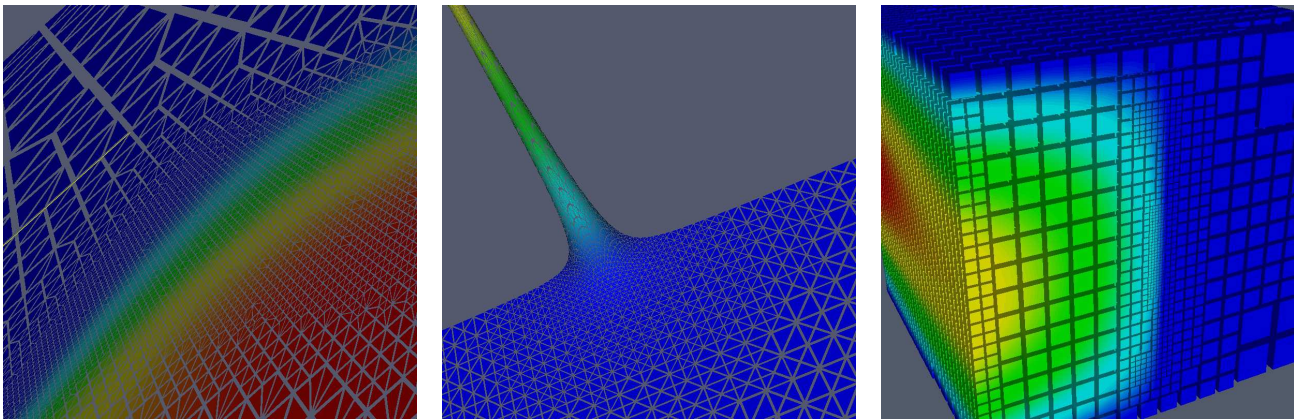


# The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO

Peter Bastian\*      Markus Blatt\*      Andreas Dedner†  
Christian Engwer\*      Robert Klöfkorn†      Martin Nolte†  
Mario Ohlberger¶      Oliver Sander‡

Version 2.3beta2, February 10, 2014



\*Abteilung ‘Simulation großer Systeme’, Universität Stuttgart,  
Universitätsstr. 38, D-70569 Stuttgart, Germany

†Abteilung für Angewandte Mathematik, Universität Freiburg,  
Hermann-Herder-Str. 10, D-79104 Freiburg, Germany

¶Institut für Numerische und Angewandte Mathematik, Universität Münster,  
Einsteinstr. 62, D-48149 Münster, Germany

‡Institut für Mathematik II,  
Freie Universität Berlin, Arnimallee 6, D-14195 Berlin, Germany

<http://www.dune-project.org/>



This document gives an introduction to the Distributed and Unified Numerics Environment (**DUNE**). **DUNE** is a template library for the numerical solution of partial differential equations. It is based on the following principles: i) Separation of data structures and algorithms by abstract interfaces, ii) Efficient implementation of these interfaces using generic programming techniques (templates) in C++ and iii) Reuse of existing finite element packages with a large body of functionality. This introduction covers only the abstract grid interface of **DUNE** which is currently the most developed part. However, part of **DUNE** are also the Iterative Solver Template Library (ISTL, providing a large variety of solvers for sparse linear systems) and a flexible class hierarchy for finite element methods. These will be described in subsequent documents. Now have fun!

Thanks to Martin Drohmann for adapting this howto to version 1.2 of the DUNE grid interface.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	What is <b>DUNE</b> anyway?	6
1.2	Download	7
1.3	Installation	7
1.4	Code documentation	8
1.5	Licence	8
<b>2</b>	<b>Getting started</b>	<b>9</b>
2.1	Creating your first grid	9
2.2	Traversing a grid — A first look at the grid interface	11
<b>3</b>	<b>The <b>DUNE</b> grid interface</b>	<b>16</b>
3.1	Grid definition	16
3.2	Concepts	18
3.2.1	Common types	18
3.2.2	Concepts of the <b>DUNE</b> grid interface	19
3.3	Propagation of type information	20
<b>4</b>	<b>Constructing grid objects</b>	<b>21</b>
4.1	Creating Structured Grids	21
4.2	Reading Unstructured Grids from Files	22
4.3	The <b>DUNE</b> Grid Format (DGF)	23
4.4	The Grid Factory	23
4.5	Attaching Data to a New Grid	25
4.6	Example: The <code>UnitCube</code> class	26
<b>5</b>	<b>Quadrature rules</b>	<b>32</b>
5.1	Numerical integration	32
5.2	Functors	33
5.3	Integration over a single element	33
5.4	Integration with global error estimation	35
<b>6</b>	<b>Attaching user data to a grid</b>	<b>38</b>
6.1	Mappers	38
6.2	Visualization of discrete functions	39
6.3	Cell centered finite volumes	44
6.3.1	Numerical Scheme	45
6.3.2	Implementation	46
6.4	A FEM example: The Poisson equation	54

## Contents

6.4.1	Implementation . . . . .	56
<b>7</b>	<b>Adaptivity</b>	<b>65</b>
7.1	Adaptive integration . . . . .	65
7.1.1	Adaptive multigrid integration . . . . .	65
7.1.2	Implementation of the algorithm . . . . .	66
7.2	Adaptive cell centered finite volumes . . . . .	69
<b>8</b>	<b>Parallelism</b>	<b>77</b>
8.1	<b>DUNE</b> Data Decomposition Model . . . . .	77
8.2	Communication Interfaces . . . . .	79
8.3	Parallel finite volume scheme . . . . .	81

# 1 Introduction

## 1.1 What is DUNE anyway?

**DUNE** is a software framework for the numerical solution of partial differential equations with grid-based methods. It is based on the following main principles:

- *Separation of data structures and algorithms by abstract interfaces.* This provides more functionality with less code and also ensures maintainability and extendability of the framework.
- *Efficient implementation of these interfaces using generic programming techniques.* Static polymorphism allows the compiler to do more optimizations, in particular function inlining, which in turn allows the interface to have very small functions (implemented by one or few machine instructions) without a severe performance penalty. In essence the algorithms are parametrized with a particular data structure and the interface is removed at compile time. Thus the resulting code is as efficient as if it would have been written for the special case.
- *Reuse of existing finite element packages with a large body of functionality.* In particular the finite element codes UG, [2], Alberta, [8], and ALU3d, [3], have been adapted to the **DUNE** framework. Thus, parallel and adaptive meshes with multiple element types and refinement rules are available. All these packages can be linked together in one executable.

The framework consists of a number of modules which are downloadable as separate packages. The current core modules are:

- **dune-common** contains the basic classes used by all **DUNE**-modules. It provides some infrastructural classes for debugging and exception handling as well as a library to handle dense matrices and vectors.
- **dune-grid** is the most mature module and is covered in this document. It defines nonconforming, hierarchically nested, multi-element-type, parallel grids in arbitrary space dimensions. Graphical output with several packages is available, e. g. file output to IBM data explorer and VTK (parallel XML format for unstructured grids). The graphics package Grape, [5] has been integrated in interactive mode.
- **dune-istl** – *Iterative Solver Template Library*. Provides generic sparse matrix/vector classes and a variety of solvers based on these classes. A special feature is the use of templates to exploit the recursive block structure of finite element matrices at compile time. Available solvers include Krylov methods, (block-) incomplete decompositions and aggregation-based algebraic multigrid.
- **dune-localfunctions** – *Library of local base functions*. Provides classes for base functions on reference elements from which global discrete function spaces can be constructed.

Before starting to work with **DUNE** you might want to update your knowledge about C++ and templates in particular. For that you should have the bible, [9], at your desk. A good introduction, besides its age, is still the book by Barton and Nackman, [1]. The definitive guide to template programming is [10]. A very useful compilation of template programming tricks with application to scientific computing is given in [11] (if you can't find it on the web, contact us).

### 1.2 Download

The source code of the **DUNE** framework can be downloaded from the web page. To get started, it is easiest to download the latest stable version of the tarballs of `dune-common`, `dune-grid` and `dune-grid-howto`. These are available on the **DUNE** download page:

`http://www.dune-project.org/download.html`

Alternatively, you can download the latest development version via anonymous SVN. For further information, please see the web page.

### 1.3 Installation

The official installation instructions are available on the web page

`http://www.dune-project.org/doc/installation-notes.html`

Obviously, we do not want to copy all this information because it might get outdated and inconsistent then. To make this document self-contained, we describe only how to install **DUNE** from the tarballs. If you prefer to use the version from SVN, see the web page for further information. Moreover, we assume that you use a UNIX system. If you have the Redmond system then ask them how to install it.

In order to build the **DUNE** framework, you need a standards compliant C++ compiler. We tested compiling with GNU `g++` in version  $\geq 4.1$ . Recent versions of Intel `icc` and `clang` ( $\geq 3.0$ ) should work, too.

Now extract the tarballs of `dune-common`, `dune-grid` and `dune-grid-howto` into a common directory, say `dune-home`. Change to this directory and call

```
> dune-common-1.0/bin/dunecontrol all
```

Replace “1.0” by the actual version number of the package you downloaded if necessary. This should configure and build all **DUNE** modules in `dune-home` with a basic configuration.

For many of the examples in this howto you need adaptive grids or the parallel features of **DUNE**. To use adaptive grids, you need to install one of the external grid packages which **DUNE** provides interfaces for, for instance Alberta, UG and ALUGrid.

- Alberta – <http://www.alberta-fem.de/>
- UG – <http://sit.iwr.uni-heidelberg.de/~ug/>
- ALUGrid – <http://www.mathematik.uni-freiburg.de/IAM/Research/alugrid/>

To use the parallel code of **DUNE**, you need an implementation of the Message Passing Interface (MPI), for example MPICH or LAM. For the **DUNE** build system to find these libraries, the `configure` scripts of the particular **DUNE** modules must be passed the locations of the respective installations. The `dunecontrol` script facilitates to pass options to the `configure` via a configuration file. Such a configuration file might look like this:

```
CONFIGURE_FLAGS="--with-alugrid=/path/to/alugrid/_"\
"--with-alberta=/path/to/alberta_\
"--with-ug=/path/to/ug_--enable-parallel"
MAKE_FLAGS="-j_2"
```

If this is saved under the name `dunecontrol.opts`, you can tell `dunecontrol` to consider the file by calling

```
> dune-common-1.0/bin/dunecontrol --opts=dunecontrol.opts all
```

For information on how to build and configure the respective grids, please see the **DUNE** web page.

### 1.4 Code documentation

Documentation of the files and classes in **DUNE** is provided in code and can be extracted using the `doxygen`<sup>1</sup> software available elsewhere. The code documentation can either be built locally on your machine (in html and other formats, e. g.  $\text{\LaTeX}$ ) or its latest version is available at

<http://www.dune-project.org/doc/>

### 1.5 Licence

The **DUNE** library and headers are licensed under version 2 of the GNU General Public License<sup>2</sup>, with a special exception for linking and compiling against **DUNE**, the so-called “runtime exception.” The license is intended to be similar to the GNU Lesser General Public License, which by itself isn’t suitable for a C++ template library.

The exact wording of the exception reads as follows:

As a special exception, you may use the **DUNE** source files as part of a software library or application without restriction. Specifically, if other files instantiate templates or use macros or inline functions from one or more of the **DUNE** source files, or you compile one or more of the **DUNE** source files and link them with other files to produce an executable, this does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

---

<sup>1</sup><http://www.doxygen.org/>

<sup>2</sup><http://www.gnu.org/licenses/gpl-2.0.html>



## 2 Getting started

In this section we will take a quick tour through the abstract grid interface provided by **DUNE**. This should give you an overview of the different classes before we go into the details.

### 2.1 Creating your first grid

Let us start with a replacement of the famous “hello world” program given below.

#### Listing 1 (File dune-grid-howto/gettingstarted.cc)

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 // $Id$
4
5 // Dune includes
6 #include "config.h" // file constructed by ./configure script
7 #include <dune/grid/sgrid.hh> // load sgrid definition
8 #include <dune/grid/common/gridinfo.hh> // definition of gridinfo
9 #include <dune/common/parallel/mpihelper.hh> // include mpi helper class
10
11
12 int main(int argc, char **argv)
13 {
14     // initialize MPI, finalize is done automatically on exit
15     Dune::MPIHelper::instance(argc, argv);
16
17     // start try/catch block to get error messages from dune
18     try{
19         // make a grid
20         const int dim=3;
21         typedef Dune::SGrid<dim,dim> GridType;
22         Dune::FieldVector<int,dim> N(3);
23         Dune::FieldVector<GridType::ctype,dim> L(-1.0);
24         Dune::FieldVector<GridType::ctype,dim> H(1.0);
25         GridType grid(N,L,H);
26
27         // print some information about the grid
28         Dune::gridinfo(grid);
29     }
30     catch (std::exception & e) {
31         std::cout << "STL_ERROR:" << e.what() << std::endl;
32         return 1;
33     }
34     catch (Dune::Exception & e) {
35         std::cout << "DUNE_ERROR:" << e.what() << std::endl;
36         return 1;
37     }
38     catch (...) {
39         std::cout << "Unknown_ERROR" << std::endl;
40         return 1;
41     }
42
43     // done
```

## 2 Getting started

```
44  return 0;
45 }
```

This program is quite simple. It starts with some includes in lines 6-8. The file `config.h` has been produced by the `configure` script in the application's build system. It contains the current configuration and can be used to compile different versions of your code depending on the configuration selected. It is important that this file is included before any other **DUNE** header files. The next file `dune/grid/sgrid.hh` includes the headers for the **SGrid** class which provides a special implementation of the **DUNE** grid interface with a structured mesh of arbitrary dimension. Then `dune/grid/common/gridinfo.hh` loads the headers of some functions which print useful information about a grid.

Since the dimension will be used as a template parameter in many places below we define it as a constant in line number 20. The **SGrid** class template takes two template parameters which are the dimension of the grid and the dimension of the space where the grid is embedded in (its world dimension). If the world dimension is strictly greater than the grid dimension the surplus coordinates of each grid vertex are set to zero. For ease of writing we define in line 21 the type **GridType** using the selected value for the dimension. All identifiers of the **DUNE** framework are within the **Dune** namespace.

Lines 22-24 prepare the arguments for the construction of an **SGrid** object. These arguments use the class template `FieldVector<T,n>` which is a vector with `n` components of type `T`. You can either assign the same value to all components in the constructor (as is done here) or you could use `operator[]` to assign values to individual components. The variable `N` defines the number of cells or elements to be used in the respective dimension of the grid. `L` defines the coordinates of the lower left corner of the cube and `H` defines the coordinates of the upper right corner of the cube. Finally in line 25 we are now able to instantiate the **SGrid** object.

The only thing we do with the grid in this little example is printing some information about it. After successfully running the executable `gettingstarted` you should see an output like this:

### Listing 2 (Output of `gettingstarted`)

```
=> SGrid(dim=3,dimworld=3)
level 0 codim[0]=27 codim[1]=108 codim[2]=144 codim[3]=64
leaf    codim[0]=27 codim[1]=108 codim[2]=144 codim[3]=64
leaf dim=3 geomTypes=((cube,3)[0]=27,(cube,2)[1]=108,(cube,1)[2]=144,(cube,0)[3]=64)
```

The first line tells you that you are looking at an **SGrid** object of the given dimensions. The **DUNE** grid interface supports unstructured, locally refined, logically nested grids. The coarsest grid is called level-0-grid or macro grid. Elements can be individually refined into a number of smaller elements. Each element of the macro grid and all its descendants obtained from refinement form a tree structure. All elements at depth  $n$  of a refinement tree form the level- $n$ -grid. All elements that are leaves of a refinement tree together form the so-called leaf grid. The second line of the output tells us that this grid object consists only of a single level (level 0) while the next line tells us that that level 0 coincides also with the leaf grid in this case. Each line reports about the number of grid entities which make up the grid. We see that there are 27 elements (codimension 0), 108 faces (codimension 1), 144 edges (codimension 2) and 64 vertices (codimension 3) in the grid. The last line reports on the different types of entities making up the grid. In this case all entities are of type "cube".

**Exercise 2.1** Try to play around with different grid sizes by assigning different values to the `N` parameter. You can also change the dimension of the grid by varying `dim`. Don't be modest. Also try dimensions 4 and 5!

**Exercise 2.2** The static methods `Dune::gridlevellist` and `Dune::gridleaflist` produce a very detailed output of the grid's elements on a specified grid level. Change the code and print out this information for the leaf grid or a grid on lower level. Try to understand the output.

## 2.2 Traversing a grid — A first look at the grid interface

After looking at very first simple example we are now ready to go on to a more complicated one. Here it is:

### Listing 3 (File `dune-grid-howto/traversal.cc`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 // $Id$
4
5 // C/C++ includes
6 #include <iostream>                // for standard I/O
7
8 // Dune includes
9 #include "config.h"                // file constructed by ./configure script
10 #include <dune/grid/sgrid.hh>      // load sgrid definition
11 #include <dune/common/parallel/mpihelper.hh> // include mpi helper class
12
13
14 // example for a generic algorithm that traverses
15 // the entities of a given mesh in various ways
16 template<class G>
17 void traversal (G& grid)
18 {
19     // first we extract the dimensions of the grid
20     const int dim = G::dimension;
21
22     // type used for coordinates in the grid
23     // such a type is exported by every grid implementation
24     typedef typename G::ctype ct;
25
26     // Leaf Traversal
27     std::cout << "***_Traverse_codim_0_leaves" << std::endl;
28
29     // type of the GridView used for traversal
30     // every grid exports a LeafGridView and a LevelGridView
31     typedef typename G :: LeafGridView LeafGridView;
32
33     // get the instance of the LeafGridView
34     LeafGridView leafView = grid.leafGridView();
35
36     // Get the iterator type
37     // Note the use of the typename and template keywords
38     typedef typename LeafGridView::template Codim<0>::Iterator ElementLeafIterator;
39
40     // iterate through all entities of codim 0 at the leaves
41     int count = 0;
42     for (ElementLeafIterator it = leafView.template begin<0>();
43          it!=leafView.template end<0>(); ++it)

```

## 2 Getting started

```
44 {
45     Dune::GeometryType gt = it->type();
46     std::cout << "visiting_Leaf_" << gt
47               << "_with_first_vertex_at_" << it->geometry().corner(0)
48               << std::endl;
49     count++;
50 }
51
52 std::cout << "there_are/is_" << count << "_leaf_element(s)" << std::endl;
53
54 // Leafwise traversal of codim dim
55 std::cout << std::endl;
56 std::cout << "***_Traverse_codim_" << dim << "_leaves" << std::endl;
57
58 // Get the iterator type
59 // Note the use of the typename and template keywords
60 typedef typename LeafGridView::template Codim<dim>
61 :: Iterator VertexLeafIterator;
62
63 // iterate through all entities of codim 0 on the given level
64 count = 0;
65 for (VertexLeafIterator it = leafView.template begin<dim>();
66      it!=leafView.template end<dim>(); ++it)
67 {
68     Dune::GeometryType gt = it->type();
69     std::cout << "visiting_" << gt
70               << "_at_" << it->geometry().corner(0)
71               << std::endl;
72     count++;
73 }
74 std::cout << "there_are/is_" << count << "_leaf_vertices(s)"
75           << std::endl;
76
77 // Levelwise traversal of codim 0
78 std::cout << std::endl;
79 std::cout << "***_Traverse_codim_0_level-wise" << std::endl;
80
81 // type of the GridView used for traversal
82 // every grid exports a LeafGridView and a LevelGridView
83 typedef typename G::LevelGridView LevelGridView;
84
85 // Get the iterator type
86 // Note the use of the typename and template keywords
87 typedef typename LevelGridView::template Codim<0>
88 :: Iterator ElementLevelIterator;
89
90 // iterate through all entities of codim 0 on the given level
91 for (int level=0; level<=grid.maxLevel(); level++)
92 {
93     // get the instance of the LeafGridView
94     LevelGridView levelView = grid.levelGridView(level);
95
96     count = 0;
97     for (ElementLevelIterator it = levelView.template begin<0>();
98          it!=levelView.template end<0>(); ++it)
99     {
100         Dune::GeometryType gt = it->type();
101         std::cout << "visiting_" << gt
102                  << "_with_first_vertex_at_" << it->geometry().corner(0)
103                  << std::endl;
104         count++;
105     }
106     std::cout << "there_are/is_" << count << "_element(s)_on_level_"
```

## 2 Getting started

```
107         << level << std::endl;
108     std::cout << std::endl;
109 }
110 }
111
112
113 int main(int argc, char **argv)
114 {
115     // initialize MPI, finalize is done automatically on exit
116     Dune::MPIHelper::instance(argc,argv);
117
118     // start try/catch block to get error messages from dune
119     try {
120         // make a grid
121         const int dim=2;
122         typedef Dune::SGrid<dim,dim> GridType;
123         Dune::FieldVector<int,dim> N(1);
124         Dune::FieldVector<GridType::ctype,dim> L(-1.0);
125         Dune::FieldVector<GridType::ctype,dim> H(1.0);
126         GridType grid(N,L,H);
127
128         // refine all elements once using the standard refinement rule
129         grid.globalRefine(1);
130
131         // traverse the grid and print some info
132         traversal(grid);
133     }
134     catch (std::exception & e) {
135         std::cout << "STL_ERROR:" << e.what() << std::endl;
136         return 1;
137     }
138     catch (Dune::Exception & e) {
139         std::cout << "DUNE_ERROR:" << e.what() << std::endl;
140         return 1;
141     }
142     catch (...) {
143         std::cout << "Unknown_ERROR" << std::endl;
144         return 1;
145     }
146
147     // done
148     return 0;
149 }
```

The `main` function near the end of the listing is pretty similar to the previous one except that we use a 2d grid for the unit square that just consists of one cell. In line 129 this cell is refined once using the standard method of grid refinement of the implementation. Here, the cell is refined into four smaller cells. The main work is done in a call to the function `traversal` in line 132. This function is given in lines 16-110.

The function `traversal` is a function template that is parametrized by a class `G` that is assumed to implement the **DUNE** grid interface. Thus, it will work on *any* grid available in **DUNE** without any changes. We now go into the details of this function.

The algorithm should work in any dimension so we extract the grid's dimension in line 20. Next, each **DUNE** grid defines a type that it uses to represent positions. This type is extracted in line 24 for later use.

A grid is considered to be a container of “entities” which are abstractions for geometric objects like vertices, edges, quadrilaterals, tetrahedra, and so on. This is very similar to the standard template

## 2 Getting started

library (STL), see e. g. [9], which is part of any C++ system. A key difference is, however, that there is not just one type of entity but several. As in the STL the elements of any container can be accessed with iterators which are generalized pointers. Again, a **DUNE** grid knows several different iterators which provide access to the different kinds of entities and which also provide different patterns of access.

As we usually do not want to use the entire hierarchy of the grid, we first define a view on that part of the grid we are interested in. This can be a level or the leaf part of the grid. In line 31 a type for a **GridView** on the leaf grid is defined.

Line 38 extracts the type of an iterator from this view class. **Codim** is a **struct** within the grid class that takes an integer template parameter specifying the codimension over which to iterate. Within the **Codim** structure the type **Iterator** is defined. Since we specified codimension 0 this iterator is used to iterate over the elements which are not refined any further, i.e. which are the leaves of the refinement trees.

The **for**-loop in line 42 now visits every such element. The **begin** and **end** on the **LeafGridView** class deliver the first leaf element and one past the last leaf element. Note that the **template** keyword must be used and template parameters are passed explicitly. Within the loop body in lines 44-50 the iterator **it** acts like a pointer to an entity of dimension **dim** and codimension 0. The exact type would be **typename G::template Codim<0>::Entity** just to mention it.

An important part of an entity is its geometrical shape and position. All geometrical information is factored out into a sub-object that can be accessed via the **geometry()** method. The geometry object is in general a mapping from a  $d$ -dimensional polyhedral reference element to  $w$  dimensional space. Here we have  $d = G::dimension$  and  $w = G::dimensionworld$ . This mapping is also called the “local to global” mapping. The corresponding reference element has a certain type which is extracted in line 45. Since the reference elements are polyhedra they consist of a finite number of corners. The images of the corners under the local to global map can be accessed via the **corner(int n)** method. Line 46 prints the geometry type and the position of the first corner of the element. Then line 49 just counts the number of elements visited.

