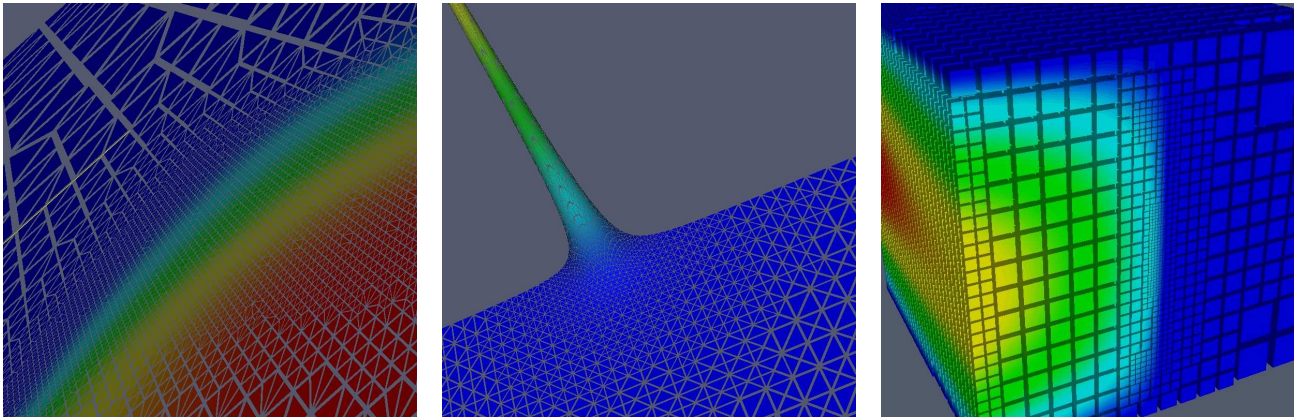


The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO

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

Version 2.1.0, April 29, 2011



*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

1	Introduction	6
1.1	What is DUNE anyway?	6
1.2	Download	7
1.3	Installation	7
1.4	Code documentation	8
1.5	Licence	8
2	Getting started	9
2.1	Creating your first grid	9
2.2	Traversing a grid — A first look at the grid interface	11
3	The DUNE grid interface	16
3.1	Grid definition	16
3.2	Concepts	18
3.2.1	Common types	18
3.2.2	Concepts of the DUNE grid interface	19
3.3	Propagation of type information	20
4	Constructing grid objects	21
4.1	Creating Structured Grids	21
4.2	Reading Unstructured Grids from Files	22
4.3	The DUNE 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
5	Quadrature rules	31
5.1	Numerical integration	31
5.2	Functors	32
5.3	Integration over a single element	32
5.4	Integration with global error estimation	33
6	Attaching user data to a grid	37
6.1	Mappers	37
6.2	Visualization of discrete functions	38
6.3	Cell centered finite volumes	43
6.3.1	Numerical Scheme	43
6.3.2	Implementation	45
6.4	A FEM example: The Poisson equation	52

Contents

6.4.1	Implementation	55
7	Adaptivity	63
7.1	Adaptive integration	63
7.1.1	Adaptive multigrid integration	63
7.1.2	Implementation of the algorithm	64
7.2	Adaptive cell centered finite volumes	67
8	Parallelism	75
8.1	DUNE Data Decomposition Model	75
8.2	Communication Interfaces	77
8.3	Parallel finite volume scheme	79

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 generic 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 $\geq 3.4.1$ and Intel `icc`, version 7.0 or 8.0.

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/>

1 Introduction

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`¹ software available elsewhere. The code documentation can either be built locally on your machine (in html and other formats, e.g. \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 License², 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.

¹<http://www.stack.nl/~dimitri/doxygen/>

²<http://www.gnu.org/licenses/gpl.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 // $Id$
2
3 // Dune includes
4 #include "config.h"           // file constructed by ./configure script
5 #include <dune/grid/sgrid.hh> // load sgrid definition
6 #include <dune/grid/common/gridinfo.hh> // definition of gridinfo
7 #include <dune/common/mpihelper.hh> // include mpi helper class
8
9
10 int main(int argc, char **argv)
11 {
12     // initialize MPI, finalize is done automatically on exit
13     Dune::MPIHelper::instance(argc,argv);
14
15     // start try/catch block to get error messages from dune
16     try{
17         // make a grid
18         const int dim=3;
19         typedef Dune::SGrid<dim,dim> GridType;
20         Dune::FieldVector<int,dim> N(3);
21         Dune::FieldVector<GridType::ctype,dim> L(-1.0);
22         Dune::FieldVector<GridType::ctype,dim> H(1.0);
23         GridType grid(N,L,H);
24
25         // print some information about the grid
26         Dune::gridinfo(grid);
27     }
28     catch (std::exception & e) {
29         std::cout << "STL_ERROR:" << e.what() << std::endl;
30         return 1;
31     }
32     catch (Dune::Exception & e) {
33         std::cout << "DUNE_ERROR:" << e.what() << std::endl;
34         return 1;
35     }
36     catch (...) {
37         std::cout << "Unknown_ERROR" << std::endl;
38         return 1;
39     }
40
41     // done
42     return 0;
43 }
```

2 Getting started

This program is quite simple. It starts with some includes in lines 4-6. 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 18. 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 19 the type `GridType` using the selected value for the dimension. All identifiers of the **DUNE** framework are within the `Dune` namespace.

Lines 20-22 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 23 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 descendents obtained from refinement form a tree structure. All elements at depth n of a refinement tree form the level- n -grid. All elements which are leafs 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 // $Id$
2
3 // C/C++ includes
4 #include<iostream>           // for standard I/O
5
6 // Dune includes
7 #include"config.h"           // file constructed by ./configure script
8 #include<dune/grid/sgrid.hh> // load sgrid definition
9 #include <dune/common/mpihelper.hh> // include mpi helper class
10
11
12 // example for a generic algorithm that traverses
13 // the entities of a given mesh in various ways
14 template<class G>
15 void traversal (G& grid)
16 {
17     // first we extract the dimensions of the grid
18     const int dim = G::dimension;
19
20     // type used for coordinates in the grid
21     // such a type is exported by every grid implementation
22     typedef typename G::ctype ct;
23
24     // Leaf Traversal
25     std::cout << "***_Traverse_codim_0_leaves" << std::endl;
26
27     // type of the GridView used for traversal
28     // every grid exports a LeafGridView and a LevelGridView
29     typedef typename G :: LeafGridView LeafGridView;
30
31     // get the instance of the LeafGridView
32     LeafGridView leafView = grid.leafView();
33
34     // Get the iterator type
35     // Note the use of the typename and template keywords
36     typedef typename LeafGridView::template Codim<0>::Iterator ElementLeafIterator;
37
38     // iterate through all entities of codim 0 at the leafs
39     int count = 0;
40     for (ElementLeafIterator it = leafView.template begin<0>();
41          it!=leafView.template end<0>(); ++it)
42     {
43         Dune::GeometryType gt = it->type();
44         std::cout << "visiting_leaf_" << gt
45                   << "_with_first_vertex_at_" << it->geometry().corner(0)
46                   << std::endl;
47         count++;
48     }
49

```

2 Getting started

```
50 std::cout << "there_are/is_" << count << "_leaf_element(s)" << std::endl;
51
52 // Leafwise traversal of codim dim
53 std::cout << std::endl;
54 std::cout << "***_Traverse_codim_" << dim << "_leaves" << std::endl;
55
56 // Get the iterator type
57 // Note the use of the typename and template keywords
58 typedef typename LeafGridView :: template Codim<dim>
59     :: Iterator VertexLeafIterator;
60
61 // iterate through all entities of codim 0 on the given level
62 count = 0;
63 for (VertexLeafIterator it = leafView.template begin<dim>();
64      it!=leafView.template end<dim>(); ++it)
65 {
66     Dune::GeometryType gt = it->type();
67     std::cout << "visiting_" << gt
68               << "_at_" << it->geometry().corner(0)
69               << std::endl;
70     count++;
71 }
72 std::cout << "there_are/is_" << count << "_leaf_vertices(s)"
73           << std::endl;
74
75 // Levelwise traversal of codim 0
76 std::cout << std::endl;
77 std::cout << "***_Traverse_codim0_level-wise" << std::endl;
78
79 // type of the GridView used for traversal
80 // every grid exports a LeafGridView and a LevelGridView
81 typedef typename G :: LevelGridView LevelGridView;
82
83 // Get the iterator type
84 // Note the use of the typename and template keywords
85 typedef typename LevelGridView :: template Codim<0>
86     :: Iterator ElementLevelIterator;
87
88 // iterate through all entities of codim 0 on the given level
89 for (int level=0; level<=grid.maxLevel(); level++)
90 {
91     // get the instance of the LeafGridView
92     LevelGridView levelView = grid.levelView(level);
93
94     count = 0;
95     for (ElementLevelIterator it = levelView.template begin<0>();
96          it!=levelView.template end<0>(); ++it)
97     {
98         Dune::GeometryType gt = it->type();
99         std::cout << "visiting_" << gt
100                << "_with_first_vertex_at_" << it->geometry().corner(0)
101                << std::endl;
102         count++;
103     }
104     std::cout << "there_are/is_" << count << "_element(s)_on_level_"
105             << level << std::endl;
106     std::cout << std::endl;
107 }
108 }
109
110
111 int main(int argc, char **argv)
112 {
```

2 Getting started

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

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 127 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 130. This function is given in lines 14-108.

The function `traversal` is a function template that is parameterized 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 18. Next, each **DUNE** grid defines a type that it uses to represent positions. This type is extracted in line 22 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 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.

2 Getting started

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 29 a type for a `GridView` on the leaf grid is defined.

Line 36 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 40 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 42-48 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 43. 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 44 prints the geometry type and the position of the first corner of the element. Then line 47 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 59 for later use. The `for`-loop starting in line 63 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 81-107 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)

```
*** 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)
```

2 Getting started

```
*** 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 nonoverlapping 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 Ω_0 is required to be identical to Ω . If $\Omega_0 = \Omega_1 = \dots = \Omega_J$ the grid is globally refined, otherwise it is locally refined. The grid that discretizes Ω_0 is called the macro grid and its elements

3 The **DUNE** grid interface

the macro elements. The grid for Ω_{l+1} is obtained from the grid for Ω_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 Ω_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 Ω 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 Ω 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.

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] = 1.0;  
  
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`¹ is an open-source geometry modeler and grid generator. It allows to define geometries using a boundary representation (interactively and via its own modelling 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` visualization toolbox.² To read `AmiraMesh` files you need to have the library `libamiramesh`³ 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

¹<http://geuz.org/gmsh/>

²<http://www.amiravis.de/>

³<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**⁴.

Here we only give a short introduction. The configuration `--with-grid-dim={1,2,3}` must be provided during configuration run in order to use the DGF parser. A default grid type can be chosen with the configuration option `--with-grid-type=ALBERTAGRID`. Further possible grid types are `ALUGRID_CUBE`, `ALUGRID_SIMPLEX`, `ALUGRID_CONFORM`, `ONEDGRID`, `SGRID`, `UGGRID`, and `YASPGRID`. Note that both values will also be changeable later. If the `--with-grid-dim` option was not provided during configuration the DGF grid type definition will not work. Nevertheless, the grid parser will work but the grid type has to be defined by the user and the appropriate DGF parser specialization has to be included. Assuming the `--with-grid-dim` was provided the DGF grid type definition works by first including `gridtype.hh`.

