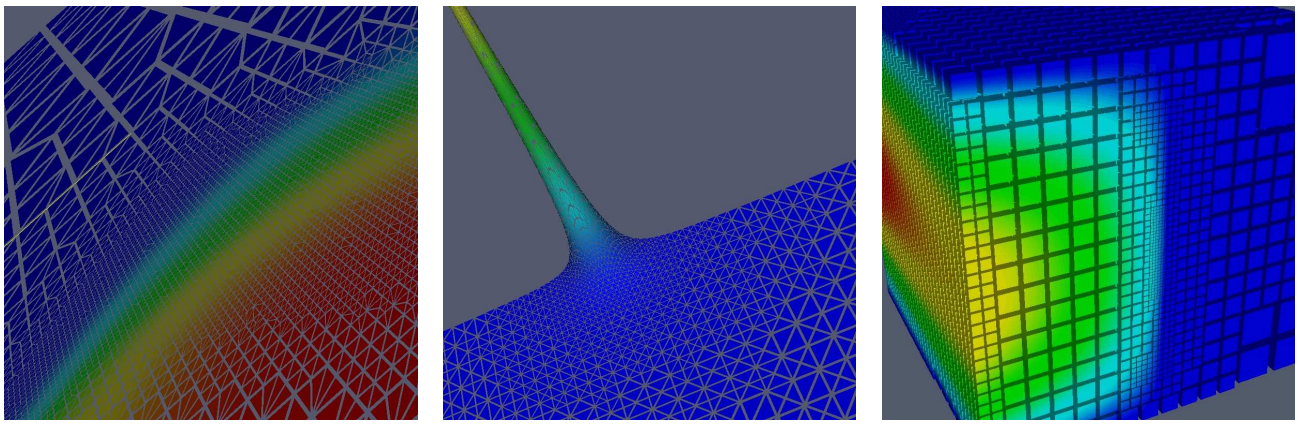
The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO

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

November 26, 2008



*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 Mathematik II,
Freie Universität Berlin, Arnimallee 2-6, 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!

Contents

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.

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/`

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 cinsider 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 License[2], with a special exception for linking and compiling against **DUNE**, the so-called "runtime exception." The license is intended to be similiar 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: gettingstarted.cc 198 2008-01-23 16:12:41Z sander $
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 }
```

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!

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: traversal.cc 188 2007-11-14 11:36:04Z sander $
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   // the grid has an iterator providing the access to
28   // all elements (better codim 0 entities) which are leafs
29   // of the refinement tree.
30   // Note the use of the typename keyword and the traits class
31   typedef typename G::template Codim<0>::LeafIterator ElementLeafIterator;
32
33   // iterate through all entities of codim 0 at the leafs
34   int count = 0;
35   for (ElementLeafIterator it = grid.template leafbegin<0>();
36        it!=grid.template leafend<0>(); ++it)
37     {
38       Dune::GeometryType gt = it->type();
39       std::cout << "visiting␣leaf␣" << gt
40                 << "␣with␣first␣vertex␣at␣" << it->geometry()[0]
41                 << std::endl;
42       count++;
43     }
44
45   std::cout << "there␣are/is␣" << count << "␣leaf␣element(s)" << std::endl;
46
47   // Leafwise traversal of codim dim
48   std::cout << std::endl;
49   std::cout << "***␣Traverse␣codim␣" << dim << "␣leaves" << std::endl;
50
51   // Get the iterator type
52   // Note the use of the typename and template keywords
53   typedef typename G::template Codim<dim>::LeafIterator VertexLeafIterator;
54
55   // iterate through all entities of codim 0 on the given level
```

```
56    count = 0;
57    for (VertexLeafIterator it = grid.template leafbegin<dim>();
58         it!=grid.template leafend<dim>(); ++it)
59      {
60        Dune::GeometryType gt = it->type();
61        std::cout << "visiting " << gt
62                  << " at " << it->geometry()[0]
63                  << std::endl;
64        count++;
65      }
66    std::cout << "there are/is " << count << " leaf vertices(s)"
67              << std::endl;
68
69    // Levelwise traversal of codim 0
70    std::cout << std::endl;
71    std::cout << "*** Traverse codim 0 level-wise" << std::endl;
72
73    // Get the iterator type
74    // Note the use of the typename and template keywords
75    typedef typename G::template Codim<0>::LevelIterator ElementLevelIterator;
76
77    // iterate through all entities of codim 0 on the given level
78    for (int level=0; level<=grid.maxLevel(); level++)
79      {
80        count = 0;
81        for (ElementLevelIterator it = grid.template lbegin<0>(level);
82             it!=grid.template lend<0>(level); ++it)
83          {
84            Dune::GeometryType gt = it->type();
85            std::cout << "visiting " << gt
86                      << " with first vertex at " << it->geometry()[0]
87                      << std::endl;
88            count++;
89          }
90        std::cout << "there are/is " << count << " element(s) on level "
91                  << level << std::endl;
92        std::cout << std::endl;
93      }
94  }
95
96
97  int main(int argc, char **argv)
98  {
99    // initialize MPI, finalize is done automatically on exit
100   Dune::MPIHelper::instance(argc,argv);
101
102   // start try/catch block to get error messages from dune
103   try {
104     // make a grid
105     const int dim=2;
106     typedef Dune::SGrid<dim,dim> GridType;
107     Dune::FieldVector<int,dim> N(1);
108     Dune::FieldVector<GridType::ctype,dim> L(-1.0);
109     Dune::FieldVector<GridType::ctype,dim> H(1.0);
110     GridType grid(N,L,H);
111
112     // refine all elements once using the standard refinement rule
113     grid.globalRefine(1);
114
115     // traverse the grid and print some info
116     traversal(grid);
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 }
```