Suppose now that we wanted to iterate over the vertices of the leaf grid instead of the elements. Now vertices have the codimension **dim** in a **dim**-dimensional grid and a corresponding iterator is provided by each grid class. It is extracted in line 61 for later use. The **for**-loop starting in line 65 is very similar to the first one except that it now uses the **VertexLeafIterator**. As you can see the different entities can be accessed with the same methods. We will see later that codimensions 0 and **dim** are specializations with an extended interface compared to all other codimensions. You can also access the codimensions between 0 and **dim**. However, currently not all implementations of the grid interface support these intermediate codimensions (though this does not restrict the implementation of finite element methods with degrees of freedom associated to, say, faces).

Finally, we show in lines 83-109 how the hierarchic structure of the mesh can be accessed. To that end a **LevelGridView** is used. It provides via an **Iterator** access to all entities of a given codimension (here 0) on a given grid level. The coarsest grid level (the initial macro grid) has number zero and the number of the finest grid level is returned by the **maxLevel()** method of the grid. The methods **begin()** and **end()** on the view deliver iterators to the first and one-past-the-last entity of a given grid level supplied as an integer argument to these methods.

The following listing shows the output of the program.

### Listing 4 (Output of traversal)

## 2 Getting started

```
*** Traverse codim 0 leaves
visiting leaf (cube, 2) with first vertex at -1 -1
visiting leaf (cube, 2) with first vertex at 0 -1
visiting leaf (cube, 2) with first vertex at -1 0
visiting leaf (cube, 2) with first vertex at 0 0
there are/is 4 leaf element(s)

*** Traverse codim 2 leaves
visiting (cube, 0) at -1 -1
visiting (cube, 0) at 0 -1
visiting (cube, 0) at 1 -1
visiting (cube, 0) at -1 0
visiting (cube, 0) at 0 0
visiting (cube, 0) at 1 0
visiting (cube, 0) at -1 1
visiting (cube, 0) at 0 1
visiting (cube, 0) at 1 1
there are/is 9 leaf vertices(s)

*** Traverse codim 0 level-wise
visiting (cube, 2) with first vertex at -1 -1
there are/is 1 element(s) on level 0

visiting (cube, 2) with first vertex at -1 -1
visiting (cube, 2) with first vertex at 0 -1
visiting (cube, 2) with first vertex at -1 0
visiting (cube, 2) with first vertex at 0 0
there are/is 4 element(s) on level 1
```

**Remark 2.3** Define the end iterator for efficiency.

**Exercise 2.4** Play with different dimensions, codimension (SGrid supports all codimensions) and refinements.

**Exercise 2.5** The method `corners()` of the geometry returns the number of corners of an entity. Modify the code such that the positions of all corners are printed.

## 3 The DUNE grid interface

### 3.1 Grid definition

There is a great variety of grids: conforming and non-conforming grids, single-element-type and multiple-element-type grids, locally and globally refined grids, nested and non-nested grids, bisection-type grids, red-green-type grids, sparse grids and so on. In this section we describe in some detail the type of grids that are covered by the **DUNE** grid interface.

#### Reference elements

A computational grid is a non-overlapping subdivision of a domain  $\Omega \subset \mathbb{R}^w$  into elements of “simple” shape. Here “simple” means that the element can be represented as the image of a reference element under a transformation. A reference element is a convex polytope, which is a bounded intersection of a finite set of half-spaces.

#### Dimension and world dimension

A grid has a dimension  $d$  which is the dimensionality of its reference elements. Clearly we have  $d \leq w$ . In the case  $d < w$  the grid discretizes a  $d$ -dimensional manifold.

#### Faces, entities and codimension

The intersection of a  $d$ -dimensional convex polytope (in  $d$ -dimensional space) with a tangent plane is called a face (note that there are faces of dimensionality  $0, \dots, d - 1$ ). Consequently, a face of a grid element is defined as the image of a face of its reference element under the transformation. The elements and faces of elements of a grid are called its entities. An entity is said to be of codimension  $c$  if it is a  $d - c$ -dimensional object. Thus the elements of the grid are entities of codimension 0, facets of an element have codimension 1, edges have codimension  $d - 1$  and vertices have codimension  $d$ .

#### Conformity

Computational grids come in a variety of flavours: A conforming grid is one where the intersection of two elements is either empty or a face of each of the two elements. Grids where the intersection of two elements may have an arbitrary shape are called nonconforming.

#### Element types

A simplicial grid is one where the reference elements are simplices. In a multi-element-type grid a finite number of different reference elements are allowed. The **DUNE** grid interface can represent conforming as well as non-conforming grids.

#### Hierarchically nested grids, macro grid

A hierarchically nested grid consists of a collection of  $J + 1$  grids that are subdivisions of nested domains

$$\Omega = \Omega_0 \supseteq \Omega_1 \supseteq \dots \supseteq \Omega_J.$$

Note that only  $\Omega_0$  is required to be identical to  $\Omega$ . If  $\Omega_0 = \Omega_1 = \dots = \Omega_J$  the grid is globally refined, otherwise it is locally refined. The grid that discretizes  $\Omega_0$  is called the macro grid and its elements



### 3 The **DUNE** grid interface

the macro elements. The grid for  $\Omega_{l+1}$  is obtained from the grid for  $\Omega_l$  by possibly subdividing each of its elements into smaller elements. Thus, each element of the macro grid and the elements that are obtained from refining it form a tree structure. The grid discretizing  $\Omega_l$  with  $0 \leq l \leq J$  is called the level- $l$ -grid and its elements are obtained from an  $l$ -fold refinement of some macro elements.

#### Leaf grid

Due to the nestedness of the domains we can partition the domain  $\Omega$  into

$$\Omega = \Omega_J \cup \bigcup_{l=0}^{J-1} \Omega_l \setminus \Omega_{l+1}.$$

As a consequence of the hierarchical construction a computational grid discretizing  $\Omega$  can be obtained by taking the elements of the level- $J$ -grid plus the elements of the level- $J-1$ -grid in the region  $\Omega_{J-1} \setminus \Omega_J$  plus the elements of the level- $J-2$ -grid in the region  $\Omega_{J-2} \setminus \Omega_{J-1}$  and so on plus the elements of the level-0-grid in the region  $\Omega_0 \setminus \Omega_1$ . The grid resulting from this procedure is called the leaf grid because it is formed by the leaf elements of the trees emanating at the macro elements.

#### Refinement rules

There is a variety of ways how to hierarchically refine a grid. The refinement is called conforming if the leaf grid is always a conforming grid, otherwise the refinement is called non-conforming. Note that the grid on each level  $l$  might be conforming while the leaf grid is not. There are also many ways how to subdivide an individual element into smaller elements. Bisection always subdivides elements into two smaller elements, thus the resulting data structure is a binary tree (independent of the dimension of the grid). Bisection is sometimes called “green” refinement. The so-called “red” refinement is the subdivision of an element into  $2^d$  smaller elements, which is most obvious for cube elements. In many practical situation anisotropic refinement, i. e. refinement in a preferred direction, may be required.

#### Summary

The **DUNE** grid interface is able to represent grids with the following properties:

- Arbitrary dimension.
- Entities of all codimensions.
- Any kind of reference elements (you could define the icosahedron as a reference element if you wish).
- Conforming and non-conforming grids.
- Grids are always hierarchically nested.
- Any type of refinement rules.
- Conforming and non-conforming refinement.
- Parallel, distributed grids.

## 3.2 Concepts

Generic algorithms are based on concepts. A concept is a kind of “generalized” class with a well defined set of members. Imagine a function template that takes a type `T` as template argument. All the members of `T`, i.e. methods, enumerations, data (rarely) and nested classes used by the function template form the concept. From that definition it is clear that the concept does not necessarily exist as program text.

A class that implements a concept is called a *model* of the concept. E.g. in the standard template library (STL) the class `std::vector<int>` is a model of the concept “container”. If all instances of a class template are a model of a given concept we can also say that the class template is a model of the concept. In that sense `std::vector` is also a model of container.

In standard OO language a concept would be formulated as an abstract base class and all the models would be implemented as derived classes. However, for reasons of efficiency we do not want to use dynamic polymorphism. Moreover, concepts are more powerful because the models of a concept can use different types, e.g. as return types of methods. As an example consider the STL where the `begin` method on a vector of `int` returns `std::vector<int>::iterator` and on a list of `int` it returns `std::list<int>::iterator` which may be completely different types.

Concepts are difficult to describe when they do not exist as concrete entities (classes or class templates) in a program. The STL way of specifying concepts is to describe the members `X::foo()` of some arbitrary model named `X`. Since this description of the concept is not processed by the compiler it can get inconsistent and there is no way to check conformity of a model to the interface. As a consequence, strange error messages from the compiler may be the result (well C++ compilers can always produce strange error messages). There are two ways to improve the situation:

- *Engines*: A class template is defined that wraps the model (which is the template parameter) and forwards all member function calls to it. In addition all the nested types and enumerations of the model are copied into the wrapper class. The model can be seen as an engine that powers the wrapper class, hence the name. Generic algorithms are written in terms of the wrapper class. Thus the wrapper class encapsulates the concept and it can be ensured formally by the compiler that all members of the concept are implemented.
- *Barton-Nackman trick*: This is a refinement of the engine approach where the models are derived from the wrapper class template in addition. Thus static polymorphism is combined with a traditional class hierarchy, see [11, 1]. However, the Barton-Nackman trick gets rather involved when the derived classes depend on additional template parameters and several types are related with each other. That is why it is not used at all places in **DUNE**.

The **DUNE** grid interface now consists of a *set of related concepts*. Either the engine or the Barton-Nackman approach are used to clearly define the concepts. In order to avoid any inconsistencies we refer as much as possible to the doxygen-generated documentation. For an overview of the grid interface see the web page

[http://www.dune-project.org/doc/doxygen/html/group\\_\\_Grid.html](http://www.dune-project.org/doc/doxygen/html/group__Grid.html).

### 3.2.1 Common types

Some types in the grid interface do not depend on a specific model, i. e. they are shared by all implementations.

#### **Dune::ReferenceElement**

describes the topology and geometry of standard entities. Any given entity of the grid can be completely specified by a reference element and a map from this reference element to world coordinate space.

#### **Dune::GeometryType**

defines names for the reference elements.

#### **Dune::CollectiveCommunication**

defines an interface to global communication operations in a portable and transparent way. In particular also for sequential grids.

### 3.2.2 Concepts of the **DUNE** grid interface

In the following a short description of each concept in the **DUNE** grid interface is given. For the details click on the link that leads you to the documentation of the corresponding wrapper class template (in the engine sense).

#### **Grid**

The grid is a container of entities that allows to access these entities and that knows the number of its entities. You create instances of a grid class in your applications, while objects of the other classes are typically aggregated in the grid class and accessed via iterators.

#### **GridView**

The GridView gives a view on a level or the leaf part of a grid. It provides iterators for access to the elements of this view and a communication method for parallel computations. Alternatively, a LevelIterator or a LeafIterator can be directly accessed from a grid. These iterator types are described below.

#### **Entity**

The entity class encapsulates the topological part of an entity, i.e. its hierarchical construction from subentities and the relation to other entities. Entities cannot be created, copied or modified by the user. They can only be read-accessed through immutable iterators.

#### **Geometry**

Geometry encapsulates the geometric part of an entity by mapping local coordinates in a reference element to world coordinates.

#### **EntityPointer**

EntityPointer is a dereferenceable type that delivers a reference to an entity. Moreover it is immutable, i.e. the referenced entity can not be modified.

#### **Iterator**

Iterator is an immutable iterator that provides access to an entity. It can be incremented to visit all entities of a given codimension of a GridView. An EntityPointer is assignable from an Iterator.

#### **IntersectionIterator**

IntersectionIterator provides access to all entities of codimension 0 that have an intersection of codimension 1 with a given entity of codimension 0. In a conforming mesh these are the face neighbors

of an element. For two entities with a common intersection the `IntersectionIterator` can be dereferenced as an `Intersection` object which in turn provides information about the geometric location of the intersection. Furthermore this `Intersection` class also provides information about intersections of an entity with the internal or external boundaries. The `IntersectionIterator` provides intersections between codimension 0 entities among the same `GridView`.

#### **LevelIndexSet, LeafIndexSet**

`LevelIndexSet` and `LeafIndexSet`, which are both models of `Dune::IndexSet`, are used to attach any kind of user-defined data to (subsets of) entities of the grid. This data is supposed to be stored in one-dimensional arrays for reasons of efficiency. An `IndexSet` is usually not used directly but through a `Mapper` (c.f. chapter 6.1).

#### **LocalIdSet, GlobalIdSet**

`LocalIdSet` and `GlobalIdSet` which are both models of `Dune::IdSet` are used to save user data during a grid refinement phase and during dynamic load balancing in the parallel case. The `LocalIdSet` is unique for all entities on the current partition, whereas the `GlobalIdSet` gives a unique mapping over all grid partitions. An `IdSet` is usually not used directly but through a `Mapper` (c.f. chapter 6.1).

### **3.3 Propagation of type information**

The types making up one grid implementation cannot be mixed with the types making up another grid implementation. Say, we have two implementations of the grid interface `XGrid` and `YGrid`. Each implementation provides a `LevelIterator` class, named `XLevelIterator` and `YLevelIterator` (in fact, these are class templates because they are parametrized by the codimension and other parameters). Although these types implement the same interface they are distinct classes that are not related in any way for the compiler. As in the Standard Template Library strange error messages may occur if you try to mix these types.

In order to avoid these problems the related types of an implementation are provided as public types by most classes of an implementation. E. g., in order to extract the `XLevelIterator` (for codimension 0) from the `XGrid` class you would write

```
XGrid::template Codim<0>::LevelIterator
```

Because most of the types are parametrized by certain parameters like dimension, codimension or partition type simple typedefs (as in the STL) are not sufficient here. The types are rather placed in a struct template, named `Codim` here, where the template parameters of the struct are those of the type. This concept may even be applied recursively.

## 4 Constructing grid objects

So far we have talked about the grid interface and how you can access and manipulate grids. This chapter will show you how you create grids in the first place. There are several ways to do this.

The central idea of **DUNE** is that all grid implementations behave equally and conform to the same interface. However, this concept fails when it comes to constructing grid objects, because grid implementations differ too much to make one construction method work for all. For example, for an unstructured grid you have to specify all vertex positions, whereas for a structured grid this would be a waste of time. On the other hand, for a structured grid you may need to give the bounding box which, for an unstructured grid, is not necessary. In practice, these differences do not pose serious problems.

In this chapter, creating a grid always means creating a grid with only a single level. Such grid is alternatively called a *coarse grid* or a *macro grid*. There is currently no functionality in **DUNE** to set up hierarchical grids directly. The underlying assumption is that the user will create a coarse grid first and then generate a hierarchy using refinement. Despite the name (and grid implementations permitting), the coarse grid can of course be as large and fine as desired.

### 4.1 Creating Structured Grids

Creating structured grids is comparatively simple, as little information needs to be provided. In general, for uniform structured grids, the grid dimension, bounding box, and number of elements in each direction suffices. Such information can be given directly with the constructor of the grid object. **DUNE** does not currently specify the signature of grid constructors, and hence they are all slightly different. For example, to create a 2D `SGrid` in  $[0, 1]^2 \subset \mathbb{R}^2$  with 10 elements in each direction call

```
Dune::FieldVector<int,2> n;  
n[0] = n[1] = 10;  
  
Dune::FieldVector<double,2> lower;  
lower[0] = lower[1] = 0.0;  
  
Dune::FieldVector<double,2> upper;  
upper[0] = upper[1] = 1.0;  
  
Dune::SGrid<2,2> grid(n, lower, upper);
```

If you want to do the same for a sequential `YaspGrid` the code is

```
Dune::FieldVector<int,2> n;  
n[0] = n[1] = 10;  
  
Dune::FieldVector<double,2> upper;
```

```
upper[0] = upper[1] = 1.0;

Dune::FieldVector<bool,dim> periodic(false);

YaspGrid<2> grid(upper, n, periodic, 0);
```

Note that you do not have to specify the lower left corner as `YaspGrid` hardwires it to zero. The unstructured one-dimensional `OneDGrid` also has a constructor

```
OneDGrid grid(10,    // number of elements
              0.0,    // left domain boundary
              1.0     // right domain boundary
              );
```

for uniform grids.

## 4.2 Reading Unstructured Grids from Files

Unstructured grids usually require much more information than what can reasonable be provided within the program code. Instead, they are usually read from special files. A large variety of different file formats for finite element grids exists, and **DUNE** provides support for some of them. Again, no interface specification exists for file readers in **DUNE**.

Arguably the most important file format currently supported by **DUNE** is the `gmsh` format. `Gmsh`<sup>1</sup> is an open-source geometry modeler and grid generator. It allows to define geometries using a boundary representation (interactively and via its own modeling language resulting in `.geo`-files), creates simplicial grids in 2d and 3d (using `tetgen` or `netgen`) and stores them in files ending in `.msh`. Current precompiled releases  $\geq 2.4.2$  of `Gmsh` have `OpenCascade`, an open-source CAD kernel, as built-in geometry modeler. Thus these releases are able to import CAD geometries, e. g. from IGES or STEP files, and to generate meshes for them to be subsequently used in **DUNE**. Be aware that most versions of `Gmsh` available in the package repositories of your operating system still lack this functionality. Further, `Gmsh` and the `Gmsh` reader of **DUNE** support second order boundary segments thus providing a rudimentary support for curved boundaries. To read such a file into a `FooGrid`, include `dune/grid/io/file/gmshreader.hh` and write

```
FooGrid* grid = GmshReader<GridType>::read(filename);
```

A second format is `AmiraMesh`, which is the native format of the `Amira`.<sup>2</sup> To read `AmiraMesh` files you need to have the library `libamiramesh`<sup>3</sup> installed. Then

```
FooGrid* grid = AmiraMeshReader<GridType>::read(filename);
```

reads the grid in `filename` into the `FooGrid`.

Further available formats are `StarCD` and the native `Alberta` format. See the class documentation of `dune-grid` for an up-to-date list. Demo grids for each format can be found in `dune-grid/doc/grids`. They exist for documentation and also as example grids for the unit tests of the file readers. The unit

---

<sup>1</sup><http://geuz.org/gmsh/>

<sup>2</sup><http://www.amira.com/>

<sup>3</sup><http://amira.com/downloads/patch-archive/patches412/81.html>

tests should not hardwire the path to the example grids. Instead, the path should be provided in the preprocessor variable `DUNE_GRID_EXAMPLE_GRIDS_PATH`.

### 4.3 The DUNE Grid Format (DGF)

Dune has its own macro grid format, the Dune Grid Format. A detailed description of the DGF and how to use it can be found on the homepage of **DUNE**<sup>4</sup>.

Here we only give a short introduction. Basically one can choose the grid manager during the make process of your program: `make GRIDTYPE=MYGRID GRIDDIM=d GRIDWORLD=w` Including `config.h` will then introduce the namespace `GridSelector` into the `Dune` namespace. This contains the typedef `GridType` which is the type of the grid. Furthermore the required header files for the grid manager are included. Through the module **DUNE-grid** the following grid managers can be used (for `MYGROD` in the example above):

`ALBERTAGRID`, `ALUGRID_CUBE`, `ALUGRID_SIMPLEX`, `ALUGRID_CONFORM`, `ONEDGRID`, `SGRID`, `UGGRID`, and `YASPGRID`.

The following example shows how an instance of the defined grid is generated. Given a DGF file, for example `unitcube2.dgf`, a grid pointer is created as follows

```
Dune::GridPtr<Dune::GridSelector::GridType> gridPtr( "unitcube2.dgf" );
```

The grid is accessed by dereferencing the grid pointer.

```
GridType& grid = *gridPtr;
```

To change the grid one simply has to re-compile the code using the following make command.

```
make GRIDDIM=2 GRIDTYPE=ALBERTAGRID integration
```

This will compile the application `integration` with grid type `ALBERTAGRID` and grid dimension 2. Note that before the re-compilation works, the corresponding object file has to be removed. If `WORLDDIM` is not provide then `WORLDDIM=GRIDDIM` is assumed. To use some grid manager by default, i.e., without providing the grid type during the make process, `GRIDTYPE` and `GRIDDIM`, `WORLDDIM` can be set in `Makefile.am`. It is then still possible to change the default during the make process.

### 4.4 The Grid Factory

While there is currently no convention on what a file reader should look like, there is a formally specified low-level interface for the construction of unstructured coarse grids. This interface, which goes by the name of `GridFactory`, provides methods to, e.g. insert vertices and elements one by one. It is the basis of the file readers described in the previous section. The main reason why you may want to program the `GridFactory` directly is when writing your own grid readers. However, in some cases it may also be most convenient to be able to specify your coarse grid entirely in the C++ code. You can do that using the `GridFactory`.

The `GridFactory` is programmed as a factory class (hence the name). You default-construct an object of the factory class, provide it with all necessary information, and it will create and hand over a grid for you. In the following we will describe the use of the `GridFactory` in more detail. Say you are interested in creating a new grid of type `FooGrid`. Then you proceed as follows:

<sup>4</sup>[http://www.dune-project.org/doc/doxygen/html/classDune\\_1\\_1DuneGridFormatParser.html](http://www.dune-project.org/doc/doxygen/html/classDune_1_1DuneGridFormatParser.html)

## 4 Constructing grid objects

### 1. Create a GridFactory Object

Get a new GridFactory object by calling

```
GridFactory< FooGrid > factory;
```

### 2. Enter the Vertices

Insert the grid vertices by calling

```
factory.insertVertex(const FieldVector<ctype,dimworld>& position);
```

for each vertex. The order of insertion determines the level- and leaf indices of your level 0 vertices.

### 3. Enter the elements

For each element call

```
factory.insertElement(Dune::GeometryType type,  
    const std::vector<int>& vertices);
```

The parameters are

- type - The element type. The grid implementation is expected to throw an exception if an element type that cannot be handled is encountered.
- vertices - The indices of the vertices of this element.

The numbering of the vertices of each element is expected to follow the **DUNE** conventions. Refer to the page on reference elements for the details.

### 4. Parametrized Domains

**FooGrid** may support parametrized domains. That means that you can provide a smooth description of your grid boundary. The actual grid may always be piecewise linear; however, as you refine, the grid will approach your prescribed boundary. You don't have to do this. If you do not specify the boundary geometry it is left to the grid implementation.

In order to create curved boundary segments, for each segment you have to write a class which implements the correct geometry. These classes are then handed over to the factory. Boundary segment implementations must be derived from

```
template <int dim, int dimworld> Dune::BoundarySegment
```

This is an abstract base class which requires you to overload the method

```
virtual FieldVector< double, dimworld >  
    operator() (const FieldVector< double, dim-1 > &local)
```

This methods must compute the world coordinates from the local ones on the boundary segment. Give these classes to your grid factory by calling

```
factory.insertBoundarySegment(const std::vector<int>& vertices,  
    const BoundarySegment<dim,dimworld> *  
    boundarySegment = NULL);
```



Control over the allocated objects is taken from you, and the grid object will take care of their destruction.

Note that you can call `insertBoundarySegment` with only the first argument. In that case, the boundary geometry is left to the grid implementation. However, the boundary segments get ordered in the way you inserted them. This may be helpful if you have data attached to your coarse grid boundary (see Sec. 4.5).

### 5. Finish construction

To finish off the construction of the `FooGrid` object call

```
FooGrid* grid = factory.createGrid();
```

This time it is you who gets full responsibility for the allocated object.

## 4.5 Attaching Data to a New Grid

In many cases there is data attached to new grids. This data may be initial values, spatially distributed material parameters, boundary conditions, etc. It is associated to elements or vertices, or the boundary segments of the coarse grid. The data may be available in a separate data file or even included in the same file with the grid.