```
#include <dune/grid/io/file/dgfparser/dgfgridtype.hh>
```

Depending on the pre-configured values of `GRIDDIM` and `GRIDTYPE` a typedef for the grid to use will be provided by including `dgfgridtype.hh`. 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.

```
GridPtr<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.

4.4 The Grid Factory

While there is currently no convention on how 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:

⁴http://www.dune-project.org/doc/doxygen/dune-grid-html/group__DuneGridFormatParser.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 `Leaf` iterators. 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  #ifndef UNITCUBE_HH
2  #define UNITCUBE_HH
3
4  #include <dune/common/exceptions.hh>
5  #include <dune/common/fvector.hh>
6  #include <dune/grid/utility/structuredgridfactory.hh>
7
8  // default implementation for any template parameter
9  template<typename T, int variant>
10 class UnitCube
11 {
12 public:
13     typedef T GridType;
14
15     static const int dim = GridType::dimension;
16
17     // constructor throwing exception
18     UnitCube ()
19     {
20         Dune::FieldVector<typename GridType::ctype,dim> lowerLeft(0);
21         Dune::FieldVector<typename GridType::ctype,dim> upperRight(1);
22         Dune::array<unsigned int,dim> elements;
23         std::fill(elements.begin(), elements.end(), 1);
24
25         switch (variant) {
26             case 1:
27                 grid_ = Dune::StructuredGridFactory<GridType>::createCubeGrid(lowerLeft, upperRight, elements);
28                 break;
29             case 2:
30                 grid_ = Dune::StructuredGridFactory<GridType>::createSimplexGrid(lowerLeft, upperRight, elements);
31                 break;
32             default:
33                 DUNE_THROW( Dune::NotImplemented, "Variant_"
34                     << variant << "_of_unit_cube_not_implemented." );
35         }
36     }
37
38     T& grid ()
39     {
40         return *grid_;
41     }
42
43 private:
44     // the constructed grid object
45     Dune::shared_ptr<T> grid_;

```

4 Constructing grid objects

```
46 };
47
48
49 // include specializations
50 #include "unitcube_sgrid.hh"
51 #include "unitcube_yaspgrid.hh"
52 #include "unitcube_albertagrid.hh"
53 #include "unitcube_alugrid.hh"
54
55 #endif
```

This is a default implementation that uses the utility class `StructuredGridFactory` (from `dune-grid/dune/grid`) 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 #ifndef UNITCUBE_SGRID_HH
2 #define UNITCUBE_SGRID_HH
3
4 #include "unitcube.hh"
5
6 #include <dune/grid/sgrid.hh>
7
8 // SGrid specialization
9 template<int dim>
10 class UnitCube<Dune::SGrid<dim,dim>,1>
11 {
12 public:
13     typedef Dune::SGrid<dim,dim> GridType;
14
15     Dune::SGrid<dim,dim>& grid ()
16     {
17         return grid_;
18     }
19
20 private:
21     Dune::SGrid<dim,dim> grid_;
22 };
23
24 #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 #ifndef UNITCUBE_YASPGRID_HH
2 #define UNITCUBE_YASPGRID_HH
3
4 #include "unitcube.hh"
5
6 #include <dune/grid/yaspgrid.hh>
7
8 // YaspGrid specialization
9 template<int dim, int size>
10 class UnitCube<Dune::YaspGrid<dim>,size>
11 {
12 public:
13     typedef Dune::YaspGrid<dim> GridType;
14
15     UnitCube () : Len(1.0), s(size), p(false),
16 #if HAVE_MPI
17     grid_(MPI_COMM_WORLD,Len,s,p,1)
18 #else
19     grid_(Len,s,p,1)
20 #endif
21     { }
22
23     Dune::YaspGrid<dim>& grid ()
24     {
25         return grid_;
26     }
27
28 private:
29     Dune::FieldVector<double,dim> Len;
30     Dune::FieldVector<int,dim> s;
31     Dune::FieldVector<bool,dim> p;
32     Dune::YaspGrid<dim> grid_;
33 };
34
35 #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 #ifndef UNITCUBE_ALBERTAGRID_HH
2 #define UNITCUBE_ALBERTAGRID_HH
3
4 #include "unitcube.hh"
5 #include "basicunitcube.hh"
6
7 #if HAVE_ALBERTA
8 #include <dune/grid/albertagrid.hh>
9 #include <dune/grid/albertagrid/gridfactory.hh>

```

4 Constructing grid objects

```
10
11 template< int dim >
12 class UnitCube< Dune::AlbertaGrid< dim, dim >, 1 >
13 : public BasicUnitCube< dim >
14 {
15 public:
16     typedef Dune::AlbertaGrid< dim, dim > GridType;
17
18 private:
19     GridType *grid_;
20
21 public:
22     UnitCube ()
23     {
24         Dune::GridFactory< GridType > factory;
25         BasicUnitCube< dim >::insertVertices( factory );
26         BasicUnitCube< dim >::insertSimplices( factory );
27         grid_ = factory.createGrid();
28     }
29
30     ~UnitCube ()
31     {
32         Dune::GridFactory< GridType >::destroyGrid( grid_ );
33     }
34
35     GridType &grid ()
36     {
37         return *grid_;
38     }
39 };
40
41 #endif // #if HAVE_ALBERTA
42
43 #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`)

```
1 #ifndef UNITCUBE_ALUGRID_HH
2 #define UNITCUBE_ALUGRID_HH
3
4 #include "unitcube.hh"
5
6 #if HAVE_ALUGRID
7 #include <dune/grid/alugrid.hh>
8 #include <dune/grid/alugrid/3d/alu3dgridfactory.hh>
9
10 // ALU3dGrid and ALU2dGrid simplex specialization.
11 // Note: element type determined by type
12 template<int dim>
13 class UnitCube<Dune::ALUSimplexGrid<dim,dim>,1>
14 {
15 public:
16     typedef Dune::ALUSimplexGrid<dim,dim> GridType;
```

4 Constructing grid objects

```
17
18 private:
19     Dune::shared_ptr<GridType> grid_;
20
21 public:
22     UnitCube ()
23     {
24         Dune::FieldVector<typename GridType::ctype,dim> lowerLeft(0);
25         Dune::FieldVector<typename GridType::ctype,dim> upperRight(1);
26         Dune::array<unsigned int,dim> elements;
27         std::fill(elements.begin(), elements.end(), 1);
28
29         grid_ = Dune::StructuredGridFactory<GridType>::createSimplexGrid(lowerLeft, upperRight, elements);
30     }
31
32     GridType &grid ()
33     {
34         return *grid_;
35     }
36 };
37
38 // ALU3dGrid hexahedra specialization. Note: element type determined by type
39 template<>
40 class UnitCube<Dune::ALUCubeGrid<3,3>,1>
41 {
42 public:
43     typedef Dune::ALUCubeGrid<3,3> GridType;
44
45 private:
46     Dune::shared_ptr<GridType> grid_;
47
48 public:
49     UnitCube ()
50     {
51         Dune::FieldVector<GridType::ctype,3> lowerLeft(0);
52         Dune::FieldVector<GridType::ctype,3> upperRight(1);
53         Dune::array<unsigned int,3> elements = {1,1,1};
54
55         grid_ = Dune::StructuredGridFactory<GridType>::createCubeGrid(lowerLeft, upperRight, elements);
56     }
57
58     GridType &grid ()
59     {
60         return *grid_;
61     }
62 };
63 #endif
64
65 #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 Δ 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 Δ . 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 ξ_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 Ω 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 ω_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 `d` is its dimension.

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

```

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

```


5 Quadrature rules

```

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

```

Line 22 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 ξ_i and weights w_i .

Line 30 shows the loop over all quadrature points in the quadrature rules. For each quadrature point i the function value at the transformed position (line 33), the weight (line 34) and the integration element (line 35) are computed and summed (line 36).

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 // $Id$
2
3 // Dune includes
4 #include "config.h" // file constructed by ./configure script
5 #include <dune/grid/sgrid.hh> // load sgrid definition
6 #include <dune/common/mpihelper.hh> // include mpi helper class
7
8 #include "functors.hh"
9 #include "integrateentity.hh"
10
11 //! uniform refinement test
12 template<class Grid>
13 void uniformintegration (Grid& grid)
14 {
15     // function to integrate
16     Exp<typename Grid::ctype, Grid::dimension> f;
17
18     // get GridView on leaf grid - type
19     typedef typename Grid :: LeafGridView GridView;
20
21     // get GridView instance
22     GridView gridView = grid.leafView();
23
24     // get iterator type
25     typedef typename GridView :: template Codim<0> :: Iterator LeafIterator;
26
27     // loop over grid sequence
28     double oldvalue=1E100;
29     for (int k=0; k<10; k++)
30     {
31         // compute integral with some order
32         double value = 0.0;
33         LeafIterator eendit = gridView.template end<0>();
34         for (LeafIterator it = gridView.template begin<0>(); it!=eendit; ++it)
35             value += integrateentity(it,f,1);
36
37         // print result and error estimate
38         std::cout << "elements="
39                 << std::setw(8) << std::right
40                 << grid.size(0)
41                 << "\nintegral="
42                 << std::scientific << std::setprecision(12)
43                 << value
44                 << "\nerror=" << std::abs(value-oldvalue)
45                 << std::endl;
46
47         // save value of integral
48         oldvalue=value;
49
50         // refine all elements
51         grid.globalRefine(1);
52     }
53 }
54
55 int main(int argc, char **argv)
56 {
57     // initialize MPI, finalize is done automatically on exit

```

5 Quadrature rules

```

58 Dune::MPIHelper::instance(argc,argv);
59
60 // start try/catch block to get error messages from dune
61 try {
62     using namespace Dune;
63
64     // the GridSelector :: GridType is defined in gridtype.hh and is
65     // set during compilation
66     typedef GridSelector :: GridType Grid;
67
68     // use unitcube from grids
69     std::stringstream dgfFileName;
70     dgfFileName << "grids/unitcube" << Grid :: dimension << ".dgf";
71
72     // create grid pointer
73     GridPtr<Grid> gridPtr( dgfFileName.str() );
74
75     // integrate and compute error with extrapolation
76     uniformintegration( *gridPtr );
77 }
78 catch (std::exception & e) {
79     std::cout << "STL_ERROR:" << e.what() << std::endl;
80     return 1;
81 }
82 catch (Dune::Exception & e) {
83     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
84     return 1;
85 }
86 catch (...) {
87     std::cout << "Unknown_ERROR" << std::endl;
88     return 1;
89 }
90
91 // done
92 return 0;
93 }

```

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

6 Attaching user data to a grid