The `main` function near the end of the listing is pretty similar to previous one except that we use a 2d grid for the unit square that just consists of one cell. In line 113 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 116. This function is given in lines 14-94.

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.

Line 31 extracts the type of an iterator from the grid 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 `LeafIterator` 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 35 now visits every such element. The `leafbegin` and `leafend` on the grid 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 explicitely. Within the loop body in lines 37-43 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

38. 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 an `operator[]`. Line 39 prints the geometry type and the position of the first corner of the element. Then line 42 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 53 for later use. The `for`-loop starting in line 57 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 75-93 how the hierarchic structure of the mesh can be accessed. To that end a `LevelIterator` is used. It provides 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 `lbegin()` and `lend()` on the grid 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)

*** 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 2.2 Define the end iterator for efficiency.

Exercise 2.3 Play with different dimensions, codimension (`SGrid` supports all codimenions) and refinements.

Exercise 2.4 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 $0, \ldots, d-1$). 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

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

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

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 decription 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.

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.

LevelIterator
LevelIterator is an immutable iterator that provides access to an entity. It can be incremented to visit all entities of a given codimension and level of the grid. An EntityPointer is assignable from a LevelIterator.

LeafIterator
LeafIterator is an immutable iterator that provides access to an entity. It can by incremented to visit all entities of a given codimension of the leaf grid. An EntityPointer is assignable from a LeafIterator.

HierarchicIterator
HierarchicIterator is an immutable iterator that provides access to an entity. It can be incremented to visit all entities of codimension 0 that resulted from subdivision of a given entity of codimension 0. An EntityPointer is assignable from a HierarchicIterator.

LevelIntersectionIterator
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 also provides information about the geometric location of the intersection. Furthermore it also provides information about intersections of an entity with the internal or external boundaries. The LevelIntersectionIterator provides intersections between codimension 0 entities having the same level.

LeafIntersectionIterator
This iterator has the same properties as the LevelIntersectionIterator but provides intersections between leaf entities of the grid.

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.

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.

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 as public types from 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 Grid implementations

4.1 Using different grids

The power of **DUNE** is the possibility of writing one algorithm that works on a large variety of grids with different features. In that chapter we show how the different available grid classes are instantiated. As an example we create grids for the unit cube $\Omega = (0,1)^d$ in various dimensions d.

The different grid classes have no common interface for instantiation, they may even have different template parameters. In order make the examples below 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
6  // default implementation for any template parameter
7  template<typename T, int variant>
8  class UnitCube
9  {
10 public:
11   typedef T GridType;
12
13   // constructor throwing exception
14   UnitCube ()
15   {
16     DUNE_THROW(Dune::Exception,"no␣specialization␣for␣this␣grid␣available");
17   }
18
19   T& grid ()
20   {
21     return grid_;
22   }
23
24 private:
25   // the constructed grid object
26   T grid_;
27 };
28
29 // include specializations
30 #include"unitcube_onedgrid.hh"
31 #include"unitcube_sgrid.hh"
32 #include"unitcube_yaspgrid.hh"
33 #include"unitcube_uggrid.hh"
34 #include"unitcube_albertagrid.hh"
35 #include"unitcube_alugrid.hh"
```

