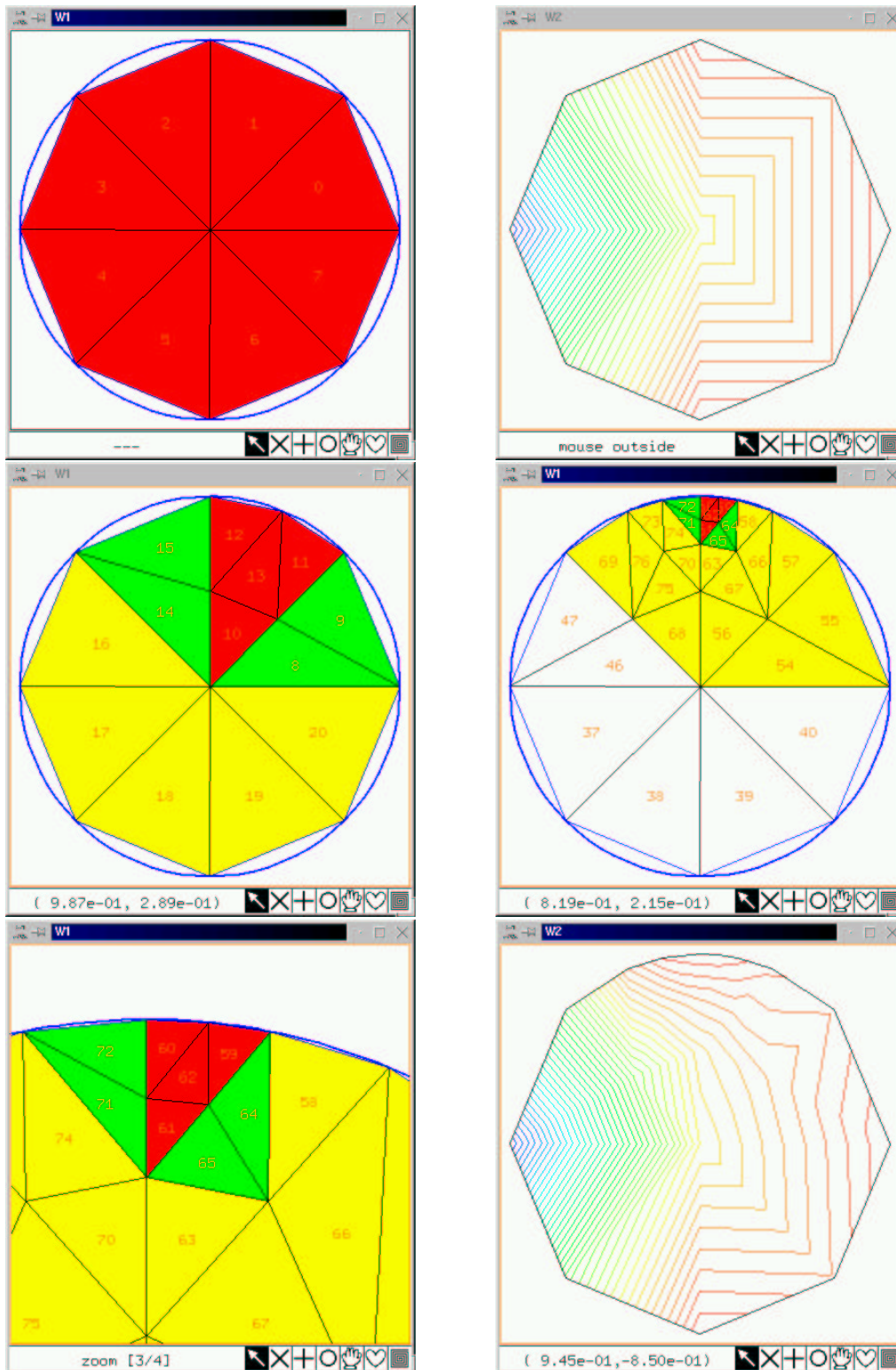


Figure 3.2: Graphics windows after various steps explained in the text.



```
* 3      3      3      18      40      23      8      18      58      0 9.802e-02 1.000e+00
surface grid up to current level:
  2      3      ---      22      50      29      13      22      65 9.802e-02 1.000e+00
32244 bytes used out of 20971520 allocated
```

which prints a summary of the multigrid hierarchy and the amount of memory used.

### 3.3 A sample script file

In this section we look at the script file `first.scr` that has been used in the previous section. It is shown in two parts in Figures 3.3 and 3.4. Let us start with Figure 3.3 where the mesh is set up.

Line 1 contains a comment. The hash character makes the rest of the line a comment line. In line 2 the variable `MAXLEVEL` is set to zero. Note that statements are always terminated with a semicolon. Like in C you can place several statements in a line.

Line 5 contains a so-called “format definition”. UG can place degrees of freedom in the nodes, edges, faces and elements of the mesh. The amount of memory (number of double values) required by the user in each geometrical object is given by the format definition. In this example the `$V` option defines vectors with one double value in each node `n1` and we will need 8 of these vectors in our numerical algorithm (therefore reserving 8 double values in each node). The `$M` option defines the amount of memory needed in the sparse matrix data structure. In our example the coupling of a node with another node will contain one double value ( $n1 \times n1$ ) and we will need two matrices with this layout (one is the stiffness matrix the other will be an incomplete decomposition). Only nodes which are corners of at least one common element will be coupled in the matrix. It is important to note that the format command does not allocate any memory, it just describes a data layout that will be used by the grid manager part of UG. You can define several different formats in a script file and select one later by giving its name (which is `FlowFormat` in our example).

Line 8 creates a new multigrid data structure in memory. The first argument (`circle`) is the name of the structure (You could open several multigrids), `$b` gives the name of the boundary value problem to be solved, `$f` specifies the format to be used and `$h` gives the amount of memory to be allocated for this multigrid data structure (20 MBytes in the example). The boundary value problem given with the `$b` option selects a geometry, boundary conditions and coefficient functions (e. g. permeability or viscosity) which have been compiled into the application. An application can include several boundary problems.

Lines 11–16 define a coarse mesh. In UG a geometry is given by a collection of boundary segments. Each boundary segment is a mapping from a parameter space to a  $d - 1$ -dimensional manifold, where  $d$  is 2 or 3. Nodes on the boundary are inserted with the `bn` command by specifying the number of the boundary segment and the parameter value. In our circle example the domain is described

```
1 # some constants
2 MAXLEVEL      = 0;          # finest level
3
4 # data format definition
5 newformat FlowFormat $V n1: nt 8 $M implicit(nt): mt 2 $I n1;
6
7 # create multigrid
8 new circle $b circle problem $f FlowFormat $h 20M;
9
10 # make mesh
11 bn 0 0.25; bn 0 0.50; bn 0 0.75;
12 bn 1 0.25; bn 1 0.50; bn 1 0.75;
13 in 0.0 0.0;
14 ie 8 0 2; ie 8 2 3; ie 8 3 4; ie 8 4 1;
15 ie 8 1 5; ie 8 5 6; ie 8 6 7; ie 8 7 0;
16 fixcoarsegrid;
17
18 # refine mesh
19 j = 0;
20 repeat {
21     if (j == MAXLEVEL) break;
22     refine $a;
23     j = j+1;
24 }
25 lexorderv ru; #order nodes
26 glist;      # print statistics
27
28 # show grid
29 openwindow 30 30 402 402 $n W1;
30 openpicture $s 1 1 400 400 $n grid;
31 setplotobject Grid $b 1 $n 0 $e 1 $c 1 $w a $p 0.9;
32 setview;
33 plot;
34 refreshon;
```

Figure 3.3: File UG/tutor/appl/scripts/first.scr: setting up the mesh

```

36 # set up discretization scheme
37 npcreate sf $c stdf;
38 npinit sf $p sol;
39
40 npcreate bf $c bf;
41 npinit bf $A MAT $x sol $b rhs $P sf;
42
43 # set up solver
44 npcreate ilu $c ilu;
45 npinit ilu $damp 1.0;
46
47 npcreate base $c ex;
48 npinit base;
49
50 npcreate basesolver $c ls;
51 npinit basesolver $red 1.0E-4 $m 50 $I base $display no;
52
53 npcreate transfer $c transfer;
54 npinit transfer $x sol;
55
56 npcreate lmgc $c lmgc;
57 npinit lmgc $S ilu ilu basesolver $T transfer
58     $n1 2 $n2 2 $g 1;
59
60 npcreate mgs $c ls;
61 npinit mgs $A MAT $x sol $b rhs $m 25 $red 1.0E-6
62     $abslimit 1.0E-15 $I lmgc $display full;
63
64 # clear, assemble, solve
65 clear sol $a $v 0.0;
66 npexecute bf $a;
67 npexecute mgs $i $d $r $s $p;
68
69 # show solution
70 openwindow 30 490 402 402 $n W2;
71 openpicture $s 1 1 400 400 $n solution;
72 setplotobject EScalar $s sol $m CONTOURS_EQ $n 40 $d 0 $f -1.0 $t 1.0;
73 setview;
74 plot;
75
76 # define solve script
77 solve = "{
78     npexecute bf $a;
79     npexecute mgs $i $d $r $s $p;
80 }";

```

Figure 3.4: File UG/tutor/appl/scripts/first.scr: The solver

by two boundary segments, the first (number 0) describing the upper half circle starts with parameter 0 at point  $(1, 0)$  and ends with parameter 1 at point  $(-1, 0)$ . The second segment describing the lower half circle starts with parameter value 0 at  $(-1, 0)$  and ends with parameter value 1 at  $(1, 0)$ . The two points  $(1, 0)$  and  $(-1, 0)$  belong to both boundary segments and are called “corners”. The new command automatically inserts nodes at those corner points of the domain. Inner nodes are inserted with the `in` command getting the coordinates of the new node as parameters. Elements are inserted with the `ie` command. The `ie` command gets the node numbers as argument. The element type is determined from the number of nodes given. The `fixcoarsegrid` command ends the specification of the mesh, computes element neighborhood and does some consistency checks.

Lines 19–24 refine the mesh uniformly `MAXLEVEL` times, where `MAXLEVEL` is set in line 2 of the script. We do this with a `repeat`-loop. This is the only type of loop available in UG’s script language. The only way to exit a loop is with the `break` statement shown in line 21. The option `$a` in the `refine` command specifies that all elements should be refined. After refining the mesh the command `lexorderv` is used to order the degrees of freedom lexicographically according to the position of the corresponding nodes of the mesh. This is not strictly necessary but may improve the convergence properties of the iterative solvers for some problems. More sophisticated ordering strategies, such as “streamline ordering” are also possible. The `glist` command has already been used in the previous section to print some statistics about the current mesh structure.

Lines 29–34 display the mesh on the screen. `Openwindow` opens a window by specifying the coordinates of the lower left corner (first two numbers) and the width and height of the window (second two numbers), the `$n` option specifies a name for the window. By default windows are opened on the screen but the same commands can be used writing the picture to a disk file in several different formats (such as `ppm`, `postscript`, ...). The `openpicture` command specifies a picture (drawing area) within the window. Several pictures can be placed into the same window. The `$s` option gives the lower left corner, width and height of the picture relative to the window and `$n` gives a name to the picture. Now we want to specify what should be drawn in the picture. This is done by associating a plot object with the picture in the `setplotobject` command. In this case we specify the `Grid` plot object which draws the grid. Several options can be given which determine whether the boundary of the domain should be drawn (`$b`) or whether node or element numbers should be given. The next command, `setview`, determines from where we want to look at the plot object. If no arguments are given default values are computed from the geometry definition. Finally, the `plot` command draws the picture. By default the contents of a UG graphics window is *not* redrawn if the mesh changes because redrawing may require a lot of time. The `refreshon` command changes this behavior and redraws the pictures whenever the mesh is changed or the window is resized.

After creating the mesh we are now able to set up and solve the numerical problem. Numerical algorithms in UG are organized in a class hierarchy (yes,

it is coded in C although it uses object-oriented concepts). Objects of a class can be created with the command `npcreate` (historically, these objects were called “numerical procedures” and all commands therefore start with `np`, also the directory containing the source code is called `np`). Every class has the three methods “init”, “display” and “execute” which are mapped to the commands `npinit`, `npdisplay` and `npexecute`. The `init` method of a class is used to set parameters of an object, the `display` method shows the current settings and the `execute` method instructs an object to do its intended job.

Each class has *two* kinds of interfaces, one is the script file interface which has been described above and the other is the C-function interface. The class hierarchy offers a C-function interface for all kinds of numerical algorithms like iterative schemes, linear solvers, nonlinear solvers, discretization schemes and more. This enables an object of one class to use an instance of another class (like a krylov method using a preconditioner or a time-stepping scheme calling a nonlinear solver) to do its job. This is also the key to reuse code written by other people.