```

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

The data vector is allocated in line 43. 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 46-57 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 56 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 61-63 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 66-79 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 #ifndef __DUNE_GRID_HOWTO_VERTEXDATA_HH__
2 #define __DUNE_GRID_HOWTO_VERTEXDATA_HH__
3
4 #include<dune/grid/common/mcmgmapper.hh>
5 #include<dune/grid/io/file/vtk/vtkwriter.hh>
6 #if HAVE_GRAPE
7 #include<dune/grid/io/visual/grapedatadisplay.hh>
8 #endif
9
10 Parameter for mapper class
11 This class is only here to show what such a class looks like — it does
12 exactly the same as Dune::MCMGVertexLayout. */
13 template<int dimgrid>
14 struct P1Layout
15 {
16     bool contains (Dune::GeometryType gt)
17     {
18         if (gt.dim()==0) return true;
19         return false;
20     }
21 };
22
23 demonstrate attaching data to elements
24 template<class G, class F>
25 void vertexdata (const G& grid, const F& f)
26 {
27     get dimension and coordinate type from Grid
28     const int dim = G::dimension;
29     typedef typename G::ctype ct;

```


6 Attaching user data to a grid

```

30 typedef typename G::LeafGridView GridView;
31 // determine type of LeafIterator for codimension = dimension
32 typedef typename GridView::template Codim<dim>::Iterator VertexLeafIterator;
33
34 // get grid view on the leaf part
35 GridView gridView = grid.leafView();
36
37 // make a mapper for codim 0 entities in the leaf grid
38 Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P1Layout>
39   mapper(grid);
40
41 // allocate a vector for the data
42 std::vector<double> c(mapper.size());
43
44 // iterate through all entities of codim 0 at the leafs
45 for (VertexLeafIterator it = gridView.template begin<dim>();
46      it!=gridView.template end<dim>(); ++it)
47 {
48     // evaluate functor and store value
49     c[mapper.map(*it)] = f(it->geometry().corner(0));
50 }
51
52 // generate a VTK file
53 // Dune::LeafP1Function<G,double> cc(grid,c);
54 Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafView());
55 vtkwriter.addVertexData(c,"data");
56 vtkwriter.write("vertexdata",Dune::VTKOptions::binaryappended);
57
58 // online visualization with Grape
59 #if HAVE_GRAPE
60 {
61     const int polynomialOrder = 1; // we piecewise linear data
62     const int dimRange = 1; // we have scalar data here
63     // create instance of data display
64     Dune::GrapeDataDisplay<G> grape(grid);
65     // display data
66     grape.displayVector("concentration", // name of data that appears in grape
67                        c, // data vector
68                        gridView.indexSet(), // used index set
69                        polynomialOrder, // polynomial order of data
70                        dimRange); // dimRange of data
71 }
72 #endif
73 }
74 #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 // $Id$
2
3 #include "config.h"
4 #include <iostream>
5 #include <iomanip>
6 #include <stdio.h>
7 #include <dune/common/mpihelper.hh> // include mpi helper class
8
9
10 #include "elementdata.hh"
11 #include "vertexdata.hh"
12 #include "functors.hh"
13 #include "unitcube.hh"
14
15
16 #ifdef GRIDDIM
17 const int dimGrid = GRIDDIM;
18 #else
19 const int dimGrid = 2;
20 #endif
21
22
23 //! supply functor
24 template<class Grid>
25 void dowork ( Grid &grid, int refSteps = 5 )
26 {
27     // make function object
28     Exp<typename Grid::ctype, Grid::dimension> f;
29
30     // refine the grid
31     grid.globalRefine( refSteps );
32
33     // call the visualization functions
34     elementdata(grid,f);
35     vertexdata(grid,f);
36 }
37
38 int main(int argc, char **argv)
39 {
40     // initialize MPI, finalize is done automatically on exit
41     Dune::MPIHelper::instance(argc,argv);
42
43     // start try/catch block to get error messages from dune
44     try
45     {
46         /*
47         UnitCube<Dune::OneDGrid,1> uc0;
48         UnitCube<Dune::YaspGrid<dimGrid>,1> uc1;
49
50         #if HAVE_UG
51         UnitCube< Dune::UGGrid< dimGrid >, 2 > uc2;
52         dowork( uc2.grid(), 3 );
53         #endif
54
55         #if HAVE_ALBERTA
56         {
57             UnitCube< Dune::AlbertaGrid< dimGrid, dimGrid >, 1 > unitcube;
58             // note: The 3d cube cannot be bisected recursively
59             dowork( unitcube.grid(), (dimGrid < 3 ? 6 : 0) );
60         }
61         #endif // #if HAVE_ALBERTA
62         */

```

6 Attaching user data to a grid

```

63
64     UnitCube< Dune::SGrid< dimGrid, dimGrid >, 1 > uc4;
65     dowork( uc4.grid(), 3 );
66
67 #if HAVE_ALUGRID
68     UnitCube< Dune::ALUSimplexGrid< dimGrid, dimGrid >, 1 > uc5;
69     dowork( uc5.grid(), 3 );
70
71 #if GRIDDIM == 3
72     UnitCube< Dune::ALUCubeGrid< dimGrid, dimGrid >, 1 > uc6;
73     dowork( uc6.grid(), 3 );
74 #endif // #if GRIDDIM == 3
75 #endif // #if HAVE_ALUGRID
76 }
77 catch (std::exception & e) {
78     std::cout << "STL_ERROR:" << e.what() << std::endl;
79     return 1;
80 }
81 catch (Dune::Exception & e) {
82     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
83     return 1;
84 }
85 catch (...) {
86     std::cout << "Unknown_ERROR" << std::endl;
87     return 1;
88 }
89
90 // done
91 return 0;
92 }

```

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.

6 Attaching user data to a grid

The grid consists of cells (elements) ω 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 ω_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 ω_i at time t_n . Moreover we subdivide the boundary $\partial\omega_i$ into facets γ_{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 γ_{ij} at time t_n , ν_{ij} is the unit outer normal of the facet γ_{ij} and ϕ 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 γ_{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 Δt^n and the update δ_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 #ifndef __DUNE_GRID_HOWTO_TRANSPORTPROBLEM_HH__
2 #define __DUNE_GRID_HOWTO_TRANSPORTPROBLEM_HH__
3
4 #include <dune/common/fvector.hh>
5 // the initial condition c0
6 template<int dimworld, class ct>
7 double c0 (const Dune::FieldVector<ct,dimworld>& x)
8 {
9     Dune::FieldVector<ct,dimworld> y(0.25);
10    y -= x;
11    if (y.two_norm()<0.125)
12        return 1.0;
13    else
14        return 0.0;
15 }
16
17 // the boundary condition b on inflow boundary
18 template<int dimworld, class ct>
19 double b (const Dune::FieldVector<ct,dimworld>& x, double t)
20 {
21     return 0.0;
22 }
23
24 // the vector field u is returned in r
25 template<int dimworld, class ct>
26 Dune::FieldVector<double,dimworld> u (const Dune::FieldVector<ct,dimworld>& x, double t)
27 {
28     Dune::FieldVector<double,dimworld> r(0.5);
29     r[0] = 1.0;
30     return r;
31 }
32 #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 #ifndef __DUNE_GRID_HOWTO_INITIALIZE_HH__
2 #define __DUNE_GRID_HOWTO_INITIALIZE_HH__
3
4 #include <dune/common/fvector.hh>
5
6 //! initialize the vector of unknowns with initial value
7 template<class G, class M, class V>

```

6 Attaching user data to a grid

```

8 void initialize (const G& grid, const M& mapper, V& c)
9 {
10 // first we extract the dimensions of the grid
11 //const int dim = G::dimension;
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 // leaf iterator type
21 typedef typename GridView::template Codim<0>::Iterator LeafIterator;
22
23 // geometry type
24 typedef typename LeafIterator::Entity::Geometry Geometry;
25
26 // get grid view on leaf part
27 GridView gridView = grid.leafView();
28
29 // iterate through leaf grid and evaluate c0 at cell center
30 LeafIterator endit = gridView.template end<0>();
31 for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
32 {
33 // get geometry
34 const Geometry &gt = it->geometry();
35
36 // get global coordinate of cell center
37 Dune::FieldVector<ct,dimworld> global = gt.center();
38
39 // initialize cell concentration
40 c[mapper.map(*it)] = c0(global);
41 }
42 }
43
44 #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 Δt^n selected by the algorithm is returned.

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

```

1 #ifndef __DUNE_GRID_HOWTO_EVOLVE_HH__
2 #define __DUNE_GRID_HOWTO_EVOLVE_HH__
3
4 #include <dune/common/fvector.hh>
5
6 template<class G, class M, class V>
7 void evolve (const G& grid, const M& mapper, V& c, double t, double& dt)
8 {
9 // first we extract the dimensions of the grid
10 const int dimworld = G::dimensionworld;
11
12 // type used for coordinates in the grid
13 typedef typename G::ctype ct;
14
15 // type of grid view on leaf part
16 typedef typename G::LeafGridView GridView;
17
18 // element iterator type

```

6 Attaching user data to a grid

```
19 typedef typename GridView::template Codim<0>::Iterator LeafIterator;
20
21 // leaf entity geometry
22 typedef typename LeafIterator::Entity::Geometry LeafGeometry;
23
24 // intersection iterator type
25 typedef typename GridView::IntersectionIterator IntersectionIterator;
26
27 // intersection geometry
28 typedef typename IntersectionIterator::Intersection::Geometry IntersectionGeometry;
29
30 // entity pointer type
31 typedef typename G::template Codim<0>::EntityPointer EntityPointer;
32
33 // get grid view on leaf part
34 GridView gridView = grid.leafView();
35
36 // allocate a temporary vector for the update
37 V update(c.size());
38 for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
39
40 // initialize dt very large
41 dt = 1E100;
42
43 // compute update vector and optimum dt in one grid traversal
44 LeafIterator endit = gridView.template end<0>();
45 for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
46 {
47     // cell geometry
48     const LeafGeometry &gt = it->geometry();
49
50
51     // cell volume, assume linear map here
52     double volume = g.volume();
53
54     // cell index
55     int indexi = mapper.map(*it);
56
57     // variable to compute sum of positive factors
58     double sumfactor = 0.0;
59
60     // run through all intersections with neighbors and boundary
61     IntersectionIterator isend = gridView.iend(*it);
62     for (IntersectionIterator is = gridView.ibegin(*it); is!=isend; ++is)
63     {
64         // get geometry type of face
65         const IntersectionGeometry &gtf = is->geometry();
66
67         // get normal vector scaled with volume
68         Dune::FieldVector<ct,dimworld> integrationOuterNormal
69             = is->centerUnitOuterNormal();
70         integrationOuterNormal *= gtf.volume();
71
72         // center of face in global coordinates
73         Dune::FieldVector<ct,dimworld> faceglobal = gtf.center();
74
75         // evaluate velocity at face center
76         Dune::FieldVector<double,dimworld> velocity = u(faceglobal,t);
77
78         // compute factor occuring in flux formula
79         double factor = velocity*integrationOuterNormal/volume;
80
81         // for time step calculation
```

6 Attaching user data to a grid