```
36
37 #endif
```

Instantiation of that template results in a class that throws an exception when an object is created.

OneDGrid

The following listing creates a `OneDGrid` object. This class has a constructor without arguments that creates a unit interval discretized with a single element. `OneDGrid` allows local mesh refinement in one space dimension.

Listing 6 (File dune-grid-howto/unitcube_onedgrid.hh)

```
1  #ifndef UNITCUBE_ONEDGRID_HH
2  #define UNITCUBE_ONEDGRID_HH
3
4  #include <dune/grid/onedgrid.hh>
5
6  // OneDGrid specialization
7  template<>
8  class UnitCube<Dune::OneDGrid,1>
9  {
10 public:
11   typedef Dune::OneDGrid GridType;
12
13   UnitCube () : grid_(1,0.0,1.0)
14   {}
15
16   Dune::OneDGrid& grid ()
17   {
18     return grid_;
19   }
20
21 private:
22   Dune::OneDGrid grid_;
23 };
24
25 #endif
```

SGrid

The following listing creates a `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 7 (File dune-grid-howto/unitcube_sgrid.hh)

```
1  #ifndef UNITCUBE_SGRID_HH
2  #define UNITCUBE_SGRID_HH
3
4  #include <dune/grid/sgrid.hh>
5
6  // SGrid specialization
7  template<int dim>
8  class UnitCube<Dune::SGrid<dim,dim>,1>
9  {
10 public:
11   typedef Dune::SGrid<dim,dim> GridType;
12
13   Dune::SGrid<dim,dim>& grid ()
14   {
```

```
15      return grid_;
16    }
17
18  private:
19    Dune::SGrid<dim,dim> grid_;
20  };
21
22  #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 8 (File dune-grid-howto/unitcube_yaspgrid.hh)

```
1  #ifndef UNITCUBE_YASPGRID_HH
2  #define UNITCUBE_YASPGRID_HH
3
4  #include <dune/grid/yaspgrid.hh>
5
6  // YaspGrid specialization
7  template<int dim, int size>
8  class UnitCube<Dune::YaspGrid<dim,dim>,size>
9  {
10 public:
11   typedef Dune::YaspGrid<dim,dim> GridType;
12
13   UnitCube () : Len(1.0), s(size), p(false),
14 #if HAVE_MPI
15   grid_(MPI_COMM_WORLD,Len,s,p,1)
16 #else
17   grid_(Len,s,p,1)
18 #endif
19   {  }
20
21   Dune::YaspGrid<dim,dim>& grid ()
22   {
23     return grid_;
24   }
25
26 private:
27   Dune::FieldVector<double,dim> Len;
28   Dune::FieldVector<int,dim> s;
29   Dune::FieldVector<bool,dim> p;
30   Dune::YaspGrid<dim,dim> grid_;
31 };
32
33 #endif
```

UGGrid

The following listing shows how to create `UGGrid` objects. Two and three-dimensional versions are available. The `variant` parameter can take on two values: 1 for quadrilateral/hexahedral grids and 2 for triangular/tetrahedral grids. The initial grids are read in AmiraMesh format.

Listing 9 (File dune-grid-howto/unitcube_uggrid.hh)

```
#ifndef UNITCUBE_UGGRID_HH
#define UNITCUBE_UGGRID_HH

#if HAVE_UG
#include <dune/grid/uggrid.hh>

// UGGrid 3d, variant 1 (hexahedra) specialization
template<>
class UnitCube<Dune::UGGrid<3>,1>
{
public:
  typedef Dune::UGGrid<3> GridType;

  UnitCube () : grid_(800)
  {
      //   Start grid creation
      grid_.createBegin();

      //   Insert vertices
      Dune::FieldVector<double,3> pos;

      pos[0] = 0;  pos[1] = 0;  pos[2] = 0;    grid_.insertVertex(pos);
      pos[0] = 1;  pos[1] = 0;  pos[2] = 0;    grid_.insertVertex(pos);
      pos[0] = 0;  pos[1] = 1;  pos[2] = 0;    grid_.insertVertex(pos);
      pos[0] = 1;  pos[1] = 1;  pos[2] = 0;    grid_.insertVertex(pos);
      pos[0] = 0;  pos[1] = 0;  pos[2] = 1;    grid_.insertVertex(pos);
      pos[0] = 1;  pos[1] = 0;  pos[2] = 1;    grid_.insertVertex(pos);
      pos[0] = 0;  pos[1] = 1;  pos[2] = 1;    grid_.insertVertex(pos);
      pos[0] = 1;  pos[1] = 1;  pos[2] = 1;    grid_.insertVertex(pos);


      // Insert element
      std::vector<unsigned int> cornerIDs(8);
      for (int i=0; i<8; i++)
          cornerIDs[i] = i;

      grid_.insertElement(Dune::GeometryType(Dune::GeometryType::cube,3), cornerIDs);

      //   Finish initialization
      grid_.createEnd();
  }

  Dune::UGGrid<3>& grid ()
  {
    return grid_;
  }

private:
  Dune::UGGrid<3> grid_;
};

// UGGrid 3d, variant 2 (tetrahedra) specialization
template<>
class UnitCube<Dune::UGGrid<3>,2>
{
public:
  typedef Dune::UGGrid<3> GridType;

  UnitCube () : grid_(800)
  {
      //   Start grid creation
      grid_.createBegin();