In our example script we will solve the problem

$$-\Delta p = 0, \quad u = x^3 + y^2 \text{ on } \partial\Omega. \quad (3.1)$$

This problem is in fact a special case of the more complicated class of problems

$$\nabla \cdot \mathbf{u} = q \text{ in } \Omega, \quad (3.2a)$$

$$\mathbf{u} = -\mu^{-1} \mathbf{K} (\nabla p - \rho \mathbf{g}), \quad (3.2b)$$

$$p = p_0 \text{ on } \Gamma_1 \subseteq \partial\Omega, \quad (3.2c)$$

$$\mathbf{u} \cdot \mathbf{n} = \phi \text{ on } \Gamma_2 = \partial\Omega \setminus \Gamma_1 \quad (3.2d)$$

which can be solved by the tutorial problem class. Eq. (3.1) is obtained from Eq. (3.2) by setting  $\mu = 1$ ,  $\mathbf{K} = Id$ ,  $\rho = 1$ ,  $\mathbf{g} = 0$  and  $q = 0$ .

In order to separate the implementation of the discretization of the general problem (3.2) from the implementation of the specific functions  $\mu$ ,  $\mathbf{K}$ , etc. , the parameter functions are encapsulated in a separate (abstract) class. The user then will implement concrete classes in his application which will then be used by the discretization class. In line 37 of the script file (see Figure 3.4) an object named `sf` of class `stdf` is created. Class `stdf` is a concrete class implementing the parameter functions  $\mu = 1$ ,  $\mathbf{K} = Id$ ,  $\rho = 1$ ,  $\mathbf{g} = 0$  and  $q = 0$  as required by our example. The object `sf` is initialized in the following line 38. Since `sf` contains a member function that computes the Darcy velocity  $\mathbf{u}$  it needs to know where the pressure field  $p$  is stored. The parameter given by `$p` is a so-called “vector data descriptor” which describes where in the little data array allocated for each node of the mesh the pressure field is stored (e. g. at array position 0, if nothing has been allocated before). Note that the amount of space needed in each node

has been determined by the `newformat` command. A vector data descriptor has a name, `sol` in this case, in order to be able to use it at other places in the script file.

Lines 40–41 create a discretization object named `bf` of class `bf` (yes, classes and objects can have the same name). `bf` stands for “boxflow” indicating that we solve the flow equation with the box method (finite volume method). The `bf` object expects a “matrix data descriptor” with the `$A` option and two vector data descriptors with the `$x` and `$b` options in order to know where the matrix, solution and right side are stored. In addition the discretization object needs an object implementing the parameter functions, i. e. our object `sf` we created above. This is done via the `$P` option. The `bf` object will then use the internal C-function interface of the `sf` object to get the values of the discretization parameters.

Lines 44–62 set up an iterative solver for the linear system. We intend to use a linear multigrid method with an `ilu` smoother, the canonical grid transfer operators given by the finite element space and an exact coarse grid solver. We start with the `ilu` smoother in line 44–45 and set up the exact solver in lines 47–48.

Internally, both the `ilu` class and the `ex` class, implement the interface of an iteration scheme. The purpose of an iteration scheme is to apply one iteration step at a time to a given linear system. A linear system solver then executes steps of an iteration scheme and checks convergence. Such a solver is set up in lines 50–51. Object `basesolver` is an instance of class `ls` which is a simple loop executing steps until the defect norm has been reduced by a certain factor (`$red` option) or a prescribed number of iterations (`$m` option) has been reached. `basesolver` uses the exact method as underlying iteration (`$I` option) and is instructed to print no messages (`$display` option).

Lines 53–54 set up a grid transfer object `transfer` of class `transfer`. Lines 56–58 set up the linear multigrid cycle. It uses the `ilu` object for pre- and postsmoothing and the `basesolver` object as coarse grid solver (`$S` option), the `transfer` for restriction and prolongation and it executes two presmoothing steps (`$n1`), two postsmoothing steps (`$n2`) and a V-cycle (`$g`).

Lines 60–62 set up a linear solver with the linear multigrid cycle as iteration scheme. Note that `mgs` and `basesolver` are instances of the same class `ls`. Since the `execute` method of `mgs` will be invoked to solve the linear system it needs to know where the linear system is stored (`$A` `$x` `$b` options). Note that the other objects building up the linear solver need not know where the linear system is stored since they will get the appropriate information from the calling object. Also, the objects will allocate additional temporary vectors internally and pass them as arguments to other objects.

Lines 65–67 finally setup and solve the linear system. Line 65 sets all components of vector `sol` on all grid levels (`$a`) to zero. Line 66 executes the `bf` object which sets up the linear system on all grid levels (`$a`). Line 67 executes the `mgs` object to solve the linear system. The options are in order: `execute preprocess`

member, replace right hand side by defect, compute defect norm, solve system and execute postprocess member function.

Lines 70–74 open a new window and set up a plot object that draws contour lines of the solution.

Finally, lines 77–80 shows how subroutines can be realized in a script file. The instructions to be executed are simple assigned to a string variable. The interpreter has an “evaluation operator” @ which evaluates the string variable to the right, i. e. the command @solve is equivalent to executing the program assigned to the variable solve.

This sample script file hopefully gave some overview of the capabilities of UG (and this was *not* all). Most of the rest of this tutorial will explain the class hierarchy for numerical algorithms in more detail. We will learn how the C-function interfaces for discretizations of linear and instationary, nonlinear discretizations look like.

### 3.4 2D flow and transport problem

Being familiar with the first simple model problem we can try a set of more complicated examples in two space dimensions which are given in the script files `simpleflow2d.scr`, `simpletransport2d.scr` and `coupled2d.scr`.

Simply start up `tutor2d` and type

```
> ex simpleflow2d
```

Alternatively you can press the Alt–Key *followed* by the letter f (do not press Alt and f together as you might think. We were too dumb to figure out how to make this work correctly). A graphics window will open with three pictures in it showing the grid, contour lines of pressure and the velocity field in a vector plot (bottom to top). This application solves a ground water flow problem Eq. (3.2) in a horizontal reservoir (i. e.  $\mathbf{g} = 0$ ). From the pressure field we can see that the reservoir contains a source on the left side and a sink on the right side (with exactly the same rate) and zero flux boundary condition on the outer boundary. The domain consists of three subdomains with different permeability values, which can also be seen from the vector plot of the flow field.

We now want to solve a transport problem of the general form

$$\frac{\partial(\Phi\rho C)}{\partial t} + \nabla \cdot \mathbf{j} = q \text{ in } \Omega, \quad (3.3a)$$

$$\mathbf{j} = \rho\mathbf{u}\phi(C) - D(\mathbf{x}, \mathbf{u})\nabla C, \quad (3.3b)$$

$$C(\mathbf{x}, 0) = C_0(\mathbf{x}) \quad (3.3c)$$

$$C = C_d(\mathbf{x}, t) \text{ on } \Gamma_1 \subseteq \partial\Omega, \quad (3.3d)$$

$$\mathbf{j} \cdot \mathbf{n} = \phi_C \text{ on } \Gamma_2 = \partial\Omega \setminus \Gamma_1. \quad (3.3e)$$

for the unknown concentration  $C$  where the velocity field  $\mathbf{u}$  is given from the first equation. Note that this problem is now nonlinear and time-dependent.

Restart your `tutor2d` application now and execute the script file `simpletransport2d.scr`. In the graphics window you will see the pressure field at the bottom and the flow field on top. In the middle you will see a contour plot of the concentration at each time step (this may take a while depending on the type of computer you are using). In the given example a fixed concentration is described at the inflow well in the left part of the domain. The flux function in the hyperbolic part is  $\varphi(C)$  as in Burger's equation, the diffusion constant (well, hydrodynamic dispersion to be correct) is set to  $10^{-6}$ . If you wait a little you can clearly see the different permeability values in the two little subdomains.

The final example in this section assumes now that the viscosity in the flow Eq. (3.2) is now a function of concentration computed in the second equation  $\mu = \mu(C)$ . This couples the two equations and is called *miscible displacement*, a model sometimes used in oil reservoir simulation. With the tools currently implemented in the tutorial problem class we can solve this problem with a simple operator splitting technique. For a given time step we first solve the flow equation for the current concentration field and then advance the concentration with the new flow field. Then we go on to the next time step. Note that there is no iteration between the two equations within a time-level to solve the coupled nonlinear problem. Clearly this imposes some restriction on the time step size but it is a simple procedure. It is also no difficulty to implement a fully coupled solution procedure but it would require to code a new discretization class in the tutorial problem class.

In order to start the simulation of the coupled problem restart `tutor2d` and execute the `coupled2d.scr` script. It displays the same plots as before but when you look carefully you will see that the pressure field changes in each time step (hint: place the mouse pointer on contour line).

### 3.5 3D flow and transport problem

The discretization code for the flow and transport problems described above is written in a dimension and element-independent way. Therefore it is now an easy task to do similar simulations also in three space dimensions. For that you need to generate the `tutor3d` application. The script files `simpleflow3d.scr`, `simpletransport3d.scr` and `coupled3d.scr` directly correspond to their two-dimensional counterparts.

Unfortunately it is not so easy to create a complicated three-dimensional domain in UG. Therefore we stick with a simple hexahedral domain. Now have fun with the three-dimensional examples!



## 3.6 A parallel example

UG has been designed from the ground up with parallel computing in mind. Parallelism is supported throughout all levels of the UG library, including grid management, numerical procedures and graphics. This section will investigate some of the steps that are necessary to run an UG application on a parallel computer.

The first step is to compile an executable for your parallel computer. Section 2.3.2 on page 10 explains the most important step, setting the `MODEL` flag with `ugconf`. You should also turn on `CHACO`. Then rebuild the UG libraries, problem class libraries and finally your application. Don't forget to clean up before the rebuild (`ugclean`).

We will assume here, that your parallel environment is a workstation cluster with  $n$  computing nodes called `node1`,  $\dots$ , `noden` and you are using `MPICH`. If your parallel application is called `ptutor3d`, then the you can start your application on 4 processors by typing

```
hal9000> mpirun -np 4 ptutor3d
```

or, if your local installation allows for the explicit selection of computing nodes, use

```
hal9000> mpirun -np 4 -machinefile machines ptutor3d
```

In the latter case, the file `machines` in your application directory lists the nodes you want to use.

```
hal9000> cat machines
node14
node15
node16
node17
```

After starting the program a UG shell will open just like on a sequential machine. There is a script file in the `scripts` directory that has been modified for parallel machines, `parallel3d.scr`. Start it by typing

```
> ex parallel3d.scr
```

in the UG shell. The script is based on `coupled3d.scr`, so you will probably already know the outcome of this computation. However, if you look at the graphic window that will open, you will notice that the topmost picture shows a disconnected grid. This is the distributed grid, where each element of the grid received its colors according to the processor it is stored on. To emphasize the distribution of the grid, the partitions have been shrunk by a factor of 0.9.

The process of partitioning a given grid that is stored on one processor into several parts that are then distributed to several processors is called load balancing. The determination of a decent distribution is a very complex task, that

will not be explained here; for an introduction see (Bastian 1998). We will only describe, how load balancing is controlled from within the UG shell.

If you compare the script files `coupled3d.scr` and `parallel3d.scr`, you will find that the grid refinement has been changed, from

```
j = 0;
repeat {
  if (j == MAXLEVEL) break;
  refine $a;
  j = j+1;
}
```

to

```
j = 0;
repeat {
  if (j==LBLEVEL) {
    if (conf:parallel) ex lb4.scr;
  }
  if (j==MAXLEVEL) break;
  refine $a;
  j=j+1;
}
```

The variable `LBLEVEL` has been set at the top of the script file. It stores the number of the multigrid level on which load balancing should be performed. Since the actual load balancing command is rather complex, it is hidden in the file `lb4.scr`. The variable `conf:parallel` stores whether we are running a parallel program, so no unnecessary execution of the load balancing script is performed in the sequential case. By using `conf:parallel` it is possible to employ the same script for sequential and parallel computations; just enclose the commands that apply only for parallel computations in `if (conf:parallel) { ... }`-blocks.

For the tutorial example it is sensible to perform load balancing only on the level `LBLEVEL`, because on coarser levels there are so few elements that distributing them would delegate so few elements to each processor that communication time would become a noticeable factor, especially if you are using a workstation cluster. (A grid with 96 elements, distributed onto 16 processors would leave only six elements on each processor.) On the levels  $l > LBLEVEL$  a new load balancing wouldn't improve the grid distribution but please bear in mind that these explanations only hold for the tutorial example with its simple grid and geometry.