```

82     if (factor>=0) sumfactor += factor;
83
84     // handle interior face
85     if (is->neighbor()) // "correct" version
86     {
87         // access neighbor
88         EntityPointer outside = is->outside();
89         int indexj = mapper.map(*outside);
90
91         // compute flux from one side only
92         // this should become easier with the new IntersectionIterator functionality!
93         if ( it->level()>outside->level() ||
94             (it->level()==outside->level() && indexi<indexj) )
95         {
96             // compute factor in neighbor
97             const LeafGeometry &nbgt = outside->geometry();
98             double nbvolume = nbgt.volume();
99             double nbfactor = velocity*integrationOuterNormal/nbvolume;
100
101             if (factor<0) // inflow
102             {
103                 update[indexi] -= c[indexj]*factor;
104                 update[indexj] += c[indexj]*nbfactor;
105             }
106             else // outflow
107             {
108                 update[indexi] -= c[indexi]*factor;
109                 update[indexj] += c[indexi]*nbfactor;
110             }
111         }
112     }
113
114     // handle boundary face
115     if (is->boundary())
116     {
117         if (factor<0) // inflow, apply boundary condition
118             update[indexi] -= b(faceglobal,t)*factor;
119         else // outflow
120             update[indexi] -= c[indexi]*factor;
121     }
122 } // end all intersections
123
124 // compute dt restriction
125 dt = std::min(dt,1.0/sumfactor);
126
127 } // end grid traversal
128
129 // scale dt with safety factor
130 dt *= 0.99;
131
132 // update the concentration vector
133 for (unsigned int i=0; i<c.size(); ++i)
134     c[i] += dt*update[i];
135
136 return;
137 }
138
139 #endif // _DUNE_GRID_HOWTO_EVOLVE_HH_

```

Lines 44-127 contain the loop over all leaf elements where the optimum Δt^n and the cell updates δ_i are computed. The update vector is allocated in line 37, where we assume that V is a container with copy constructor and size method.

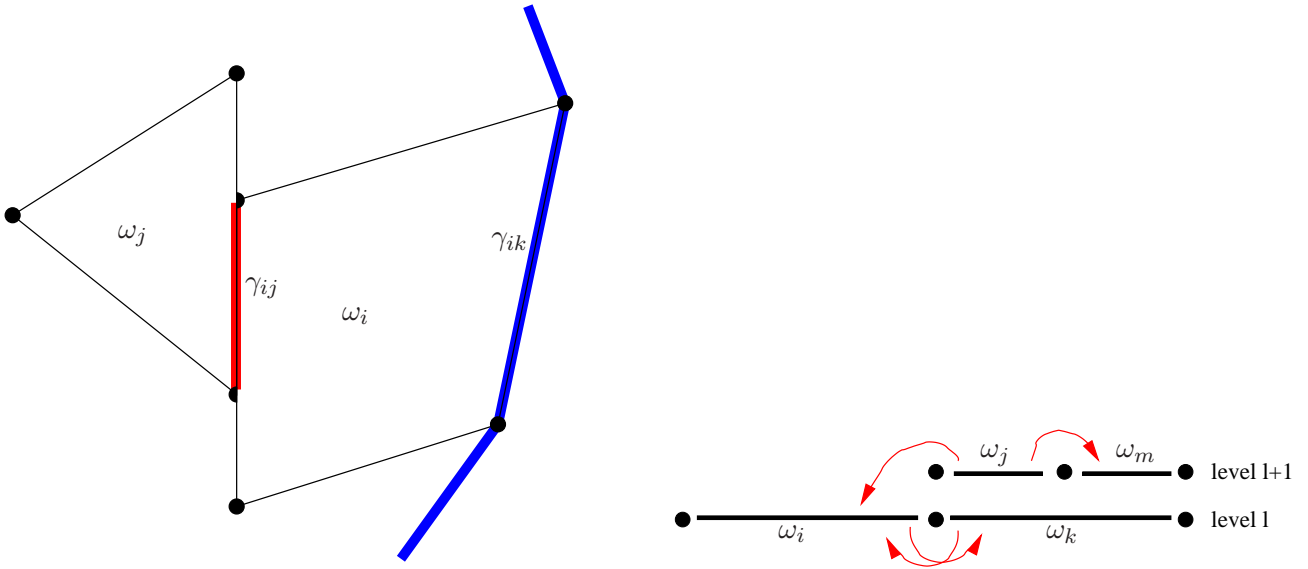


Figure 6.1: Left: intersection with other elements and the boundary, right: intersections in the case of locally refined grids.

The computation of the fluxes is done in lines 61-122. An `IntersectionIterator` is used to access all intersections γ_{ij} of a leaf element ω_i . For a full documentation on the `Intersection` class, we refer to the doxygen module page on Intersections¹. An `Intersection` is with another element ω_j if the `neighbor()` method of the iterator returns true (line 85) or with the external boundary if `boundary()` returns true (line 115), see also left part of Figure 6.1. An intersection γ_{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 ω_i and ω_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 ω_j is a leaf element on level $l+1$. The intersection iterator on ω_j delivers two intersections, one with ω_i which is on level l and one with ω_m which is also on level $l+1$. However, the intersection iterator started on ω_i will deliver an intersection with ω_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 γ_{ji} visited by the intersection iterator started on ω_j , because only there the two concentration values C_j and C_i are both accessible. Note also that the outside element delivered by an intersection iterator need not be a leaf element (such as ω_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 to the situation of ω_j referring to ω_i in the right part of Figure 6.1) or when the levels are equal and the index of the outside element is larger.

¹http://www.dune-project.org/doc/doxygen/dune-grid-html/group_GIIntersectionIterator.html

The latter condition with the indices just ensures that the flux is only computed once.

The Δt^n calculation is done in line 125 where the minimum over all cells is taken. Then, line 130 multiplies the optimum Δt^n with a safety factor to avoid any instability due to round-off errors.

Finally, line 134 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 #ifndef __DUNE_GRID_HOWTO_VTKOUT_HH__
2 #define __DUNE_GRID_HOWTO_VTKOUT_HH__
3
4 #include <dune/grid/io/file/vtk/vtkwriter.hh>
5 #include <stdio.h>
6
7 template<class G, class V>
8 void vtkout (const G& grid, const V& c, const char* name, int k, double time=0.0, int rank=0)
9 {
10     Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafView());
11     char fname[128];
12     char sername[128];
13     sprintf(fname, "%s-%05d", name, k);
14     sprintf(sername, "%s.series", name);
15     vtkwriter.addCellData(c, "celldata");
16     vtkwriter.write(fname, Dune::VTKOptions::ascii);
17
18     if ( rank == 0 )
19     {
20         std::ofstream serstream(sername, (k==0 ? std::ios_base::out : std::ios_base::app));
21         serstream << k << " " << fname << ".vtu" << time << std::endl;
22         serstream.close();
23     }
24 }
25
26 #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 #include "config.h" // know what grids are present
2 #include <iostream> // for input/output to shell
3 #include <fstream> // for input/output to files
4 #include <vector> // STL vector class
5 #include <dune/grid/common/mcmgmapper.hh> // mapper class
6 #include <dune/common/mpihelper.hh> // include mpi helper class
7
8 #include "vtkout.hh"
9 #include "transportproblem2.hh"
10 #include "initialize.hh"
11 #include "evolve.hh"
12

```

6 Attaching user data to a grid

```
13 //=====
14 // the time loop function working for all types of grids
15 //=====
16
17 template<class G>
18 void timeloop (const G& grid, double tend)
19 {
20     // make a mapper for codim 0 entities in the leaf grid
21     Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,Dune::MCMGElementLayout>
22     mapper(grid);
23
24     // allocate a vector for the concentration
25     std::vector<double> c(mapper.size());
26
27     // initialize concentration with initial values
28     initialize(grid,mapper,c);
29     vtkout(grid,c,"concentration",0,0.0);
30
31     // now do the time steps
32     double t=0,dt;
33     int k=0;
34     const double saveInterval = 0.1;
35     double saveStep = 0.1;
36     int counter = 1;
37
38     while (t<tend)
39     {
40         // augment time step counter
41         ++k;
42
43         // apply finite volume scheme
44         evolve(grid,mapper,c,t,dt);
45
46         // augment time
47         t += dt;
48
49         // check if data should be written
50         if (t >= saveStep)
51         {
52             // write data
53             vtkout(grid,c,"concentration",counter,t);
54
55             // increase counter and saveStep for next interval
56             saveStep += saveInterval;
57             ++counter;
58         }
59
60         // print info about time, timestep size and counter
61         std::cout << "s=" << grid.size(0)
62             << "k=" << k << "t=" << t << "dt=" << dt << std::endl;
63     }
64
65     // output results
66     vtkout(grid,c,"concentration",counter,tend);
67 }
68
69 //=====
70 // The main function creates objects and does the time loop
71 //=====
72
73 int main (int argc , char ** argv)
74 {
75     // initialize MPI, finalize is done automatically on exit
```

6 Attaching user data to a grid

```

76 Dune::MPIHelper::instance(argc,argv);
77
78 // start try/catch block to get error messages from dune
79 try {
80     using namespace Dune;
81
82     // the GridSelector :: GridType is defined in gridtype.hh and is
83     // set during compilation
84     typedef GridSelector :: GridType Grid;
85
86     // use unitcube from dgf grids
87     std::stringstream dgfFileName;
88     dgfFileName << "grids/unitcube" << Grid :: dimension << ".dgf";
89
90     // create grid pointer
91     GridPtr<Grid> gridPtr( dgfFileName.str() );
92
93     // grid reference
94     Grid& grid = *gridPtr;
95
96     // half grid width 4 times
97     int level = 4 * DGFGGridInfo<Grid>::refineStepsForHalf();
98
99     // refine grid until upper limit of level
100    grid.globalRefine(level);
101
102    // do time loop until end time 0.5
103    timeloop(grid, 0.5);
104 }
105 catch (std::exception & e) {
106     std::cout << "STL_ERROR:" << e.what() << std::endl;
107     return 1;
108 }
109 catch (Dune::Exception & e) {
110     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
111     return 1;
112 }
113 catch (...) {
114     std::cout << "Unknown_ERROR" << std::endl;
115     return 1;
116 }
117
118 // done
119 return 0;
120 }

```

The function `timeloop` constructs a mapper and allocates the concentration vector with one entry per element in the leaf grid. In line 28 this vector is initialized with the initial concentration and the loop in line 38-63 evolves the concentration in time. Finally, the simulation result is written to a file in line 66.