The connection with the grid in the grid file is usually made implicitly. For example, vertex data is ordered in the same order as the vertices itself. Hence the grid-reading process must guarantee that vertices and elements are not reordered during grid creation. More specifically, **DUNE** guarantees the following: *the level and leaf indices of zero-level vertices and elements are defined by the order in which they were inserted into the grid factory*. Note that this does not mean that the vertices and elements are traversed in this order by the `Level-` and `LeafIterators`. What matters are the indices. Note also that no such promise is made concerning edges, faces and the like. Hence it is currently not possible to read edge and face data along with a grid without some trickery.

It is also possible to attach data to boundary segments of the coarse grids. For this, the method `Intersection::boundaryId` (which should really be called `boundaryIndex`) returns an index when called for a boundary intersection. If the boundary intersection is on level zero the index is consecutive and zero-starting. For all other boundary intersections it is the index of the zero-level ancestor boundary segment of the intersection.

If you have a list of data associated to certain boundary segments of your coarse grid, you need some control on how the boundary ids are set. Remember from Sec. 4.4 that you can create a grid without mentioning the boundary at all. If you do that, the boundary ids are set automatically by the grid implementation and the exact order is implementation-specific. If you set boundary segments explicitly using the `insertBoundarySegment` method, then *the boundary segments are numbered in the order of their insertion*. If you do not set all boundary segments the remaining ones get automatic, implementation-specific ids. This is why the second argument of `insertBoundarySegment` is optional: you may want to influence the ordering of the boundary segments, but leave the boundary geometry to the grid implementation. Calling `insertBoundarySegment` with a single argument allows you to do just this.

## 4.6 Example: The UnitCube class

In this chapter we give example code that shows how the different available grid classes are instantiated. We create grids for the unit cube  $\Omega = (0,1)^d$  in various dimensions  $d$ .

Not all grid classes have the same interface for instantiation. Unstructured grids are created using the `GridFactory` class, but for structured grids there is more variation. In order to make the examples in later chapters easier to write we want to have a class template `UnitCube` that we parametrize with a type `T` and an integer parameter `variant`. `T` should be one of the available grid types and `variant` can be used to generate different grids (e.g. triangular or quadrilateral) for the same type `T`. The advantage of the `UnitCube` template is that the instantiation is hidden from the user.

The definition of the general template is as follows.

### Listing 5 (File `dune-grid-howto/unitcube.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef UNITCUBE_HH
4 #define UNITCUBE_HH
5
6 #include <dune/common/exceptions.hh>
7 #include <dune/common/fvector.hh>
8 #include <dune/grid/utility/structuredgridfactory.hh>
9
10 // default implementation for any template parameter
11 template<typename T, int variant>
12 class UnitCube
13 {
14 public:
15     typedef T GridType;
16
17     static const int dim = GridType::dimension;
18
19     // constructor throwing exception
20     UnitCube ()
21     {
22         Dune::FieldVector<typename GridType::ctype,dim> lowerLeft(0);
23         Dune::FieldVector<typename GridType::ctype,dim> upperRight(1);
24         Dune::array<unsigned int,dim> elements;
25         std::fill(elements.begin(), elements.end(), 1);
26
27         switch (variant) {
28         case 1 :
29             grid_ = Dune::StructuredGridFactory<GridType>::createCubeGrid(lowerLeft, upperRight,
30                                     elements);
31             break;
32         case 2 :
33             grid_ = Dune::StructuredGridFactory<GridType>::createSimplexGrid(lowerLeft, upperRight,
34                                     elements);
35             break;
36         default :
37             DUNE_THROW( Dune::NotImplemented, "Variant_"
38                         << variant << " of unit cube not implemented." );
39         }
40     }
41
42     T& grid ()
43     {
44         return *grid_;
45     }
46 }

```

## 4 Constructing grid objects

```
44
45 private:
46     // the constructed grid object
47     Dune::shared_ptr<T> grid_;
48 };
49
50
51 // include specializations
52 #include "unitcube_sgrid.hh"
53 #include "unitcube_yaspgrid.hh"
54 #include "unitcube_albertagrid.hh"
55 #include "unitcube_alugrid.hh"
56
57 #endif
```

This is a default implementation that uses the utility class `StructuredGridFactory` (from the header `dune-grid/dune/grid/utility/structuredgridfactory.hh`) to create grids for the unit cube. The `StructuredGridFactory` uses the `GridFactory` class (Section 4.4) internally to create structured simplicial and hexahedral grids. Depending on the template parameter `variant`, a hexahedral (`variant==1`) or simplicial (`variant==2`) grid is created.

The `GridFactory` class is a required part of the grid interface for all unstructured grids. Hence the default implementation of `UnitCube` should work for all unstructured grids, namely `UGGrid`, `OneDGrid`, `ALUGrid`, and `AlbertaGrid`. The construction of structured grid objects is currently not standardized. Therefore `UnitCube` is specialized for each structured grid type. We now look at each specialization in turn.

For historic reasons, there are also specializations for `ALUGrid` and `AlbertaGrid`.

### SGrid

The following listing creates an `SGrid` object. This class template also has a constructor without arguments that results in a cube with a single element. `SGrid` supports all dimensions.

#### Listing 6 (File `dune-grid-howto/unitcube_sgrid.hh`)

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef UNITCUBE_SGRID_HH
4 #define UNITCUBE_SGRID_HH
5
6 #include "unitcube.hh"
7
8 #include <dune/grid/sgrid.hh>
9
10 // SGrid specialization
11 template<int dim>
12 class UnitCube<Dune::SGrid<dim,dim>,1>
13 {
14 public:
15     typedef Dune::SGrid<dim,dim> GridType;
16
17     Dune::SGrid<dim,dim>& grid ()
18     {
19         return grid_;
20     }
21
22 private:
23     Dune::SGrid<dim,dim> grid_;
24 };
```

```

25
26 #endif

```

### YaspGrid

The following listing instantiates a **YaspGrid** object. The **variant** parameter specifies the number of elements in each direction of the cube. In the parallel case all available processes are used and the overlap is set to one element. Periodicity is not used.

#### Listing 7 (File dune-grid-howto/unitcube\_yaspgrid.hh)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef UNITCUBE_YASPGRID_HH
4 #define UNITCUBE_YASPGRID_HH
5
6 #include "unitcube.hh"
7
8 #include <dune/grid/yaspgrid.hh>
9
10 // YaspGrid specialization
11 template<int dim, int size>
12 class UnitCube<Dune::YaspGrid<dim>,size>
13 {
14 public:
15     typedef Dune::YaspGrid<dim> GridType;
16
17     UnitCube ()
18     {
19         Dune::FieldVector<double,dim> length(1.0);
20         Dune::array<int,dim> elements;
21         std::fill(elements.begin(), elements.end(), size);
22         std::bitset<dim> periodicity(0);
23
24         grid_ = std::auto_ptr<Dune::YaspGrid<dim> >(new Dune::YaspGrid<dim>(
25 #if HAVE_MPI
26                                                         MPI_COMM_WORLD,
27 #endif
28                                                         length,elements,periodicity,1));
29     }
30
31     Dune::YaspGrid<dim>& grid ()
32     {
33         return *grid_;
34     }
35
36 private:
37     std::auto_ptr<Dune::YaspGrid<dim> > grid_;
38 };
39
40 #endif

```

### AlbertaGrid

The following listing contains specializations of the **UnitCube** template for Alberta in two and three dimensions. When using Alberta versions less than 2.0 the **DUNE** framework has to be configured with a dimension (**--with-alberta-dim=2**, **--with-alberta-world-dim=2**) and only this dimension can then be used. The dimension from the configure run is available in the macro **ALBERTA\_DIM** and

ALBERTA\_WORLD\_DIM in the file `config.h` (see next section). The `variant` parameter must be 1. The grid factory concept is used by the base class `BasicUnitCube`.

#### Listing 8 (File `dune-grid-howto/unitcube_albertagrid.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef UNITCUBE_ALBERTAGRID_HH
4 #define UNITCUBE_ALBERTAGRID_HH
5
6 #include "unitcube.hh"
7 #include "basicunitcube.hh"
8
9 #if HAVE_ALBERTA
10 #include <dune/grid/albertagrid.hh>
11 #include <dune/grid/albertagrid/gridfactory.hh>
12
13 template< int dim >
14 class UnitCube< Dune::AlbertaGrid< dim, dim >, 1 >
15   : public BasicUnitCube< dim >
16 {
17 public:
18   typedef Dune::AlbertaGrid< dim, dim > GridType;
19
20 private:
21   GridType *grid_;
22
23 public:
24   UnitCube ()
25   {
26     Dune::GridFactory< GridType > factory;
27     BasicUnitCube< dim >::insertVertices( factory );
28     BasicUnitCube< dim >::insertSimplices( factory );
29     grid_ = factory.createGrid();
30   }
31
32   ~UnitCube ()
33   {
34     Dune::GridFactory< GridType >::destroyGrid( grid_ );
35   }
36
37   GridType &grid ()
38   {
39     return *grid_;
40   }
41 };
42
43 #endif // #if HAVE_ALBERTA
44
45 #endif

```

#### ALUGrid

The next listing shows the instantiation of `ALUSimplexGrid` or `ALUCubeGrid` objects. The `ALU-Grid` implementation supports either simplicial grids, i.e. tetrahedral or triangular grids, and hexahedral grids and the element type has to be chosen at compile-time. This is done by choosing either `ALUSimplexGrid` or `ALUCubeGrid`. The `variant` parameter must be 1. As in the default implementation, grid objects are set up with help of the `StructuredGridFactory` class.

#### Listing 9 (File `dune-grid-howto/unitcube_alugrid.hh`)

## 4 Constructing grid objects

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef UNITCUBE_ALUGRID_HH
4 #define UNITCUBE_ALUGRID_HH
5
6 #include "unitcube.hh"
7
8 #if HAVE_ALUGRID
9 #include <dune/grid/alugrid.hh>
10 #include <dune/grid/alugrid/3d/alu3dgridfactory.hh>
11
12 // ALU3dGrid and ALU2dGrid simplex specialization.
13 // Note: element type determined by type
14 template<int dim>
15 class UnitCube<Dune::ALUGrid<dim,dim,Dune::simplex,Dune::nonconforming>,1>
16 {
17 public:
18     typedef Dune::ALUGrid<dim,dim,Dune::simplex,Dune::nonconforming> GridType;
19
20 private:
21     Dune::shared_ptr<GridType> grid_;
22
23 public:
24     UnitCube ()
25     {
26         Dune::FieldVector<typename GridType::ctype,dim> lowerLeft(0);
27         Dune::FieldVector<typename GridType::ctype,dim> upperRight(1);
28         Dune::array<unsigned int,dim> elements;
29         std::fill(elements.begin(), elements.end(), 1);
30
31         grid_ = Dune::StructuredGridFactory<GridType>::createSimplexGrid(lowerLeft, upperRight,
32             elements);
33     }
34
35     GridType &grid ()
36     {
37         return *grid_;
38     }
39 };
40
41 // ALU3dGrid hexahedra specialization. Note: element type determined by type
42 template<>
43 class UnitCube<Dune::ALUGrid<3,3,Dune::cube,Dune::nonconforming>,1>
44 {
45 public:
46     typedef Dune::ALUGrid<3,3,Dune::cube,Dune::nonconforming> GridType;
47
48 private:
49     Dune::shared_ptr<GridType> grid_;
50
51 public:
52     UnitCube ()
53     {
54         Dune::FieldVector<GridType::ctype,3> lowerLeft(0);
55         Dune::FieldVector<GridType::ctype,3> upperRight(1);
56         Dune::array<unsigned int,3> elements = { {1,1,1} };
57
58         grid_ = Dune::StructuredGridFactory<GridType>::createCubeGrid(lowerLeft, upperRight,
59             elements);
60     }
61
62     GridType &grid ()
63     {
64         return *grid_;
65     }
66 };
```

#### *4 Constructing grid objects*

```
62     return *grid_;  
63 }  
64 };  
65 #endif  
66  
67 #endif
```

## 5 Quadrature rules

In this chapter we explore how an integral

$$\int_{\Omega} f(x) \, dx$$

over some function  $f : \Omega \rightarrow \mathbb{R}$  can be computed numerically using a **DUNE** grid object.

### 5.1 Numerical integration

Assume first the simpler task that  $\Delta$  is a reference element and that we want to compute the integral over some function  $\hat{f} : \Delta \rightarrow \mathbb{R}$  over the reference element:

$$\int_{\Delta} \hat{f}(\hat{x}) \, d\hat{x}.$$

A quadrature rule is a formula that approximates integrals of functions over a reference element  $\Delta$ . In general it has the form

$$\int_{\Delta} \hat{f}(\hat{x}) \, d\hat{x} = \sum_{i=1}^n \hat{f}(\xi_i) w_i + \text{error}.$$

The positions  $\xi_i$  and weight factors  $w_i$  are dependent on the type of reference element and the number of quadrature points  $n$  is related to the error.

Using the transformation formula for integrals we can now compute integrals over domains  $\omega \subseteq \Omega$  that are mapped from a reference element, i. e.  $\omega = \{x \in \Omega \mid x = g(\hat{x}), \hat{x} \in \Delta\}$ , by some function  $g : \Delta \rightarrow \Omega$ :

$$\int_{\Omega} f(x) \, dx = \int_{\Delta} f(g(\hat{x})) \mu(\hat{x}) \, d\hat{x} = \sum_{i=1}^n f(g(\xi_i)) \mu(\xi_i) w_i + \text{error}. \quad (5.1)$$

Here  $\mu(\hat{x}) = \sqrt{|\det J^T(\hat{x}) J(\hat{x})|}$  is the integration element and  $J(\hat{x})$  the Jacobian matrix of the map  $g$ .

The integral over the whole domain  $\Omega$  requires a grid  $\overline{\Omega} = \bigcup_k \overline{\omega}_k$ . Using (5.1) on each element we obtain finally

$$\int_{\Omega} f(x) \, dx = \sum_k \sum_{i=1}^{n_k} f(g^k(\xi_i^k)) \mu^k(\xi_i^k) w_i^k + \sum_k \text{error}^k. \quad (5.2)$$

Note that each element  $\omega_k$  may in principle have its own reference element which means that quadrature points and weights as well as the transformation and integration element may depend on  $k$ . The total error is a sum of the errors on the individual elements.

In the following we show how the formula (5.2) can be realised within **DUNE**.



## 5.2 Functors

The function  $f$  is represented as a functor, i. e. a class having an `operator()` with appropriate arguments. A point  $x \in \Omega$  is represented by an object of type `FieldVector<ct,dim>` where `ct` is the type for each component of the vector and `dim` is its dimension.

**Listing 10 (dune-grid-howto/functors.hh)** Here are some examples for functors.

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_FUNCTORS_HH__
4 #define __DUNE_GRID_HOWTO_FUNCTORS_HH__
5
6 #include <dune/common/fvector.hh>
7 // a smooth function
8 template<typename ct, int dim>
9 class Exp {
10 public:
11     Exp () {midpoint = 0.5;}
12     double operator() (const Dune::FieldVector<ct,dim>& x) const
13     {
14         Dune::FieldVector<ct,dim> y(x);
15         y -= midpoint;
16         return exp(-3.234*(y*y));
17     }
18 private:
19     Dune::FieldVector<ct,dim> midpoint;
20 };
21
22 // a function with a local feature
23 template<typename ct, int dim>
24 class Needle {
25 public:
26     Needle ()
27     {
28         midpoint = 0.5;
29         midpoint[dim-1] = 1;
30     }
31     double operator() (const Dune::FieldVector<ct,dim>& x) const
32     {
33         Dune::FieldVector<ct,dim> y(x);
34         y -= midpoint;
35         return 1.0/(1E-4+y*y);
36     }
37 private:
38     Dune::FieldVector<ct,dim> midpoint;
39 };
40
41 #endif // __DUNE_GRID_HOWTO_FUNCTORS_HH__

```

## 5.3 Integration over a single element

The function `integrateentity` in the following listing computes the integral over a single element of the mesh with a quadrature rule of given order. This relates directly to formula (5.1) above.

**Listing 11 (dune-grid-howto/integrateentity.hh)**

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-

```

## 5 Quadrature rules

```

2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef DUNE_INTEGRATE_ENTITY_HH
4 #define DUNE_INTEGRATE_ENTITY_HH
5
6 #include <dune/common/exceptions.hh>
7 #include <dune/geometry/quadraturerules.hh>
8
9 //! compute integral of function over entity with given order
10 template<class Entity, class Function>
11 double integrateEntity (const Entity &entity, const Function &f, int p)
12 {
13     // dimension of the entity
14     const int dim = Entity::dimension;
15
16     // type used for coordinates in the grid
17     typedef typename Entity::ctype ctype;
18
19     // get geometry
20     const typename Entity::Geometry geometry = entity.geometry();
21
22     // get geometry type
23     const Dune::GeometryType gt = geometry.type();
24
25     // get quadrature rule of order p
26     const Dune::QuadratureRule<ctype,dim>&
27     rule = Dune::QuadratureRules<ctype,dim>::rule(gt,p);
28
29     // ensure that rule has at least the requested order
30     if (rule.order()<p)
31         DUNE_THROW(Dune::Exception,"order not available");
32
33     // compute approximate integral
34     double result=0;
35     for (typename Dune::QuadratureRule<ctype,dim>::const_iterator i=rule.begin();
36          i!=rule.end(); ++i)
37     {
38         double fval = f(geometry.global(i->position()));
39         double weight = i->weight();
40         double detjac = geometry.integrationElement(i->position());
41         result += fval * weight * detjac;
42     }
43
44     // return result
45     return result;
46 }
47
48 #endif

```

Line 27 extracts a reference to a `Dune::QuadratureRule` from the `Dune::QuadratureRules` singleton which is a container containing quadrature rules for all the different reference element types and different orders of approximation. Both classes are parametrized by dimension and the basic type used for the coordinate positions. `Dune::QuadratureRule` in turn is a container of `Dune::QuadraturePoint` supplying positions  $\xi_i$  and weights  $w_i$ .

Line 35 shows the loop over all quadrature points in the quadrature rules. For each quadrature point  $i$  the function value at the transformed position (line 38), the weight (line 39) and the integration element (line 40) are computed and summed (line 41).

## 5.4 Integration with global error estimation

In the listing below function `uniformintegration` computes the integral over the whole domain via formula (5.2) and in addition provides an estimate of the error. This is done as follows. Let  $I_c$  be the value of the numerically computed integral on some grid and let  $I_f$  be the value of the numerically computed integral on a grid where each element has been refined. Then

$$E \approx |I_f - I_c| \quad (5.3)$$

is an estimate for the error. If the refinement is such that every element is bisected in every coordinate direction, the function to be integrated is sufficiently smooth and the order of the quadrature rule is  $p + 1$ , then the error should be reduced by a factor of  $(1/2)^p$  after each mesh refinement.

### Listing 12 (dune-grid-howto/integration.cc)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 // $Id$
4
5 // Dune includes
6 #include "config.h" // file constructed by ./configure script
7 #include <dune/grid/sgrid.hh> // load sgrid definition
8 #include <dune/common/parallel/mpihelper.hh> // include mpi helper class
9
10 #include "functors.hh"
11 #include "integrateentity.hh"
12
13 //! uniform refinement test
14 template<class Grid>
15 void uniformintegration (Grid& grid)
16 {
17     // function to integrate
18     Exp<typename Grid::ctype, Grid::dimension> f;
19
20     // get GridView on leaf grid - type
21     typedef typename Grid :: LeafGridView GridView;
22
23     // get GridView instance
24     GridView gridView = grid.leafGridView();
25
26     // get iterator type
27     typedef typename GridView :: template Codim<0> :: Iterator LeafIterator;
28
29     // loop over grid sequence
30     double oldvalue=1E100;
31     for (int k=0; k<10; k++)
32     {
33         // compute integral with some order
34         double value = 0.0;
35         LeafIterator eendit = gridView.template end<0>();
36         for (LeafIterator it = gridView.template begin<0>(); it!=eendit; ++it)
37             value += integrateEntity(*it,f,1);
38
39         // print result and error estimate
40         std::cout << "elements="
41                 << std::setw(8) << std::right
42                 << gridView.size(0)
43                 << "┐integral="
44                 << std::scientific << std::setprecision(12)
45                 << value

```

## 5 Quadrature rules

```

46         << "␣error=" << std::abs(value-oldvalue)
47         << std::endl;
48
49     // save value of integral
50     oldvalue=value;
51
52     // refine all elements
53     grid.globalRefine(1);
54 }
55 }
56
57 int main(int argc, char **argv)
58 {
59     // initialize MPI, finalize is done automatically on exit
60     Dune::MPIHelper::instance(argc,argv);
61
62     // start try/catch block to get error messages from dune
63     try {
64         using namespace Dune;
65
66         // the GridSelector :: GridType is defined in gridtype.hh and is
67         // set during compilation
68         typedef GridSelector :: GridType Grid;
69
70         // use unitcube from grids
71         std::stringstream dgfFileName;
72         dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
73         << "unitcube" << Grid::dimension << ".dgf";
74
75         // create grid pointer
76         GridPtr<Grid> gridPtr( dgfFileName.str() );
77
78         // integrate and compute error with extrapolation
79         uniformintegration( *gridPtr );
80     }
81     catch (std::exception & e) {
82         std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
83         return 1;
84     }
85     catch (Dune::Exception & e) {
86         std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
87         return 1;
88     }
89     catch (...) {
90         std::cout << "Unknown␣ERROR" << std::endl;
91         return 1;
92     }
93
94     // done
95     return 0;
96 }

```

Running the executable `integration` on a `YaspGrid` in two space dimensions with a quadrature rule of order two the following output is obtained:

```

elements=      1  integral=1.000000000000e+00  error=1.000000000000e+100
elements=      4  integral=6.674772311008e-01  error=3.325227688992e-01
elements=     16  integral=6.283027311366e-01  error=3.917449996419e-02
elements=     64  integral=6.192294777551e-01  error=9.073253381426e-03
elements=    256  integral=6.170056966109e-01  error=2.223781144285e-03
elements=   1024  integral=6.164524949226e-01  error=5.532016882082e-04
elements=   4096  integral=6.163143653145e-01  error=1.381296081435e-04
elements=  16384  integral=6.162798435779e-01  error=3.452173662133e-05

```

## 5 Quadrature rules

```
elements= 65536 integral=6.162712138101e-01 error=8.629767731416e-06
elements= 262144 integral=6.162690564098e-01 error=2.157400356695e-06
elements= 1048576 integral=6.162685170623e-01 error=5.393474630244e-07
elements= 4194304 integral=6.162683822257e-01 error=1.348366243104e-07
```

The ratio of the errors on two subsequent grids nicely approaches the value  $1/4$  as the grid is refined.

**Exercise 5.1** Try different quadrature orders. For that just change the last argument of the call to `integrateentity` in line 37 in file `integration.cc`.

**Exercise 5.2** Try different grid implementations and dimensions and compare the run-time.

**Exercise 5.3** Try different integrands  $f$  and look at the development of the (estimated) error in the integral.

## 6 Attaching user data to a grid

In most useful applications there will be the need to associate user-defined data with certain entities of a grid. The standard example are, of course, the degrees of freedom of a finite element function. But it could be as simple as a boolean value that indicates whether an entity has already been visited by some algorithm or not. In this chapter we will show with some examples how arbitrary user data can be attached to a grid.

### 6.1 Mappers

The general situation is that a user wants to store some arbitrary data with a subset of the entities of a grid. Remember that entities are all the vertices, edges, faces, elements, etc., on all the levels of a grid.

An important design decision in the **DUNE** grid interface was that user-defined data is stored in user space. This has a number of implications:

- **DUNE** grid objects do not need to know anything about the user data.
- Data structures used in the implementation of a **DUNE** grid do not have to be extensible.
- Types representing the user data can be arbitrary.
- The user is responsible for possibly reorganizing the data when a grid is modified (i. e. refined, coarsened, load balanced).

Since efficiency is important in scientific computing the second important design decision was that user data is stored in arrays (or random access containers) and that the data is accessed via an index. The set of indices starts at zero and is consecutive.