Now we will take a quick look the file `lb4.scr`. As we have noted above, this file exists to hide the complexity of the actual load balancing command `lb4`. `lb4` is based on the load balancing toolkit CHACO, which has been integrated into UG and replaced `lb4s` predecessors `lb1 ... lb3`.

```
hal9000> head lb4.scr
# set depth of clusters
```

```

StartLevel = @LBLEVEL;
Depth = 2;
MinElem = 1;
LB = 0;

if (LB == 0)
{
    # partition with RCB
    lb4 @StartLevel @Depth 1 @MinElem 500 10 0 0 0 1 0 0 0;
}

```

The most important settings for `lb4` have been parameterized in the script:

*StartLevel* This is the level, from which `lb4` starts load balancing. If the multigrid hierarchy is already five levels deep, calling `lb4` with `StartLevel=2` will generate a load-balanced multigrid hierarchy from level 2.

*MinElem* The minimal number of elements that each processor should store.

*LB* Determines the load balancing strategy that is used. Recursive Coordinate Bisection (RCB) and Recursive Spectral Bisection (RSB) with the Kernhigan-Lin algorithm (KL) are among the most common choices.

Another difference between the sequential and the parallel example is the initialization of the multigrid smoother.

```
npinit ilu $damp 0.92;
```

The same applies for the `t_smooth numproc`, a symmetric Gauss-Seidel-Smoothen. Damping becomes necessary in the parallel case because the parallel smoother isn't implemented as an exact copy of the sequential smoother. Apart from the fact that implementing a parallel smoother that yields exactly the same results as its sequential counterpart is complicated for unstructured grids, the real reason for using a different approach are the considerably greater communication requirements of such a smoother. The method implemented in UG performs one smoothing step in each partition of the grid and then performs an update of the solution (which requires only one communication of each processor with its neighbours). The resulting method is a block-Jacobi method in which the matrix blocks correspond to grid partitions, and the Jacobi-like nature of the algorithm explains why damping becomes necessary.<sup>1</sup>

---

<sup>1</sup>Another implication of this method is, that if you are using an exact solver on your coarsest grid, you will have to make sure that it is stored on one processor.



# 4

## Basic data types

This chapter provides a short introduction into basic UG data types as far as they are important to understand the implementation of the tutor problem class. We will not discuss the implementation of the data types in detail but we will rather concentrate on what data types there are and how they can be accessed. Most data types to be discussed here are defined in the file `UG/ug/gm/gm.h`. Table 4.1 gives the names of the basic data types together with a short description.

### 4.1 Unstructured mesh data structure

The finite element mesh is represented by the first seven data types in Table 4.1. We shortly describe each datatype and its most important attributes. Access to components of the individual data types is usually done via macros defined in `UG/ug/gm/gm.h`. We will write these macros here in the form of functions since this makes clear the types of the parameters and the return value. Typically, “functions” with all upper case names are macros in UG. Basic data types such as `short`, `int`, `float` and `double` are used in UG via the upper case names `SHORT`, `INT`, `FLOAT` and `DOUBLE`. These macros are mapped to appropriate machine-dependent data types in the file `UG/ug/arch/compiler.h`.

Table 4.1: Overview of UG basic data types.

data type	short description
MULTIGRID	mother of all information
GRID	access all data on a single grid level
ELEMENT	generic element type, provides local access
NODE	level-dependent part of a mesh node
VERTEX	level-independent part of a mesh node
LINK	makes a list of all neighboring nodes
EDGE	bidirectional connection of two nodes
BNDP	point located on a boundary segment
BNDS	element face located on a boundary segment
VECTOR	data associated with a geometric object
MATRIX	a (block) matrix entry
VECDATA_DESC	describes layout of a vector
MATDATA_DESC	describes layout of a matrix

### 4.1.1 MULTIGRID

The MULTIGRID data type is a container type representing a complete hierarchical mesh structure. It provides access to the data objects on each mesh level and some general statistics such as number of elements, number of nodes, etc..

```
GRID *GRID_ON_LEVEL (MULTIGRID *mg, int l);
```

This macro provides access to individual levels of the mesh. It returns a pointer to a GRID structure which is defined below. The highest level available and the “current” level maintained by the user interface are accessible via

```
INT TOPLEVEL (MULTIGRID *mg);
```

```
INT CURRENTLEVEL (MULTIGRID *mg);
```

### 4.1.2 GRID

The GRID data type provides access to all data objects on a mesh level. All ELEMENT, NODE, VERTEX and VECTOR objects on a mesh level are connected in a double linked list structures (each type in a different list). The first elements of these lists are accessible via

```
ELEMENT *FIRSTELEMENT (GRID *g);
```

```
NODE *FIRSTNODE (GRID *g);
```

```
VERTEX *FIRSTVERTEX (GRID *g);
```

```
VECTOR *FIRSTVECTOR (GRID *g);
```

### 4.1.3 ELEMENT

The ELEMENT is an abstract data type. It can represent a triangle or a quadrilateral in 2D or a tetrahedron, a pyramid, a prism or a hexahedron in 3D. The size of each element is determined individually by the UG grid manager. The size also depends on whether the element is at the boundary of the domain and where degrees of freedom are required. It is therefore mandatory to access an element only via the macros explained in the following.

```
INT TAG (ELEMENT *e);
```

Returns the element tag indicating the element type. Tag values are defined in UG/ug/gm/gm.h.

```
INT SUBDOMAIN (ELEMENT *e);
```

A domain can be subdivided into subdomains by defining internal boundaries. Each element can only be part of exactly one subdomain.

```
ELEMENT *SUCCE (ELEMENT *e);
```

Returns a pointer to the next element in the list of elements on a grid level.

```
INT CORNERS_OF_ELEM (ELEMENT *e);
```

Returns the number of corners of the element.

```
INT EDGES_OF_ELEM (ELEMENT *e);
```

Returns the number of edges of the element.

```
INT SIDES_OF_ELEM (ELEMENT *e);
```

Returns the number of sides (faces) of the element.

```
NODE *CORNER (ELEMENT *e, INT i);
```

Returns a pointer to the *i*'th corner node of the element.

```
ELEMENT *NBELEM (ELEMENT *e, INT i);
```

Returns a pointer to the neighboring element over face *i*. A NULL pointer indicates that this face is a boundary face.

```
INT NSONS (ELEMENT *e);
```

Returns the number of son elements on the next higher element that originated from refinement of the given element.

```
ELEMENT *SON (ELEMENT *e, INT i);
```

Returns a pointer to the *i*'th son element.

```
ELEMENT *EFATHER (ELEMENT *e);
```

Returns a pointer to the element on the next coarser level from which the given element originated during refinement.

```
VECTOR *EVECTOR (ELEMENT *e);
```

Returns a pointer to the VECTOR structure attached to the element.

```
BNDS *ELEM_BNDS (ELEMENT *e, INT i);
```

Returns a pointer to a BNDS structure containing information where face *i* is located on the boundary.

#### 4.1.4 NODE

The node data type represents a mesh node. Nodes on a fine grid level that already existed in the coarser level are represented by separate NODE objects but share a common VERTEX object.

```
VERTEX *MYVERTEX (NODE *n);
```

provides access to this corresponding VERTEX object.

```
NODE *SUCCN (NODE *n);
```

Returns a pointer to the next node in the double linked list of nodes on each mesh level.

```
VECTOR *NVECTOR (NODE *n);
```

Returns a pointer to the VECTOR structure attached to the node.

#### 4.1.5 VERTEX

The VERTEX data type provides common information shared by nodes at the same position but on different grid levels.

```
VERTEX *SUCCV (VERTEX *v);
```

Returns a pointer to the next vertex in the double linked list of vertices on each mesh level.

```
INT OBJT (VERTEX *v);
```

Returns BVOBJ if the vertex is at the boundary of the domain and IVOBJ else.

```
DOUBLE XC (VERTEX *v);
```

```
DOUBLE YC (VERTEX *v);
```

```
DOUBLE ZC (VERTEX *v);
```

These macros return the  $x$ ,  $y$  and  $z$  coordinate of the position of the vertex. The position of the vertex can also be accessed as an array of length DIM via

```
DOUBLE *CVECT (VERTEX *v);
```

```
BNDP *V_BNDP (VERTEX *v);
```

Returns a pointer to a data structure holding information about the position on the domain boundary.

#### 4.1.6 LINK AND EDGE

These data types provide a list of neighboring nodes for each NODE object. Nodes are neighbors if they are connected by an edge in the mesh. Degrees of freedom (in the form of a VECTOR object) can be attached to an EDGE object.

These data structures are not explained here since they will not be needed in the tutorial problem class.



## 4.2 Geometry data structure

A domain  $\Omega$  in UG is described by its boundary  $\partial\Omega$  and possibly by internal boundaries which divide the domain into subdomains. The boundary is supposed to be piecewise smooth and is therefore described by a collection of boundary segments. Each boundary segment is given by a function mapping a parameter interval to a  $d - 1$ -dimensional manifold,  $d = 2, 3$ . The topology of the domain (number of boundary segments, connectivity of boundary segments) cannot be changed during the computation.

The representation of domains is handled by a domain module in UG and is separated from the representation and management of the mesh data structure. Two different domain modules are available and can be selected at compilation time: The “standard domain” and the “linear grid model” domain. In the standard domain the user has to code a C-function for each boundary segment. This is appropriate for simple piecewise smooth objects such as a circle, a cylinder or a torus. In the linear grid model a boundary segment is given by a simplex mesh (a polygon in 2D, a triangular surface mesh in 3D).

The mesh data structure is independent of the domain representation and accesses the domain definition via the domain interface. If a VERTEX object or a face (edge in 2D) of an ELEMENT object is at the boundary of the domain, the corresponding position is maintained in a corresponding BNDP or BNDS structure which is filled by the domain module.

Boundary conditions are provided by the user separately for each boundary segment. The parametrization can be used to change the type and value of the boundary condition accordingly.

In the discretization code it is necessary to access this boundary condition data. This is done via the following functions exported by each domain module. In this way discretization code is independent of the domain module used.

```
BNDP_BndCond(BNDP *p, INT *n, INT i, DOUBLE *in,
             DOUBLE *v, INT *t);
```

Evaluates user boundary condition function for boundary position given by  $p$ . Since a boundary node can be located on several boundary segments simultaneously, this number of segments is returned in the parameter  $n$ . One then usually calls the function with  $i$  set to all values from 0 to  $*n-1$  and checks whether all boundary conditions matches or prefers Dirichlet over Neumann boundary conditions. The array  $in$  contains additional parameters to be passed to the users function and results are returned in  $v$  and  $t$ .

```
BNDS_BndCond(BNDS *s, DOUBLE *local, DOUBLE*in,
             DOUBLE *v, INT *t);
```

Evaluates user boundary condition function for the boundary face  $s$ . Position with this face is given by  $local$  in the local coordinate system of the boundary face. Parameters  $in$ ,  $v$  and  $t$  are the same as above.

### 4.3 Sparse matrix vector data structure

Numerical data can be associated with nodes, edges, faces and elements of the mesh. Numerical data for each such object is stored in the VECTOR data type as small array. The size of this array depends on the *type* of geometrical object the VECTOR object is associated with. In 3D there can be four different sizes of the VECTOR objects. The sizes are determined by the format associated with the multigrid (see the `format` command in the introductory example).

Sparse matrices are stored in a list based block compressed row storage scheme. A VECTOR object has a list of MATRIX objects. Each MATRIX object stores the matrix entries coupling the degrees of freedom in two corresponding VECTOR objects. A MATRIX object may itself contain a sparse matrix which is handy in high-dimensional PDE systems.

#### 4.3.1 VECTOR

These are the most important aspects of the VECTOR data type:

```
DOUBLE VVALUE (VECTOR *v, INT i);
```

Accesses the *i*'th value stored in the VECTOR object. This macro can also be used as an lvalue in order to assign new values.

```
VECTOR *SUCCV (VECTOR *v);
```

All VECTOR objects on a grid level are connected in a double linked list structure. This macro returns a pointer to the next element in the list.

```
void SET_SKIP_BIT (VECTOR *v, INT eq, INT val);
```

Each VECTOR object maintains a list of so-called “skip bits”. These are used to inform the solver about dirichlet boundary conditions and are necessary for proper operation of the multigrid transfer operators. Parameter *eq* corresponds to the component in case of a system of PDEs and *val* is either 0 or 1.

```
INT VECSKIPBIT (VECTOR *v, INT eq);
```

Reads the skip bits.

```
MATRIX *VSTART (VECTOR *v);
```

Returns first element of the matrix list for this vector. This list contains the row of the matrix for all degrees of freedom stored in *v*.

```
MATRIX *GetMatrix (VECTOR *v1, VECTOR *v2);
```

Returns the MATRIX object in the list of *v1* the contains the coupling with *v2*.