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 Ω 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 Δ_i, Δ_j

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

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

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.

As a basis ϕ_1, \dots, ϕ_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 ϕ_1, \dots, ϕ_N with coefficients u_1, \dots, u_N . We assemble the stiffness Matrix A and the vector b :

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

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 ξ_k be the quadrature points of the chosen rule and ω_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.

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:

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

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

This is possible as - using the nodal basis - the coefficients match the value of the numerical solution at the correspondant 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  #ifndef SHAPEFUNCTIONS_HH
2  #define SHAPEFUNCTIONS_HH
3
4  #include <dune/common/fvector.hh>
5
6  // LinearShapeFunction:
7  // represents a shape function and provides methods to evaluate the function
8  // and its gradient
9  template<class ctype, class rtype, int dim>
10 class LinearShapeFunction
11 {
12 public:
13     enum { dimension = dim };
14
15     LinearShapeFunction() : coeff0(0.0), coeff1(0.0) {}
16
17     LinearShapeFunction(rtype coeff0_, const Dune::FieldVector<rtype,dim>& coeff1_)
18         : coeff0(coeff0_), coeff1(coeff1_) {}
19
20     void setCoeff(rtype coeff0_, const Dune::FieldVector<rtype,dim>& coeff1_)
21     {
22         coeff0 = coeff0_;
23         coeff1 = coeff1_;
24     }
25
26     rtype evaluateFunction(const Dune::FieldVector<ctype,dim>& local) const
27     {
28         ctype result = coeff0;
29         for (int i = 0; i < dim; ++i)
30             result += coeff1[i] * local[i];
31         return result;
32     }
33
34     Dune::FieldVector<rtype,dim>
35     evaluateGradient(const Dune::FieldVector<ctype,dim>& local) const
36     {
37         return coeff1;
38     }
39
40 private:
41     rtype coeff0;
42     Dune::FieldVector<rtype,dim> coeff1;
43 };
44
45 // P1ShapeFunctionSet
46 // initializes one and only one set of LinearShapeFunction
47 template<class ctype, class rtype, int dim>
48 class P1ShapeFunctionSet
49 {
50 public:
51     enum { n = dim + 1 };
52
53     typedef LinearShapeFunction<ctype, rtype, dim> ShapeFunction;

```

6 Attaching user data to a grid

```

54     typedef rtype resultttype;
55
56     // get the only instance of this class
57     static const P1ShapeFunctionSet& instance()
58     {
59         static const P1ShapeFunctionSet sfs;
60         return sfs;
61     }
62
63     const ShapeFunction& operator[](int i) const
64     {
65         if (!i)
66             return f0;
67         else
68             return f1[i - 1];
69     }
70
71 private:
72     // private constructor prevents additional instances
73     P1ShapeFunctionSet()
74     {
75         Dune::FieldVector<rtype,dim> e(-1.0);
76         f0.setCoeff(1.0, e);
77         for (int i = 0; i < dim; ++i)
78         {
79             Dune::FieldVector<rtype,dim> e(0.0);
80             e[i] = 1.0;
81             f1[i].setCoeff(0.0, e);
82         }
83     }
84
85     ShapeFunction f0;
86     ShapeFunction f1[dim];
87 };
88
89 #endif

```

And now the actual FEM code:

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

```

1 #include "config.h"
2 #include <iostream>
3 #include <vector>
4 #include <set>
5 #include <dune/common/fvector.hh>
6 #include <dune/common/fmatrix.hh>
7 #include <dune/grid/common/quadraturerules.hh>
8 #include <dune/grid/io/file/vtk/vtkwriter.hh>
9 #include <dune/istl/bvector.hh>
10 #include <dune/istl/bcrsmatrix.hh>
11 #include <dune/istl/ilu.hh>
12 #include <dune/istl/operators.hh>
13 #include <dune/istl/solvers.hh>
14 #include <dune/istl/preconditioners.hh>
15 #include <dune/istl/io.hh>
16 #include <dune/grid/albertagrid.hh>
17 #include "shapefunctions.hh"
18
19 // P1Elements:
20 // a P1 finite element discretization for elliptic problems Dirichlet
21 // boundary conditions on simplicial conforming grids
22 template<class GV, class F>

```


6 Attaching user data to a grid

```

23 class P1Elements
24 {
25 public:
26     static const int dim = GV::dimension;
27
28     typedef typename GV::ctype ctype;
29     typedef Dune::BCSRMatrix<Dune::FieldMatrix<ctype,1,1> > Matrix;
30     typedef Dune::BlockVector<Dune::FieldVector<ctype,1> > ScalarField;
31
32 private:
33     typedef typename GV::template Codim<0>::Iterator LeafIterator;
34     typedef typename GV::IntersectionIterator IntersectionIterator;
35     typedef typename GV::IndexSet LeafIndexSet;
36
37     const GV& gv;
38     const F& f;
39
40 public:
41     Matrix A;
42     ScalarField b;
43     ScalarField u;
44     std::vector< std::set<int> > adjacencyPattern;
45
46     P1Elements(const GV& gv_, const F& f_) : gv(gv_), f(f_) {}
47
48     // store adjacency information in a vector of sets
49     void determineAdjacencyPattern();
50
51     // assemble stiffness matrix A and right side b
52     void assemble();
53
54     // finally solve Au = b for u
55     void solve();
56 };
57
58 template<class GV, class F>
59 void P1Elements<GV, F>::determineAdjacencyPattern()
60 {
61     const int N = gv.size(dim);
62     adjacencyPattern.resize(N);
63
64     const LeafIndexSet& set = gv.indexSet();
65     const LeafIterator itend = gv.template end<0>();
66
67     for (LeafIterator it = gv.template begin<0>(); it != itend; ++it)
68     {
69         Dune::GeometryType gt = it->type();
70         const Dune::template GenericReferenceElement<ctype,dim> &ref =
71             Dune::GenericReferenceElements<ctype,dim>::general(gt);
72
73         // traverse all codim-1-entities of the current element and store all
74         // pairs of vertices in adjacencyPattern
75         const IntersectionIterator isend = gv.iend(*it);
76         for (IntersectionIterator is = gv.ibegin(*it) ; is != isend ; ++is)
77         {
78             int vertexsize = ref.size(is->indexInInside(),1,dim);
79             for (int i=0; i < vertexsize; i++)
80             {
81                 int indexi = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,i,dim),dim);
82                 for (int j=0; j < vertexsize; j++)
83                 {
84                     int indexj = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,j,dim),dim);
85                     adjacencyPattern[indexi].insert(indexj);

```

6 Attaching user data to a grid

```

86         }
87     }
88 }
89 }
90 }
91
92 template<class GV, class F>
93 void P1Elements<GV, F>::assemble()
94 {
95     const int N = gv.size(dim);
96
97     const LeafIndexSet& set = gv.indexSet();
98     const LeafIterator itend = gv.template end<0>();
99
100    // set sizes of A and b
101    A.setSize(N, N, N + 2*gv.size(dim-1));
102    A.setBuildMode(Matrix::random);
103    b.resize(N, false);
104
105    for (int i = 0; i < N; i++)
106        A.setrowsize(i, adjacencyPattern[i].size());
107    A.endrowsizes();
108
109    // set sparsity pattern of A with the information gained in determineAdjacencyPattern
110    for (int i = 0; i < N; i++)
111    {
112        std::template set<int>::iterator setend = adjacencyPattern[i].end();
113        for (std::template set<int>::iterator setit = adjacencyPattern[i].begin();
114             setit != setend; ++setit)
115            A.addindex(i,*setit);
116    }
117
118    A.endindices();
119
120    // initialize A and b
121    A = 0.0;
122    b = 0.0;
123
124    // get a set of P1 shape functions
125    P1ShapeFunctionSet<ctype,ctype,dim> basis = P1ShapeFunctionSet<ctype,ctype,dim>::instance();
126
127    for (LeafIterator it = gv.template begin<0>(); it != itend; ++it)
128    {
129        // determine geometry type of the current element and get the matching reference element
130        Dune::GeometryType gt = it->type();
131        const Dune::template GenericReferenceElement<ctype,dim> &ref =
132            Dune::GenericReferenceElements<ctype,dim>::general(gt);
133        int vertexsize = ref.size(dim);
134
135        // get a quadrature rule of order one for the given geometry type
136        const Dune::QuadratureRule<ctype,dim>& rule = Dune::QuadratureRules<ctype,dim>::rule(gt,1);
137        for (typename Dune::QuadratureRule<ctype,dim>::const_iterator r = rule.begin();
138             r != rule.end() ; ++r)
139        {
140            // compute the jacobian inverse transposed to transform the gradients
141            Dune::FieldMatrix<ctype,dim,dim> jacInvTra =
142                it->geometry().jacobianInverseTransposed(r->position());
143
144            // get the weight at the current quadrature point
145            ctype weight = r->weight();
146
147            // compute Jacobian determinant for the transformation formula
148            ctype detjac = it->geometry().integrationElement(r->position());

```

6 Attaching user data to a grid

```

149     for (int i = 0; i < vertexsize; i++)
150     {
151         // compute transformed gradients
152         Dune::FieldVector<ctype,dim> grad1;
153         jacInvTra.mv(basis[i].evaluateGradient(r->position()),grad1);
154         for (int j = 0; j < vertexsize; j++)
155         {
156             Dune::FieldVector<ctype,dim> grad2;
157             jacInvTra.mv(basis[j].evaluateGradient(r->position()),grad2);
158
159             // gain global inidices of vertices i and j and update associated matrix entry
160             A[set.subIndex(*it,i,dim)][set.subIndex(*it,j,dim)]
161                 += (grad1*grad2) * weight * detjac;
162         }
163     }
164 }
165
166 // get a quadrature rule of order two for the given geometry type
167 const Dune::QuadratureRule<ctype,dim>& rule2 = Dune::QuadratureRules<ctype,dim>::rule(gt,2);
168 for (typename Dune::QuadratureRule<ctype,dim>::const_iterator r = rule2.begin();
169      r != rule2.end() ; ++r)
170 {
171     ctype weight = r->weight();
172     ctype detjac = it->geometry().integrationElement(r->position());
173     for (int i = 0 ; i<vertexsize; i++)
174     {
175         // evaluate the integrand of the right side
176         ctype fval = basis[i].evaluateFunction(it->geometry().global(r->position()))
177             * f(it->geometry().global(r->position())) ;
178         b[set.subIndex(*it,i,dim)] += fval * weight * detjac;
179     }
180 }
181 }
182
183 // Dirichlet boundary conditions:
184 // replace lines in A related to Dirichlet vertices by trivial lines
185 for ( LeafIterator it = gv.template begin<0>() ; it != itend ; ++it)
186 {
187     const IntersectionIterator isend = gv.iend(*it);
188     for (IntersectionIterator is = gv.ibegin(*it) ; is != isend ; ++is)
189     {
190         // determine geometry type of the current element and get the matching reference element
191         Dune::GeometryType gt = it->type();
192         const Dune::template GenericReferenceElement<ctype,dim> &ref =
193             Dune::GenericReferenceElements<ctype,dim>::general(gt);
194
195         // check whether current intersection is on the boundary
196         if ( is->boundary() )
197         {
198             // traverse all vertices the intersection consists of
199             for (int i=0; i < ref.size(is->indexInInside(),1,dim); i++)
200             {
201                 // and replace the associated line of A and b with a trivial one
202                 int indexi = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,i,dim),dim);
203
204                 A[indexi] = 0.0;
205                 A[indexi][indexi] = 1.0;
206                 b[indexi] = 0.0;
207             }
208         }
209     }
210 }
211 }

```

6 Attaching user data to a grid