Let us assume that the set of all entities in the grid is  $E$  and that  $E' \subseteq E$  is the subset of entities for which data is to be stored. E. g. this could be all the vertices in the leaf grid in the case of  $P_1$  finite elements. Then the access from grid entities to user data is a two stage process: A so-called *mapper* provides a map

$$m : E' \rightarrow I_{E'} \quad (6.1)$$

where  $I_{E'} = \{0, \dots, |E'| - 1\} \subset \mathbb{N}$  is the consecutive and zero-starting index set associated to the entity set. The user data  $D(E') = \{d_e \mid e \in E'\}$  is stored in an array, which is another map

$$a : I_{E'} \rightarrow D(E'). \quad (6.2)$$

In order to get the data  $d_e \in D(E')$  associated to entity  $e \in E'$  we therefore have to evaluate the two maps:

$$d_e = a(m(e)). \quad (6.3)$$

**DUNE** provides different implementations of mappers that differ in functionality and cost (with respect to storage and run-time). Basically there are two different kinds of mappers.

### Index based mappers

An index-based mapper is allocated for a grid and can be used as long as the grid is not changed (i.e. refined, coarsened or load balanced). The implementation of these mappers is based on a `Dune::IndexSet` and evaluation of the map  $m$  is typically of  $O(1)$  complexity with a very small constant. Index-based mappers are only available for restricted (but usually sufficient) entity sets. They will be used in the examples shown below.

### Id based mappers

Id-based mappers can also be used while a grid changes, i.e. it is ensured that the map  $m$  can still be evaluated for all entities  $e$  that are still in the grid after modification. For that it has to be implemented on the basis of a `Dune::IdSet`. This may be relatively slow because the data type used for ids is usually not an `int` and the non-consecutive ids require more complicated search data structures (typically a map). Evaluation of the map  $m$  therefore typically costs  $O(\log |E'|)$ . On the other hand, id-based mappers are not restricted to specific entity sets  $E'$ .

In adaptive applications one would use an index-based mapper to do in the calculations on a certain grid and only in the adaption phase an id-based mapper would be used to transfer the required data (e. g. only the finite element solution) from one grid to the next grid.

## 6.2 Visualization of discrete functions

Let us use mappers to evaluate a function  $f : \Omega \rightarrow \mathbb{R}$  for certain entities and store the values in a vector. Then, in order to do something useful, we use the vector to produce a graphical visualization of the function.

The first example evaluates the function at the centers of all elements of the leaf grid and stores this value. Here is the listing:

### Listing 13 (File `dune-grid-howto/elementdata.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_ELEMENT_DATA_HH
4 #define __DUNE_GRID_HOWTO_ELEMENT_DATA_HH
5
6 #include <dune/grid/common/mcmgmapper.hh>
7 #include <dune/grid/io/file/vtk/vtkwriter.hh>
8 #if HAVE_GRAPE
9 #include <dune/grid/io/visual/grapedatadisplay.hh>
10 #endif
11
12 //! Parameter for mapper class
13 /** This class is only here to show what such a class looks like — it does
14     exactly the same as Dune::MCMGElementLayout. */
15 template<int dimgrid>
16 struct P0Layout
17 {
18     bool contains (Dune::GeometryType gt)
19     {
20         if (gt.dim()==dimgrid) return true;
21         return false;
22     }

```

## 6 Attaching user data to a grid

```

23 };
24
25 // demonstrate attaching data to elements
26 template<class G, class F>
27 void elementdata (const G& grid, const F& f)
28 {
29     // the usual stuff
30     //const int dim = G::dimension;
31     const int dimworld = G::dimensionworld;
32     typedef typename G::ctype ct;
33     typedef typename G::LeafGridView GridView;
34     typedef typename GridView::template Codim<0>::Iterator ElementLeafIterator;
35     typedef typename ElementLeafIterator::Entity::Geometry LeafGeometry;
36
37     // get grid view on leaf part
38     GridView gridView = grid.leafGridView();
39
40     // make a mapper for codim 0 entities in the leaf grid
41     Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,POLayout>
42     mapper(grid);
43
44     // allocate a vector for the data
45     std::vector<double> c(mapper.size());
46
47     // iterate through all entities of codim 0 at the leaves
48     for (ElementLeafIterator it = gridView.template begin<0>();
49          it!=gridView.template end<0>(); ++it)
50     {
51         // cell geometry
52         const LeafGeometry geo = it->geometry();
53
54         // get global coordinate of cell center
55         Dune::FieldVector<ct,dimworld> global = geo.center();
56
57         // evaluate functor and store value
58         c[mapper.map(*it)] = f(global);
59     }
60
61     // generate a VTK file
62     // Dune::LeafP0Function<G,double> cc(grid,c);
63     Dune::VTKWriter<typename G::LeafGridView> vtkwriter(gridView);
64     vtkwriter.addCellData(c,"data");
65     vtkwriter.write( "elementdata", Dune::VTK::appendedraw );
66
67     // online visualization with Grape
68     #if HAVE_GRAPE
69     {
70         const int polynomialOrder = 0; // we piecewise constant data
71         const int dimRange = 1; // we have scalar data here
72         // create instance of data display
73         Dune::GrapeDataDisplay<G> grape(grid);
74         // display data
75         grape.displayVector("concentration", // name of data that appears in grape
76                             c, // data vector
77                             gridView.indexSet(), // used index set
78                             polynomialOrder, // polynomial order of data
79                             dimRange); // dimRange of data
80     }
81     #endif
82 }
83
84 #endif // _DUNE_GRID.HOWTO_ELEMENT_DATA_HH

```



The class template `Dune::LeafMultipleCodimMultipleGeomTypeMapper` provides an index-based mapper where the entities in the subset  $E'$  are all leaf entities and can further be selected depending on the codimension and the geometry type. To that end the second template argument has to be a class template with one integer template parameter containing a method `contains`. Just look at the example `P0Layout`. When the method `contains` returns true for a combination of dimension, codimension and geometry type then all leaf entities with that dimension, codimension and geometry type will be in the subset  $E'$ . The mapper object is constructed in line 42. A similar mapper is available also for the entities of a grid level.

The data vector is allocated in line 45. Here we use a `std::vector<double>`. The `size()` method of the mapper returns the number of entities in the set  $E'$ . Instead of the STL vector one can use any other type with an `operator[]`, even built-in arrays (however, built-in arrays will not work in this example because the VTK output below requires a container with a `size()` method).

Now the loop in lines 48-59 iterates through all leaf elements. The next three statements within the loop body compute the position of the center of the element in global coordinates. Then the essential statement is in line 58 where the function is evaluated and the value is assigned to the corresponding entry in the `c` array. The evaluation of the map  $m$  is performed by `mapper.map(*it)` where `*it` is the entity which is passed as a const reference to the mapper.

The remaining lines of code produce graphical output. Lines 63-65 produce an output file for the Visualization Toolkit (VTK), [7], in its XML format. If the grid is distributed over several processes the `Dune::VTKWriter` produces one file per process and the corresponding XML metafile. Using Paraview, [6], you can visualize these files. Lines 68-81 enable online interactive visualization with the Grape, [5], graphics package, if it is installed on your machine.

The next list shows a function `vertexdata` that does the same job except that the data is associated with the vertices of the grid.

#### Listing 14 (File `dune-grid-howto/vertexdata.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_VERTEXDATA_HH__
4 #define __DUNE_GRID_HOWTO_VERTEXDATA_HH__
5
6 #include <dune/grid/common/mcmgmapper.hh>
7 #include <dune/grid/io/file/vtk/vtkwriter.hh>
8 #if HAVE_GRAPE
9 #include <dune/grid/io/visual/grapedatadisplay.hh>
10 #endif
11
12 //! Parameter for mapper class
13 /** This class is only here to show what such a class looks like — it does
14     exactly the same as Dune::MCMGVertexLayout. */
15 template<int dimgrid>
16 struct P1Layout
17 {
18     bool contains (Dune::GeometryType gt)
19     {
20         if (gt.dim()==0) return true;
21         return false;
22     }
23 };
24
25 // demonstrate attaching data to elements
26 template<class G, class F>
27 void vertexdata (const G& grid, const F& f)

```

## 6 Attaching user data to a grid

```

28 {
29 // get dimension and coordinate type from Grid
30 const int dim = G::dimension;
31 typedef typename G::ctype ct;
32 typedef typename G::LeafGridView GridView;
33 // determine type of LeafIterator for codimension = dimension
34 typedef typename GridView::template Codim<dim>::Iterator VertexLeafIterator;
35
36 // get grid view on the leaf part
37 GridView gridView = grid.leafGridView();
38
39 // make a mapper for codim 0 entities in the leaf grid
40 Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P1Layout>
41 mapper(grid);
42
43 // allocate a vector for the data
44 std::vector<double> c(mapper.size());
45
46 // iterate through all entities of codim 0 at the leaves
47 for (VertexLeafIterator it = gridView.template begin<dim>();
48      it!=gridView.template end<dim>(); ++it)
49 {
50     // evaluate functor and store value
51     c[mapper.map(*it)] = f(it->geometry().corner(0));
52 }
53
54 // generate a VTK file
55 // Dune::LeafP1Function<G,double> cc(grid,c);
56 Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafGridView());
57 vtkwriter.addVertexData(c,"data");
58 vtkwriter.write( "vertexdata", Dune::VTK::appenddraw );
59
60 // online visualization with Grape
61 #if HAVE_GRAPE
62 {
63     const int polynomialOrder = 1; // we piecewise linear data
64     const int dimRange = 1; // we have scalar data here
65     // create instance of data display
66     Dune::GrapeDataDisplay<G> grape(grid);
67     // display data
68     grape.displayVector("concentration", // name of data that appears in grape
69                        c, // data vector
70                        gridView.indexSet(), // used index set
71                        polynomialOrder, // polynomial order of data
72                        dimRange); // dimRange of data
73 }
74 #endif
75 }
76 #endif // _DUNE_GRID_HOWTO_VERTEXDATA_HH_

```

The differences to the `elementdata` example are the following:

- In the `P1Layout` struct the method `contains` returns true if `codim==dim`.
- Use a leaf iterator for codimension `dim` instead of 0.
- Evaluate the function at the vertex position which is directly available via `it->geometry()[0]`.
- Use `addVertexData` instead of `addCellData` on the `Dune::VTKWriter`.
- Pass `polynomialOrder=1` instead of 0 as the second last argument of `grape.displayVector`. This argument is the polynomial degree of the approximation.

Finally the following listing shows the main program that calls the two functions just discussed:

**Listing 15 (File dune-grid-howto/visualization.cc)**

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 // $Id$
4
5 #include <config.h>
6 #include <iostream>
7 #include <iomanip>
8 #include <stdio.h>
9 #include <dune/common/parallel/mpihelper.hh> // include mpi helper class
10 #include <dune/grid/io/file/dgfparsers/dgfparsers.hh>
11
12
13 #include "elementdata.hh"
14 #include "vertexdata.hh"
15 #include "functors.hh"
16 #include "unitcube.hh"
17
18
19 #ifdef GRIDDIM
20 const int dimGrid = GRIDDIM;
21 #else
22 const int dimGrid = 2;
23 #endif
24
25
26 //! supply functor
27 template<class Grid>
28 void dowork ( Grid &grid, int refSteps = 5 )
29 {
30     // make function object
31     Exp<typename Grid::ctype, Grid::dimension> f;
32
33     // refine the grid
34     grid.globalRefine( refSteps );
35
36     // call the visualization functions
37     elementdata(grid,f);
38     vertexdata(grid,f);
39 }
40
41 int main(int argc, char **argv)
42 {
43     // initialize MPI, finalize is done automatically on exit
44     Dune::MPIHelper::instance(argc,argv);
45
46     // start try/catch block to get error messages from dune
47     try
48     {
49         if( argc > 1 )
50         {
51             typedef Dune::GridSelector::GridType DGFGGridType;
52             // create grid pointer
53             Dune::GridPtr< DGFGGridType > gridPtr( argv[ 1 ] );
54             dowork( *gridPtr, 3 );
55         }
56
57         /*
58             UnitCube<Dune::OneDGrid,1> uc0;
59             UnitCube<Dune::YaspGrid<dimGrid>,1> uc1;
60
```

## 6 Attaching user data to a grid

```

61     #if HAVE_UG
62     UnitCube< Dune::UGGrid< dimGrid >, 2 > uc2;
63     dowork( uc2.grid(), 3 );
64     #endif
65
66     #if HAVE_ALBERTA
67     {
68     UnitCube< Dune::AlbertaGrid< dimGrid, dimGrid >, 1 > unitcube;
69     // note: The 3d cube cannot be bisected recursively
70     dowork( unitcube.grid(), (dimGrid < 3 ? 6 : 0) );
71     }
72     #endif // #if HAVE_ALBERTA
73 */
74
75     UnitCube< Dune::SGrid< dimGrid, dimGrid >, 1 > uc4;
76     dowork( uc4.grid(), 3 );
77
78 #if HAVE_ALUGRID
79     UnitCube< Dune::ALUGrid< dimGrid, dimGrid, Dune::simplex,
80             Dune::nonconforming >, 1 > uc5;
81     dowork( uc5.grid(), 3 );
82
83 #if GRIDDIM == 2 || GRIDDIM == 3
84     UnitCube< Dune::ALUGrid< dimGrid, dimGrid, Dune::cube,
85             Dune::nonconforming >, 1 > uc6;
86     dowork( uc6.grid(), 3 );
87 #endif // #if GRIDDIM == 2 || GRIDDIM == 3
88 #endif // #if HAVE_ALUGRID
89 }
90 catch (std::exception & e) {
91     std::cout << "STL_ERROR:" << e.what() << std::endl;
92     return 1;
93 }
94 catch (Dune::Exception & e) {
95     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
96     return 1;
97 }
98 catch (...) {
99     std::cout << "Unknown_ERROR" << std::endl;
100    return 1;
101 }
102
103 // done
104 return 0;
105 }

```

### 6.3 Cell centered finite volumes

In this section we show a first complete example for the numerical solution of a partial differential equation (PDE), although a very simple one.

We will solve the linear hyperbolic PDE

$$\frac{\partial c}{\partial t} + \nabla \cdot (uc) = 0 \quad \text{in } \Omega \times T \quad (6.4)$$

where  $\Omega \subset \mathbb{R}^d$  is a domain,  $T = (0, t_{\text{end}})$  is a time interval,  $c : \Omega \times T \rightarrow \mathbb{R}$  is the unknown concentration and  $u : \Omega \times T \rightarrow \mathbb{R}^d$  is a given velocity field. We require that the velocity field is divergence free for

all times. The equation is subject to the initial condition

$$c(x, 0) = c_0(x) \quad x \in \Omega \quad (6.5)$$

and the boundary condition

$$c(x, t) = b(x, t) \quad t > 0, x \in \Gamma_{\text{in}}(t) = \{y \in \partial\Omega \mid u(y, t) \cdot \nu(y) < 0\}. \quad (6.6)$$

Here  $\nu(x)$  is the unit outer normal at a point  $y \in \partial\Omega$  and  $\Gamma_{\text{in}}(t)$  is the inflow boundary at time  $t$ .

### 6.3.1 Numerical Scheme

To keep the presentation simple we use a cell-centered finite volume discretization in space, full upwind evaluation of the fluxes and an explicit Euler scheme in time.

The grid consists of cells (elements)  $\omega$  and the time interval  $T$  is discretized into discrete steps  $0 = t_0, t_1, \dots, t_n, t_{n+1}, \dots, t_N = t_{\text{end}}$ . Cell centered finite volume schemes integrate the PDE (6.4) over a cell  $\omega_i$  and a time interval  $(t_n, t_{n+1})$ :

$$\int_{\omega_i} \int_{t_n}^{t_{n+1}} \frac{\partial c}{\partial t} dt dx + \int_{\omega_i} \int_{t_n}^{t_{n+1}} \nabla \cdot (uc) dt dx = 0 \quad \forall i. \quad (6.7)$$

Using integration by parts we arrive at

$$\int_{\omega_i} c(x, t_{n+1}) dx - \int_{\omega_i} c(x, t_n) dx + \int_{t_n}^{t_{n+1}} \int_{\partial\omega_i} cu \cdot \nu ds dt = 0 \quad \forall i. \quad (6.8)$$

Now we approximate  $c$  by a cell-wise constant function  $C$ , where  $C_i^n$  denotes the value in cell  $\omega_i$  at time  $t_n$ . Moreover we subdivide the boundary  $\partial\omega_i$  into facets  $\gamma_{ij}$  which are either intersections with other cells  $\partial\omega_i \cap \partial\omega_j$ , or intersections with the boundary  $\partial\omega_i \cap \partial\Omega$ . Evaluation of the fluxes at time level  $t_n$  leads to the following equation for the unknown cell values at  $t_{n+1}$ :

$$C_i^{n+1}|\omega_i| - C_i^n|\omega_i| + \sum_{\gamma_{ij}} \phi(C_i^n, C_j^n, u_{ij}^n \cdot \nu_{ij}; \gamma_{ij}, t_n) |\gamma_{ij}| \Delta t^n = 0 \quad \forall i, \quad (6.9)$$

where  $\Delta t^n = t_{n+1} - t_n$ ,  $u_{ij}^n$  is the velocity on the facet  $\gamma_{ij}$  at time  $t_n$ ,  $\nu_{ij}$  is the unit outer normal of the facet  $\gamma_{ij}$  and  $\phi$  is the flux function defined as

$$\phi(C_i^n, C_j^n, u_{ij}^n \cdot \nu_{ij}; \gamma_{ij}, t_n) = \begin{cases} b(\gamma_{ij}) u_{ij}^n \cdot \nu_{ij} & \gamma_{ij} \subset \Gamma_{\text{in}}(t) \\ C_j^n u_{ij}^n \cdot \nu_{ij} & \gamma_{ij} = \partial\omega_i \cap \partial\omega_j \wedge u_{ij}^n \cdot \nu_{ij} < 0 \\ C_i^n u_{ij}^n \cdot \nu_{ij} & u_{ij}^n \cdot \nu_{ij} \geq 0 \end{cases} \quad (6.10)$$

Here  $b(\gamma_{ij})$  denotes evaluation of the boundary condition on an inflow facet  $\gamma_{ij}$ . If we formally set  $C_j^n = b(\gamma_{ij})$  on an inflow facet  $\gamma_{ij} \subset \Gamma_{\text{in}}(t)$  we can derive the following shorthand notation for the flux function:

$$\phi(C_i^n, C_j^n, u_{ij}^n \cdot \nu_{ij}; \gamma_{ij}, t_n) = C_i^n \max(0, u_{ij}^n \cdot \nu_{ij}) - C_j^n \max(0, -u_{ij}^n \cdot \nu_{ij}). \quad (6.11)$$

Inserting this into (6.9) and solving for  $C_i^{n+1}$  we obtain

$$C_i^{n+1} = C_i^n \left( 1 - \Delta t^n \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^n \cdot \nu_{ij}) \right) + \Delta t^n \sum_{\gamma_{ij}} C_j^n \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, -u_{ij}^n \cdot \nu_{ij}) \quad \forall i. \quad (6.12)$$

One can show that the scheme is stable provided the following condition holds:

$$\forall i : 1 - \Delta t^n \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^n \cdot \nu_{ij}) \geq 0 \Leftrightarrow \Delta t^n \leq \min_i \left( \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} \max(0, u_{ij}^n \cdot \nu_{ij}) \right)^{-1}. \quad (6.13)$$

When we rewrite 6.12 in the form

$$C_i^{n+1} = C_i^n - \underbrace{\Delta t^n \sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|\omega_i|} (C_i^n \max(0, u_{ij}^n \cdot \nu_{ij}) + C_j^n \max(0, -u_{ij}^n \cdot \nu_{ij}))}_{\delta_i} \quad \forall i \quad (6.14)$$

then it becomes clear that the optimum time step  $\Delta t^n$  and the update  $\delta_i$  for each cell can be computed in a single iteration over the grid. The computation  $C^{n+1} = C^n - \Delta t^n \delta$  can then be realized with a simple vector update. In this form, the algorithm can also be parallelized in a straightforward way.

### 6.3.2 Implementation

First, we need to specify the problem parameters, i.e. initial condition, boundary condition and velocity field. This is done by the following functions.

#### Listing 16 (File dune-grid-howto/transportproblem.hh)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_TRANSPORTPROBLEM_HH__
4 #define __DUNE_GRID_HOWTO_TRANSPORTPROBLEM_HH__
5
6 #include <dune/common/fvector.hh>
7 // the initial condition c0
8 template<int dimworld, class ct>
9 double c0 (const Dune::FieldVector<ct,dimworld>& x)
10 {
11     Dune::FieldVector<ct,dimworld> y(0.25);
12     y -= x;
13     if (y.two_norm() < 0.125)
14         return 1.0;
15     else
16         return 0.0;
17 }
18
19 // the boundary condition b on inflow boundary
20 template<int dimworld, class ct>
21 double b (const Dune::FieldVector<ct,dimworld>& x, double t)
22 {
23     return 0.0;
24 }
25
26 // the vector field u is returned in r
    
```

## 6 Attaching user data to a grid

```
27 template<int dimworld, class ct>
28 Dune::FieldVector<double,dimworld> u (const Dune::FieldVector<ct,dimworld>& x, double t)
29 {
30     Dune::FieldVector<double,dimworld> r(0.5);
31     r[0] = 1.0;
32     return r;
33 }
34 #endif // _DUNE_GRID_HOWTO_TRANSPORTPROBLEM2_HH_
```

The initialization of the concentration vector with the initial condition should also be straightforward now. The function `initialize` works on a concentration vector `c` that can be stored in any container type with a vector interface (`operator[]`, `size()` and copy constructor are needed). Moreover the grid and a mapper for element-wise data have to be passed as well.