#### 4.3.2 MATRIX

MATRIX objects are only accessible via the list head in the VECTOR objects.

```
DOUBLE MVALUE (MATRIX *m, INT i);
```

Provides access to the matrix entries. Can be used for reading and writing.

```
MATRIX *MNEXT (MATRIX *m);
```

Returns the next MATRIX object or a NULL pointer if the end of the list is reached.

```
INT MDIAG (MATRIX *m);
```

Returns 1 if the given MATRIX object is a diagonal entry.

```
VECTOR *MDEST (MATRIX *m);
```

If the given MATRIX object *m* is in the list of VECTOR object *v1* and *v2*=MDEST(*m*) then *m* contains all the matrix entries coupling degrees of freedom in *v1* with those in *v2*. In a standard compressed row storage scheme this is equivalent to the column index.

### 4.3.3 VECDATA\_DESC

The VECTOR data type contains *all* numerical data at a geometrical object that is ever needed during a computation. E. g. in case of a PDE system with two components it might contain two values for the solution, two values for the right hand sides and any other quantities needed in the linear/nonlinear solvers. When a solver is called we need to specify exactly where the right hand side and the solution is stored in each VECTOR object. This is the job of the VECDATA\_DESC structure (“vector data descriptor”).

We only describe one macro here to access a VECDATA\_DESC structure.

```
SHORT *VD_ncomp_cmpptr_of_otype(VECDATA_DESC *vd,  
    INT loc, INT *n);
```

Given a VECDATA\_DESC and a geometric location, e. g. NODEVEC this macro returns the number of double values used by the given VECDATA\_DESC in the parameter *n*. The indices of the individual components are given in the return value of type SHORT \*.

### 4.3.4 MATDATA\_DESC

A MATRIX object may contain the entries of more than one matrix, e. g. a stiffness matrix and an incomplete decomposition. The MATDATA\_DESC (“matrix data descriptor”) describes the layout of a matrix, i. e. which values in the MATRIX objects are used for the matrix.

```
SHORT *MD_nr_nc_mcmptr_of_ro_co(MATDATA_DESC *md,  
    INT sloc, INT dloc, INT *nr, INT *nc);
```

Given a `MATDATA_DESC` and a source and destination location this function returns the size of the block matrix coupling degrees of freedom in these two locations and the individual positions of these entries in the `MATRIX` data structure. E. g. `sloc` and `dloc` may both be specified as `NODEVEC`.

# 5

## The tutorial application

### 5.1 Overview

The tutorial code customizes the UG framework to solve a certain set of partial differential equations – a groundwater flow equation and a solute transport equation. It is structured into two parts: the problem class containing the discretization scheme and the application containing the description of domains, boundary conditions and coefficient functions. Together they make up a complete simulation model.

In this section we will look at the application part, the next section will treat the problem class implementation. The tutorial code can be found in the `UG/tutor` directory. The subdirectories `appl` and `pclib` contain the application and problem class part, respectively.

### 5.2 The `main()` function

Every C (and C++) program starts execution at the function called `main()`. This function is contained in `tutor.c` in the application directory and is shown in Fig. 5.2.

The `main()` function is quite short. Its purpose is to initialize the UG framework (line 6 in the figure), the tutorial problem class (line 9) and setup the application specific parts (lines 17 to 26). The application customizes the framework by adding descriptions of domains, boundary condition functions and providing application specific parts of the simulation model (like a viscosity value, for example).

After that is done `main()` passes control to UG by calling the function `CommandLoop()` in line 29. This brings up the shell and your application is ready to receive commands.

In the following we will look into how a domain is set up and how the application specific part of a simulation model is set up.

### 5.3 Setting up a domain

Consider the two-dimensional domain in the left part of Fig. 5.2. It is multiply connected and consists of three different subdomains with an interface between them. Such a (complicated) domain is represented in UG by specifying its boundary which is supposed to consist of piecewise smooth parts called “boundary segments”. Each boundary segment is given by a mapping from a parameter

```
1  int main (int argc, char **argv)
2  {
3      INT err;
4
5      /* init ug library */
6      if (InitUg(&argc,&argv)!=0) return(1);
7
8      /* init problem class library next */
9      if ((err=InitTutor())!=0)
10     {
11         printf("ERROR in main while: called routine line %d\n",
12              (int) HiWrd(err), (int) LoWrd(err));
13         printf ("aborting ug\n");
14         return (1);
15     }
16
17     /* domains */
18     #ifdef __TWOEDIM__
19         if (InitDomains2d()!=0) return(1);
20     #endif
21     #ifdef __THREEDIM__
22         if (InitDomains3d()!=0) return(1);
23     #endif
24
25     /* problems */
26     if (InitProblems()!=0) return(1);
27
28     /* execute commands till quit */
29     CommandLoop(argc,argv);
30
31     return(0);
32 }
```

Figure 5.1: The main() routine of the tutorial application

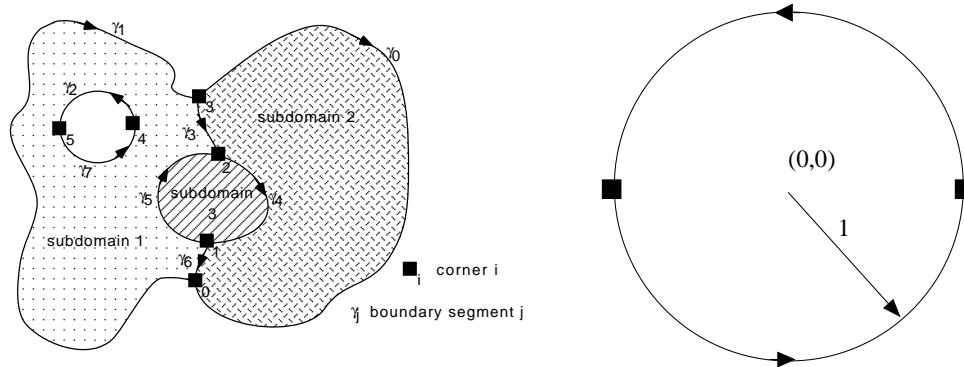


Figure 5.2: A two-dimensional domain.

space (in 2D a line segment) to a  $d - 1$ -dimensional manifold (a curve or a surface). In addition we have to specify how the individual boundary segments are connected to make up the subdomains. The points in space where two or more boundary segments meet are called “corners”. Typically this is where the boundary is non-smooth and where you want to have a node in your mesh.

We consider the process of describing a two-dimensional domain by looking at a simple unit circle (right part of Fig. 5.2). The code for this domain is given in the file `domains2d.c` in the application directory.

The domain description in UG requires that the two corners associated with a (two-dimensional) boundary segment are *different*. Therefore we must split up the boundary of the circle into two segments. In our implementation we use the upper and the lower part, i. e. the corners will be  $(1, 0)$  and  $(-1, 0)$ .

For each segment we have to write a C-function with a specific format that

```

1 static INT circleBoundaryUpper (void *data, DOUBLE *param, DOUBLE *result)
2 {
3     DOUBLE lambda;
4
5     /* retrieve parameter value */
6     lambda = param[0];
7
8     /* check range */
9     if ((lambda<0.0)|| (lambda>1.0)) return(1);
10
11     /* fill result */
12     result[0] = cos(PI*lambda); /* x */ /* PI defined in misc.h */
13     result[1] = sin(PI*lambda); /* y */
14
15     /* return ok */
16     return(0);
17 }

```

Figure 5.3: Function defining upper part of the unit circle

implements the mapping. For the upper half this function is shown in Fig. 5.3. This function is called with a parameter value given in `param` and returns the corresponding point in 2D-space in `result`. The parameter range can be chosen arbitrarily and is  $[0, 1]$  here.

The `main()` function shown above calls the function `InitDomains2d()`, this in turn calls the function `InitUnitCircle()` which is shown in Fig. Fig:UnitCircleFunction. As its name implies this function creates a description of the unit circle domain within the UG framework. There it is stored in the (standard) domain module in a certain data structure which is not of interest to us. Note that the creation of a domain does not mean that the application will use the unit circle domain for the simulation. You can create an arbitrary number of different domains within your application at initialization time. The domain to be used for a simulation is selected later by the new command in your script file.

A domain description is created by first calling the (dimension-independent) function `CreateDomain()`. For the case of the unit circle, see Fig. 5.3. As parameters one has to specify the name, a ball containing the domain, the number of boundary segments and the number of corners the domain consists of. Then the function `CreateBoundarySegment2D` must be called for each boundary segment to be defined. Parameters are: the name of the boundary segment, the left and right subdomain (the segment has an orientation by walking on it in the direction of increasing parameter values), the number of the segment (starting with 0), The corner where the segment starts (i. e. the point corresponding to the smallest parameter value) and where it ends (i. e. the point corresponding to the largest parameter value), the resolution to be used for drawing the boundary segment, the range of the parameter, a pointer to the function doing the parameter mapping and finally a user definable pointer. This user definable pointer is given to the mapping function as its first argument (see Fig. 5.3).

Going back to Fig. 3.3 you can see how this domain is used for a simulation in a script file (line 8) and how an initial mesh is created (lines 11–16). The shell-command `bn` inserts boundary nodes on a given segment (by specifying its number) and the parameter value. As you can see six points are inserted at the boundary at positions  $(\sqrt{2}/2, \sqrt{2}/2)$ ,  $(0, 1)$ ,  $(-\sqrt{2}/2, \sqrt{2}/2)$ ,  $(-\sqrt{2}/2, -\sqrt{2}/2)$ ,  $(0, -1)$  and  $(\sqrt{2}/2, -\sqrt{2}/2)$ . The nodes at corner points  $(1, 0)$  and  $(-1, 0)$  are not defined in the script file, they are already inserted by the new command by default. This feature can not be switched off. This is because a side (edge or face) of an element must always correspond to exactly one boundary segment. Therefore you always need to have nodes at the corner points of a domain.

As you might suspect, this description of a domain gets quite tedious for complicated domains. Especially in three dimensions where it is sometimes not so easy to find the mapping functions. For that reason we also omit the description of the three-dimensional case here, maybe I add it later.



```

1  static INT InitUnitCircle (void)
2  {
3      DOUBLE radius, MidPoint[2];
4
5      /* allocate new domain structure */
6      MidPoint[0] = MidPoint[1] = 0.0;
7      radius = 1.05;
8      if (CreateDomain(
9          "circle",          /* name of the new domain      */
10         MidPoint, radius,  /* circle containing the domain */
11         2,                 /* number of boundary segments */
12         2,                 /* number of corners          */
13         YES                /* true if domain is convex    */
14     )!=NULL) return(1);
15
16     /* allocate the boundary segments, segment allocation must      */
17     /* immediately follow the domain definition.                    */
18     if (CreateBoundarySegment2D(
19         "circle bnd upper", /* name of the boundary segment */
20         1,                 /* number of left subdomain     */
21         0,                 /* number of right subdomain    */
22         0,                 /* number of segment, starting with 0 */
23         0,                 /* number of corner where segm start */
24         1,                 /* number of corner where segm ends */
25         40,                /* resolution, use 1 for straight li */
26         0.0,               /* begin of parameter interval   */
27         1.0,               /* end of parameter interval     */
28         circleBoundaryUpper, /* function mapping parameter to wor */
29         NULL                /* user defined pointer to be suppli */
30     )!=NULL) return(1);
31     if (CreateBoundarySegment2D(
32         "circle bnd lower", /* name of the boundary segment   */
33         1,                 /* number of left subdomain       */
34         0,                 /* number of right subdomain      */
35         1,                 /* number of segment, starting with 0 */
36         1,                 /* number of corner where segment st */
37         0,                 /* number of corner where segment en */
38         40,                /* resolution, use 1 for straight li */
39         0.0,               /* begin of parameter interval     */
40         1.0,               /* end of parameter interval       */
41         circleBoundaryLower, /* function mapping parameter to wor */
42         NULL                /* user defined pointer to be suppli */
43     )!=NULL) return(1);
44
45     /* return ok */
46     return(0);
47 }

```

Figure 5.4: Function making the unit circle domain know to the UG framework.

## 5.4 Setting up a problem

### 5.4.1 NUMPROCS

The tutorial problem class solves the general problem given in Eqs. (3.2) and (3.3). We want the different coefficient functions (like viscosity, density, permeability, source terms etc.) in these equations not to be hardcoded into the discretization scheme but rather to be supplied by the application. The same also applies for boundary and initial conditions. From the point of view of the UG framework it is arbitrary how the discretization scheme gets this information from the application: It involves only communication between the problem class and the application. Nevertheless UG provides (at least two) different ways to ease this communication. Boundary conditions are treated differently since they involve the geometry representation. The way how to prescribe boundary conditions depends on the domain module that is used (in our case standard domain).