```

```
64        //   Insert vertices
65        Dune::FieldVector<double,3> pos;
66
67        pos[0] = 0;  pos[1] = 0;  pos[2] = 0;    grid_.insertVertex(pos);
68        pos[0] = 1;  pos[1] = 0;  pos[2] = 0;    grid_.insertVertex(pos);
69        pos[0] = 0;  pos[1] = 1;  pos[2] = 0;    grid_.insertVertex(pos);
70        pos[0] = 1;  pos[1] = 1;  pos[2] = 0;    grid_.insertVertex(pos);
71        pos[0] = 0;  pos[1] = 0;  pos[2] = 1;    grid_.insertVertex(pos);
72        pos[0] = 1;  pos[1] = 0;  pos[2] = 1;    grid_.insertVertex(pos);
73        pos[0] = 0;  pos[1] = 1;  pos[2] = 1;    grid_.insertVertex(pos);
74        pos[0] = 1;  pos[1] = 1;  pos[2] = 1;    grid_.insertVertex(pos);
75
76
77        // Insert element
78        std::vector<unsigned int> cornerIDs(4);
79
80        cornerIDs[0] = 0;  cornerIDs[1] = 1;  cornerIDs[2] = 2;  cornerIDs[3] = 4;
81        grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,3), cornerIDs);
82
83        cornerIDs[0] = 1;  cornerIDs[1] = 3;  cornerIDs[2] = 2;  cornerIDs[3] = 7;
84        grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,3), cornerIDs);
85
86        cornerIDs[0] = 1;  cornerIDs[1] = 7;  cornerIDs[2] = 2;  cornerIDs[3] = 4;
87        grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,3), cornerIDs);
88
89        cornerIDs[0] = 1;  cornerIDs[1] = 7;  cornerIDs[2] = 4;  cornerIDs[3] = 5;
90        grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,3), cornerIDs);
91
92        cornerIDs[0] = 4;  cornerIDs[1] = 7;  cornerIDs[2] = 2;  cornerIDs[3] = 6;
93        grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,3), cornerIDs);
94
95        //   Finish initialization
96        grid_.createEnd();
97    }
98
99    Dune::UGGrid<3>& grid ()
100   {
101     return grid_;
102   }
103
104 private:
105   Dune::UGGrid<3> grid_;
106 };
107
108 // UGGrid 2d, variant 1 (quadrilaterals) specialization
109 template<>
110 class UnitCube<Dune::UGGrid<2>,1>
111 {
112 public:
113   typedef Dune::UGGrid<2> GridType;
114
115   UnitCube () : grid_(800)
116   {
117       //   Start grid creation
118       grid_.createBegin();
119
120       //   Insert vertices
121       Dune::FieldVector<double,2> pos;
122
123       pos[0] = 0;  pos[1] = 0;
124       grid_.insertVertex(pos);
125
126       pos[0] = 1;  pos[1] = 0;
```

```
127       grid_.insertVertex(pos);
128
129       pos[0] = 0;  pos[1] = 1;
130       grid_.insertVertex(pos);
131
132       pos[0] = 1;  pos[1] = 1;
133       grid_.insertVertex(pos);
134
135       // Insert element
136       std::vector<unsigned int> cornerIDs(4);
137       cornerIDs[0] = 0;
138       cornerIDs[1] = 1;
139       cornerIDs[2] = 2;
140       cornerIDs[3] = 3;
141
142       grid_.insertElement(Dune::GeometryType(Dune::GeometryType::cube,2), cornerIDs);
143
144       //   Finish initialization
145       grid_.createEnd();
146   }
147
148   Dune::UGGrid<2>& grid ()
149   {
150     return grid_;
151   }
152
153 private:
154   Dune::UGGrid<2> grid_;
155 };
156
157 // UGGrid 2d, variant 2 (triangles) specialization
158 template<>
159 class UnitCube<Dune::UGGrid<2>,2>
160 {
161 public:
162   typedef Dune::UGGrid<2> GridType;
163
164   UnitCube () : grid_(800)
165   {
166       //   Start grid creation
167       grid_.createBegin();
168
169       //   Insert vertices
170       Dune::FieldVector<double,2> pos;
171
172       pos[0] = 0;  pos[1] = 0;
173       grid_.insertVertex(pos);
174
175       pos[0] = 1;  pos[1] = 0;
176       grid_.insertVertex(pos);
177
178       pos[0] = 0;  pos[1] = 1;
179       grid_.insertVertex(pos);
180
181       pos[0] = 1;  pos[1] = 1;
182       grid_.insertVertex(pos);
183
184       // Insert element
185       std::vector<unsigned int> cornerIDs(3);
186
187       cornerIDs[0] = 0;  cornerIDs[1] = 1;  cornerIDs[2] = 2;
188       grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,2), cornerIDs);
189
```

```
190       cornerIDs[0] = 2;  cornerIDs[1] = 1;  cornerIDs[2] = 3;
191       grid_.insertElement(Dune::GeometryType(Dune::GeometryType::simplex,2), cornerIDs);
192
193       //   Finish initialization
194       grid_.createEnd();
195   }
196
197   Dune::UGGrid<2>& grid ()
198   {
199     return grid_;
200   }
201
202 private:
203   Dune::UGGrid<2> grid_;
204 };
205 #endif
206
207 #endif
```