### Listing 17 (File `dune-grid-howto/initialize.hh`)

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_INITIALIZE_HH__
4 #define __DUNE_GRID_HOWTO_INITIALIZE_HH__
5
6 #include <dune/common/fvector.hh>
7
8 //! initialize the vector of unknowns with initial value
9 template<class G, class M, class V>
10 void initialize (const G& grid, const M& mapper, V& c)
11 {
12     // first we extract the dimensions of the grid
13     //const int dim = G::dimension;
14     const int dimworld = G::dimensionworld;
15
16     // type used for coordinates in the grid
17     typedef typename G::ctype ct;
18
19     // type of grid view on leaf part
20     typedef typename G::LeafGridView GridView;
21
22     // leaf iterator type
23     typedef typename GridView::template Codim<0>::Iterator LeafIterator;
24
25     // geometry type
26     typedef typename LeafIterator::Entity::Geometry Geometry;
27
28     // get grid view on leaf part
29     GridView gridView = grid.leafGridView();
30
31     // iterate through leaf grid and evaluate c0 at cell center
32     LeafIterator endit = gridView.template end<0>();
33     for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
34     {
35         // get geometry
36         const Geometry geo = it->geometry();
37
38         // get global coordinate of cell center
39         Dune::FieldVector<ct,dimworld> global = geo.center();
40
41         // initialize cell concentration
42         c[mapper.map(*it)] = c0(global);
43     }
44 }
45
46 #endif // __DUNE_GRID_HOWTO_INITIALIZE_HH__
```

The main work is now done in the function which implements the evolution (6.14) with optimal time step control via (6.13). In addition to grid, mapper and concentration vector the current time  $t_n$  is passed and the optimum time step  $\Delta t^n$  selected by the algorithm is returned.

#### Listing 18 (File dune-grid-howto/evolve.hh)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_EVOLVE_HH__
4 #define __DUNE_GRID_HOWTO_EVOLVE_HH__
5
6 #include <dune/common/fvector.hh>
7
8 template<class G, class M, class V>
9 void evolve (const G& grid, const M& mapper, V& c, double t, double& dt)
10 {
11     // first we extract the dimensions of the grid
12     const int dimworld = G::dimensionworld;
13
14     // type used for coordinates in the grid
15     typedef typename G::ctype ct;
16
17     // type of grid view on leaf part
18     typedef typename G::LeafGridView GridView;
19
20     // element iterator type
21     typedef typename GridView::template Codim<0>::Iterator LeafIterator;
22
23     // leaf entity geometry
24     typedef typename LeafIterator::Entity::Geometry LeafGeometry;
25
26     // intersection iterator type
27     typedef typename GridView::IntersectionIterator IntersectionIterator;
28
29     // intersection geometry
30     typedef typename IntersectionIterator::Intersection::Geometry IntersectionGeometry;
31
32     // entity pointer type
33     typedef typename G::template Codim<0>::EntityPointer EntityPointer;
34
35     // get grid view on leaf part
36     GridView gridView = grid.leafGridView();
37
38     // allocate a temporary vector for the update
39     V update(c.size());
40     for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
41
42     // initialize dt very large
43     dt = 1E100;
44
45     // compute update vector and optimum dt in one grid traversal
46     LeafIterator endit = gridView.template end<0>();
47     for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
48     {
49         // cell geometry
50         const LeafGeometry geo = it->geometry();
51
52
53         // cell volume, assume linear map here
54         double volume = geo.volume();
55
56         // cell index
57         int indexi = mapper.map(*it);

```



## 6 Attaching user data to a grid

```

58
59 // variable to compute sum of positive factors
60 double sumfactor = 0.0;
61
62 // run through all intersections with neighbors and boundary
63 IntersectionIterator isend = gridView.iend(*it);
64 for (IntersectionIterator is = gridView.ibegin(*it); is!=isend; ++is)
65 {
66     // get geometry type of face
67     const IntersectionGeometry igeo = is->geometry();
68
69     // get normal vector scaled with volume
70     Dune::FieldVector<ct,dimworld> integrationOuterNormal
71     = is->centerUnitOuterNormal();
72     integrationOuterNormal *= igeo.volume();
73
74     // center of face in global coordinates
75     Dune::FieldVector<ct,dimworld> faceglobal = igeo.center();
76
77     // evaluate velocity at face center
78     Dune::FieldVector<double,dimworld> velocity = u(faceglobal,t);
79
80     // compute factor occuring in flux formula
81     double factor = velocity*integrationOuterNormal/volume;
82
83     // for time step calculation
84     if (factor>=0) sumfactor += factor;
85
86     // handle interior face
87     if (is->neighbor()) // "correct" version
88     {
89         // access neighbor
90         EntityPointer outside = is->outside();
91         int indexj = mapper.map(*outside);
92
93         // compute flux from one side only
94         // this should become easier with the new IntersectionIterator functionality!
95         if ( it->level()>outside->level() ||
96             (it->level()==outside->level() && indexi<indexj) )
97         {
98             // compute factor in neighbor
99             const LeafGeometry nbgeo = outside->geometry();
100             double nbvolume = nbgeo.volume();
101             double nbfactor = velocity*integrationOuterNormal/nbvolume;
102
103             if (factor<0) // inflow
104             {
105                 update[indexi] -= c[indexj]*factor;
106                 update[indexj] += c[indexj]*nbfactor;
107             }
108             else // outflow
109             {
110                 update[indexi] -= c[indexi]*factor;
111                 update[indexj] += c[indexi]*nbfactor;
112             }
113         }
114     }
115
116     // handle boundary face
117     if (is->boundary())
118     {
119         if (factor<0) // inflow, apply boundary condition
120             update[indexi] -= b(faceglobal,t)*factor;

```

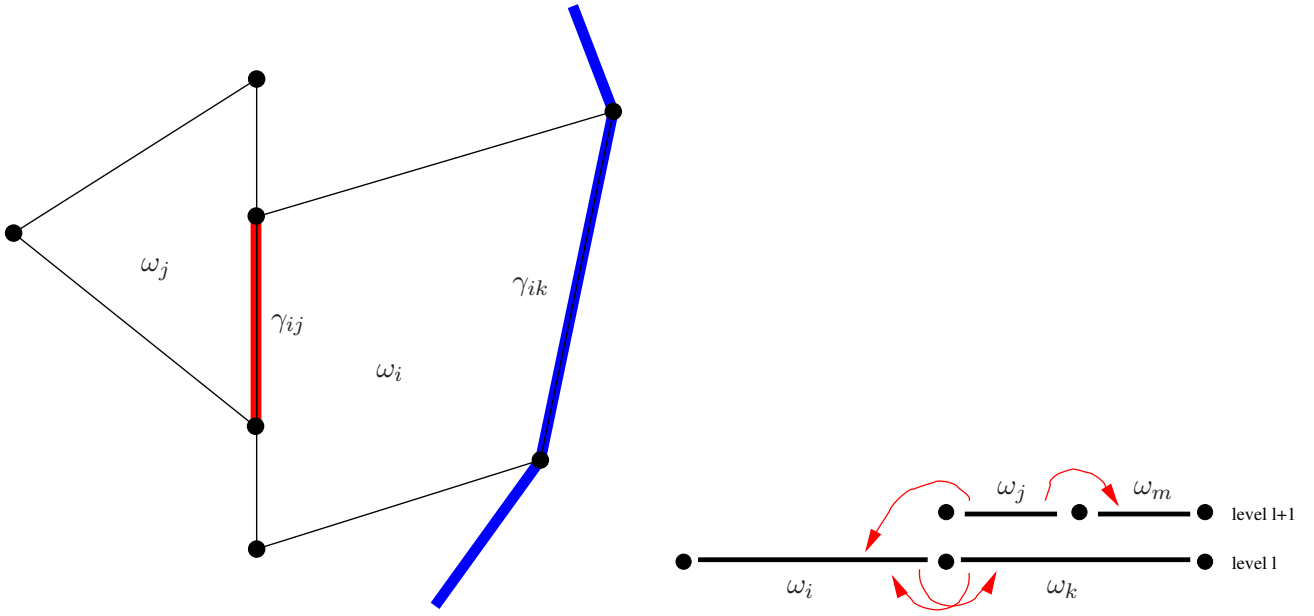


Figure 6.1: Left: intersection with other elements and the boundary, right: intersections in the case of locally refined grids.

```

121         else // outflow
122             update[indexi] -= c[indexi]*factor;
123         }
124     } // end all intersections
125
126     // compute dt restriction
127     dt = std::min(dt, 1.0/sumfactor);
128
129 } // end grid traversal
130
131 // scale dt with safety factor
132 dt *= 0.99;
133
134 // update the concentration vector
135 for (unsigned int i=0; i<c.size(); ++i)
136     c[i] += dt*update[i];
137
138 return;
139 }
140
141 #endif // _DUNE_GRID_HOWTO_EVOLVE_HH_

```

Lines 46-129 contain the loop over all leaf elements where the optimum  $\Delta t^n$  and the cell updates  $\delta_i$  are computed. The update vector is allocated in line 39, where we assume that **V** is a container with copy constructor and size method.

The computation of the fluxes is done in lines 63-124. An **IntersectionIterator** is used to access all intersections  $\gamma_{ij}$  of a leaf element  $\omega_i$ . For a full documentation on the **Intersection** class, we refer to the doxygen module page on Intersections<sup>1</sup> An **Intersection** is with another element  $\omega_j$  if the

<sup>1</sup>[http://www.dune-project.org/doc/doxygen/html/classDune\\_1\\_1IntersectionIterator.html](http://www.dune-project.org/doc/doxygen/html/classDune_1_1IntersectionIterator.html)

`neighbor()` method of the iterator returns true (line 87) or with the external boundary if `boundary()` returns true (line 117), see also left part of Figure 6.1. An intersection  $\gamma_{ij}$  is described by several mappings: (i) from a reference element of the intersection (with a dimension equal to the grid's dimension minus 1) to the reference elements of the two elements  $\omega_i$  and  $\omega_j$  and (ii) from a reference element of the intersection to the global coordinate system (with the world dimension). If an intersection is with another element then the `outside()` method returns an `EntityPointer` to an entity of codimension 0.

In the case of a locally refined grid special care has to be taken in the flux evaluation because the intersection iterator is not symmetric. This is illustrated for a one-dimensional situation in the right part of Figure 6.1. Element  $\omega_j$  is a leaf element on level  $l+1$ . The intersection iterator on  $\omega_j$  delivers two intersections, one with  $\omega_i$  which is on level  $l$  and one with  $\omega_m$  which is also on level  $l+1$ . However, the intersection iterator started on  $\omega_i$  will deliver an intersection with  $\omega_k$  and one with the external boundary (which is not shown). This means that the correct flux for the intersection  $\partial\omega_i \cap \partial\omega_j$  can only be evaluated from the intersection  $\gamma_{ji}$  visited by the intersection iterator started on  $\omega_j$ , because only there the two concentration values  $C_j$  and  $C_i$  are both accessibly. Note also that the outside element delivered by an intersection iterator need not be a leaf element (such as  $\omega_k$ ).

Therefore, in the code it is first checked that the outside element is actually a leaf element (line 89). Then the flux can be evaluated if the level of the outside element is smaller than that of the element where the intersection iterator was started (this corresponds the the situation of  $\omega_j$  referring to  $\omega_i$  in the right part of Figure 6.1) or when the levels are equal and the index of the outside element is larger. The latter condition with the indices just ensures that the flux is only computed once.

The  $\Delta t^n$  calculation is done in line 127 where the minimum over all cells is taken. Then, line 132 multiplies the optimum  $\Delta t^n$  with a safety factor to avoid any instability due to round-off errors.

Finally, line 136 computes the new concentration by adding the scaled update to the current concentration.

The function `vtkout` in the following listing provides an output of the grid and the solution using the Visualization Toolkit's [7] XML file format.

#### Listing 19 (File `dune-grid-howto/vtkout.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_VTKOUT_HH__
4 #define __DUNE_GRID_HOWTO_VTKOUT_HH__
5
6 #include <dune/grid/io/file/vtk/vtkwriter.hh>
7 #include <stdio.h>
8
9 template<class G, class V>
10 void vtkout (const G& grid, const V& c, const char* name, int k, double time=0.0, int rank=0)
11 {
12     Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafGridView());
13     char fname[128];
14     char sername[128];
15     sprintf(fname, "%s-%05d", name, k);
16     sprintf(sername, "%s.series", name);
17     vtkwriter.addCellData(c, "celldata");
18     vtkwriter.write( fname, Dune::VTK::ascii );
19
20     if ( rank == 0 )
21     {
22         std::ofstream serstream(sername, (k==0 ? std::ios_base::out : std::ios_base::app));

```

## 6 Attaching user data to a grid

```
23     serstream << k << "_" << fname << ".vtu_" << time << std::endl;
24     serstream.close();
25 }
26 }
27
28 #endif // _DUNE_GRID_HOWTO_VTKOUT_HH_
```

In addition to the snapshots that are produced at each timestep, this function also generates a series file which stores the actual time of an evolution scheme together with the snapshots' filenames. After executing the shell script `writePVD` on this series file, we get a Paraview Data (PVD) file with the same name as the snapshots. This file opened with paraview then gives us a neat animation over the time period.

Finally, the main program:

### Listing 20 (File `dune-grid-howto/finitevolume.cc`)

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #include "config.h" // know what grids are present
4 #include <iostream> // for input/output to shell
5 #include <fstream> // for input/output to files
6 #include <vector> // STL vector class
7 #include <dune/grid/common/mcmgmapper.hh> // mapper class
8 #include <dune/common/parallel/mpihelper.hh> // include mpi helper class
9
10 #include "vtkout.hh"
11 #include "transportproblem2.hh"
12 #include "initialize.hh"
13 #include "evolve.hh"
14
15 //=====
16 // the time loop function working for all types of grids
17 //=====
18
19 template<class G>
20 void timeloop (const G& grid, double tend)
21 {
22     // make a mapper for codim 0 entities in the leaf grid
23     Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,Dune::MCMGElementLayout>
24     mapper(grid);
25
26     // allocate a vector for the concentration
27     std::vector<double> c(mapper.size());
28
29     // initialize concentration with initial values
30     initialize(grid,mapper,c);
31     vtkout(grid,c,"concentration",0,0.0);
32
33     // now do the time steps
34     double t=0,dt;
35     int k=0;
36     const double saveInterval = 0.1;
37     double saveStep = 0.1;
38     int counter = 1;
39
40     while (t<tend)
41     {
42         // augment time step counter
43         ++k;
44     }
```

## 6 Attaching user data to a grid

```
45 // apply finite volume scheme
46 evolve(grid,mapper,c,t,dt);
47
48 // augment time
49 t += dt;
50
51 // check if data should be written
52 if (t >= saveStep)
53 {
54     // write data
55     vtkout(grid,c,"concentration",counter,t);
56
57     // increase counter and saveStep for next interval
58     saveStep += saveInterval;
59     ++counter;
60 }
61
62 // print info about time, timestep size and counter
63 std::cout << "s=" << grid.size(0)
64           << "k=" << k << "t=" << t << "dt=" << dt << std::endl;
65 }
66
67 // output results
68 vtkout(grid,c,"concentration",counter,tend);
69 }
70
71 //=====
72 // The main function creates objects and does the time loop
73 //=====
74
75 int main (int argc , char ** argv)
76 {
77     // initialize MPI, finalize is done automatically on exit
78     Dune::MPIHelper::instance(argc,argv);
79
80     // start try/catch block to get error messages from dune
81     try {
82         using namespace Dune;
83
84         // the GridSelector :: GridType is defined in gridtype.hh and is
85         // set during compilation
86         typedef GridSelector :: GridType Grid;
87
88         // use unitcube from dgf grids
89         std::stringstream dgfFileName;
90         dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
91                   << "unitcube" << Grid::dimension << ".dgf";
92
93         // create grid pointer
94         GridPtr<Grid> gridPtr( dgfFileName.str() );
95
96         // grid reference
97         Grid& grid = *gridPtr;
98
99         // half grid width 4 times
100        int level = 4 * DGFGGridInfo<Grid>::refineStepsForHalf();
101
102        // refine grid until upper limit of level
103        grid.globalRefine(level);
104
105        // do time loop until end time 0.5
106        timeloop(grid, 0.5);
107    }
```

```

108 catch (std::exception & e) {
109     std::cout << "STL_ERROR:" << e.what() << std::endl;
110     return 1;
111 }
112 catch (Dune::Exception & e) {
113     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
114     return 1;
115 }
116 catch (...) {
117     std::cout << "Unknown_ERROR" << std::endl;
118     return 1;
119 }
120
121 // done
122 return 0;
123 }

```

The function `timeloop` constructs a mapper and allocates the concentration vector with one entry per element in the leaf grid. In line 30 this vector is initialized with the initial concentration and the loop in line 40-65 evolves the concentration in time. Finally, the simulation result is written to a file in line 68.

## 6.4 A FEM example: The Poisson equation

In this section we will put together our knowledge about the **DUNE** grid interface acquired in previous chapters to solve the Poisson equation with Dirichlet boundary conditions on the domain  $\Omega = (0, 1)^d$ :

$$-\Delta u = f \text{ in } \Omega \quad (6.15)$$

$$u = 0 \text{ on } \partial\Omega \quad (6.16)$$

The equation will be solved using P1-Finite-Elements on a simplicial grid. The implementation aims to be easy to understand and yet show the power of the **DUNE** grid interface and its generic approach.

The starting point of the Finite Element Method is the variational formulation of 6.15, which is obtained by partial integration:

$$\underbrace{\int_{\Omega} \nabla u \cdot \nabla v dx}_{=:a(u,v)} = \underbrace{\int_{\Omega} f v dx}_{=:l(v)} \quad v \in V_h \quad (6.17)$$

Let now  $\mathcal{T}$  be a conforming triangulation of the domain  $\Omega$  with simplices:

- (i)  $\bigcup_{\Delta \in \mathcal{T}} \overline{\Delta} = \overline{\Omega}$
- (ii)  $\Delta_i \cap \Delta_j$   $i \neq j$  is an entity of higher codimension of the elements  $\Delta_i, \Delta_j$

As we want to use linear finite elements we choose our test function space to be

$$V_h = \{u \in C(\bar{\Omega}) \mid u|_{\Delta} \in \mathcal{P}_1(\Delta) \ \forall \Delta \in \mathcal{T}\} \quad (6.18)$$

We will not incorporate the Dirichlet boundary conditions into this function space. Instead, we will implement them in an easier way as described later on.

## 6 Attaching user data to a grid

As a basis  $\phi_1, \dots, \phi_N$  of  $V_h$  we choose the nodal basis - providing us small supports and thus a sparse stiffness matrix. After transformation onto the reference element we can use the shape functions

$$N_0(x) = 1 - \sum_{i=1}^d x_i \quad (6.19)$$

$$N_i(x) = x_i \quad i = 1, \dots, d \quad (6.20)$$

to evaluate the basis functions and their gradients.

The numerical solution  $u_h$  is a linear combination of  $\phi_1, \dots, \phi_N$  with coefficients  $u_1, \dots, u_N$ . We assemble the stiffness Matrix  $A$  and the vector  $b$ :

$$A_{ij} = a(\phi_i, \phi_j) \quad b_i = \ell(\phi_i) \quad (6.21)$$

The coefficients  $u_1, \dots, u_N$  are then obtained by solving  $Au = b$ .

The integrals are transformed onto the reference element  $\hat{\Delta}$  and computed with an appropriate quadrature rule. Let the transformation map be given by  $g : \hat{\Delta} \rightarrow \Delta$

$$A_{ij} = \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j \, dx \quad (6.22)$$

$$= \sum_{\Delta \in \mathcal{T}} \int_{\Delta} \nabla \phi_i \cdot \nabla \phi_j \, dx \quad (6.23)$$

$$= \sum_{\Delta \in \mathcal{T}} \int_{\hat{\Delta}} \nabla \phi_i(g(\hat{x})) \cdot \nabla \phi_j(g(\hat{x})) \mu(\hat{x}) d\hat{x} \quad (6.24)$$

$$= \sum_{\Delta \in \mathcal{T}} \int_{\hat{\Delta}} (J_g^{-T} \hat{\nabla} \phi_i)(\hat{x}) \cdot (J_g^{-T} \hat{\nabla} \phi_j)(\hat{x}) \mu(\hat{x}) d\hat{x} \quad (6.25)$$

$J_g$  is the Jacobian of the map  $g$  and  $\mu(\hat{x}) := \sqrt{\det J_g^T J_g}$  the Jacobian determinant. Let now  $\xi_k$  be the quadrature points of the chosen rule and  $\omega_k$  the associated weights. We assume that there are  $p_1$  quadrature points to evaluate:

$$\Rightarrow A_{ij} = \sum_{\Delta \in \mathcal{T}} \sum_{k=1}^{p_1} \omega_k (J_g^{-T} \hat{\nabla} \phi_i)(\xi_k) \cdot (J_g^{-T} \hat{\nabla} \phi_j)(\xi_k) \mu(\xi_k) \quad (6.26)$$

Simultaneously, the right side  $b$  is treated in the same manner. As we might want to use another quadrature rule here that better suits our function  $f$ , we use  $p_2$  quadrature points:

$$b_i = \sum_{\Delta \in \mathcal{T}} \sum_{k=1}^{p_2} \omega_k f(g(\xi_k)) \phi_i(g(\xi_k)) \mu(\xi_k) \quad (6.27)$$

In our implementation we will of course not compute the matrix entries one after another but rather iterate over all elements of the grid and update all matrix entries with a non-vanishing contribution on that element.

$$\rightarrow i \quad \begin{matrix} & & i \downarrow \\ \begin{pmatrix} \star & \star & \star & \star & \star & \star & \star \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ \star & \star & \star & \star & \star & \star & \star \end{pmatrix} & \begin{pmatrix} \star \\ u_i \\ \star \end{pmatrix} = \begin{pmatrix} \star \\ 0 \\ \star \end{pmatrix}$$

Figure 6.2: Lines of  $A$  and  $b$  are replaced by trivial lines.

After assembling the matrix we implement the Dirichlet boundary conditions by overwriting the lines of the equation system associated with boundary nodes with trivial lines:

This is possible as—using the nodal basis—the coefficients match the value of the numerical solution at the corresponding node.

### 6.4.1 Implementation

In this implementation we will restrict ourselves to a 2-dimensional grid. However, the code works on simplicial grids of any dimension. Try this later!

Lets first have a look at the implementation of the shape functions. This class only provides the methods to evaluate the shape functions and their gradients:

#### Listing 21 (File `dune-grid-howto/shapefunctions.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef SHAPEFUNCTIONS_HH
4 #define SHAPEFUNCTIONS_HH
5
6 #include <dune/common/fvector.hh>
7
8 // LinearShapeFunction:
9 // represents a shape function and provides methods to evaluate the function
10 // and its gradient
11 template<class ctype, class rtype, int dim>
12 class LinearShapeFunction
13 {
14 public:
15     enum { dimension = dim };
16
17     LinearShapeFunction() : coeff0(0.0), coeff1(0.0) {}
18
19     LinearShapeFunction(rtype coeff0_, const Dune::FieldVector<rtype,dim>& coeff1_)
20         : coeff0(coeff0_), coeff1(coeff1_) {}
21
22     void setCoeff(rtype coeff0_, const Dune::FieldVector<rtype,dim>& coeff1_)
23     {
24         coeff0 = coeff0_;
25         coeff1 = coeff1_;
26     }
27
28     rtype evaluateFunction(const Dune::FieldVector<ctype,dim>& local) const
29     {
30         rtype result = coeff0;
31         for (int i = 0; i < dim; ++i)
32             result += coeff1[i] * local[i];
33         return result;
34     }
35

```



## 6 Attaching user data to a grid

```
36 Dune::FieldVector<rtype,dim>
37 evaluateGradient(const Dune::FieldVector<ctype,dim>& local) const
38 {
39     return coeff1;
40 }
41
42 private:
43     rtype coeff0;
44     Dune::FieldVector<rtype,dim> coeff1;
45 };
46
47 // P1ShapeFunctionSet
48 // initializes one and only one set of LinearShapeFunction
49 template<class ctype, class rtype, int dim>
50 class P1ShapeFunctionSet
51 {
52 public:
53     enum { n = dim + 1 };
54
55     typedef LinearShapeFunction<ctype,rtype,dim> ShapeFunction;
56     typedef rtype resulttype;
57
58     // get the only instance of this class
59     static const P1ShapeFunctionSet& instance()
60     {
61         static const P1ShapeFunctionSet sfs;
62         return sfs;
63     }
64
65     const ShapeFunction& operator[](int i) const
66     {
67         if (!i)
68             return f0;
69         else
70             return f1[i - 1];
71     }
72
73 private:
74     // private constructor prevents additional instances
75     P1ShapeFunctionSet()
76     {
77         Dune::FieldVector<rtype,dim> e(-1.0);
78         f0.setCoeff(1.0, e);
79         for (int i = 0; i < dim; ++i)
80         {
81             Dune::FieldVector<rtype,dim> e(0.0);
82             e[i] = 1.0;
83             f1[i].setCoeff(0.0, e);
84         }
85     }
86
87     P1ShapeFunctionSet(const P1ShapeFunctionSet& other)
88     {}
89
90     ShapeFunction f0;
91     ShapeFunction f1[dim];
92 };
93
94 #endif
```

And now the actual FEM code:

**Listing 22 (File dune-grid-howto/finiteelements.cc)**

## 6 Attaching user data to a grid

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #include "config.h"
4 #include <iostream>
5 #include <vector>
6 #include <set>
7 #include <dune/common/fvector.hh>
8 #include <dune/common/fmatrix.hh>
9 #include <dune/geometry/quadraturerules.hh>
10 #include <dune/grid/io/file/vtk/vtkwriter.hh>
11 #include <dune/grid/albertagrid.hh>
12
13 #if HAVE_DUNE_ISTL
14 #include <dune/istl/bvector.hh>
15 #include <dune/istl/bcrsmatrix.hh>
16 #include <dune/istl/ilu.hh>
17 #include <dune/istl/operators.hh>
18 #include <dune/istl/solvers.hh>
19 #include <dune/istl/preconditioners.hh>
20 #include <dune/istl/io.hh>
21 #else
22 #include <dune/common/dynvector.hh>
23 #include <dune/common/dynmatrix.hh>
24 #endif // HAVE_DUNE_ISTL
25
26 #include "shapefunctions.hh"
27
28 // P1Elements:
29 // a P1 finite element discretization for elliptic problems Dirichlet
30 // boundary conditions on simplicial conforming grids
31 template<class GV, class F>
32 class P1Elements
33 {
34 public:
35     static const int dim = GV::dimension;
36
37     typedef typename GV::ctype ctype;
38     #if HAVE_DUNE_ISTL
39     typedef Dune::BCRSMMatrix<Dune::FieldMatrix<ctype,1,1> > Matrix;
40     typedef Dune::BlockVector<Dune::FieldVector<ctype,1> > ScalarField;
41     #else
42     typedef Dune::DynamicMatrix<ctype> Matrix;
43     typedef Dune::DynamicVector<ctype> ScalarField;
44     #endif // HAVE_DUNE_ISTL
45
46 private:
47     typedef typename GV::template Codim<0>::Iterator LeafIterator;
48     typedef typename GV::template Codim<0>::Geometry::JacobianInverseTransposed
49         JacobianInverseTransposed;
50     typedef typename GV::IntersectionIterator IntersectionIterator;
51     typedef typename GV::IndexSet LeafIndexSet;
52
53     const GV& gv;
54     const F& f;
55
56 public:
57     Matrix A;
58     ScalarField b;
59     ScalarField u;
60     std::vector< std::set<int> > adjacencyPattern;
61
62     P1Elements(const GV& gv_, const F& f_) : gv(gv_), f(f_) {}
63 }
```