Let us first consider the flow equation (3.2) now. Besides boundary conditions there are five parameters for the flow problem: Permeability, viscosity, density, gravity vector and source term. Using an object-oriented approach we design an abstract class containing these methods and then provide different realizations in the application code. In C we mimick a class definition by a structure with function pointers. This C structure is given in Fig. 5.4.1 and can be found in the file `flow.h` in `UG/tutor/pclib`. Historically these “class definitions” have been known as “numerical procedures”, “numprocs” for short, in UG.

Looking closely, we find a sixth method in this class definition: the Darcy velocity  $\mathbf{u}$ . This function is not used in the discretization of the flow problem but it is used in the discretization of the transport problem where it is one of the parameters.

Another thing to mention about the methods defined in `NP_FLOW_PARAM` is that *every method receives a pointer to the object as its first argument. This corresponds to the `this` pointer in C++.*

Class `NP_FLOW_PARAM` shown in Fig. 5.4.1 is derived from a more general class called `NP_BASE` shown in Fig. 5.4.1. In fact *all* numprocs are derived from this abstract base class. `NP_BASE` adds three more methods to our class: setting of parameters with `Init()`, display of status with `Display()` and execution of some action with `Execute`. These methods are very general since they apply to all numprocs, e. g. all numerical algorithms in UG are derived from this class. The `Init()`, `Display()` and `Execute` functions will never be called by the discretization scheme, rather these methods are called by the shell commands `npinit`, `npdisplay` and `npexecute` which we already encountered in our first script file in Chapter 3.

Now how are objects of class `NP_FLOW_PARAM` instantiated? They are instantiated by the UG framework when the command `npcreate` is issued on the UG shell. Since objects are created by the UG framework it needs to know the class

```

1 struct np_flow_param {
2
3     NP_BASE base;                /* inherits base class          */
4
5     /* functions */
6     INT (*Permeability)
7         (struct np_flow_param *,    /* pointer to (derived) object */
8          DOUBLE *,                  /* position vector              */
9          INT ,                      /* subdomain id                */
10         DOUBLE *);                /* result tensor               */
11     DOUBLE (*Viscosity)
12         (struct np_flow_param *,    /* pointer to (derived) object */
13          VECTOR *);                /* may depend on any nodal quantity*/
14     DOUBLE (*Density)
15         (struct np_flow_param *);   /* is constant                  */
16     INT (*Gravity)
17         (struct np_flow_param *,    /* pointer to (derived) object */
18          DOUBLE *);                /* result vector                */
19     DOUBLE (*Source)
20         (struct np_flow_param *,    /* pointer to (derived) object */
21          DOUBLE *,                  /* position vector              */
22          INT);                      /* subdomain id                */
23     INT (*DarcyVelocity)
24         (struct np_flow_param *,    /* pointer to (derived) object */
25          const ELEMENT *,           /* element                      */
26          DOUBLE *,                  /* position in local coordinates */
27          DOUBLE *);                /* result vector                */
28 };
29 typedef struct np_flow_param NP_FLOW_PARAM;

```

Figure 5.5: A “class definition” providing parameters for the flow problem.

```

1 struct np_base {
2
3     /* data */
4     ENVVAR v;                    /* is an environment variable   */
5     MULTIGRID *mg;               /* associated multigrid         */
6     INT status;                  /* has a status, NO type and size...*/
7
8     /* functions */
9     INT (*Init) (struct np_base *, INT, char **); /* initializing routine */
10    INT (*Display) (struct np_base *);           /* Display routine */
11    INT (*Execute) (struct np_base *, INT, char **); /* Execute routine */
12 };
13 typedef struct np_base NP_BASE;

```

Figure 5.6: The NP\_BASE class.

```

1 typedef struct {
2
3     NP_FLOW_PARAM fp;           /* extend this class          */
4
5     /* variables */
6     VECDATA_DESC *p;           /* we need pressure field    */
7     INT p_comp;                /* for fast access           */
8
9 } NP_SIMPLE_FLOW;

```

Figure 5.7: The NP\_SIMPLE\_FLOW class (to be found in problems.c in UG/tutor/appl).

definitions. A numproc class definition is made known to the framework via the `CreateClass()` function call (see the example below).

#### 5.4.2 AN EXAMPLE

We are now in a position to do a complete example how a flow problem is described using class NP\_FLOW\_PARAM. The code to be described in this subsection is contained in the file `problems.c` in `UG/tutor/appl`. In order to be able to compute the Darcy velocity we will need to know the pressure field. In Fig. 5.4.2 a new class is derived from NP\_FLOW\_PARAM where we added a vector data descriptor variable of type VECDATA\_DESC, see Chapter 4. This essentially contains information where the pressure field is stored in memory.

Now we have to give concrete implementations of all the abstract methods in NP\_SIMPLE\_FLOW. These are the functions from NP\_BASE and from NP\_FLOW\_PARAM.

Let us start with the `Init()` method from NP\_BASE. It is shown in Fig. 5.4.2. It gets a pointer to the object it has to initialize as first argument. Since it is a base class method this object is of type NP\_BASE. The other arguments contain the command line given after the `npinit` command on the shell. Like the arguments to the C `main()` function this is the number of options and an array of strings containing each option. The body of the function then looks for an option of the form `$p name`, where `name` is then interpreted as the name of a vector data descriptor. This is done by the function `ReadArgvVecDesc()`. If no such vector data descriptor is found the status NP\_NOT\_ACTIVE is returned, indicating that initialization is not complete. If one is found the component number is stored in the `p_comp` attribute for later use.

There is no room here to discuss all functions in detail. Let us look at one more function, `SimpleFlowPerm()`, which is shown in Fig. 5.4.2. It returns the permeability tensor in the `out` array. As arguments it gets the position in array `x` and the subdomain number `sd`. As you can see, we assign different permeability values according to the subdomain number. The flow model allows for a full

```

1 static INT NPSimpleFlowInit (NP_BASE *theNP, INT argc , char **argv)
2 {
3     NP_SIMPLE_FLOW *np;
4     INT n;
5
6     np = (NP_SIMPLE_FLOW *) theNP;
7     np->p = ReadArgvVecDesc(theNP->mg, "p", argc, argv);
8
9     if (np->p == NULL)
10        return(NP_NOT_ACTIVE);
11
12    np->p_comp = VD_ncmp_cmpptr_of_otype(np->p, NODEVEC, &n)[0];
13    if (n!=1) return(NP_NOT_ACTIVE);
14
15    return(NP_ACTIVE);
16 }

```

Figure 5.8: The Init() method.

permeability tensor but we return only a diagonal matrix with identical values on the diagonal.

Imagine an object of type NP\_SIMPLE\_FLOW in memory. How do the function pointers in this structure get assigned their values. This is done by the constructor function SimpleFlowConstruct shown in Fig. 5.4.2. As discussed, Creation of objects is done by the UG framework. The code segment

```

if (CreateClass(FLOW_CLASS_NAME ".sf", sizeof(NP_SIMPLE_FLOW),
    SimpleFlowConstruct))
    return (__LINE__);

```

makes a class definition known to the UG framework. It tells UG the name of the class, in this case sf for “simple flow”, the size of an object of this class and a pointer to the constructor function. When the command

```
> npcreate sfobj $c sf;
```

is issued on the shell an object of type NP\_SIMPLE\_FLOW is created in memory and the creator function is called for it, i. e. the proper function pointers are assigned. This object is known in the UG framework under the name sfobj. In that way it can be passed to other objects as parameters. E. g. consider the following sequence of script commands:

```

npcreate sfobj $c sf;
npinit sfobj $p sol;

npcreate bfobj $c bf;
npinit bfobj $A MAT $x sol $b rhs $P sfobj;

```

The first line creates the object sfobj of class sf which is NP\_SIMPLE\_FLOW. The second line sets the parameters of sfobj. In that case it tells that the location

```
1 static INT SimpleFlowPerm (NP_FLOW_PARAM *fp, DOUBLE *x, INT sd, DOUBLE *out)
2 {
3     DOUBLE k;
4
5     switch (sd) {
6         case 1: k = 1.0E-10; break;
7         case 2: k = 1.0E-14; break;
8         case 3: k = 1.0E-8; break;
9         default:
10            k = 1.0E-10; break;
11    }
12
13    #ifdef __TWO DIM__
14        out[0] = k; out[1] = 0.0;
15        out[2] = 0.0; out[3] = k;
16    #endif
17
18    #ifdef __THREEDIM__
19        out[0] = k ; out[1] = 0.0; out[2] = 0.0;
20        out[3] = 0.0; out[4] = k ; out[5] = 0.0;
21        out[6] = 0.0; out[7] = 0.0; out[8] = k ;
22    #endif
23
24    return(0);
25 }
```

Figure 5.9: The SimpleFlowPerm() method.

```

1  static INT SimpleFlowConstruct (NP_BASE *theNP)
2  {
3      NP_FLOW_PARAM *np;
4
5      np = (NP_FLOW_PARAM *) theNP;
6
7      /* base functions */
8      theNP->Init          = NPSimpleFlowInit;
9      theNP->Display      = NPSimpleFlowDisplay;
10     theNP->Execute       = NPSimpleFlowExecute;
11
12     /* flow problem functions */
13     np->Permeability     = SimpleFlowPerm;
14     np->Viscosity        = SimpleFlowViscosity;
15     np->Density          = SimpleFlowDensity;
16     np->Gravity          = SimpleFlowGravity;
17     np->DarcyVelocity    = SimpleFlowDarcyVelocity;
18     np->Source           = SimpleFlowSource;
19
20     return(0);
21 }

```

Figure 5.10: The constructor function SimpleFlowConstruct.

of the pressure field in memory is stored in a vector data descriptor named `sol`. In the third line an object `bfobj` is created which is of class `bf` which is the discretization scheme. Finally the discretization object is initialized with the descriptors for the matrix, solution, which is the pressure field `sol`, and the right hand side. With the `$P sfobj` option we tell the discretization object that it should use `sfobj` to get its coefficient functions.

File `problem.c` contains four different class definitions. In order to find out what they do it is best to first look at the `CreateClass()` calls and then look at the corresponding constructor function to see what function pointers it assigns.

The coefficient functions for the transport problem are contained in class `NP_TRANSPORT_PARAM` shown in Fig. 5.4.2. It contains methods for the non-linearity of the flux function,  $\phi(C)$ , porosity, density, dispersion tensor, source term and initial conditions.

### 5.4.3 BOUNDARY CONDITIONS

Boundary conditions are set up similar to a domain definition. For each boundary segment a corresponding boundary condition function has to be written that uses the same parametrization and returns type and value of the boundary condition for a given parameter value.

Fig. 5.4.3 shows such a function for the upper half circle and corresponds to the boundary segment function in Fig. 5.3. The `in` argument contains the parameter value, value and type of the boundary condition are returned in `outValues`

```

1 struct np_transport_param {
2
3     NP_BASE base;                /* inherits base class          */
4
5     /* functions */
6     DOUBLE (*Phi)
7         (struct np_transport_param *, /* pointer to (derived) object */
8          DOUBLE);                  /* Concentration                */
9     DOUBLE (*Porosity)
10        (struct np_transport_param *, /* pointer to (derived) object */
11         DOUBLE *,                  /* position vector              */
12         INT);                      /* subdomain id                 */
13     DOUBLE (*TDensity)
14        (struct np_transport_param *); /* is constant                  */
15     INT (*Dispersion)
16        (struct np_transport_param *, /* pointer to (derived) object */
17         DOUBLE *,                  /* position vector              */
18         INT ,                      /* subdomain id                 */
19         DOUBLE *,                  /* velocity                     */
20         DOUBLE *);                /* result tensor                */
21     DOUBLE (*TSource)
22        (struct np_transport_param *, /* pointer to (derived) object */
23         DOUBLE *,                  /* position vector              */
24         INT);                      /* subdomain id                 */
25     DOUBLE (*Initial)
26        (struct np_transport_param *, /* pointer to (derived) object */
27         DOUBLE *);                /* position vector              */
28 };
29 typedef struct np_transport_param NP_TRANSPORT_PARAM;

```

Figure 5.11: Interface class for the transport problem.

```

1 static INT CircleProblemBoundaryUpper (void *segdata, void *conddata,
2         DOUBLE *in, DOUBLE *outValues, INT *bndType)
3 {
4     DOUBLE lambda,x,y;
5
6     /* retrieve parameter value */
7     lambda = in[0];
8
9     /* return type of boundary condition in bndType array */
10    bndType[0] = DIRICHLET;
11
12    /* return value on outValues array, here exact solution */
13    x = cos(PI*lambda); y = sin(PI*lambda);
14    outValues[0] = x*x*x+y*y;
15
16    /* everything ok */
17    return(0);
18 }

```

Figure 5.12: Boundary condition definition for the upper half of the unit circle.



```

1  static INT InitCircleProblem (void)
2  {
3      /* allocate new problem structure */
4      if (CreateProblem(
5          "circle",          /* name of domain where problem lives */
6          "circle problem", /* name of the problem */
7          FLOW_PROBLEMID,   /* problem ID exported by problem class */
8          NULL,             /* the configuration function (or NULL) */
9          0,                /* number of coeff. functions supplied */
10         NULL,             /* array with coefficient function ptrs */
11         0,                /* number of user functions supplied */
12         NULL              /* array with user function ptrs */
13     )==NULL) return(1);
14
15     /* allocate the boundary conditions, boundary condition allocation must */
16     /* immediately follow the problem definition. */
17     if (CreateBoundaryCondition(
18         "circle bnd cond upper", /* name of boundary condition segmen*/
19         0,                      /* number of corresponding bnd segme*/
20         CircleProblemBoundaryUpper, /* the function giving type and valu*/
21         NULL                    /* user supplied conddata pointer */
22     )==NULL) return(1);
23     if (CreateBoundaryCondition(
24         "circle bnd cond lower", /* name of boundary condition segmen*/
25         1,                      /* number of corresponding bnd segme*/
26         CircleProblemBoundaryLower, /* the function giving type and valu*/
27         NULL                    /* user supplied conddata pointer */
28     )==NULL) return(1);
29
30     /* make bvp, still old style */
31     if (CreateBVP("circle problem","circle","circle problem")==NULL) return (1);
32
33     return(0);
34 }

```

Figure 5.13: Setting up a boundary value problem.

and `bndType` respectively. The identifier `DIRICHLET` is exported by the problem class in `flow.h` and indicates Dirichlet boundary conditions. You can also specify Neumann (or flux) boundary conditions by specifying `NEUMANN`.