```

212
213 template<class GV, class E>
214 void P1Elements<GV, E>::solve()
215 {
216     // make linear operator from A
217     Dune::MatrixAdapter<Matrix,ScalarField,ScalarField> op(A);
218
219     // initialize preconditioner
220     Dune::SeqILUn<Matrix,ScalarField,ScalarField> ilu1(A, 1, 0.92);
221
222     // the inverse operator
223     Dune::BiCGSTABSolver<ScalarField> bcgs(op, ilu1, 1e-15, 5000, 0);
224     Dune::InverseOperatorResult r;
225
226     // initialue u to some arbitrary value to avoid u being the exact
227     // solution
228     u.resize(b.N(), false);
229     u = 2.0;
230
231     // finally solve the system
232     bcgs.apply(u, b, r);
233 }
234
235
236 // an example right hand side function
237 template<class ctype, int dim>
238 class Bump {
239 public:
240     ctype operator() (Dune::FieldVector<ctype,dim> x) const
241     {
242         ctype result = 0;
243         for (int i=0 ; i < dim ; i++)
244             result += 2.0 * x[i]* (1-x[i]);
245         return result;
246     }
247 };
248
249 int main(int argc, char** argv)
250 {
251     #if HAVE_ALBERTA
252     #if ALBERTA_DIM==2
253         static const int dim = 2;
254         const char* gridfile = "grids/2dgrid.al";
255
256         typedef Dune::AlbertaGrid<dim,dim> GridType;
257         typedef GridType::LeafGridView GV;
258
259         typedef GridType::ctype ctype;
260         typedef Bump<ctype,dim> Func;
261
262         GridType grid(gridfile);
263         const GV& gv=grid.leafView();
264
265         Func f;
266         P1Elements<GV,Func> p1(gv, f);
267
268         grid.globalRefine(16);
269
270         std::cout << "-----" << "\n";
271         std::cout << "number of unknowns: " << grid.size(dim) << "\n";
272
273         std::cout << "determine adjacency pattern..." << "\n";
274         p1.determineAdjacencyPattern();

```

6 Attaching user data to a grid

```
275
276     std::cout << "assembling..." << "\n";
277     p1.assemble();
278
279     std::cout << "solving..." << "\n";
280     p1.solve();
281
282     std::cout << "visualizing..." << "\n";
283     Dune::VTKWriter<GridType::LeafGridView> vtkwriter(grid.leafView());
284     vtkwriter.addVertexData(p1.u, "u");
285     vtkwriter.write("fem2d", Dune::VTKOptions::binaryappended);
286
287 #endif
288 #endif
289 }
```

The function `determineAdjacencyPattern()` in lines 58 to 90 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 110 on. Only after finishing this in line 118 we can start to fill the matrix with values.

From line 127 to 181 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 160 and 178 - 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 185 to 210. We have to traverse the whole grid once again and check for each intersection of elements whether it is on the boundary. In line 204 we overwrite the line correspondent to a node on the boundary as shown in figure 6.2.

When you visualize your results, you should get something like figure 6.4.1 or 6.4.1!

Exercise 6.1 Try a 3-dimensional grid! Just change the dimension in line 253 and the name of the gridfile in line 254 to `3dgrid.a1`. 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!

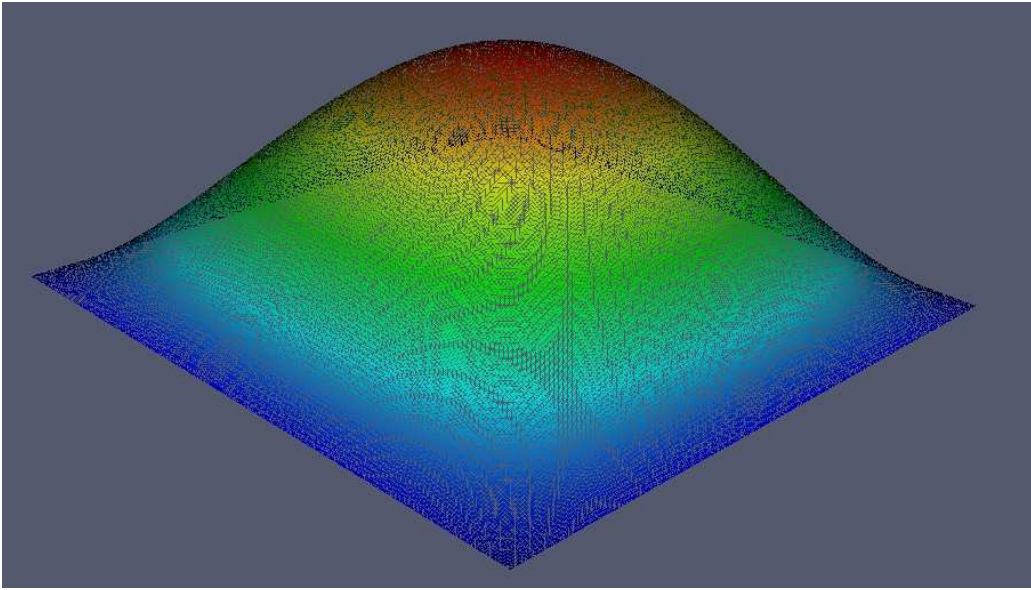


Figure 6.3: Solution in 2D

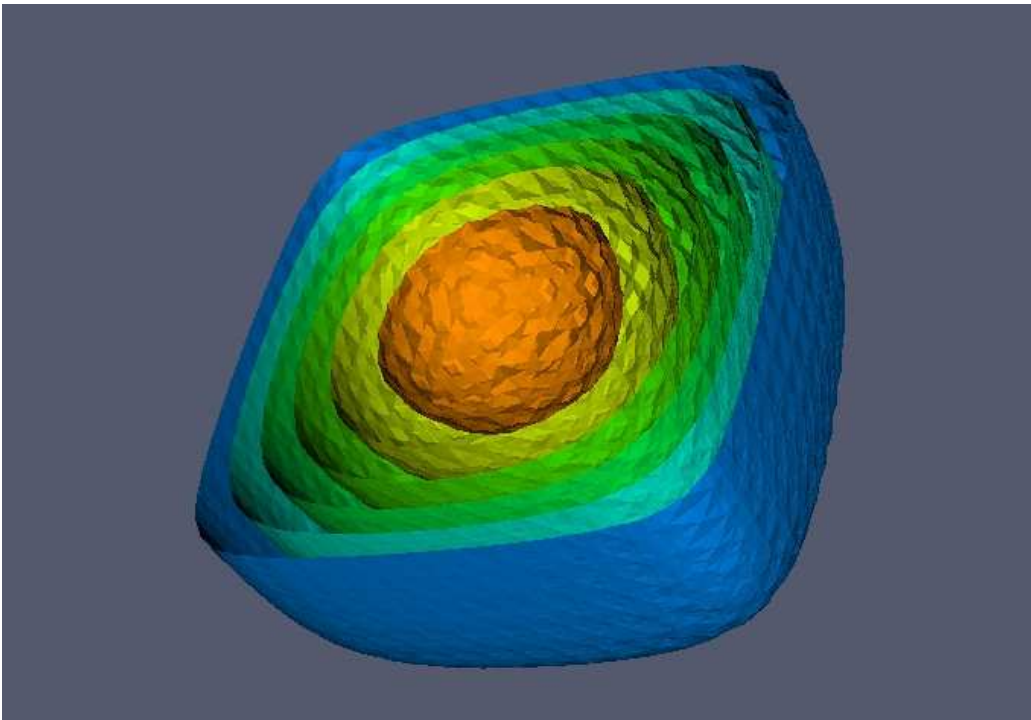


Figure 6.4: Solution in 3D

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 ω .

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 would know 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 ω and its father element ω^- , i. e. the refinement of ω^- resulted in ω . Moreover, assume that ω^+ is a (virtual) element that would result from a refinement of ω . 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 ω^+ , 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 κ 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 κ as defined above.
 - Refine all elements ω 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 // $Id$
2
3 #include "config.h"
4 #include <iostream>
5 #include <iomanip>
6 #include <dune/grid/io/file/vtk/vtkwriter.hh> // VTK output routines
7 #include <dune/common/mpihelper.hh> // include mpi helper class
8
9 #include "unitcube.hh"
10 #include "functors.hh"
11 #include "integrateentity.hh"
12
13
14 //! adaptive refinement test
15 template<class Grid, class Functor>
16 void adaptiveintegration (Grid& grid, const Functor& f)
17 {
18   // get grid view type for leaf grid part
19   typedef typename Grid::LeafGridView GridView;
20   // get iterator type
21   typedef typename GridView::template Codim<0>::Iterator ElementLeafIterator;
22
23   // get grid view on leaf part
24   GridView gridView = grid.leafView();
25
26   // algorithm parameters
27   const double tol=1E-8;
28   const int loworder=1;
29   const int highorder=3;
30
31   // loop over grid sequence
32   double oldvalue=1E100;
33   for (int k=0; k<100; k++)
34   {
35     // compute integral on current mesh
36     double value=0;
37     for (ElementLeafIterator it = gridView.template begin<0>();
38          it!=gridView.template end<0>(); ++it)
39       value += integrateentity(it,f,highorder);

```


7 Adaptivity

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

7 Adaptivity

```

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

```

The work is done in the function `adaptiveintegration`. Lines 36-39 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 54. The extrapolation strategy relies on the fact that every element has a father. To ensure this, the grid is at least once refined globally in the first step (line 61). Now the refinement threshold κ can be computed in lines 66-88. Finally the last loop in lines 91-98 marks elements for refinement