## 6 Attaching user data to a grid

```

63 // store adjacency information in a vector of sets
64 void determineAdjacencyPattern();
65
66 // assemble stiffness matrix A and right side b
67 void assemble();
68
69 // finally solve  $Au = b$  for u
70 void solve();
71 };
72
73 template<class GV, class F>
74 void P1Elements<GV, F>::determineAdjacencyPattern()
75 {
76     const int N = gv.size(dim);
77     adjacencyPattern.resize(N);
78
79     const LeafIndexSet& set = gv.indexSet();
80     const LeafIterator itend = gv.template end<0>();
81
82     for (LeafIterator it = gv.template begin<0>(); it != itend; ++it)
83     {
84         Dune::GeometryType gt = it->type();
85         const Dune::template ReferenceElement<ctype,dim> &ref =
86             Dune::ReferenceElements<ctype,dim>::general(gt);
87
88         // traverse all codim-1-entities of the current element and store all
89         // pairs of vertices in adjacencyPattern
90         const IntersectionIterator isend = gv.iend(*it);
91         for (IntersectionIterator is = gv.ibegin(*it) ; is != isend ; ++is)
92         {
93             int vertexsize = ref.size(is->indexInInside(),1,dim);
94             for (int i=0; i < vertexsize; i++)
95             {
96                 int indexi = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,i,dim),dim);
97                 for (int j=0; j < vertexsize; j++)
98                 {
99                     int indexj = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,j,dim),dim);
100                     adjacencyPattern[indexi].insert(indexj);
101                 }
102             }
103         }
104     }
105 }
106
107 template<class GV, class F>
108 void P1Elements<GV, F>::assemble()
109 {
110     const int N = gv.size(dim);
111
112     const LeafIndexSet& set = gv.indexSet();
113     const LeafIterator itend = gv.template end<0>();
114
115     // set sizes of A and b
116 #if HAVE_DUNE_ISTL
117     A.setSize(N, N, N + 2*gv.size(dim-1));
118     A.setBuildMode(Matrix::random);
119     b.resize(N, false);
120
121     for (int i = 0; i < N; i++)
122         A.setrowsize(i,adjacencyPattern[i].size());
123     A.endrowsizes();
124
125     // set sparsity pattern of A with the information gained in determineAdjacencyPattern

```

## 6 Attaching user data to a grid

```

126 for (int i = 0; i < N; i++)
127 {
128     std::template set<int>::iterator setend = adjacencyPattern[i].end();
129     for (std::template set<int>::iterator setit = adjacencyPattern[i].begin();
130         setit != setend; ++setit)
131         A.addindex(i,*setit);
132 }
133
134 A.endindices();
135 #else
136 A.resize(N, N);
137 b.resize(N);
138 #endif // HAVE_DUNE_ISTL
139
140 // initialize A and b
141 A = 0.0;
142 b = 0.0;
143
144 // get a set of P1 shape functions
145 const P1ShapeFunctionSet<ctype,ctype,dim>& basis = P1ShapeFunctionSet<ctype,ctype,dim>::
146     instance();
147
148 for (LeafIterator it = gv.template begin<0>(); it != itend; ++it)
149 {
150     // determine geometry type of the current element and get the matching reference element
151     Dune::GeometryType gt = it->type();
152     const Dune::template ReferenceElement<ctype,dim> &ref =
153         Dune::ReferenceElements<ctype,dim>::general(gt);
154     int vertexsize = ref.size(dim);
155
156     // get a quadrature rule of order one for the given geometry type
157     const Dune::QuadratureRule<ctype,dim>& rule = Dune::QuadratureRules<ctype,dim>::rule(gt,1);
158     for (typename Dune::QuadratureRule<ctype,dim>::const_iterator r = rule.begin();
159         r != rule.end(); ++r)
160     {
161         // compute the jacobian inverse transposed to transform the gradients
162         JacobianInverseTransposed jacInvTra =
163             it->geometry().jacobianInverseTransposed(r->position());
164
165         // get the weight at the current quadrature point
166         ctype weight = r->weight();
167
168         // compute Jacobian determinant for the transformation formula
169         ctype detjac = it->geometry().integrationElement(r->position());
170         for (int i = 0; i < vertexsize; i++)
171         {
172             // compute transformed gradients
173             Dune::FieldVector<ctype,dim> grad1;
174             jacInvTra.mv(basis[i].evaluateGradient(r->position()),grad1);
175             for (int j = 0; j < vertexsize; j++)
176             {
177                 Dune::FieldVector<ctype,dim> grad2;
178                 jacInvTra.mv(basis[j].evaluateGradient(r->position()),grad2);
179
180                 // gain global indices of vertices i and j and update associated matrix entry
181                 A[set.subIndex(*it,i,dim)][set.subIndex(*it,j,dim)]
182                     += (grad1*grad2) * weight * detjac;
183             }
184         }
185
186         // get a quadrature rule of order two for the given geometry type
187         const Dune::QuadratureRule<ctype,dim>& rule2 = Dune::QuadratureRules<ctype,dim>::rule(gt,2)

```

## 6 Attaching user data to a grid

```

188     ;
189     for (typename Dune::QuadratureRule<ctype,dim>::const_iterator r = rule2.begin();
190          r != rule2.end() ; ++r)
191     {
192         ctype weight = r->weight();
193         ctype detjac = it->geometry().integrationElement(r->position());
194         for (int i = 0 ; i<vertexsize; i++)
195         {
196             // evaluate the integrand of the right side
197             ctype fval = basis[i].evaluateFunction(r->position())
198                 * f(it->geometry().global(r->position())) ;
199             b[set.subIndex(*it,i,dim)] += fval * weight * detjac;
200         }
201     }
202
203     // Dirichlet boundary conditions:
204     // replace lines in A related to Dirichlet vertices by trivial lines
205     for ( LeafIterator it = gv.template begin<0>() ; it != itend ; ++it)
206     {
207         const IntersectionIterator isend = gv.iend(*it);
208         for (IntersectionIterator is = gv.ibegin(*it) ; is != isend ; ++is)
209         {
210             // determine geometry type of the current element and get the matching reference element
211             Dune::GeometryType gt = it->type();
212             const Dune::template ReferenceElement<ctype,dim> &ref =
213                 Dune::ReferenceElements<ctype,dim>::general(gt);
214
215             // check whether current intersection is on the boundary
216             if ( is->boundary() )
217             {
218                 // traverse all vertices the intersection consists of
219                 for (int i=0; i < ref.size(is->indexInInside(),1,dim); i++)
220                 {
221                     // and replace the associated line of A and b with a trivial one
222                     int indexi = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,i,dim),dim);
223
224                     A[indexi] = 0.0;
225                     A[indexi][indexi] = 1.0;
226                     b[indexi] = 0.0;
227                 }
228             }
229         }
230     }
231 }
232
233 #if HAVE_DUNE_ISTL
234 template<class GV, class E>
235 void P1Elements<GV, E>::solve()
236 {
237     // make linear operator from A
238     Dune::MatrixAdapter<Matrix,ScalarField,ScalarField> op(A);
239
240     // initialize preconditioner
241     Dune::SeqILUn<Matrix,ScalarField,ScalarField> ilu1(A, 1, 0.92);
242
243     // the inverse operator
244     Dune::BiCGSTABSolver<ScalarField> bcgs(op, ilu1, 1e-15, 5000, 0);
245     Dune::InverseOperatorResult r;
246
247     // initialize u to some arbitrary value to avoid u being the exact
248     // solution
249     u.resize(b.N(), false);

```

## 6 Attaching user data to a grid

```
250  u = 2.0;
251
252  // finally solve the system
253  bcgs.apply(u, b, r);
254 }
255 #endif // HAVE_DUNE_ISTL
256
257 // an example right hand side function
258 template<class ctype, int dim>
259 class Bump {
260 public:
261     ctype operator() (Dune::FieldVector<ctype,dim> x) const
262     {
263         ctype result = 0;
264         for (int i=0 ; i < dim ; i++)
265             result += 2.0 * x[i]* (1-x[i]);
266         return result;
267     }
268 };
269
270 int main(int argc, char** argv)
271 {
272     #if HAVE_ALBERTA && ALBERTA_DIM==2
273         static const int dim = 2;
274         const char* gridfile = "grids/2dgrid.al";
275
276         typedef Dune::AlbertaGrid<dim,dim> GridType;
277         typedef GridType::LeafGridView GV;
278
279         typedef GridType::ctype ctype;
280         typedef Bump<ctype,dim> Func;
281
282         GridType grid(gridfile);
283         const GV& gv = grid.leafGridView();
284
285         Func f;
286         P1Elements<GV,Func> p1(gv, f);
287
288         #if HAVE_DUNE_ISTL
289             grid.globalRefine(16);
290         #else
291             grid.globalRefine(10);
292         #endif // HAVE_DUNE_ISTL
293
294         std::cout << "-----" << "\n";
295         std::cout << "number_of_unknows:_" << grid.size(dim) << "\n";
296
297         std::cout << "determine_adjacency_pattern..." << "\n";
298         p1.determineAdjacencyPattern();
299
300         std::cout << "assembling..." << "\n";
301         p1.assemble();
302
303         #if HAVE_DUNE_ISTL
304             std::cout << "solving..." << "\n";
305             p1.solve();
306
307             std::cout << "visualizing..." << "\n";
308             Dune::VTKWriter<GridType::LeafGridView> vtkwriter(grid.leafGridView());
309             vtkwriter.addVertexData(p1.u, "u");
310             vtkwriter.write("fem2d", Dune::VTK::appenddraw);
311         #else
312             std::cout << "for_solving_and_visualizing_dune-istl_is_necessary." << "\n";
```

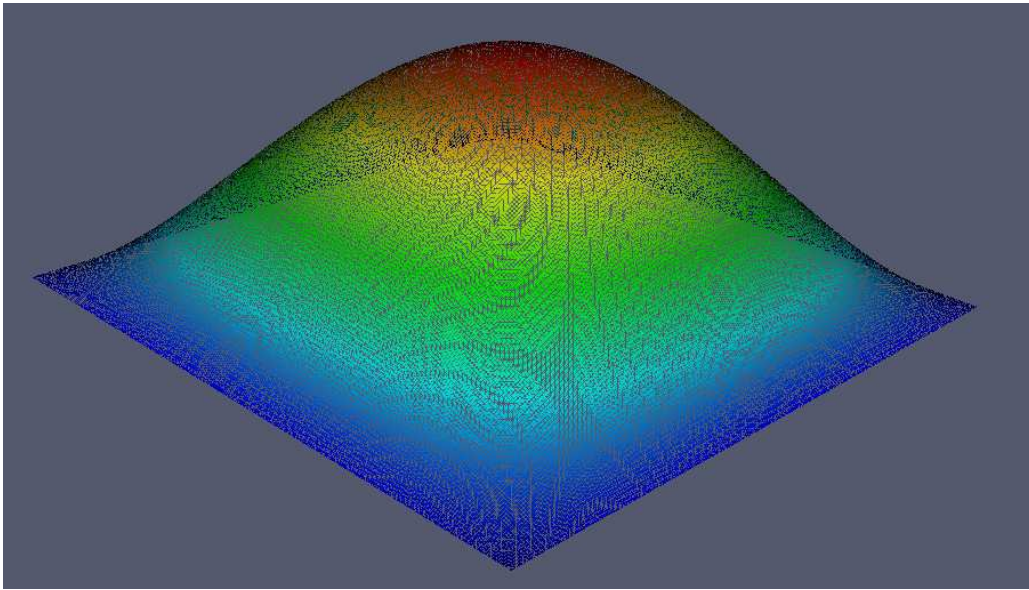


Figure 6.3: Solution in 2D

```

313 #endif // HAVE_DUNE_ISTL
314 #endif // HAVE_ALBERTA && ALBERTA_DIM==2
315 }

```

The function `determineAdjacencyPattern()` in lines 73 to 105 does traverse the grid and stores all adjacency information in a `std::vector< std::set<int> >`. You might wonder why this is necessary before the actual computing of the matrix entries. The reason for this is that, as data structure for the matrix  $A$ , we use `BCRSMatrix` - which is specialized to hold large sparse matrices. Using this type, information about which entries do not vanish has to be known when assembling. We do give this information to the matrix from line 126 on. Only after finishing this in line 134 we can start to fill the matrix with values.

From line 147 to 201 we have the main loop traversing the whole grid and updating the matrix entries. This does strictly follow the procedure described in previous chapters. The main calculation is done in line 180 and 198 - which are one-to-one implementations of 6.26 and 6.27.

As already said above, we do directly implement Dirichlet boundaries into our matrix. This is done in lines 205 to 230. We have to traverse the whole grid once again and check for each intersection of elements whether it is on the boundary. In line 224 we overwrite the line corresponding to a node on the boundary as shown in figure 6.2.

When you visualize your results, you should get something like figure 6.3 or 6.4!

**Exercise 6.1** Try a 3-dimensional grid! Just change the dimension in line 273 and the name of the gridfile in line 274 to `3dgrid.al`. You can compile the new code without reconfiguring by running `make ALBERTA_DIM=3`

**Exercise 6.2** Modify the code in order to make it handle Neumann boundary conditions too!

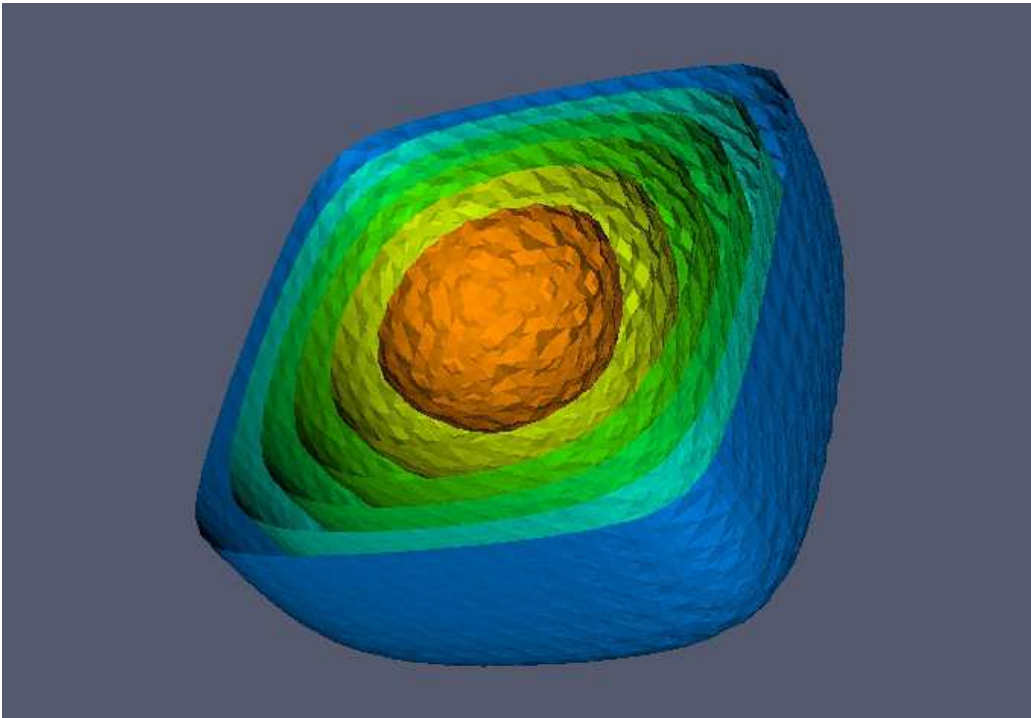


Figure 6.4: Solution in 3D



## 7 Adaptivity

### 7.1 Adaptive integration

#### 7.1.1 Adaptive multigrid integration

In this section we describe briefly the adaptive multigrid integration algorithm presented in [4].

##### Global error estimation

The global error can be estimated by taking the difference of the numerically computed value for the integral on a fine and a coarse grid as given in (5.3).

##### Local error estimation

Let  $I_f^p(\omega)$  and  $I_f^q(\omega)$  be two integration formulas of different orders  $p > q$  for the evaluation of the integral over some function  $f$  on the element  $\omega \subseteq \Omega$ . If we assume that the higher order rule is locally more accurate then

$$\bar{\epsilon}(\omega) = |I_f^p(\omega) - I_f^q(\omega)| \quad (7.1)$$

is an estimator for the local error on the element  $\omega$ .

##### Refinement strategy

If the estimated global error is not below a user tolerance the grid is to be refined in those places where the estimated local error is “high”. To be more specific, we want to achieve that each element in the grid contributes about the same local error to the global error. Suppose we knew the maximum local error on all the new elements that resulted from refining the current mesh (without actually doing so). Then it would be a good idea to refine only those elements in the mesh where the local error is not already below that maximum local error that will be attained anyway. In [4] it is shown that the local error after mesh refinement can be effectively computed without actually doing the refinement. Consider an element  $\omega$  and its father element  $\omega^-$ , i. e. the refinement of  $\omega^-$  resulted in  $\omega$ . Moreover, assume that  $\omega^+$  is a (virtual) element that would result from a refinement of  $\omega$ . Then it can be shown that under certain assumptions the quantity

$$\epsilon^+(\omega) = \frac{\bar{\epsilon}(\omega)^2}{\bar{\epsilon}(\omega^-)} \quad (7.2)$$

is an estimate for the local error on  $\omega^+$ , i. e.  $\bar{\epsilon}(\omega^+)$ .

Another idea to determine the refinement threshold is to look simply at the maximum of the local errors on the current mesh and to refine only those elements where the local error is above a certain fraction of the maximum local error.

By combining the two approaches we get the threshold value  $\kappa$  actually used in the code:

$$\kappa = \min \left( \max_{\omega} \epsilon^+(\omega), \frac{1}{2} \max_{\omega} \bar{\epsilon}(\omega) \right). \quad (7.3)$$

**Algorithm**

The complete multigrid integration algorithm then reads as follows:

- Choose an initial grid.
- Repeat the following steps
  - Compute the value  $I$  for the integral on the current grid.
  - Compute the estimate  $E$  for the global error.
  - If  $E < \text{tol} \cdot I$  we are done.
  - Compute the threshold  $\kappa$  as defined above.
  - Refine all elements  $\omega$  where  $\bar{\epsilon}(\omega) \geq \kappa$ .

**7.1.2 Implementation of the algorithm**

The algorithm above is realized in the following code.

**Listing 23 (File dune-grid-howto/adaptiveintegration.cc)**

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 // $Id$
4
5 #include "config.h"
6 #include <iostream>
7 #include <iomanip>
8 #include <dune/grid/io/file/vtk/vtkwriter.hh> // VTK output routines
9 #include <dune/common/parallel/mphelper.hh> // include mpi helper class
10
11 #include "unitcube.hh"
12 #include "functors.hh"
13 #include "integrateentity.hh"
14
15
16 //! adaptive refinement test
17 template<class Grid, class Functor>
18 void adaptiveintegration (Grid& grid, const Functor& f)
19 {
20   // get grid view type for leaf grid part
21   typedef typename Grid::LeafGridView GridView;
22   // get iterator type
23   typedef typename GridView::template Codim<0>::Iterator ElementLeafIterator;
24
25   // get grid view on leaf part
26   GridView gridView = grid.leafGridView();
27
28   // algorithm parameters
29   const double tol=1E-8;
30   const int loworder=1;
31   const int highorder=3;
32
33   // loop over grid sequence
34   double oldvalue=1E100;
35   for (int k=0; k<100; k++)
36   {
37     // compute integral on current mesh
38     double value=0;
39     for (ElementLeafIterator it = gridView.template begin<0>();

```

## 7 Adaptivity

```

40         it!=gridView.template end<0>(); ++it)
41         value += integrateEntity(*it,f,highorder);
42
43     // print result
44     double estimated_error = std::abs(value-oldvalue);
45     oldvalue=value;           // save value for next estimate
46     std::cout << "elements="
47                 << std::setw(8) << std::right
48                 << grid.size(0)
49                 << "□integral="
50                 << std::scientific << std::setprecision(8)
51                 << value
52                 << "□error=" << estimated_error
53                 << std::endl;
54
55     // check convergence
56     if (estimated_error <= tol*value)
57         break;
58
59     // refine grid globally in first step to ensure
60     // that every element has a father
61     if (k==0)
62     {
63         grid.globalRefine(1);
64         continue;
65     }
66
67     // compute threshold for subsequent refinement
68     double maxerror=-1E100;
69     double maxextrapolatederror=-1E100;
70     for (ElementLeafIterator it = gridView.template leafbegin<0>();
71          it!=gridView.template leafend<0>(); ++it)
72     {
73         // error on this entity
74         double lowresult=integrateEntity(*it,f,loworder);
75         double highresult=integrateEntity(*it,f,highorder);
76         double error = std::abs(lowresult-highresult);
77
78         // max over whole grid
79         maxerror = std::max(maxerror,error);
80
81         // error on father entity
82         double fatherlowresult=integrateEntity(*(it->father()),f,loworder);
83         double fatherhighresult=integrateEntity(*(it->father()),f,highorder);
84         double fathererror = std::abs(fatherlowresult-fatherhighresult);
85
86         // local extrapolation
87         double extrapolatederror = error*error/(fathererror+1E-30);
88         maxextrapolatederror = std::max(maxextrapolatederror,extrapolatederror);
89     }
90     double kappa = std::min(maxextrapolatederror,0.5*maxerror);
91
92     // mark elements for refinement
93     for (ElementLeafIterator it = gridView.template begin<0>();
94          it!=gridView.template end<0>(); ++it)
95     {
96         double lowresult=integrateEntity(*it,f,loworder);
97         double highresult=integrateEntity(*it,f,highorder);
98         double error = std::abs(lowresult-highresult);
99         if (error>kappa) grid.mark(1,*it);
100     }
101
102     // adapt the mesh

```

## 7 Adaptivity