As with domains and `numprocs`, the boundary condition function must be published to the UG framework. This is done in function `InitCircleProblem()` shown in Fig. 5.4.3. First a call to `CreateProblem()` starts a new problem definition. This problem will be called “circle problem” and will use the domain named “circle” which has been defined above. Next we have to give the boundary condition function corresponding to each boundary segment with a call to `CreateBoundaryCondition`. Finally, a call to `CreateBVP()` combines the domain “circle” and the problem “circle problem” into a boundary value problem which is also called “circle problem”. This actually is the name supplied to the new command when a simulation is started. For example in `first.scr` we typed:

```
new circle $b circle problem $f FlowFormat $h 20M;
```

to make a multigrid structure called `circle` using solving the boundary value problem (`$b` option) “circle problem”.

# **6**

## **The tutorial problem class**



# 7

## Graphics

In this chapter we give an introduction to the basic understanding of the graphical subsystem and its usage. At the end you will be able to handle the graphical facilities. In section 7.1 we will explain the basic data structures (plotobject, viewedobject, picture, window). It will be explained how to define and initialize them. In a separate section we describe the high-level methods available for pictures (section 7.2). Finally a list of the basic objects (plotobjects) implemented in UG is given in section 7.3. Here the various options of these objects are described in detail. On the one hand this section serves as an overview which objects are available. Beyond that it should be consulted to figure out the exact options to initialize the object.

### 7.1 Data structures

The data structures follow, apart of some odd features, the human intuition of how a picture (of whatever) is made. The basic data structure is the plotobject. The corresponding data structure, defined in `wpm.h` is `PLOT_OBJ`. It is a representation of an object. The aim of the graphics subsystem is display some of the features of these objects. The objects can have geometrical properties but they don't have to. The features of the plotobject will be displayed graphically, so they have to have geometric representations. Let us consider the scalar solution of a partial differential equation on a two-dimensional grid. There are many ways to give a geometrical representation of such a solution or of parts of it. We like to mention a few of them.

- Rep. 1: Graph of the solution along a line (2-dimensional),
- Rep. 2: Graph of the solution (3-dimensional),
- Rep. 3: Contour lines of some fixed values,
- Rep. 4: coding the values of the scalar as colors.

Each of these can be thought of being configurable. Representation 1 depends on the position of the line. Representation 2 depends on the view one has to the 3-dimensional graph (of course only a projection to a plane can be displayed) and on the scaling factor mapping the value of the scalar into space. For representation 3 the values of the contours may be configurable etc. We have chosen an object (the scalar solution) which has some geometric parts (it is the solution on a 2-dimensional domain) and some abstract parts (its values). One can think of objects, which are purely abstract, like matrices or purely geometrical, like

a single grid. Lets consider again the four representations of a scalar solution. It is a matter of choice what to consider a plotobject having in mind something like this scalar solution. One could keep every of these representations within one plotobject or one can create four (or even more) seperate ones. In the first case an appropriate name would be

Rep. 1/2/3/4: Scalar solution,

having four different solutions appropriate names would be

Rep. 1: Line,

Rep. 2: Graph,

Rep. 3: Contours,

Rep. 4: Colors.

Often the choice is clear from technical problems involved. Sometime it is just a matter of taste or of history. The choice we took in UG w.r.t. the scalar solution is

Rep. 1: Line,

Rep. 3/4: Scalar.

The idea of representation 2 has no realization in UG. In the sequel we will speak about a `Line-plotobject`, a `Scalar-plotobject` etc. Of course every plotobject has a number of configuration parameters. The name `Line-plotobject` is chosen to explain well, what the plotobject represents, whereas the name `Scalar-plotobject` is historical. In the past it was the only way to display a scalar field. One general guide line is that every plotobject has a well-defined dimension. It is 2 in the cases of the `Line-plotobject` and the `Scalar-plotobject`. This dimension, called the dimesion of the plotobject, does not coincide the the so-called application-dimension, i.e. the dimension of space of the underlying partial differential equation.

We now discuss the functionality of the plotobjects. The graphical subsystem is written in C-language following loosely object-oriented concepts. The reader familiar with those concepts will notice a lot of differences to a strict realization. The plotobjects are the basic classes of the graphics. Whereas the creation and distruction is done for higher level objects inheriting the plotobjects, two methods are applicable to each plotobject. It can be initialized using the UG-command

```
> setplotobject <class> [$clearOn|$clearOff] [$option1 $option2 ...].
```

The class of the plotobject is specified at initialization time, not at creation time! Once created an plotobject can change its class. There is only on option common to all plotobjects. If '`$clearOn`' is specified (with is default) a previous plot

on the same picture is erased before the new content is displayed. The options differ from class to class and will be described below. In section 7.3 we give a complete list of the plotobjects together with its configuration parameters. The command refers to a "current" plotobject, which is related to the "current" picture. It links the "current" picture to the "current" multigrid. A sample may serve as an illustration:

```
> setplotobject Matrix $M A
picture 'pic_0' and multigrid 'test' coupled
```

The second method displays the initialization. It is accessible via the UG-command

```
> polist.
```

Again this command acts on the "current" plotobject. Common to all plotobjects a header is printed. As a sample may serve a fully initialized Matrix-plotobject:

```
> polist
-----
Display of PlotObject
-----
PO-NAME          = Matrix
MG-NAME          = test
STATUS           = ACTIVE:2D
CLEAR FIRST     = YES
MIDPOINT        = 144.5    144.5
RADIUS          = 144.5

range           = -4      4
regular conn    = YES
extra conn     = NO
use log        = NO
rel values     = NO
Thresh        = 0
BV blocks      = NO
ind to vec     = NO
Matrix        = A
```

The first (significant) line gives the name of the plotobject. The second tells that the corresponding picture is linked to the multigrid 'test'. The third line reports the status. It is ACTIVE, i.e. fully initialized and has the plotobject dimension 2 (if it is not fully initialized the status reads 'NOT\_ACTIVE:2D'). Line four reports the [`$clearOn`|`$clearOff`]-option. Finally the midpoint and the radius of the bounding sphere of the plotobject is displayed. What follows is specific to the Matrix-plotobject. Have a look to section 7.3 to identify the output and the options.

The plotobject is inherited by the viewedobject (data structure VIEWEDOBJ in wpm.h). It adds a view to plotobject, i.e. a plane is defined where to the

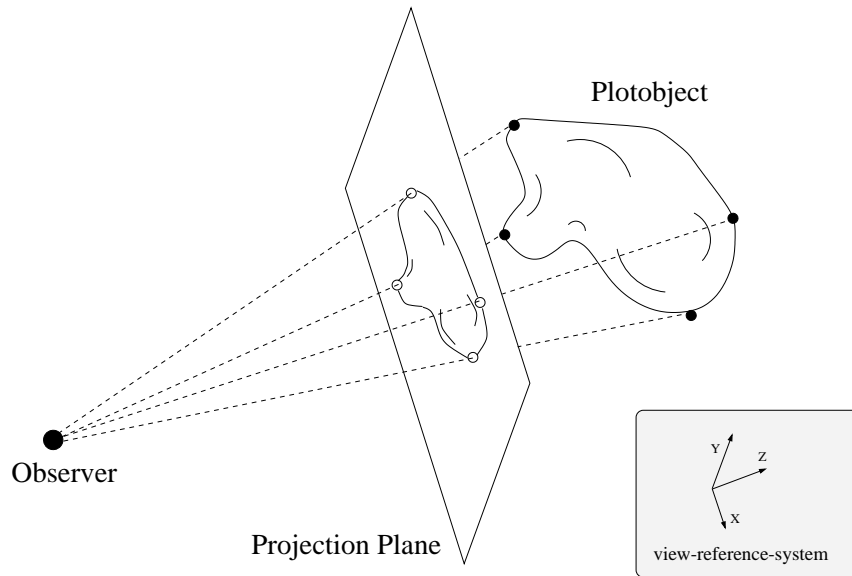


Figure 7.1: Projection of a 3d-plotobject.

plotobject is projected. The plane is a rectangular surface with well-defined position and orientation in  $\mathbb{R}^2$  or  $\mathbb{R}^3$ , according to the plotobject-dimension. It is specified by its midpoint and the direction of its x- and y-axis. In the 2d-case the projection is the restriction of the plotobject to that plane. Here and in the sequel we identify the plotobject with its geometric representation. In the 3d-case additionally a observer viewpoint has to be specified. Two modi of projection are available. The parallel projection maps the plotobject parallel to the perpendicular line onto the plane passing through the observer viewpoint. The central projection maps the plotobject onto the plane using the central perspective of the observer. Only in the latter case the distance observer-plotobject is of importance. The position of the plane (3D: and of the observer) fixes the view-reference-system (VRS). The VRS is an orthonormal coordinate system with the x- and yaxis parallel to x- and yaxis of the plane. Its z-axis points from the observer perpendicular to the plane. Figure 7.1 may illustrate the situation in the 3d-case. The methods available for viewedobjects initialize, modify and display the view to the plotobject. Table 7.1 and table refTab:Graphics:VO:cmd3 give a list of the commands related to these methods together with a short description for 2d- and 3d-plotobjects. First we discuss the command `setview` in detail. For 2d-plotobjects the syntax of the command is

```
> setview [ $\$i$ ]
    [ $\$t$  <x> <y>]
    [ $\$s$  <x> <y>]
    [ $\$x$  <x> <y>];
```

The `setview`-command provides for every option reasonable default-values. The `t`-option sets the target-point, i.e. the midpoint of the projection plane. The default is the midpoint of the domain's bounding sphere. The `s`-option allows



Table 7.1: Commands related to the viewedobject (2d-plotobject)

COMMAND	OPTIONS, DESCRIPTION
setview	Vaiious options, see below. Initialization/modification.
vdisplay	Displays the setting of the view, see below.
walk	<x> <y>; Move the observer in VRS.
zoom	<factor>; Scaling of the plane by <factor>.
drag	<x> <y>; Drags the plane in VRS by <x> <y>.
rotate	<angle>; Rotates the plane by <angle> (mathematically positive)

to set scaling-factors for each of the spatial directions (e.g. using 'setview \$s 1 0.5' leads to a in y-direction flattened display of the plotobject). The x-option allows to specify the x-direction of the VRS. This fixes the orientation as well as the size of the projection-plane. The default is a vector parallel to the x-direction and a length large enough to display the hole plotobject. The y-direction is chosen according to that choice. The i-option reinitializes the view.