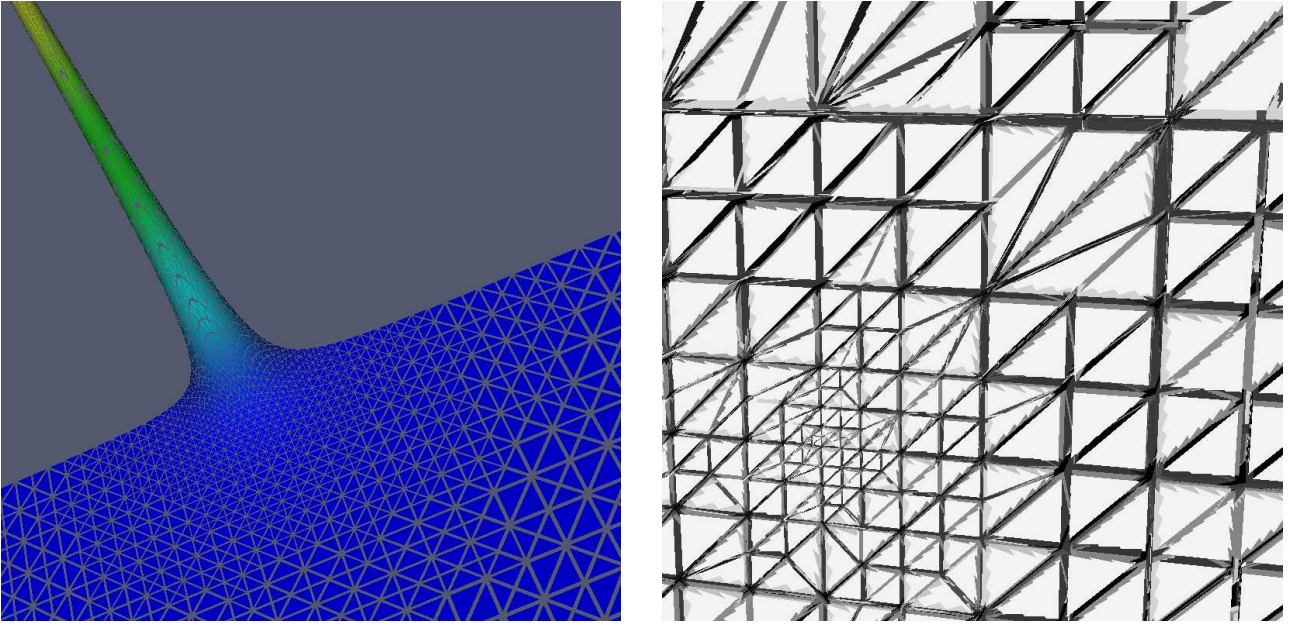


Figure 7.1: Two and three-dimensional grids generated by the adaptive integration algorithm applied to the needle pulse. Left grid is generated using Alberta, right grid is generated using UG.

and lines 101-103 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 7.1.2 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:

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

- As the local indicator in cell ω_i we define

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

7 Adaptivity

Here γ_{ij} denotes intersections with other elements in the leaf grid.

- If for ω_i we have $\eta_i > \overline{\text{tol}} \cdot (\overline{C} - \underline{C})$ and ω_i has not been refined more than \overline{M} times then mark ω_i and all its neighbors for refinement.
- Mark all elements ω_i for coarsening where $\eta_i < \underline{\text{tol}} \cdot (\overline{C} - \underline{C})$ and ω_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 refinetol = 0.05;
25     const double coarsentol = 0.001;
26
27     // type used for coordinates in the grid
28     typedef typename G::ctype ct;
29
30     // grid view types

```

7 Adaptivity

```

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.leafView();
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>();
89      it!=leafView.template end<0>(); ++it)
90 {
91     if ( indicator[mapper.map(*it)]>refinetol*globaldelta
92         && (it.level()<lmax || !it->isRegular()))
93     {

```

7 Adaptivity

```

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.levelView(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
152 // adapt mesh and mapper
153 bool rv=grid.adapt();
154 mapper.update();
155 restrictionmap.reserve();
156 c.resize(mapper.size());

```

7 Adaptivity

```

157
158 // interpolate new cells, restrict coarsened cells
159 for (int level=0; level<=grid.maxLevel(); level++)
160 {
161     LevelGridView levelView = grid.levelView(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 η_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 #include "config.h"           // know what grids are present
2 #include <iostream>           // for input/output to shell
3 #include <fstream>            // for input/output to files
4 #include <vector>             // STL vector class
5
6 #include <dune/grid/common/mcmgmapper.hh> // mapper class
7 #include <dune/common/mpihelper.hh> // include mpi helper class
8
9 #include "vtkout.hh"
10 #include "transportproblem2.hh"
11 #include "initialize.hh"
12 #include "evolve.hh"
13 #include "finitevolumeadapt.hh"
14
15 //=====
16 // the time loop function working for all types of grids
17 //=====
18
19 template<class G>
20 void timeloop (G& grid, double tend, int lmin, int lmax)
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     for (int i=grid.maxLevel(); i<lmax; i++)
32     {
33         if (grid.maxLevel()>=lmax) break;
34         finitevolumeadapt(grid,mapper,c,lmin,lmax,0);
35         initialize(grid,mapper,c);
36     }
37
38     // write initial data
39     vtkout(grid,c,"concentration",0,0);
40
41     // variables for time, timestep etc.
42     double dt, t=0;
43     double saveStep = 0.1;
44     const double saveInterval = 0.1;
45     int counter = 0;
46     int k = 0;
47
48     std::cout << "s=" << grid.size(0) << " \nk=" << k << " \nt=" << t << std::endl;
49     while (t<tend)
50     {
51         // augment time step counter
52         ++k;
53
54         // apply finite volume scheme
55         evolve(grid,mapper,c,t,dt);
56
57         // augment time

```


7 Adaptivity

```

58     t += dt;
59
60     // check if data should be written
61     if (t >= saveStep)
62     {
63         // write data
64         vtkout(grid,c,"concentration",counter,t);
65
66         // increase counter and saveStep for next interval
67         saveStep += saveInterval;
68         ++counter;
69     }
70
71     // print info about time, timestep size and counter
72     std::cout << "s=" << grid.size(0)
73               << "k=" << k << "t=" << t << "dt=" << dt << std::endl;
74
75     // for unstructured grids call adaptation algorithm
76     finitevolumeadapt(grid,mapper,c,lmin,lmax,k);
77 }
78
79 // write last time step
80 vtkout(grid,c,"concentration",counter,tend);
81
82 // write
83 }
84
85 //=====
86 // The main function creates objects and does the time loop
87 //=====
88
89 int main (int argc , char ** argv)
90 {
91     // initialize MPI, finalize is done automatically on exit
92     Dune::MPIHelper::instance(argc,argv);
93
94     // start try/catch block to get error messages from dune
95     try {
96         using namespace Dune;
97
98         // the GridSelector :: GridType is defined in gridtype.hh and is
99         // set during compilation
100         typedef GridSelector :: GridType Grid;
101
102         // use unitcube from grids
103         std::stringstream dgfFileName;
104         dgfFileName << "grids/unitcube" << Grid :: dimension << ".dgf";
105
106         // create grid pointer
107         GridPtr<Grid> gridPtr( dgfFileName.str() );
108
109         // grid reference
110         Grid& grid = *gridPtr;
111
112         // minimal allowed level during refinement
113         int minLevel = 2 * DGFGGridInfo<Grid>::refineStepsForHalf();
114
115         // refine grid until upper limit of level
116         grid.globalRefine(minLevel);
117
118         // maximal allowed level during refinement
119         int maxLevel = minLevel + 3 * DGFGGridInfo<Grid>::refineStepsForHalf();
120

```

7 Adaptivity

```
121 // do time loop until end time 0.5
122 timeloop(grid, 0.5, minLevel, maxLevel);
123 }
124 catch (std::exception & e) {
125     std::cout << "STL_ERROR:" << e.what() << std::endl;
126     return 1;
127 }
128 catch (Dune::Exception & e) {
129     std::cout << "DUNE_ERROR:" << e.what() << std::endl;
130     return 1;
131 }
132 catch (...) {
133     std::cout << "Unknown_ERROR" << std::endl;
134     return 1;
135 }
136
137 // done
138 return 0;
139 }
```

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 34 and after each time step in line 76 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 Ω_i , $0 \leq i < P$, form a nonoverlapping decomposition of the computational domain Ω . 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 6×4 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

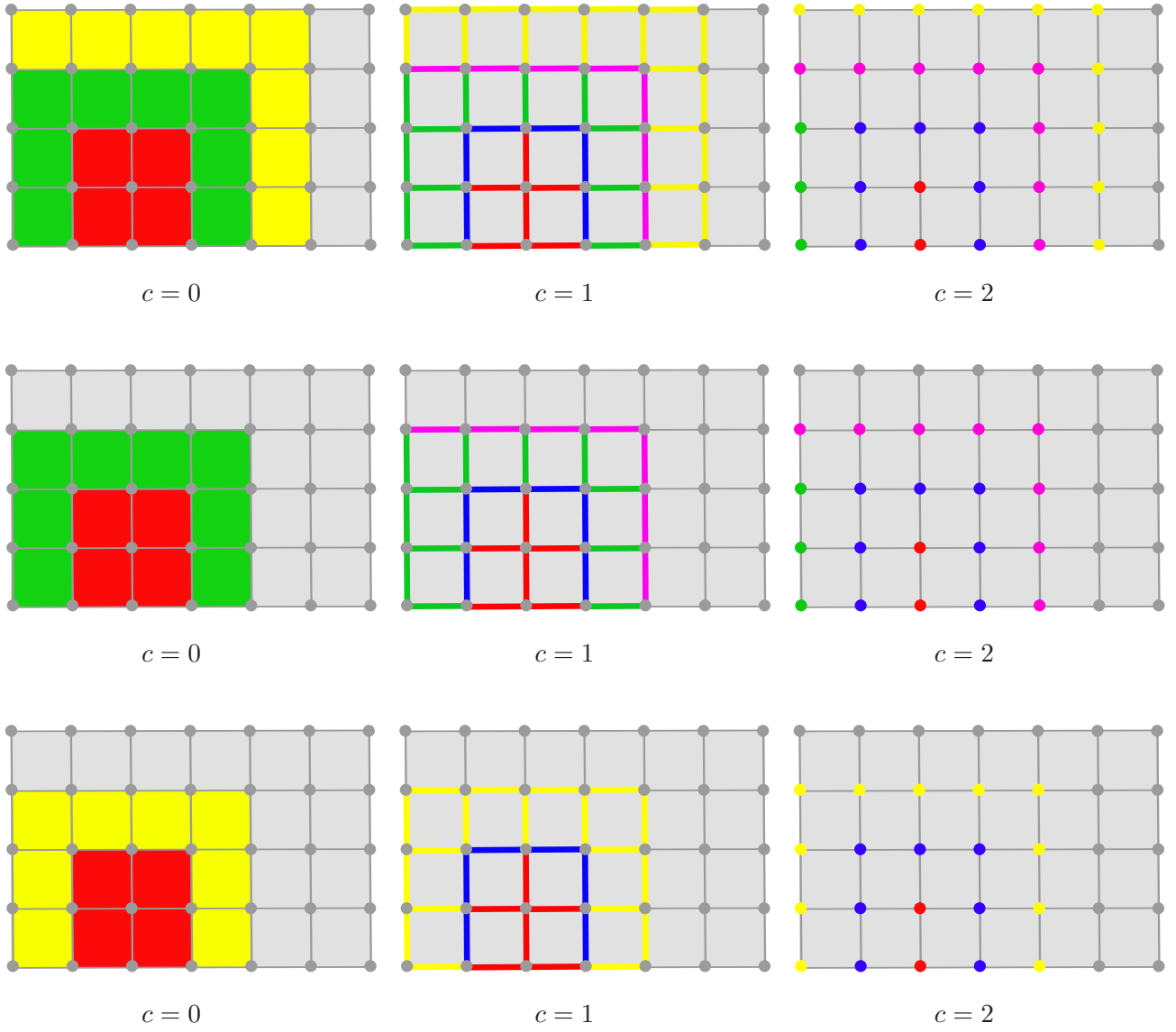


Figure 8.1: Color coded illustration of different data decompositions: interior (red), border (blue), overlap (green), front (magenta) and ghost (yellow), gray encodes entities not stored by the process. First row shows case with interior, overlap and ghost entities, second row shows a case with interior 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 #ifndef __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__
2 #define __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__
3
4 #include <dune/grid/common/datahandleif.hh>
5
6 // A DataHandle class to exchange entries of a vector
7 template<class M, class V> // mapper type and vector type
8 class VectorExchange
9 : public Dune::CommDataHandleIF<VectorExchange<M,V>,
10                               typename V::value_type>
11 {
12 public:
13     //! export type of data for message buffer
14     typedef typename V::value_type DataType;
15
16     //! returns true if data for this codim should be communicated
17     bool contains (int dim, int codim) const
18     {
19         return (codim==0);
20     }
21
22     //! returns true if size per entity of given dim and codim is a constant