AlbertaGrid

The following listing contains specializations of the `UnitCube` template for Alberta in two and three dimensions. When using Alberta 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.

Listing 10 (File dune-grid-howto/unitcube_albertagrid.hh)

```
1  #ifndef UNITCUBE_ALBERTAGRID_HH
2  #define UNITCUBE_ALBERTAGRID_HH
3
4  #if HAVE_ALBERTA
5  #include <dune/grid/albertagrid.hh>
6
7  // AlbertaGrid 2d, variant 1 (2 triangles) specialization
8  #if ALBERTA_DIM == 2 && ALBERTA_WORLD_DIM == 2
9  template<>
10 class UnitCube<Dune::AlbertaGrid<2,2>,1>
11 {
12 public:
13   typedef Dune::AlbertaGrid<2,2> GridType;
14
15   UnitCube () : grid_("grids/2dgrid.al")
16   {
17   }
18
19   Dune::AlbertaGrid<2,2>& grid ()
20   {
21     return grid_;
22   }
23
24 private:
25   Dune::AlbertaGrid<2,2> grid_;
26 };
27 #endif
28
29 // AlbertaGrid 3d, variant 1 (6 tetrahedra) specialization
30 #if ALBERTA_DIM == 3 && ALBERTA_WORLD_DIM == 3
31 template<>
32 class UnitCube<Dune::AlbertaGrid<3,3>,1>
```

```
33 {
34 public:
35   typedef Dune::AlbertaGrid<3,3> GridType;
36
37   UnitCube () : grid_("grids/3dgrid.al")
38   {
39   }
40
41   Dune::AlbertaGrid<3,3>& grid ()
42   {
43     return grid_;
44   }
45
46 private:
47   Dune::AlbertaGrid<3,3> grid_;
48 };
49 #endif
50 #endif
51 #endif
```

ALUGrid

The next listing shows the instantiation of `ALUSimplexGrid` or `ALUCubeGrid` objects. The ALUGrid 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.