For 3d-plotobject the syntax of the setview-command is

```
> setview [$i]
    [$o <x> <y> <z>]
    [$t <x> <y> <z>]
    [$s <x> <y> <z>]
    [$x <x> <y> <z>]
    [$p =|<]
    [$C]
    [$R]
    [$P <x> <y> <z>]
    [$N <x> <y> <z>];
```

The options *i*, *t*, *s*, *x* are like the one for the 2d-plotobjects. The *o*-option specifies the observer viewpoint. The default is an intelligent choice according to the main axis of inertia of the plotobject. The *p*-option sets the projection-mode ('\$p <' for central perspective, '\$p =' for parallel perspective). The central perspective is default. The last four options are related to plotobjects which allow a cut-specification. A cut-plane is defined by one point in space and a plane-normal, specified by the P- and N-option. If specified only the part of the plotobject behind the cut-plane is kept (e.g. for displaying it). To make life eas-

Table 7.2: Commands related to the viewedobject (3d-plotobject)

COMMAND	OPTIONS, DESCRIPTION
setview	Vaious options, see below. Initialization/modification.
vdisplay	Displays the setting of the view, see below.
walk	<x> <y> <z>; Move the observer in VRS.
walkaround	< $\alpha$ > < $\beta$ >; Walk around the target point on a sphere. The direction in the VRS is determined by $\alpha$ (mathematical positive form x-axis). Walks around by an angle of $\beta$ .
zoom	<factor>; Scaling of the plane by <factor>.
drag	<x> <y>; Drags the plane in VRS by <x> <y>. No z-component in VRS.
rotate	<angle>; Rotates the plane by <angle> (mathematicaly positive) around an vector perpendicular to the plane.

ier, the C-option sets an intelligent default. The R-option removes the cut-plane. Default is no specification of a cut-plane.

The vdisplay-command displays the configuration of the view. A typical example of a view for a 3d-plotobject is

```
> vdisplay
-----
Display of View of VO
-----
VO_STATUS      = ACTIVE
Dim            = TYPE_3D
Observer       = 0.3544   0.526   0.9473
Target        = 0.1414   0.1     0.09526
PlaneXDir     = 0.2079  -0.02079  -0.04158
WinWidth      = 0.426
```

The VO\_STATUS = ACTIVE tells that the view is specified correctly. A status NOT\_ACTIVE means the observer has a non-valid position w.r.t. the plotobject (within its bounding sphere or behind the object). NOT\_INIT obviously means the view is not initialized. The output for Observer, Target, PlaneXDir should be clear. WinWidth gives the total extension of the projection-plane in VRS' x-direction. The second way to use the command is

```
> vdisplay $s
setview $i
  $o 0.354435 0.526028 0.947319
  $t 0.141421 0.1 0.0952628
  $x 0.206859 -0.00182105 -0.0508042
  $p <;
```

leading to an output which can be used to initialize the view. This is useful since UG provides interactive graphical facilities to adjust the view.

All commands discussed so far refer to a "current" plotobject or viewedobject. The commands related to pictures and UG-windows (data structure `PICTURE` and `UGWINDOW` in `wpm.h`) can be accessed using unique names. Since explaining the commands related to pictures take arguments related to UG-windows both structures are best explained together. An UG-window can be thought of being a canvas where one or more pictures can be placed on. UG can handle one or more UG-windows at a time. Both UG-windows and pictures have unique names and every picture is related to exactly one UG-window. Both UG-windows and pictures can be opened and closed at runtime. If a UG-window is closed, all associated pictures will be closed too. The picture inherits the viewedobject, so everytime a picture is created/disposed, the inherited structures (viewedobject, plotobject) is created/disposed too. UG-windows are opened for a specified outputdevice. The devices available are

1. `screen`,
2. `meta`,
3. `ps`,
4. `psbw`,
5. `ppm`.

The `screen`-outputdevice opens a window on the screen (i.e. X11 on unix-type systems, ...) and is the only device with interactive facilities. All other outputdevices map the opening of a window to opening of a file and are therefore not interactive. They differ in the file format they use for writing the graphical information. The `meta`-device is related to a UG-specific binary graphics format for which postprocessing tools are available. The `ps`- and `psbw`-outputdevices write a file in postscript-format, the latter converts all colors to greyscale. Finally the `ppm`-outputdevice writes in the portable-pixmap format.

The UG-command to open a UG-window is

```
openwindow <x> <y> <width> <height>
  [$n <window name>]
  [$d <outputdevice>]
  [$r [0|1]].
```

The parameters `<x>`, `<y>` fix position the lower left corner of the window. For the `screen` outputdevice this is a position on the screen's pixelspace having the origin on the lower left corner of the screen and having increasing values in the right and upward direction. `<width>` and `<height>` fixes the size of the window. After creation UG-window is current. UG-windows are closed using the `UG`-command

```
closewindow [$n <window name> | $a];
```

This command closes the current UG-window (no option), the named UG-window (n-option) or all UG-windows (a-option). A UG-window is made current using the command

```
setcurrwindow <window name>.
```

On the well-defined pixel space of an UG-window an arbitrary number of pictures can be opened by

```
openpicture [$n <picture name>] [$s <x> <y> <width> <height>]
[$w <window name>].
```

If specified it is named `<picture name>` otherwise a defaultname is chosen. The s-option specifies size and position relative to the pixel space of the UG-window (`<x>`, `<y>` is the lower left corner of the picture). Pictures do not have to lie (entirely) within the UG-window and are allowed to overlap. If the s-option is not specified the picture takes the size of the UG-window. The picture is opened in the current UG-window or in `<window name>` if specified. After creation a picture is current. Pictures can be close via the command

```
closepicture [$a | {$w <window name> {<picture name> | $a}}].
```

It closes the current picture (no options), all pictures of the current UG-window (only a-option specified), all pictures of `<window name>` (a-option and w-option specified) or a specified picture in a specified UG-window (both names specified). A picture can be made current by

```
setcurrpicture <picture name> [$w <window name>].
```

This command can access pictures in the current UG-window or in the specified one. As a sample we consider the sequence of commands

```
openwindow 0 0 300 300 $n w1 $d screen;
  openpicture $s 10 10 50 50 $n p1;
  openpicture $s 50 50 70 70 $n p2;

openwindow 0 0 800 800 $n w2 $d meta;
  openpicture $n m1;

openwindow 0 0 800 800 $n w3 $d ps;
  openpicture $s 0 0 800 400 $n ps1;
  openpicture $s 0 400 800 400 $n ps2;

openwindow 0 0 1000 1000 $n w4 $d psbw;
  openpicture $s 0 0 1000 500 $n psbw1;
  openpicture $s 0 0 500 1000 $n psbw2;
  openpicture $s 100 100 500 500 $n psbw3;

openwindow 0 0 200 200 $n w5 $d ppm;
  openpicture $n ppml;
```

creating 9 pictures on 5 UG-windows, each on a different outputdevice. A list of the currently open pictures is given by the command

```
> wplist
UgWindow  Device  Picture  VO_Status  PlotObjType  PO_Status  Multigrid
-----  -
w1        screen
w1        screen  p1       NOT_INIT   ---         NOT_INIT   ---
w1        screen  p2       NOT_INIT   ---         NOT_INIT   ---
w2        meta
w2        meta   m1       NOT_INIT   ---         NOT_INIT   ---
w3        ps
w3        ps     ps1     NOT_INIT   ---         NOT_INIT   ---
w3        ps     ps2     NOT_INIT   ---         NOT_INIT   ---
w4        psbw
w4        psbw  psbw1   NOT_INIT   ---         NOT_INIT   ---
w4        psbw  psbw2   NOT_INIT   ---         NOT_INIT   ---
w4        psbw  psbw3   NOT_INIT   ---         NOT_INIT   ---
# w5      ppm
* w5      ppm    ppm1    NOT_INIT   ---         NOT_INIT   ---
```

giving the names of the UG-windows, used outputdevices and pictures. The stati of the viewobject and plotobject etc. is shown. The current UG-window is marked by a hash, the current picture by an asterisk. Remember that the the methods for the viewedobject and the plotobject refer to the objects inherited by the current picture.

Other commands related to UG-windows and pictures are given with a short description in table 7.3

Table 7.3: Commands related to UG-windows and pictures

COMMAND	OPTIONS, DESCRIPTION
screensize	; <p>Prints the size of the screen. The values are stored in the variables :screensize:width and :screensize:height.</p>
drawtext	<x> <y> <text>; <p>Draws the text &lt;text&gt; at the pixelcoordinates &lt;x&gt; &lt;y&gt;.</p>
clearpicture	; <p>Erases the current picture.</p>
picframe	0 1; <p>Switches Off/On display of the frame for every picture. The border of a picture is called frame.</p>
picwin	; <p>Create a new UG-window and move the current picture there.</p>

## 7.2 High-level methods for pictures

In this section we give an overview over the high-level methods available for pictures. The methods available depend on the class of the inherited plotobject. The method `draw` is available for all plotobjects. The method `findrange` is available to all plotobjects representing (at least in parts) a numerical solution. It evaluates the range of numerical values the numerical solution takes. The rest of the methods are interactive methods which can be used only by an interactive device as explained in section 7.1.

Before we explain the methods we briefly introduce the underlying concept. All high-level methods are implemented in the file `wop.c`. The main functionality is exported by the c-function

```
INT WorkOnPicture (PICTURE *thePicture, WORK *theWork).
```

The first argument refers to a (fully initialized) picture as described in the previous section. Depending on the initialization a picture can represent a view to any of the plotobjects available in UG. The data structure `WORK` defined in `wop.h` can be initialized to be any kind of method available for pictures. In that way the function `WorkOnPicture` can be considered a matrix. Every matrix entry represents a method for a (view to a) plotobject. These matrix entries are mapped to UG-commands or interactive facilities of UG. In table 7.4 the available methods are marked by an asterisk. Only the first two methods (`draw` and `findrange`) are non-interactive and are accessible via an UG-command. The latter ones are interactive and accessible only via graphical tools.

Table 7.4: High-level methods for plotobjects

	draw	find-range	select-node	select-vector	select-element	mark-element	insert-node	insert-bndnode	movenode
Matrix	*	*							
Line(2/3d)	*	*							
Grid(2d)	*		*		*	*	*	*	*
Grid(3d)	*				*				
HGrid(2d)	*								
EScalar(2/3d)	*	*							
EVector(2/3d)	*	*							
VecMat(2d)	*			*					

The method `draw` is mapped to the UG-command

```
> plot [%o 0|1|2] [%b [bullet value]].
```

The command displays the plotobject using the associated view of the current picture. The graphics uses hidden lines/hidden surfaces requiring an ordering of the elements (if option `$b` is not specified). In that case the option `$o` offers the choice of different ordering strategies for the coarse grid elements. In case of 0 (default) an extended datastructure resulting in an ordering of optimal complexity is used. `$o 1` results in an algorithm from Newell & Sancha whereas `$o 2` uses an modified insertion sort of horrible complexity. The latter one we kept for historical reasons. If the `$b` option is specified, a z-buffer is used (optimal complex).

The method `findrange` is mapped to the `ugcommand`

```
> findrange [$s] [$p] [$z <factor>]
```

The command finds the range of the values associated with the plotobject. In the case of the plotobjects `Matrix`, `Line` and `EScalar` the minimal and maximal values are detected. The output looks like

```
> findrange
FR_min = 1.2
FR_max = 0.8.
```

In the case of the plotobject `EVector` only the maximal value is detected. The minimal is set to 0. The values are stored in the variables

```
:findrange:min
:findrange:max.
```

If the option `$p` is specified, the values are used to specify the range of the plotobject. The option `$s` symetrizes the result. The above example again serves to explain the behaviour

```
> findrange
FR_min = 1.2
FR_max = -1.2 .
```

The option `$z <value>` zooms the resulting range w.r.t. its midpoint. For the example above we get

```
> findrange $z 0.4
FR_min = 1.08
FR_max = 0.92,
> findrange $s $z 0.4
FR_min = 0.48
FR_max = -0.48.
```

The rest of the methods are interactive. At the bottom of each UG-window opened on the screen a tool-chest offers the interactive tools. Entering the mouse in the appropriate region, the functionality of the corresponding tool is shown.

## 7.3 Plotobjects

The plotobjects fall into three cathegories. The ones independent of the application dimension, the ones available for application dimension 2 and the ones for the application dimension 3. Table 7.5 show the plotobjects available and gives a reference to the table explaining its usage.

Table 7.5: The plotobjects available in UG depending on the application dimension

PLOBJECT	INDEPENDENT	2D	3D
Matrix	Table 7.6	–	–
Line	–	Table 7.7	Table 7.7
Grid	–	Table 7.8	Table 7.9
HGrid	–	Table 7.10	–
Escalar	–	Table 7.11	Table 7.12
EVector	–	Table 7.13	Table 7.14
VecMat	–	Table 7.15	–



Table 7.6: The Matrix plotobject - options for setplotobject

DESCRIPTION	The PO represents a matrix corresponding to the sparse matrix in the UG data structure.
PLOTOBJECT-DIMENSION: 2, APPLICATION-DIMENSION: arbitrary.	
OPTIONS	
\$f <from> \$t <to>	Specifies the color of matrix entries, values smaller than <from> are mapped to UG-blue, values higher than <to> are mapped to UG-red.
Default:	-4 4.
\$T <tresh>	If the absolute value is smaller than <tresh>, the value is omitted, a space indicates its existence.
Default:	0.
\$l [0 1]	Use the natural logarithm of absolute values to determine range and color of matrix-entries, if 1.
Default:	0.
\$BV [0 1] <dash> <space>	Display UG-blockvectors by dashed lines. The dashes have the length of <dash> pixels, the space inbetween the length of <space> pixels.
Default:	0 0 0.
\$rel [0 1]	Scale each row of the matrix by the inverse of its diagonal, if 1.
Default:	0.
\$C [0 1]	Include the regular matrix entries,if 1.
Default:	1.
\$E [0 1]	Include the extra matrix entries, if 1.
Default:	0.
\$M <A>	Include the matrix corresponding to the symbol <A>.
Default:	none.
\$e <MatrixEvalProc>	Use matrix evaluation callback function to specify the value of the matrix.
Default:	none.