```

103     grid.preAdapt();
104     grid.adapt();
105     grid.postAdapt();
106 }
107
108 // write grid in VTK format
109 Dune::VTKWriter<typename Grid::LeafGridView> vtkwriter(gridView);
110 vtkwriter.write( "adaptivegrid", Dune::VTK::appenddraw );
111 }
112
113 //! supply functor
114 template<class Grid>
115 void dowork (Grid& grid)
116 {
117     adaptiveintegration(grid, Needle<typename Grid::ctype, Grid::dimension>());
118 }
119
120 int main(int argc, char **argv)
121 {
122     // initialize MPI, finalize is done automatically on exit
123     Dune::MPIHelper::instance(argc, argv);
124
125     // start try/catch block to get error messages from dune
126     try {
127         using namespace Dune;
128
129         // the GridSelector :: GridType is defined in gridtype.hh and is
130         // set during compilation
131         typedef GridSelector :: GridType Grid;
132
133         // use unitcube from grids
134         std::stringstream dgfFileName;
135         dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
136             << "unitcube" << Grid::dimension << ".dgf";
137
138         // create grid pointer
139         GridPtr<Grid> gridPtr( dgfFileName.str() );
140
141         // do the adaptive integration
142         // NOTE: for structured grids global refinement will be used
143         dowork( *gridPtr );
144     }
145     catch (std::exception & e) {
146         std::cout << "STL_ERROR:" << e.what() << std::endl;
147         return 1;
148     }
149     catch (Dune::Exception & e) {
150         std::cout << "DUNE_ERROR:" << e.what() << std::endl;
151         return 1;
152     }
153     catch (...) {
154         std::cout << "Unknown_ERROR" << std::endl;
155         return 1;
156     }
157
158     // done
159     return 0;
160 }

```

The work is done in the function `adaptiveintegration`. Lines 38-41 compute the value of the integral on the current mesh. After printing the result the decision whether to continue or not is done in line 56. The extrapolation strategy relies on the fact that every element has a father. To ensure

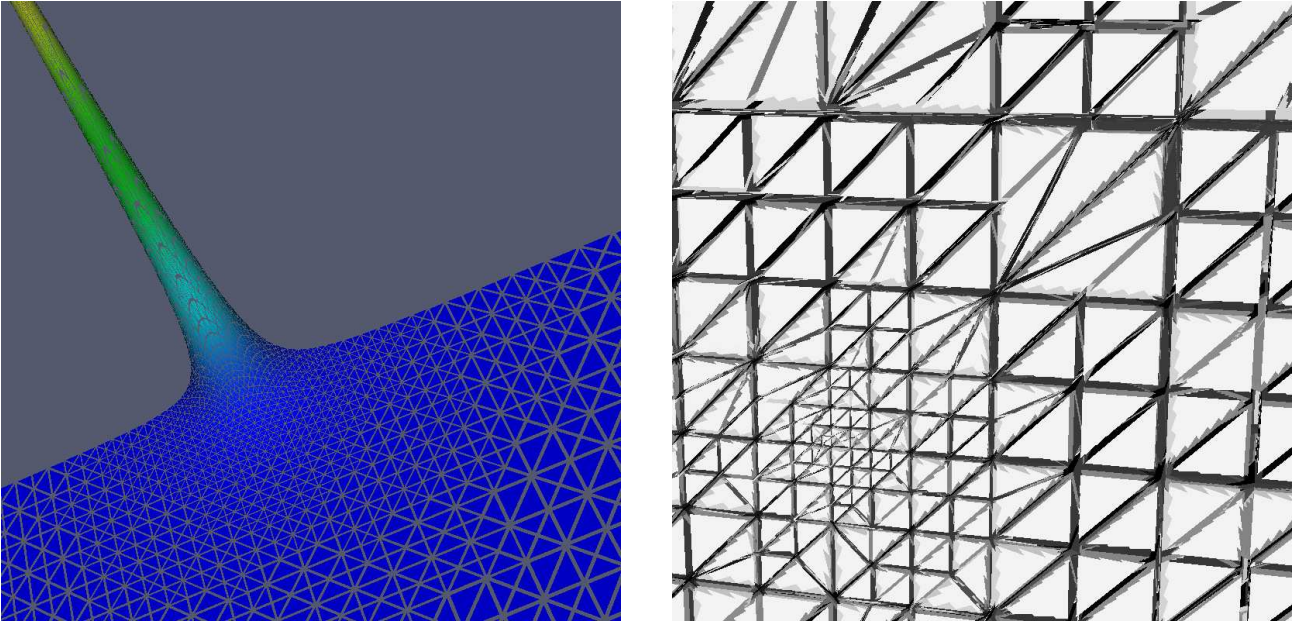


Figure 7.1: Two and three-dimensional grids generated by the adaptive integration algorithm applied to the needle pulse. Left grid is generated using Alberta, right grid is generated using UG.

this, the grid is at least once refined globally in the first step (line 63). Now the refinement threshold  $\kappa$  can be computed in lines 68-90. Finally the last loop in lines 93-100 marks elements for refinement and lines 103-105 actually do the refinement. The reason for dividing refinement into three functions `preAdapt()`, `adapt()` and `postAdapt()` will be explained with the next example. Note the flexibility of this algorithm: It runs in any space dimension on any kind of grid and different integration orders can easily be incorporated. And that with just about 100 lines of code including comments.

Figure 7.1 shows two grids generated by the adaptive integration algorithm.

**Warning 7.1** The quadrature rules for prisms and pyramids are currently only implemented for order two. Therefore adaptive calculations with UGGrid and hexahedral elements do not work.

## 7.2 Adaptive cell centered finite volumes

In this section we extend the example of Section 6.3 by adaptive mesh refinement. This requires two things: (i) a method to select cells for refinement or coarsening (derefinement) and (ii) the transfer of a solution on a given grid to the adapted grid. The finite volume algorithm itself has already been implemented for adaptively refined grids in Section 6.3.

For the adaptive refinement and coarsening we use a very simple heuristic strategy that works as follows:

- Compute global maximum and minimum of element concentrations:

$$\overline{C} = \max_i C_i, \quad \underline{C} = \min_i C_i.$$

## 7 Adaptivity

- As the local indicator in cell  $\omega_i$  we define

$$\eta_i = \max_{\gamma_{ij}} |C_i - C_j|.$$

Here  $\gamma_{ij}$  denotes intersections with other elements in the leaf grid.

- If for  $\omega_i$  we have  $\eta_i > \overline{\text{tol}} \cdot (\overline{C} - \underline{C})$  and  $\omega_i$  has not been refined more than  $\overline{M}$  times then mark  $\omega_i$  and all its neighbors for refinement.
- Mark all elements  $\omega_i$  for coarsening where  $\eta_i < \underline{\text{tol}} \cdot (\overline{C} - \underline{C})$  and  $\omega_i$  has been refined at least  $\underline{M}$  times.

This strategy refines an element if the local gradient is “large” and it coarsens elements (which means it removes a previous refinement) if the local gradient is “small”. In addition any element is refined at least  $\underline{M}$  times and at most  $\overline{M}$  times.

After mesh modification the solution from the previous grid must be transferred to the new mesh. Thereby the following situations do occur for an element:

- The element is a leaf element in the new mesh and was a leaf element in the old mesh: keep the value.
- The element is a leaf element in the new mesh and existed in the old mesh as a non-leaf element: Compute the cell value as an average of the son elements in the old mesh.
- The element is a leaf element in the new mesh and is obtained through refining some element in the old mesh: Copy the value from the element in the old mesh to the new mesh.

The complete mesh adaptation is done by the function `finitevolumeadapt` in the following listing:

### Listing 24 (File `dune-grid-howto/finitevolumeadapt.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_FINITEVOLUMEADAPT_HH__
4 #define __DUNE_GRID_HOWTO_FINITEVOLUMEADAPT_HH__
5
6 #include <cmath>
7 #include <dune/grid/utility/persistentcontainer.hh>
8
9 struct RestrictedValue
10 {
11     double value;
12     int count;
13     RestrictedValue ()
14     {
15         value = 0;
16         count = 0;
17     }
18 };
19
20 template<class G, class M, class V>
21 bool finitevolumeadapt (G& grid, M& mapper, V& c, int lmin, int lmax, int k)
22 {
23     // tol value for refinement strategy
24     const double refinertol = 0.05;
25     const double coarsentol = 0.001;

```

## 7 Adaptivity

```

26
27 // type used for coordinates in the grid
28 typedef typename G::ctype ct;
29
30 // grid view types
31 typedef typename G::LeafGridView LeafGridView;
32 typedef typename G::LevelGridView LevelGridView;
33
34 // iterator types
35 typedef typename LeafGridView::template Codim<0>::Iterator LeafIterator;
36 typedef typename LevelGridView::template Codim<0>::Iterator LevelIterator;
37
38 // entity and entity pointer
39 typedef typename G::template Codim<0>::Entity Entity;
40 typedef typename G::template Codim<0>::EntityPointer EntityPointer;
41
42 // intersection iterator type
43 typedef typename LeafGridView::IntersectionIterator LeafIntersectionIterator;
44
45 // get grid view on leaf grid
46 LeafGridView leafView = grid.leafGridView();
47
48 // compute cell indicators
49 V indicator(c.size(),-1E100);
50 double globalmax = -1E100;
51 double globalmin = 1E100;
52 for (LeafIterator it = leafView.template begin<0>();
53      it!=leafView.template end<0>(); ++it)
54 {
55     // my index
56     int indexi = mapper.map(*it);
57
58     // global min/max
59     globalmax = std::max(globalmax,c[indexi]);
60     globalmin = std::min(globalmin,c[indexi]);
61
62     LeafIntersectionIterator isend = leafView.iend(*it);
63     for (LeafIntersectionIterator is = leafView.ibegin(*it); is!=isend; ++is)
64     {
65         const typename LeafIntersectionIterator::Intersection &intersection = *is;
66         if( !intersection.neighbor() )
67             continue;
68
69         // access neighbor
70         const EntityPointer pOutside = intersection.outside();
71         const Entity &outside = *pOutside;
72         int indexj = mapper.map( outside );
73
74         // handle face from one side only
75         if ( it.level() > outside.level() ||
76             (it.level() == outside.level() && indexi<indexj) )
77         {
78             double localdelta = std::abs(c[indexj]-c[indexi]);
79             indicator[indexi] = std::max(indicator[indexi],localdelta);
80             indicator[indexj] = std::max(indicator[indexj],localdelta);
81         }
82     }
83 }
84
85 // mark cells for refinement/coarsening
86 double globaldelta = globalmax-globalmin;
87 int marked=0;
88 for (LeafIterator it = leafView.template begin<0>();

```

## 7 Adaptivity

```

89     it!=leafView.template end<0>(); ++it)
90 {
91     if (indicator[mapper.map(*it)]>refinetol*globaldelta
92         && (it.level()<lmax || !it->isRegular()))
93     {
94         const Entity &entity = *it;
95         grid.mark( 1, entity );
96         ++marked;
97         LeafIntersectionIterator isend = leafView.iend(entity);
98         for( LeafIntersectionIterator is = leafView.ibegin(entity); is != isend; ++is )
99         {
100             const typename LeafIntersectionIterator::Intersection &intersection = *is;
101             if( !intersection.neighbor() )
102                 continue;
103
104             const EntityPointer pOutside = intersection.outside();
105             const Entity &outside = *pOutside;
106             if( (outside.level() < lmax) || !outside.isRegular() )
107                 grid.mark( 1, outside );
108         }
109     }
110     if (indicator[mapper.map(*it)]<coarsentol*globaldelta && it.level()>lmin)
111     {
112         grid.mark( -1, *it );
113         ++marked;
114     }
115 }
116 if( marked==0 )
117     return false;
118
119 grid.preAdapt();
120
121 typedef Dune::PersistentContainer<G,RestrictedValue> RestrictionMap;
122 RestrictionMap restrictionmap(grid,0); // restricted concentration
123
124 for (int level=grid.maxLevel(); level>=0; level--)
125 {
126     // get grid view on level grid
127     LevelGridView levelView = grid.levelGridView(level);
128     for (LevelIterator it = levelView.template begin<0>();
129         it!=levelView.template end<0>(); ++it)
130     {
131         // get your map entry
132         RestrictedValue& rv = restrictionmap[*it];
133         // put your value in the map
134         if (it->isLeaf())
135         {
136             int indexi = mapper.map(*it);
137             rv.value = c[indexi];
138             rv.count = 1;
139         }
140
141         // average in father
142         if (it.level()>0)
143         {
144             EntityPointer ep = it->father();
145             RestrictedValue& rvf = restrictionmap[*ep];
146             rvf.value += rv.value/rv.count;
147             rvf.count += 1;
148         }
149     }
150 }
151

```



## 7 Adaptivity

```

152 // adapt mesh and mapper
153 bool rv=grid.adapt();
154 mapper.update();
155 restrictionmap.resize();
156 c.resize(mapper.size());
157
158 // interpolate new cells, restrict coarsened cells
159 for (int level=0; level<=grid.maxLevel(); level++)
160 {
161     LevelGridView levelView = grid.levelGridView(level);
162     for (LevelIterator it = levelView.template begin<0>();
163          it!=levelView.template end<0>(); ++it)
164     {
165         // get your id
166
167         // check map entry
168         if (! it->isNew() )
169         {
170             // entry is in map, write in leaf
171             if (it->isLeaf())
172             {
173                 RestrictedValue& rv = restrictionmap[*it];
174                 int indexi = mapper.map(*it);
175                 c[indexi] = rv.value/rv.count;
176             }
177         }
178         else
179         {
180             // value is not in map, interpolate from father element
181             assert (it.level()>0);
182             EntityPointer ep = it->father();
183             RestrictedValue& rvf = restrictionmap[*ep];
184             if (it->isLeaf())
185             {
186                 int indexi = mapper.map(*it);
187                 c[indexi] = rvf.value/rvf.count;
188             }
189             else
190             {
191                 // create new entry
192                 RestrictedValue& rv = restrictionmap[*it];
193                 rv.value = rvf.value/rvf.count;
194                 rv.count = 1;
195             }
196         }
197     }
198 }
199 grid.postAdapt();
200
201 return rv;
202 }
203
204 #endif // _DUNE_GRID_HOWTO.FINITEVOLUMEADAPT_HH_

```

The loop in lines 52-83 computes the indicator values  $\eta_i$  as well as the global minimum and maximum  $\overline{C}, \underline{C}$ . Then the next loop in lines 88-115 marks the elements for refinement. Lines 122-149 construct a map that stores for each element in the mesh (on all levels) the average of the element values in the leaf elements of the subtree of the given element. This is accomplished by descending from the fine grid levels to the coarse grid levels and thereby adding the value in an element to the father element. The key into the map is the global id of an element. Thus the value is accessible also after mesh

modification.

Now the grid can really be modified in line 153 by calling the `adapt()` method on the grid object. The mapper is updated to reflect the changes in the grid in line 154 and the concentration vector is resized to the new size in line 156. Then the values have to be interpolated to the new elements in the mesh using the map and finally to be transferred to the resized concentration vector. This is done in the loop in lines 159-197.

Here is the new main program with an adapted `timeloop`:

#### Listing 25 (File `dune-grid-howto/adativefinitevolume.cc`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #include "config.h"           // know what grids are present
4 #include <iostream>           // for input/output to shell
5 #include <fstream>           // for input/output to files
6 #include <vector>            // STL vector class
7
8 #include <dune/grid/common/mcmgmapper.hh> // mapper class
9 #include <dune/common/parallel/mphelper.hh> // include mpi helper class
10
11 #include "vtkout.hh"
12 #include "transportproblem2.hh"
13 #include "initialize.hh"
14 #include "evolve.hh"
15 #include "finitevolumeadapt.hh"
16
17 //=====
18 // the time loop function working for all types of grids
19 //=====
20
21 template<class G>
22 void timeloop (G& grid, double tend, int lmin, int lmax)
23 {
24     // make a mapper for codim 0 entities in the leaf grid
25     Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,Dune::MCMGElementLayout>
26     mapper(grid);
27
28     // allocate a vector for the concentration
29     std::vector<double> c(mapper.size());
30
31     // initialize concentration with initial values
32     initialize(grid,mapper,c);
33     for (int i=grid.maxLevel(); i<lmax; i++)
34     {
35         if (grid.maxLevel()>=lmax) break;
36         finitevolumeadapt(grid,mapper,c,lmin,lmax,0);
37         initialize(grid,mapper,c);
38     }
39
40     // write initial data
41     vtkout(grid,c,"concentration",0,0);
42
43     // variables for time, timestep etc.
44     double dt, t=0;
45     double saveStep = 0.1;
46     const double saveInterval = t + 0.1;
47     int counter = 1;
48     int k = 0;
49
50     std::cout << "s=" << grid.size(0) << "\nk=" << k << "\nt=" << t << std::endl;
51     while (t<tend)

```

## 7 Adaptivity

```

52 {
53     // augment time step counter
54     ++k;
55
56     // apply finite volume scheme
57     evolve(grid,mapper,c,t,dt);
58
59     // augment time
60     t += dt;
61
62     // check if data should be written
63     if (t >= saveStep)
64     {
65         // write data
66         vtkout(grid,c,"concentration",counter,t);
67
68         // increase counter and saveStep for next interval
69         saveStep += saveInterval;
70         ++counter;
71     }
72
73     // print info about time, timestep size and counter
74     std::cout << "s=" << grid.size(0)
75               << "k=" << k << "t=" << t << "dt=" << dt << std::endl;
76
77     // for unstructured grids call adaptation algorithm
78     finitevolumeadapt(grid,mapper,c,lmin,lmax,k);
79 }
80
81 // write last time step
82 vtkout(grid,c,"concentration",counter,tend);
83
84 // write
85 }
86
87 //=====
88 // The main function creates objects and does the time loop
89 //=====
90
91 int main (int argc , char ** argv)
92 {
93     // initialize MPI, finalize is done automatically on exit
94     Dune::MPIHelper::instance(argc,argv);
95
96     // start try/catch block to get error messages from dune
97     try {
98         using namespace Dune;
99
100        // the GridSelector :: GridType is defined in gridtype.hh and is
101        // set during compilation
102        typedef GridSelector :: GridType Grid;
103
104        // use unitcube from grids
105        std::stringstream dgfFileName;
106        dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
107                  << "unitcube" << Grid::dimension << ".dgf";
108
109        // create grid pointer
110        GridPtr<Grid> gridPtr( dgfFileName.str() );
111
112        // grid reference
113        Grid& grid = *gridPtr;
114

```

## 7 Adaptivity

```

115 // minimal allowed level during refinement
116 int minLevel = 2 * DGFGGridInfo<Grid>::refineStepsForHalf();
117
118 // refine grid until upper limit of level
119 grid.globalRefine(minLevel);
120
121 // maximal allowed level during refinement
122 int maxLevel = minLevel + 3 * DGFGGridInfo<Grid>::refineStepsForHalf();
123
124 // do time loop until end time 0.5
125 timeloop(grid, 0.5, minLevel, maxLevel);
126 }
127 catch (std::exception & e) {
128     std::cout << "STL_ERROR:" << e.what() << std::endl;
129     return 1;
130 }
131 catch (Dune::Exception & e) {
132     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
133     return 1;
134 }
135 catch (...) {
136     std::cout << "Unknown_ERROR" << std::endl;
137     return 1;
138 }
139
140 // done
141 return 0;
142 }

```

The program works analogously to the non adaptive `finitevolume` version from the previous chapter. The only differences are inside the `timeloop` function. During the initialization of the concentration vector in line 36 and after each time step in line 78 the function `finitevolumeadapt` is called in order to refine the grid. The initial adaptation is repeated  $\overline{M}$  times. Note that adaptation after each time steps is deactivated during the compiler phase for unstructured grids with help of the `Capabilities` class. This is because structured grids do not allow a conforming refinement and are therefore unusable for adaptive schemes. In fact, the `adapt` method on a grid of `YaspGrid` e.g. results in a *global* grid refinement.

**Exercise 7.2** Compile the program with the `gridtype` set to `ALUGRID_SIMPLEX` and `ALUGRID_CONFORM` and compare the results visually.

## 8 Parallelism

### 8.1 DUNE Data Decomposition Model

The parallelization concept in **DUNE** follows the Single Program Multiple Data (SPMD) data parallel programming paradigm. In this programming model each process executes the same code but on different data. The parallel program is parametrized by the rank of the individual process in the set and the number of processes  $P$  involved. The processes communicate by exchanging messages, but you will rarely have the need to bother with sending messages.

A parallel **DUNE** grid, such as YaspGrid, is a collective object which means that all processes participating in the computations on the grid instantiate the grid object at the same time (collectively). Each process stores a subset of all the entities that the same program running on a single process would have. An entity may be stored in more than one process, in principle it may be even stored in all processes. An entity stored in more than one process is called a distributed entity. **DUNE** allows quite general data decompositions but not arbitrary data decompositions. Each entity in a process has a partition type value assigned to it. There are five different possible partition type values:

*interior, border, overlap, front and ghost.*

Entities of codimension 0 are restricted to the three partition types *interior*, *overlap* and *ghost*. Entities of codimension greater than 0 may take all partition type values. The codimension 0 entities with partition type *interior* form a non-overlapping decomposition of the entity set, i.e. for each entity of codimension 0 there is exactly one process where this entity has partition type *interior*. Moreover, the codimension 0 leaf entities in process number  $i$  form a subdomain  $\Omega_i \subseteq \Omega$  and all the  $\Omega_i$ ,  $0 \leq i < P$ , form a nonoverlapping decomposition of the computational domain  $\Omega$ . The leaf entities of codimension 0 in a process  $i$  with partition types *interior* or *overlap* together form a subdomain  $\hat{\Omega}_i \subseteq \Omega$ .

Now the partition types of the entities in process  $i$  with codimension greater 0 can be determined according to the following table:

Entity located in	Partition Type value
$B_i = \partial\Omega_i \setminus \partial\Omega$	<i>border</i>
$\overline{\Omega_i} \setminus B_i$	<i>interior</i>
$F_i = \partial\hat{\Omega}_i \setminus \partial\Omega \setminus B_i$	<i>front</i>
$\hat{\Omega}_i \setminus (B_i \cup F_i)$	<i>overlap</i>
Rest	<i>ghost</i>

The assignment of partition types is illustrated for three different examples in Figure 8.1. Each example shows a two-dimensional structured grid with  $6 \times 4$  elements (in gray). The entities stored in some process  $i$  are shown in color, where color indicates the partition type as explained in the caption. The first row shows an example where process  $i$  has codimension 0 entities of all three partition types *interior*, *overlap* and *ghost* (leftmost picture in first row). The corresponding assignment of partition

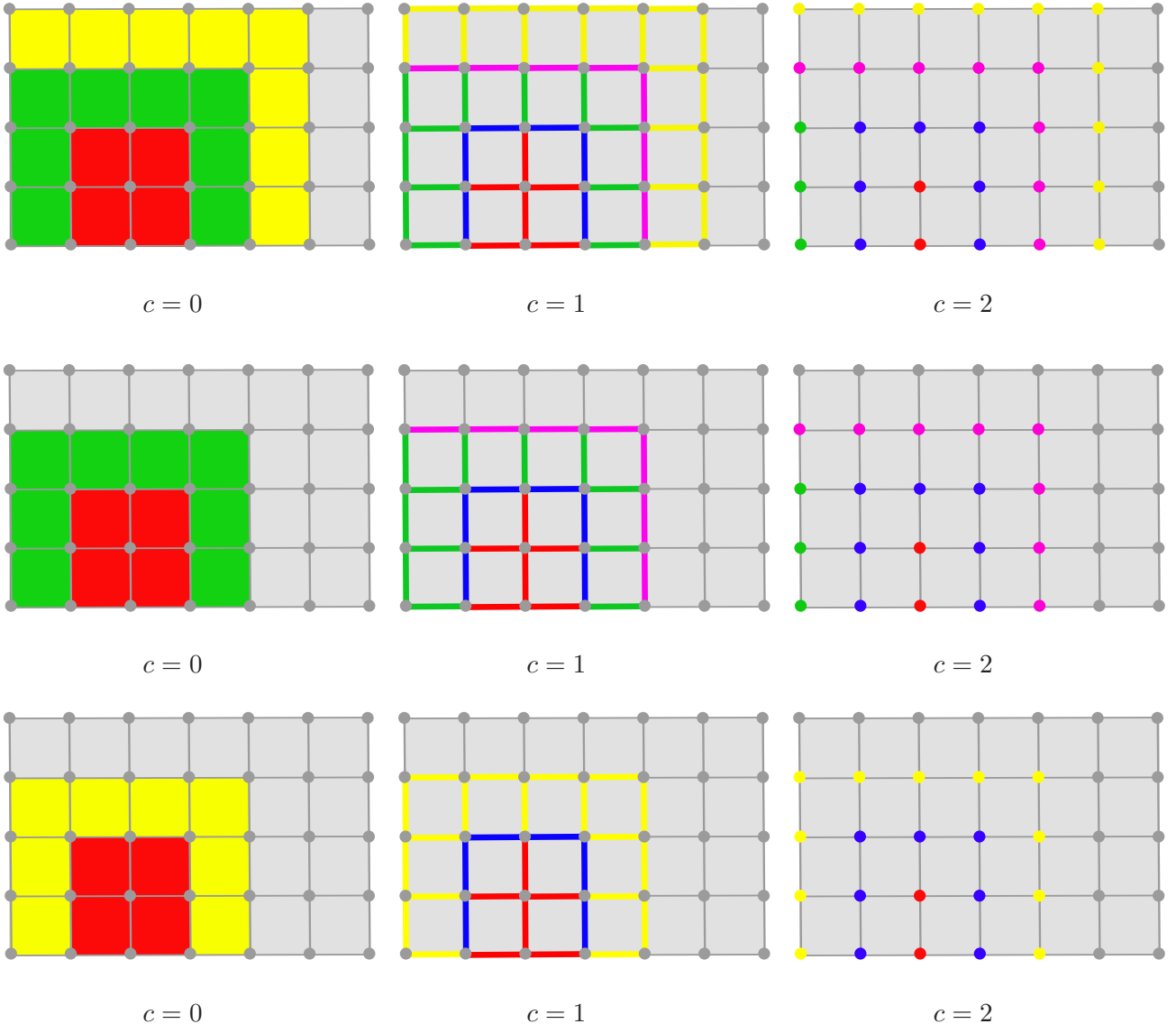


Figure 8.1: Color coded illustration of different data decompositions: interior (red), border (blue), overlap (green), front (magenta) and ghost (yellow), gray encodes entities not stored by the process. First row shows case with interior, overlap and ghost entities, second row shows a case with interior and overlap without ghost and the last row shows a case with interior and ghost only.

types to entities of codimension 1 and 2 is then shown in the middle and right most picture. A grid implementation can choose to omit the partition type *overlap* or *ghost* or both, but not *interior*. The middle row shows an example where an *interior* partition is extended by an *overlap* and no *ghost* elements are present. This is the model used in YaspGrid. The last row shows an example where the *interior* partition is extended by one row of *ghost* cells. This is the model used in UGGrid and ALUGrid.

## 8.2 Communication Interfaces

This section explains how the exchange of data between the partitions in different processes is organized in a flexible and portable way.

The abstract situation is that data has to be sent from a copy of a distributed entity in a process to one or more copies of the same entity in other processes. Usually data has to be sent not only for one entity but for many entities at a time, thus it is more efficient pack all data that goes to the same destination process into a single message. All entities for which data has to be sent or received form a so-called *communication interface*. As an example let us define the set  $X_{i,j}^c$  as the set of all entities of codimension  $c$  in process  $i$  with partition type *interior* or *border* that have a copy in process  $j$  with any partition type. Then in the communication step process  $i$  will send one message to any other process  $j$  when  $X_{i,j}^c \neq \emptyset$ . The message contains some data for every entity in  $X_{i,j}^c$ . Since all processes participate in the communication step, process  $i$  will receive data from a process  $j$  whenever  $X_{j,i}^c \neq \emptyset$ . This data corresponds to entities in process  $i$  that have a copy in  $X_{j,i}^c$ .

A **DUNE** grid offers a selection of predefined interfaces. The example above would use the parameter `InteriorBorder_All_Interface` in the communication function. After the selection of the interface it remains to specify the data to be sent per entity and how the data should be processed at the receiving end. Since the data is in user space the user has to write a small class that encapsulates the processing of the data at the sending and receiving end. The following listing shows an example for a so-called data handle:

### Listing 26 (File `dune-grid-howto/parfvdatahandle.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__
4 #define __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__
5
6 #include <dune/grid/common/datahandleif.hh>
7
8 // A DataHandle class to exchange entries of a vector
9 template<class M, class V> // mapper type and vector type
10 class VectorExchange
11 : public Dune::CommDataHandleIF<VectorExchange<M,V>,
12     typename V::value_type>
13 {
14 public:
15     //! export type of data for message buffer
16     typedef typename V::value_type DataType;
17
18     //! returns true if data for this codim should be communicated
19     bool contains (int dim, int codim) const
20     {
21         return (codim==0);
22     }