Listing 11 (File dune-grid-howto/unitcube_alugrid.hh)

```
1  #ifndef UNITCUBE_ALU3DGRID_HH
2  #define UNITCUBE_ALU3DGRID_HH
3
4  #if HAVE_ALUGRID
5  #include <dune/grid/alugrid.hh>
6
7  // ALU3dGrid tetrahedra specialization. Note: element type determined by type
8  template<>
9  class UnitCube<Dune::ALUSimplexGrid<3,3>,1>
10 {
11 public:
12   typedef Dune::ALUSimplexGrid<3,3> GridType;
13
14   UnitCube () : filename("grids/cube.tetra"), grid_(filename.c_str())
15   {}
16
17   GridType& grid ()
18   {
19     return grid_;
20   }
21
22 private:
23   std::string filename;
24   GridType grid_;
25 };
26
27 // ALU2SimplexGrid 2d specialization. Note: element type determined by type
28 template<>
29 class UnitCube<Dune::ALUSimplexGrid<2,2>,1>
30 {
31 public:
32   typedef Dune::ALUSimplexGrid<2,2> GridType;
```

```
33
34   UnitCube () : filename("grids/2dsimplex.alu"), grid_(filename.c_str())
35   {}
36
37   GridType& grid ()
38   {
39     return grid_;
40   }
41
42 private:
43   std::string filename;
44   GridType grid_;
45 };
46
47 // ALU3dGrid hexahedra specialization. Note: element type determined by type
48 template<>
49 class UnitCube<Dune::ALUCubeGrid<3,3>,1>
50 {
51 public:
52   typedef Dune::ALUCubeGrid<3,3> GridType;
53
54   UnitCube () : filename("grids/cube.hexa"), grid_(filename.c_str())
55   {}
56
57   GridType& grid ()
58   {
59     return grid_;
60   }
61
62 private:
63   std::string filename;
64   GridType grid_;
65 };
66 #endif
67
68 #endif
```

4.2 Using configuration information provided by configure

The `./configure` script in the application (`dune-grid-howto` here) produces a file `config.h` that contains information about the configuration parameters. E. g. which of the optional grid implementations is available and which dimension has been selected (if applicable). This information can then be used at compile-time to include header files or code that depend on optional packages.

As an example, the macro `HAVE_UG` can be used to compile UG-specific code as in

```
#if HAVE_UG
#include"dune/grid/uggrid.hh"
#include"dune/io/file/amirameshreader.hh"
#endif
```

It is important that the file `config.h` is the first include file in your application!

4.3 The DGF Parser – reading common macro grid files

Dune has it's 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 under the documentation section (see `http://www.dune-project.org/doc/doxygen/dune-grid-html/group__DuneGridFormatParser.html`).

Here we only give a short introduction. To use the DGF parser the configuration option `--with-grid-dim={1,2,3}` must be provided during configuration run. Optional `--with-grid-type=ALBERTAGRID`. Furthermore, `ALUGRID_CUBE,ALUGRID_SIMPLEX,ALUGRID_CONFORM`, `ONEDGRID,SGRID,UGGRID`, and `YASPGRID` can be chosen as grid types. can be chosen. Note that both values will also be changeable later. If the `--with-grid-dim` option was not provided during configuration the DFG 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 `dgfgridtype.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 follwoing example show 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 be 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.

5 Quadrature rules

In this chapter we explore how an integral

$$\int_\Omega f(x)\ dx$$

over some function $f : \Omega \to \mathbb{R}$ 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 $\hat{f} : \Delta \to \mathbb{R}$ 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 $g : \Delta \to \Omega$:

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

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

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

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

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

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 13 (dune-grid-howto/integrateentity.hh)

```
1  #ifndef DUNE_INTEGRATE_ENTITY_HH
2  #define DUNE_INTEGRATE_ENTITY_HH
3
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

$$E \approx |I_f - I_c| \tag{5.3}$$

is an estimate for the error. If the refinement is such that every element is halfened 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 14 (dune-grid-howto/integration.cc)

```
1  // $Id: integration.cc 184 2007-10-16 12:18:29Z robertk $
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  // checks for defined gridtype and inlcudes appropriate dgfparser implementation
9  #include <dune/grid/io/file/dgfparser/dgfgridtype.hh>
10
11 #include"functors.hh"
12 #include"integrateentity.hh"
13
14 //! uniform refinement test
15 template<class Grid>
16 void uniformintegration (Grid& grid)
17 {
18   // function to integrate
19   Exp<typename Grid::ctype,Grid::dimension> f;
20
21   // get iterator type
22   typedef typename Grid::template Codim<0>::LeafIterator LeafIterator;
23
24   // loop over grid sequence
25   double oldvalue=1E100;
26   for (int k=0; k<10; k++)
27     {
28       // compute integral with some order
29       double value = 0.0;
30       LeafIterator eendit = grid.template leafend<0>();
31       for (LeafIterator it = grid.template leafbegin<0>(); it!=eendit; ++it)
32               value += integrateentity(it,f,1);
33
34       // print result and error estimate
35       std::cout << "elements="
36                 << std::setw(8) << std::right
37                 << grid.size(0)
38                 << "␣integral="
39                 << std::scientific << std::setprecision(12)
40                 << value
41                 << "␣error=" << std::abs(value-oldvalue)
42                 << std::endl;
43
44       // save value of integral
45       oldvalue=value;
46
47       // refine all elements
48       grid.globalRefine(1);
49     }
50 }
51
52 int main(int argc, char **argv)
53 {
54   // initialize MPI, finalize is done automatically on exit
55   Dune::MPIHelper::instance(argc,argv);
56
57   // start try/catch block to get error messages from dune
58   try {
59     using namespace Dune;
60
61     // use unitcube from grids
62     std::stringstream dgfFileName;
```