Table 7.7: The Line plotobject - options for setplotobject

DESCRIPTION	The PO represents the graph of a scalar field evaluated on a line within the domain.
PLOTOBJECT-DIMENSION: 2, APPLICATION-DIMENSION: arbitrary.	
OPTIONS	
2d: \$l < $l_x$ > < $l_y$ >,      3d: \$l < $l_x$ > < $l_y$ > < $l_z$ >	Position of the first point of the line, mapped on the left point of the graph.
Default:	0 0 (0).
2d: \$r < $r_x$ > < $r_y$ >,      3d: \$l < $r_x$ > < $r_y$ > < $r_z$ >	Position of the second point of the line, mapped on the right point of the graph.
Default:	0 0 (0).
\$c < color >	Sets the color of the graph, 0 means UG-blue, 1 means UG-red.
Default:	0.
\$Ly [0 1]	If 1 take the logarithm (base 10) of the absolute value of the scalar field.
Default:	0.
\$d < depth >	Depth of the recursive halvening of each line intersection with an element.
Default:	0.
\$e < eval proc >	Name of the element evaluation procedure providing the scalar field.
Default:	none.
\$s < symbol >	Alternative to \$e: specification of a scalar vector symbol.
Default:	none.
\$G < filename >	Gnuplot-option: writes graphical information to <filename> in gnuplot-format if set.
Default:	none.

Table 7.8: The Grid plotobject - options for setplotobject

DESCRIPTION	The PO represents a 2d multigrid hierarchy.
PLOTOBJECT-DIMENSION: 2, APPLICATION-DIMENSION: 2.	
OPTIONS	
\$w [c i r a]	Including all elements on TOPLEVEL (c), red and green elements on TOPLEVEL (i), red elements on TOPLEVEL (r) or all elements (a).
Default:	a.
\$s <shrink>	Shrinks each element by a factor of <shrink>.
Default:	1.
\$c [0 1 2]	Color of the elements: No color (0), color according to the regularity (1) or color according to the MG_NPROPERTY of the multigrid.
Default:	1.
\$b [0 1]	Sets boundary to a distinct color if 1.
Default:	1.
\$r [0 1]	Shows refinement marks on the grid if 1.
Default:	0.
\$e [0 1]	Shows element id's if 1.
Default:	0.
\$S [0 1]	Shows subdomain of the elements if 1.
Default:	0.
\$n [0 1]	Shows node id's if 1.
Default:	0.
\$m [0 1]	Includes marks for nodes.
Default:	0.

Table 7.9: The Grid plotobject - options for setplotobject

DESCRIPTION	The PO represents a 3d multigrid hierarchy. Allows the specification of a cut plane.
PLOTOBJECT-DIMENSION: 3, APPLICATION-DIMENSION: 3.	
OPTIONS	
\$w [c i r a]	Including all elements on TOPLEVEL (c), red and green elements on TOPLEVEL (i), red elements on TOPLEVEL (r) or all elements (a).
Default:	a.
\$s <shrink>	Shrinks each element by a factor of <shrink>.
Default:	1.
\$c [0 1 2]	Color of the elements: No color (0), color according to the regularity (1) or color according to the MG_NPROPERTY of the multigrid.
Default:	1.

Table 7.10: The HGrid plotobject - options for setplotobject

DESCRIPTION	The PO represents a 2d multigrid hierarchy in 3d view to all levels.
PLOTOBJECT-DIMENSION: 3, APPLICATION-DIMENSION: 2.	
OPTIONS	
\$w [c i r a]	Including all elements on TOPLEVEL (c), red and green elements on TOPLEVEL (i), red elements on TOPLEVEL (r) or all elements (a).
Default:	a.
\$s <shrink>	Shrinks each element by a factor of <shrink>.
Default:	1.
\$c [0 1 2]	Color of the elements: No color (0), color according to the regularity (1) or color according to the MG_NPROPERTY of the multigrid.
Default:	1.
\$e [0 1]	Shows element id's if 1.
Default:	0.
\$S [0 1]	Shows subdomain of the elements if 1.
Default:	0.
\$z <z max>	The maximum value of z-component to which the CURRENTLEVEL of the multigrid is scaled.
Default:	Radius of the BVP.

Table 7.11: The `EScalar` plotobject - options for `setplotobject`

DESCRIPTION	The PO represents the scalar field on a 2d multigrid hierarchy.
PLOTOBJECT-DIMENSION: 2, APPLICATION-DIMENSION: 2.	
OPTIONS	
<code>\$f &lt;from&gt; \$t &lt;to&gt;</code>	Range. Specifies the coloring of the scalar field, values smaller than <code>&lt;from&gt;</code> are mapped to UG-blue, values higher than <code>&lt;to&gt;</code> are mapped to UG-red.
Default:	0 1.
<code>\$m [COLOR CONTOUR_EQ]</code>	Mode of the coloring. <code>COLOR</code> takes one color for every part of an element. The parts result from a <code>&lt;depth&gt;</code> -times recursive subdivision of one element into four pieces (see <code>\$d</code> option). <code>CONTOUR_EQ</code> species contourlines. Additionally the number (option <code>\$n</code> ) or values (option <code>\$v</code> ) has to be specified. The intersection of each contour with an element is subdivided recursively <code>&lt;depth&gt;</code> -times (see <code>\$d</code> option) into two part. The coloring is according to the value and range.
Default:	COLOR.
<code>\$d &lt;depth&gt;</code>	Depth.
Default:	0.
<code>\$n &lt;nb&gt;</code>	Number of contour lines if <code>\$m CONTOUR_EQ</code> is specified. The values of the contour lines are equally spaced, starting from the lower bound of the range (option <code>\$f</code> ) and ending at the upper bound (option <code>\$t</code> ). At least two have to be specified.
Default:	10.
<code>\$v &lt;v<sub>1</sub>&gt;, ..., &lt;v<sub>n</sub>&gt;, 1 ≤ n ≤ 10</code>	Alternative way to specify the values of the contour lines. Has higher priority than option <code>\$n</code> .
Default:	none.
<code>\$g [0 1]</code>	Includes the grid if specified.
Default:	0.
<code>\$e &lt;eval proc&gt;</code>	Name of the element evaluation procedure providing the scalar field.
Default:	none.
<code>\$s &lt;symbol&gt;</code>	Alternative to <code>\$e</code> : specification of a scalar vector symbol.
Default:	none.

Table 7.12: The EScalar plotobject - options for setplotobject

DESCRIPTION	The PO represents the scalar field on a 2d cut plane intersecting a 3d multigrid hierarchy. The part of the grid behind the cut is included. The intersection of an element with the cut plane is referred to as a <i>cutted element</i> .
PLOTOBJECT-DIMENSION: 2, APPLICATION-DIMENSION: 3.	
OPTIONS	
\$f <from> \$t <to>	Range. Specifies the coloring of the scalar field, values smaller than <from> are mapped to UG-blue, values higher than <to> are mapped to UG-red.
Default:	0 1.
\$m [COLOR CONTOUR_EQ]	Mode of the coloring. COLOR takes one color for every part of a cutted element. The parts result from a <depth>-times recursive subdivision of one cutted element into four pieces (see \$d option). CONTOUR_EQ species contourlines. Additionally the number (option \$n) or values (option \$v) has to be specified. The intersection of each contour with an cutted element is subdivided recursively <depth>-times (see \$d option) into two part. The coloring is according to the value and range.
Default:	COLOR.
\$d <depth>	Depth.
Default:	0.
\$n <nb>	Number of contour lines if \$m CONTOUR_EQ is specified. The values of the contour lines are equally spaced, starting from the lower bound of the range (option \$f) and ending at the upper bound (option \$t). At least two have to be specified.
Default:	10.
\$e <eval proc>	Name of the element evaluation procedure providing the scalar field.
Default:	none.
\$s <symbol>	Alternative to \$e: specification of a scalar vector symbol.
Default:	none.
\$a <ambient>	Control of the ambient shading of the grid behind the cut plane. 1 leads to no shading, 0 to full shading.
Default:	1.

Table 7.13: The EVector plotobject - options for setplotobject

DESCRIPTION	The PO represents 2d vector field on a 2d hierarchie of grids. The field is evaluated on a quadratic raster. The values of the field are displayed by arrows having the direction and length of the vector.
PLOTOBJECT-DIMENSION: 2, APPLICATION-DIMENSION: 2.	
OPTIONS	
\$t <to> Default:	Upper bound of the range. Lower bound is 0. 1.
\$g [0 1] Default:	Include grid if 1. 0.
\$r <raster> Default:	Raster size in pixels. 20.
\$c [0 1]	If set: Cut vector if it exceeds <length>-times the <raster>. In the case of cutting the vector is displayed red, otherwise black.
\$l <length> Default:	see option \$c. 1.
\$e <eval proc> Default:	Name of the element evaluation procedure providing the vector field. none.
\$s <symbol>	Alternative to \$e: specification of a vector-valued vector symbol.



Table 7.14: The EVector plotobject - options for setplotobject

DESCRIPTION	The PO represents the 3d vector field on a 2d cut plane intersecting a 3d multigrid hierarchy. The field is evaluated on a quadratic raster on the cut plane. It is displayed using arrows having the direction and length of the vector. The color depends on the angle of the vector with the cut plane. A vector parallel to the cut plane is displayed in UG-green, a vector pointing out in UG-red and a vector pointing into the plane in UG-blue. The vectors are displayed as 3d objects or its projection onto the cut plane (see option \$p below). The part of the grid behind the cut is included.
PLOTOBJECT-DIMENSION: 3, APPLICATION-DIMENSION: 3.	
OPTIONS	
\$t <to>	Upper bound of the range. Lower bound is 0.
Default:	1.
\$r <raster>	Raster size in pixels.
Default:	20.
\$c [0 1]	If set: Cut vector if it exceeds <length>-times the <raster>. In the case of cutting the vector is displayed black.
\$l <length>	see option \$c.
Default:	0.9.
\$p [0 1]	Project the vectors onto the cut plane if 1.
Default:	1.
\$a <ambient>	Control of the ambient shading of the grid behind the cut plane. 1 leads to no shading, 0 to full shading.
Default:	1.
\$e <eval proc>	Name of the element evaluation procedure providing the vector field.
Default:	none.
\$s <symbol>	Alternative to \$e: specification of a vector-valued vector symbol.

Table 7.15: The VecMat plotobject - options for setplotobject

DESCRIPTION	The PO represents the graph structure of the VECTORS and MATRIXs of UG.
PLOT OBJECT-DIMENSION: 2, APPLICATION-DIMENSION: 2.	
OPTIONS	
\$m [0 1] Default:	Place a square marker at the position of each vector. 0.
\$c [0 1] Default:	Include connections if 1. 1.
\$C [0 1] Default:	Shows the 1d ordering of the vectors. Specification of 1 resets option \$c. 0.
\$e [0 1] Default:	Include extra connections if 1. 0.
\$i [0 1] Default:	Include the index of the vectors. 0.
\$p [0 1] Default:	Shows part information of the vectors. Has higher priority than option \$i. 0.
\$d [0 1] Default:	Shows dependencies of the MATRIXs, indicated by an arrow. Non-dependent connections are not displayed. 0.
\$o [0 1 2 3] Default:	Display ordering w.r.t. UG-blockvectors. Consult your local UG-specialist for usage. 0.
\$b [0 1] Default:	Show boundary if 1. 1.
\$f [0 1] Default:	Consider only VCFLAGged vectors. 0.
\$V <vec sym> Default:	Vector symbol used to evaluate data for vectors in selection list. Displays vector entries for selected vectors. none.
\$M <mat sym> Default:	Matrix symbol used to evaluate data for vectors in selection list. Displays matrix entries for all connections associated with the selected vectors. none.

# Bibliography

- Bastian, P. (1998). Load balancing for adaptive multigrid methods. *SIAM J. Sci. Stat. Comput.* 19(4), 1303–1321.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns*. Addison–Wesley.