```

## 8 Parallelism

```

23
24 //! returns true if size per entity of given dim and codim is a constant
25 bool fixedsize (int dim, int codim) const
26 {
27     return true;
28 }
29
30 //! how many objects of type DataType have to be sent for a given entity
31
32     Note: Only the sender side needs to know this size.
33     */
34 template<class EntityType>
35 size_t size (EntityType& e) const
36 {
37     return 1;
38 }
39
40 //! pack data from user to message buffer
41 template<class MessageBuffer, class EntityType>
42 void gather (MessageBuffer& buff, const EntityType& e) const
43 {
44     buff.write(c[mapper.map(e)]);
45 }
46
47 //! unpack data from message buffer to user
48
49     n is the number of objects sent by the sender
50     */
51 template<class MessageBuffer, class EntityType>
52 void scatter (MessageBuffer& buff, const EntityType& e, size_t n)
53 {
54     DataType x;
55     buff.read(x);
56     c[mapper.map(e)]=x;
57 }
58
59 //! constructor
60 VectorExchange (const M& mapper_, V& c_)
61 : mapper(mapper_), c(c_)
62 {}
63
64 private:
65     const M& mapper;
66     V& c;
67 };
68
69 #endif // _DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH_

```

Every instance of the `VectorExchange` class template conforms to the data handle concept. It defines a type `DataType` which is the type of objects that are exchanged in the messages between the processes. The method `contains` should return true for all codimensions that participate in the data exchange. Method `fixedsize` should return true when, for the given codimension, the same number of data items per entity is sent. If `fixedsize` returns false the method `size` is called for each entity in order to ask for the number of items of type `DataType` that are to be sent for the given entity. Note that this information has only to be given at the sender side. Then the method `gather` is called for each entity in a communication interface on the sender side in order to pack the data for this entity into the message buffer. The message buffer itself is realized as an output stream that accepts data of type `DataType`. After exchanging the data via message passing the `scatter` method is called for each entity at the receiving end. Here the data is read from the message buffer and stored in the user's data



structures. The message buffer is realized as an input stream delivering items of type `DataType`. In the `scatter` method it is up to the user how the data is to be processed, e. g. one can simply overwrite (as is done here), add or compute a maximum.

### 8.3 Parallel finite volume scheme

In this section we parallelize the (nonadaptive!) cell centered finite volume scheme. Essentially only the `evolve` method has to be parallelized. The following listing shows the parallel version of this method. Compare this with listing 18 on page 48.

#### Listing 27 (File `dune-grid-howto/parevolve.hh`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #ifndef __DUNE_GRID_HOWTO_PAREVOLVE_HH__
4 #define __DUNE_GRID_HOWTO_PAREVOLVE_HH__
5
6 #include "parfvdatahandle.hh"
7
8 #include <dune/grid/common/gridenums.hh>
9 #include <dune/common/fvector.hh>
10
11 template<class G, class M, class V>
12 void parevolve (const G& grid, const M& mapper, V& c, double t, double& dt)
13 {
14     // check data partitioning
15     assert(grid.overlapSize(0)>0 || (grid.ghostSize(0)>0));
16
17     // first we extract the dimensions of the grid
18     const int dim = G::dimension;
19     const int dimworld = G::dimensionworld;
20
21     // type used for coordinates in the grid
22     typedef typename G::ctype ct;
23
24     // type for grid view on leaf part
25     typedef typename G::LeafGridView GridView;
26
27     // iterator type
28     typedef typename GridView::template Codim<0>::
29     template Partition<Dune::All_Partition>::Iterator LeafIterator;
30
31     // leaf entity geometry
32     typedef typename LeafIterator::Entity::Geometry LeafGeometry;
33
34     // intersection iterator type
35     typedef typename GridView::IntersectionIterator IntersectionIterator;
36
37     // type of intersection
38     typedef typename IntersectionIterator::Intersection Intersection;
39
40     // intersection geometry
41     typedef typename Intersection::Geometry IntersectionGeometry;
42
43     // entity pointer type
44     typedef typename G::template Codim<0>::EntityPointer EntityPointer;
45
46     // allocate a temporary vector for the update
47     V update(c.size());
48     for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;

```

## 8 Parallelism

```

49
50 // initialize dt very large
51 dt = 1E100;
52
53 // get grid view instance on leaf grid
54 GridView gridView = grid.leafGridView();
55
56 // compute update vector and optimum dt in one grid traversal
57 // iterate over all entities, but update is only used on interior entities
58 LeafIterator endit = gridView.template end<0,Dune::All_Partition>();
59 for (LeafIterator it = gridView.template begin<0,Dune::All_Partition>(); it!=endit; ++it)
60 {
61     // cell geometry
62     const LeafGeometry geo = it->geometry();
63
64     // cell volume
65     double volume = geo.volume();
66
67     // cell index
68     int indexi = mapper.map(*it);
69
70     // variable to compute sum of positive factors
71     double sumfactor = 0.0;
72
73     // run through all intersections with neighbors and boundary
74     const IntersectionIterator isend = gridView.iend(*it);
75     for( IntersectionIterator is = gridView.ibegin(*it); is != isend; ++is )
76     {
77         const Intersection &intersection = *is;
78
79         // get geometry type of face
80         const IntersectionGeometry igeo = intersection.geometry();
81
82         // get normal vector scaled with volume
83         Dune::FieldVector< ct, dimworld > integrationOuterNormal
84             = intersection.centerUnitOuterNormal();
85         integrationOuterNormal *= igeo.volume();
86
87         // center of face in global coordinates
88         Dune::FieldVector< ct, dimworld > faceglobal = igeo.center();
89
90         // evaluate velocity at face center
91         Dune::FieldVector<double,dim> velocity = u(faceglobal,t);
92
93         // compute factor occuring in flux formula
94         double factor = velocity*integrationOuterNormal/volume;
95
96         // for time step calculation
97         if (factor>=0) sumfactor += factor;
98
99         // handle interior face
100         if( intersection.neighbor() )
101         {
102             // access neighbor
103             EntityPointer outside = intersection.outside();
104             int indexj = mapper.map(*outside);
105
106             const int insideLevel = it->level();
107             const int outsideLevel = outside->level();
108
109             // handle face from one side
110             if( (insideLevel > outsideLevel)
111                 || ((insideLevel == outsideLevel) && (indexi < indexj)) )

```

```

112     {
113         // compute factor in neighbor
114         const LeafGeometry nbgeo = outside->geometry();
115         double nbvolume = nbgeo.volume();
116         double nbfactor = velocity*integrationOuterNormal/nbvolume;
117
118         if( factor < 0 )          // inflow
119         {
120             update[indexi] -= c[indexj]*factor;
121             update[indexj] += c[indexj]*nbfactor;
122         }
123         else          // outflow
124         {
125             update[indexi] -= c[indexi]*factor;
126             update[indexj] += c[indexi]*nbfactor;
127         }
128     }
129 }
130
131 // handle boundary face
132 if( intersection.boundary() )
133 {
134     if( factor < 0 )          // inflow, apply boundary condition
135         update[indexi] -= b(faceglobal,t)*factor;
136     else          // outflow
137         update[indexi] -= c[indexi]*factor;
138 }
139 }          // end all intersections
140
141 // compute dt restriction
142 if (it->partitionType()==Dune::InteriorEntity)
143     dt = std::min(dt,1.0/sumfactor);
144
145 }          // end grid traversal
146
147 // global min over all partitions
148 dt = grid.comm().min(dt);
149 // scale dt with safety factor
150 dt *= 0.99;
151
152 // exchange update
153 VectorExchange<M,V> dh(mapper,update);
154 grid.template
155 communicate<VectorExchange<M,V> >(dh,Dune::InteriorBorder_All_Interface,
156                                     Dune::ForwardCommunication);
157
158 // update the concentration vector
159 for (unsigned int i=0; i<c.size(); ++i)
160     c[i] += dt*update[i];
161
162 return;
163 }
164
165 #endif // _DUNE_GRID_HOWTO_PAREVOLVE_HH_

```

The first difference to the sequential version is in line 15 where it is checked that the grid provides an overlap of at least one element. The overlap may be either of partition type *overlap* or *ghost*. The finite volume scheme itself only computes the updates for the elements with partition type *interior*.

In order to iterate over entities with a specific partition type the leaf and level iterators can be parametrized by an additional argument `PartitionIteratorType` as shown in line 29. If the argument `All_Partition` is given then all entities are processed, regardless of their partition type. This is also

the default behavior of the level and leaf iterators. If the partition iterator type is specified explicitly in an iterator the same argument has also to be specified in the begin and end methods on the grid as shown in lines 58-59.

The next change is in line 142 where the computation of the optimum stable time step is restricted to elements of partition type *interior* because only those elements have all neighboring elements locally available. Next, the global minimum of the time steps sizes determined in each process is taken in line 148. For collective communication each grid returns a collective communication object with its `comm()` method which allows to compute global minima and maxima, sums, broadcasts and other functions.

Finally the updates computed on the *interior* cells in each process have to be sent to all copies of the respective entities in the other processes. This is done in lines 153-156 using the data handle described above. The `communicate` method on the grid uses the data handle to assemble the message buffers, exchanges the data and writes the data into the user's data structures.

Finally, we need a new main program, which is in the following listing:

### Listing 28 (File `dune-grid-howto/parfinitevolume.cc`)

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 // vi: set et ts=4 sw=2 sts=2:
3 #include "config.h"           // know what grids are present
4 #include <iostream>           // for input/output to shell
5 #include <fstream>           // for input/output to files
6 #include <vector>             // STL vector class
7 #include <dune/grid/common/mcmgmapper.hh> // mapper class
8 #include <dune/common/parallel/mpihelper.hh> // include mpi helper class
9
10
11 // checks for defined gridtype and includes appropriate dgfparser implementation
12 #include "vtkout.hh"
13 #include "unitcube.hh"
14 #include "transportproblem2.hh"
15 #include "initialize.hh"
16 #include "parfvdatahandle.hh"
17 #include "parevolve.hh"
18
19
20 //=====
21 // the time loop function working for all types of grids
22 //=====
23
24 template<class G>
25 void partimeloop (const G& grid, double tend)
26 {
27     // make a mapper for codim 0 entities in the leaf grid
28     Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,Dune::MCMGElementLayout>
29     mapper(grid);
30
31     // allocate a vector for the concentration
32     std::vector<double> c(mapper.size());
33
34     // initialize concentration with initial values
35     initialize(grid,mapper,c);
36     vtkout(grid,c,"pconc",0,0.0,grid.comm().rank());
37
38     // now do the time steps
39     double t=0,dt;
40     int k=0;
41     const double saveInterval = 0.1;
42     double saveStep = 0.1;

```

## 8 Parallelism

```

43 int counter = 1;
44 while (t<tend)
45 {
46     // augment time step counter
47     k++;
48
49     // apply finite volume scheme
50     parevolve(grid,mapper,c,t,dt);
51
52     // augment time
53     t += dt;
54
55     // check if data should be written
56     if (t >= saveStep)
57     {
58         // write data
59         vtkout(grid,c,"pconc",counter,t,grid.comm().rank());
60
61         //increase counter and saveStep for next interval
62         saveStep += saveInterval;
63         ++counter;
64     }
65
66     // print info about time, timestep size and counter
67     if (grid.comm().rank()==0)
68         std::cout << "k=" << k << "┐t=" << t << "┐dt=" << dt << std::endl;
69 }
70 vtkout(grid,c,"pconc",counter,tend,grid.comm().rank());
71 }
72
73 //=====
74 // The main function creates objects and does the time loop
75 //=====
76
77 int main (int argc , char ** argv)
78 {
79     // initialize MPI, finalize is done automatically on exit
80     Dune::MPIHelper::instance(argc,argv);
81
82     // start try/catch block to get error messages from dune
83     try {
84         using namespace Dune;
85
86         UnitCube<YaspGrid<2>,64> uc;
87         uc.grid().globalRefine(2);
88         partimeloop(uc.grid(),0.5);
89
90         /* To use an alternative grid implementations for parallel computations,
91            uncomment exactly one definition of uc2 and the line below. */
92         // #define LOAD_BALANCING
93
94         // UGGrid supports parallelization in 2 or 3 dimensions
95 #if HAVE_UG
96         // typedef UGGrid< 2 > GridType;
97         // typedef UnitCube< GridType, 2 > uc2;
98 #endif
99
100         // ALUGRID supports parallelization in 3 dimensions only
101 #if HAVE_ALUGRID
102         // typedef Dune::ALUGrid< 3, 3, Dune::cube, Dune::nonconforming > GridType;
103         // typedef Dune::ALUGrid< 3, 3, Dune::simplex, Dune::nonconforming > GridType;
104         // typedef UnitCube< GridType, 1 > uc2;
105 #endif

```

```

106
107 #ifndef LOAD_BALANCING
108
109     // refine grid until upper limit of level
110     uc2.grid().globalRefine( 6 );
111
112     // re-partition grid for better load balancing
113     uc2.grid().loadBalance();
114
115     // do time loop until end time 0.5
116     partimeloop(uc2.grid(), 0.5);
117 #endif
118
119 }
120 catch (std::exception & e) {
121     std::cout << "STL_ERROR:" << e.what() << std::endl;
122     return 1;
123 }
124 catch (Dune::Exception & e) {
125     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
126     return 1;
127 }
128 catch (...) {
129     std::cout << "Unknown_ERROR" << std::endl;
130     return 1;
131 }
132
133 // done
134 return 0;
135 }

```

A difference to the sequential program can be found in line 67 where the printing of the data of the current time step is restricted to the process with rank 0. **YaspGrid** does not support dynamical load balancing and therefore needs to start with a sufficiently fine grid that allows a reasonable partition where each processes gets a non-empty part of grid. This is why we do not use DGF Files in the parallel example and initialize the grid by the **UnitCube** class instead. For **YaspGrid** this allows an easy selection of the grid's initial coarseness through the second template argument of the **UnitCube**. This argument should be chosen sufficiently high, because after each global refinement step the overlap region grows and therefore the communication overhead increases.

If you want to use a grid with support for dynamical load balancing, uncomment one of the possible definitions for such a grid in the code and define the macro **LOAD\_BALANCING**. In this case in line 113 the method **loadBalance** is called on the grid. This method re-partitions the grid in a way such that on every partition there is an equal amount of grid elements.



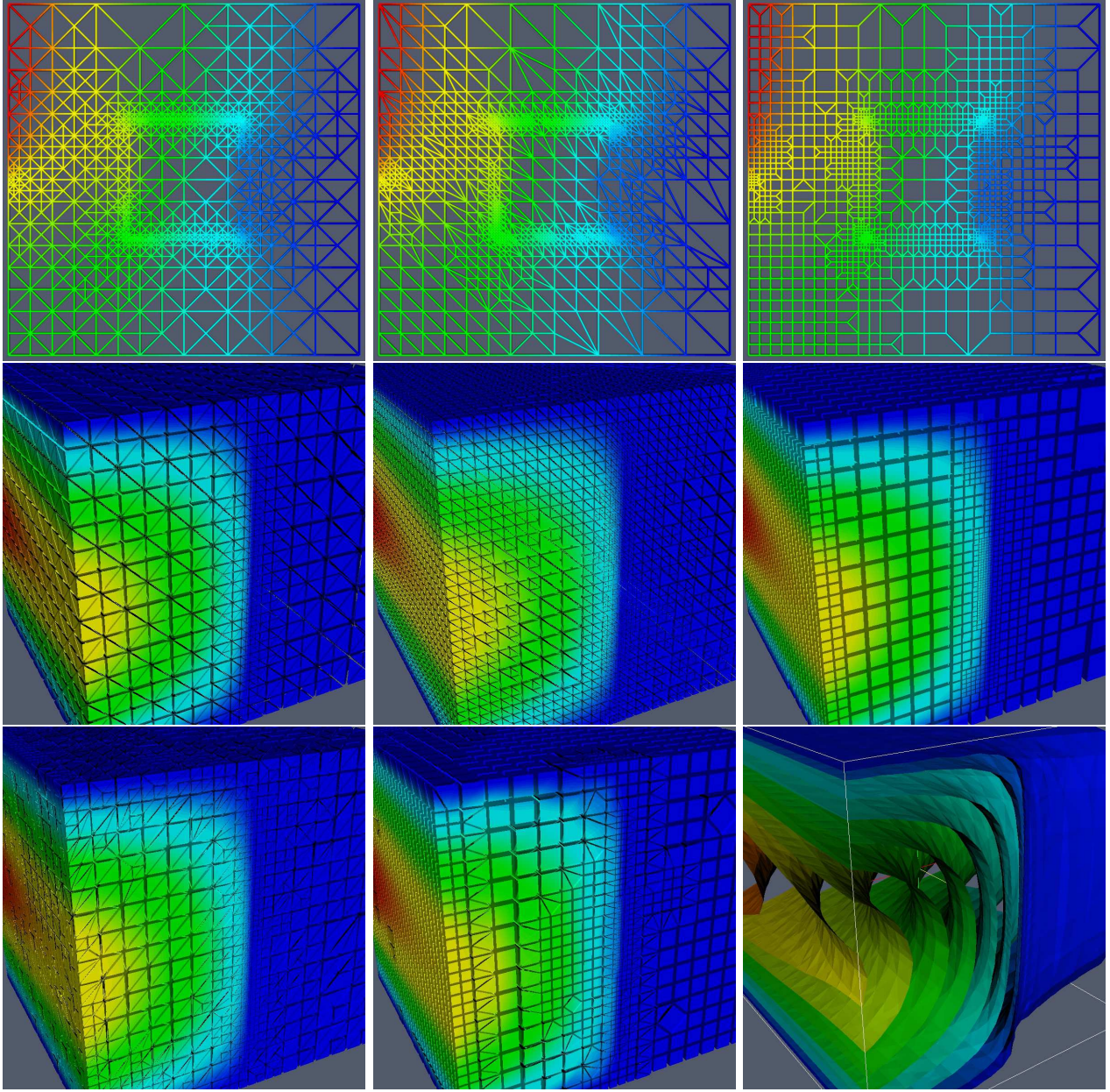


Figure 8.2: Adaptive solution of an elliptic model problem with  $P_1$  conforming finite elements and residual based error estimator. Illustrates that adaptive finite element algorithm can be formulated independent of dimension, element type and refinement scheme. From top to bottom, left to right: Alberta (bisection, 2d), UG (red/green on triangles), UG (red/green on quadrilaterals), Alberta (bisection, 3d), ALU (hanging nodes on tetrahedra), ALU (hanging nodes on hexahedra), UG (red/green on tetrahedra), UG (red/green on hexahedra, pyramids and tetrahedra), isosurfaces of solution.

## Bibliography

- [1] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.
- [2] P. Bastian, K. Birken, S. Lang, K. Johannsen, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG: A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1:27–40, 1997.
- [3] A. Dedner, C. Rohde, B. Schupp, and M. Wesenberg. A parallel, load balanced mhd code on locally adapted, unstructured grids in 3d. *Computing and Visualization in Science*, 7:79–96, 2004.
- [4] P. Deuffhard and A. Hohmann. *Numerische Mathematik I*. Walter de Gruyter, 1993.
- [5] Grape Web Page. <http://www.iam.uni-bonn.de/grape/main.html>.
- [6] Paraview Web Page. <http://www.paraview.org/HTML/Index.html>.
- [7] Visualization Toolkit Web Page. <http://public.kitware.com/VTK/>.
- [8] K. Siebert and A. Schmidt. *Design of adaptive finite element software: The finite element toolbox ALBERTA*. Springer, 2005.
- [9] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.
- [10] D. Vandervoorde and N. M. Josuttis. *C++ Templates — The complete guide*. Addison-Wesley, 2003.
- [11] T. Veldhuizen. Techniques for scientific C++. Technical report, Indiana University, 1999. Computer Science Department.