```
63      dgfFileName << "grids/unitcube" << GridType :: dimension << ".dgf";
64
65      // create grid pointer, GridType is defined by gridtype.hh
66      GridPtr<GridType> gridPtr( dgfFileName.str() );
67
68      // integrate and compute error with extrapolation
69      uniformintegration( *gridPtr );
70    }
71    catch (std::exception & e) {
72      std::cout << "STL ERROR: " << e.what() << std::endl;
73      return 1;
74    }
75    catch (Dune::Exception & e) {
76      std::cout << "DUNE ERROR: " << e.what() << std::endl;
77      return 1;
78    }
79    catch (...) {
80      std::cout << "Unknown ERROR" << std::endl;
81      return 1;
82    }
83
84    // done
85    return 0;
86  }
```

Running the executable `integration` on a YaspGrid in two space dimensions with a qudature 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 32 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

$$m : E' \to I_{E'} \tag{6.1}$$

where $I_{E'} = \{0, \ldots, |E'|-1\} \subset \mathbb{N}$ 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

$$a : I_{E'} \to D(E'). \tag{6.2}$$

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:

$$d_e = a(m(e)). \tag{6.3}$$

DUNE provides different implementations of mappers that differ in functionality and cost (with respect to sorage 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 mapper 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 use mappers to evaluate a function $f : \Omega \to \mathbb{R}$ 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 15 (File dune-grid-howto/elementdata.hh)

```
1  #include<dune/grid/common/referenceelements.hh>
2  #include<dune/grid/common/mcmgmapper.hh>
3  #include<dune/grid/io/file/vtk/vtkwriter.hh>
4  #if HAVE_GRAPE
5  #include<dune/grid/io/visual/grapedatadisplay.hh>
6  #endif
7
8  //! Parameter for mapper class
9  template<int dim>
10 struct P0Layout
11 {
12   bool contains (Dune::GeometryType gt)
13   {
14     if (gt.dim()==dim) return true;
15     return false;
16   }
17 };
18
19 // demonstrate attaching data to elements
20 template<class G, class F>
21 void elementdata (const G& grid, const F& f)
22 {
```

```
23   // the usual stuff
24   const int dim = G::dimension;
25   const int dimworld = G::dimensionworld;
26   typedef typename G::ctype ct;
27   typedef typename G::template Codim<0>::LeafIterator ElementLeafIterator;
28
29   // make a mapper for codim 0 entities in the leaf grid
30   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
31       mapper(grid);
32
33   // allocate a vector for the data
34   std::vector<double> c(mapper.size());
35
36   // iterate through all entities of codim 0 at the leafs
37   for (ElementLeafIterator it = grid.template leafbegin<0>();
38        it!=grid.template leafend<0>(); ++it)
39     {
40       // cell geometry type
41       Dune::GeometryType gt = it->type();
42
43       // cell center in reference element
44       const Dune::FieldVector<ct,dim>&
45         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
46
47       // get global coordinate of cell center
48       Dune::FieldVector<ct,dimworld> global = it->geometry().global(local);
49
50       // evaluate functor and store value
51       c[mapper.map(*it)] = f(global);
52     }
53
54   // generate a VTK file
55   // Dune::LeafP0Function<G,double> cc(grid,c);
56   Dune::VTKWriter<G> vtkwriter(grid);
57   vtkwriter.addCellData(c,"data");
58   vtkwriter.write("elementdata",Dune::VTKOptions::binaryappended);
59
60   // online visualization with Grape
61 #if HAVE_GRAPE
62   {
63     const int polynomialOrder = 0; // we piecewise constant data
64     const int dimRange = 1; // we have scalar data here
65     // create instance of data display
66     Dune::GrapeDataDisplay<G> grape(grid);
67     // display data
68     grape.displayVector("concentration", // name of data that appears in grape
69                         c,  // data vector
70                         grid.leafIndexSet(), // used index set
71                         polynomialOrder, // polynomial order of data
72                         dimRange); // dimRange of data
73   }
74 #endif
75 }
```

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 31. A similar mapper is available also for the entities of a grid level.

The data vector is allocated in line 34. 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 37-52 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 51 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 56-58 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 correponding XML metafile. Using Paraview, [6], you can visualize these files. Lines 61-74 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 16 (File dune-grid-howto/vertexdata.hh)