```

```

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

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

```

1 #ifndef __DUNE_GRID_HOWTO_PAREVOLVE_HH__
2 #define __DUNE_GRID_HOWTO_PAREVOLVE_HH__
3
4 #include "parfvdatahandle.hh"
5
6 #include <dune/grid/common/gridenums.hh>
7 #include <dune/common/fvector.hh>
8
9 template<class G, class M, class V>
10 void parevolve (const G& grid, const M& mapper, V& c, double t, double& dt)
11 {
12     // check data partitioning
13     assert(grid.overlapSize(0)>0 || (grid.ghostSize(0)>0));
14
15     // first we extract the dimensions of the grid
16     const int dim = G::dimension;
17     const int dimworld = G::dimensionworld;
18
19     // type used for coordinates in the grid
20     typedef typename G::ctype ct;
21
22     // type for grid view on leaf part
23     typedef typename G::LeafGridView GridView;
24
25     // iterator type
26     typedef typename GridView::template Codim<0>::
27         template Partition<Dune::All_Partition>::Iterator LeafIterator;
28
29     // leaf entity geometry
30     typedef typename LeafIterator::Entity::Geometry LeafGeometry;
31
32     // intersection iterator type
33     typedef typename GridView::IntersectionIterator IntersectionIterator;
34
35     // type of intersection
36     typedef typename IntersectionIterator::Intersection Intersection;
37
38     // intersection geometry
39     typedef typename Intersection::Geometry IntersectionGeometry;
40
41     // entity pointer type
42     typedef typename G::template Codim<0>::EntityPointer EntityPointer;
43
44     // allocate a temporary vector for the update
45     V update(c.size());
46     for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
47
48     // initialize dt very large
49     dt = 1E100;

```

8 Parallelism

```

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

```


8 Parallelism

```

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

```

The first difference to the sequential version is in line 13 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 27. 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 56-57.

The next change is in line 140 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 146. 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 151-154 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 #include "config.h" // know what grids are present
2 #include <iostream> // for input/output to shell
3 #include <fstream> // for input/output to files
4 #include <vector> // STL vector class
5 #include <dune/grid/common/mcmgmapper.hh> // mapper class
6 #include <dune/common/mpihelper.hh> // include mpi helper class
7
8
9 // checks for defined gridtype and includes appropriate dgfparser implementation
10 #include "vtkout.hh"
11 #include "unitcube.hh"
12 #include "transportproblem2.hh"
13 #include "initialize.hh"
14 #include "parfvdatahandle.hh"
15 #include "parevolve.hh"
16
17
18 //=====
19 // the time loop function working for all types of grids
20 //=====
21
22 template<class G>
23 void partimeloop (const G& grid, double tend)
24 {
25     // make a mapper for codim 0 entities in the leaf grid
26     Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,Dune::MCMGElementLayout>
27         mapper(grid);
28
29     // allocate a vector for the concentration
30     std::vector<double> c(mapper.size());
31
32     // initialize concentration with initial values
33     initialize(grid,mapper,c);
34     vtkout(grid,c,"pconc",0,0.0,grid.comm().rank());
35
36     // now do the time steps
37     double t=0,dt;
38     int k=0;
39     const double saveInterval = 0.1;
40     double saveStep = 0.1;
41     int counter = 1;
42     while (t<tend)
43     {
44         // augment time step counter
45         k++;

```

8 Parallelism

```

46
47 // apply finite volume scheme
48 parevolve(grid,mapper,c,t,dt);
49
50 // augment time
51 t += dt;
52
53 // check if data should be written
54 if (t >= saveStep)
55 {
56     // write data
57     vtkout(grid,c,"pconc",counter,t,grid.comm().rank());
58
59     //increase counter and saveStep for next interval
60     saveStep += saveInterval;
61     ++counter;
62 }
63
64 // print info about time, timestep size and counter
65 if (grid.comm().rank()==0)
66     std::cout << "k=" << k << " t=" << t << " dt=" << dt << std::endl;
67 }
68 vtkout(grid,c,"pconc",counter,tend,grid.comm().rank());
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         UnitCube<YaspGrid<2>,64> uc;
85         uc.grid().globalRefine(2);
86         partimeloop(uc.grid(),0.5);
87
88         /* To use an alternative grid implementations for parallel computations,
89            uncomment exactly one definition of uc2 and the line below. */
90         // #define LOAD_BALANCING
91
92         // UGGrid supports parallelization in 2 or 3 dimensions
93         #if HAVE_UG
94             typedef UGGrid< 2 > GridType;
95             typedef UnitCube< GridType, 2 > uc2;
96         #endif
97
98         // ALUGRID supports parallelization in 3 dimensions only
99         #if HAVE_ALUGRID
100             typedef ALUCubeGrid< 3, 3 > GridType;
101             typedef ALUSimplexGrid< 3, 3 > GridType;
102             typedef UnitCube< GridType , 1 > uc2;
103         #endif
104
105         #ifndef LOAD_BALANCING
106
107             // refine grid until upper limit of level
108             uc2.grid().globalRefine( 6 );

```

```

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

```

A difference to the sequential program can be found in line 65 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, define the macro **LOAD_BALANCING** and uncomment one of the possible definitions for such a grid in the code. In this case in line 111 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.

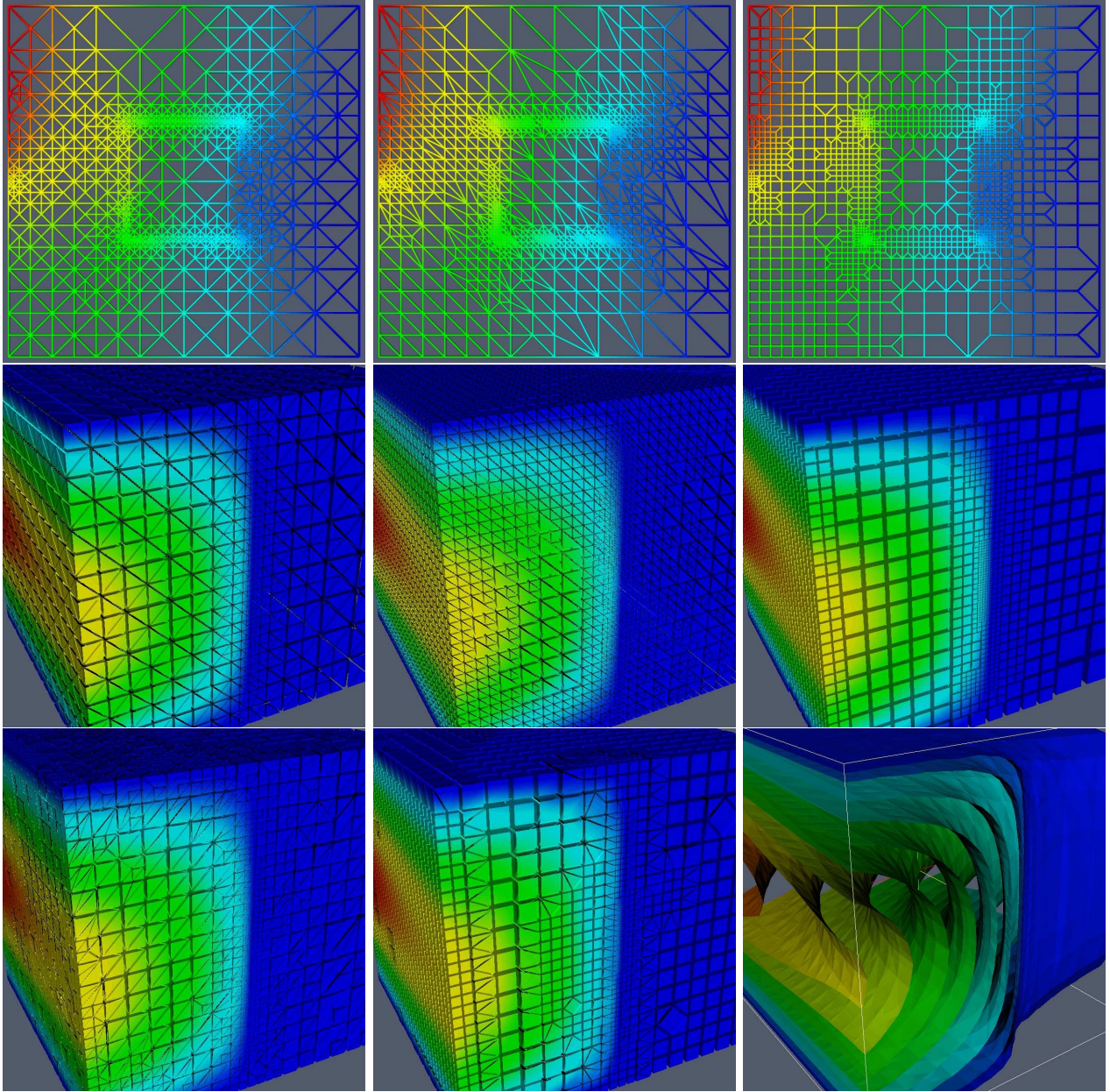


Figure 8.2: Adaptive solution of an elliptic model problem with P_1 conforming finite elements and residual based error estimator. Illustrates that adaptive finite element algorithm can be formulated independent of dimension, element type and refinement scheme. From top to bottom, left to right: Alberta (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.