```
1  #include<dune/grid/common/referenceelements.hh>
2  #include<dune/grid/common/mcmgmapper.hh>
3  #include<dune/grid/io/file/vtk/vtkwriter.hh>
4  #if HAVE_GRAPE
5  #include<dune/grid/io/visual/grapedatadisplay.hh>
6  #endif
7
8  //! Parameter for mapper class
9  template<int dim>
10 struct P1Layout
11 {
12   bool contains (Dune::GeometryType gt)
13   {
14     if (gt.dim()==0) return true;
15     return false;
16   }
17 };
18
19 // demonstrate attaching data to elements
20 template<class G, class F>
21 void vertexdata (const G& grid, const F& f)
22 {
23   // get dimension and coordinate type from Grid
24   const int dim = G::dimension;
25   typedef typename G::ctype ct;
26   // dertermine type of LeafIterator for codimension = dimension
27   typedef typename G::template Codim<dim>::LeafIterator VertexLeafIterator;
28
29   // make a mapper for codim 0 entities in the leaf grid
30   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P1Layout>
31     mapper(grid);
32
33   // allocate a vector for the data
34   std::vector<double> c(mapper.size());
35
```

```
36   // iterate through all entities of codim 0 at the leafs
37   for (VertexLeafIterator it = grid.template leafbegin<dim>();
38        it!=grid.template leafend<dim>(); ++it)
39     {
40       // evaluate functor and store value
41       c[mapper.map(*it)] = f(it->geometry()[0]);
42     }
43
44   // generate a VTK file
45   //   Dune::LeafP1Function<G,double> cc(grid,c);
46   Dune::VTKWriter<G> vtkwriter(grid);
47   vtkwriter.addVertexData(c,"data");
48   vtkwriter.write("vertexdata",Dune::VTKOptions::binaryappended);
49
50   // online visualization with Grape
51 #if HAVE_GRAPE
52   {
53     const int polynomialOrder = 1; // we piecewise linear data
54     const int dimRange = 1; // we have scalar data here
55     // create instance of data display
56     Dune::GrapeDataDisplay<G> grape(grid);
57     // display data
58     grape.displayVector("concentration", // name of data that appears in grape
59                         c,  // data vector
60                         grid.leafIndexSet(), // used index set
61                         polynomialOrder, // polynomial order of data
62                         dimRange); // dimRange of data
63   }
64 #endif
65 }
```

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 programm that drives the two functions just discussed:

Listing 17 (File dune-grid-howto/visualization.cc)

```
1  // $Id: visualization.cc 165 2007-07-29 18:46:12Z robertk $
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 //! supply functor
16 template<class Grid>
17 void dowork (Grid& grid)
18 {
19   // make function object
20   Exp<typename Grid::ctype,Grid::dimension> f;
21
22   // refine the grid
23   grid.globalRefine(5);
24
25   // call the visualization functions
26   elementdata(grid,f);
27   vertexdata(grid,f);
28 }
29
30 int main(int argc, char **argv)
31 {
32   // initialize MPI, finalize is done automatically on exit
33   Dune::MPIHelper::instance(argc,argv);
34
35   // start try/catch block to get error messages from dune
36   try {
37     /*
38     UnitCube<Dune::OneDGrid,1> uc0;
39     UnitCube<Dune::YaspGrid<3,3>,1> uc1;
40     UnitCube<Dune::YaspGrid<2,2>,1> uc2;
41     UnitCube<Dune::SGrid<1,1>,1> uc3;
42     UnitCube<Dune::SGrid<2,2>,1> uc4;
43     UnitCube<Dune::SGrid<3,3>,1> uc5;
44 #if HAVE_UG
45       UnitCube<Dune::UGGrid<3>,2> uc6;
46 #endif
47 #if HAVE_ALBERTA
48 #if ALBERTA_DIM==2
49       UnitCube<Dune::AlbertaGrid<2,2>,1> uc7;
50 #endif
51 #endif
52     */
53     UnitCube<Dune::SGrid<2,2>,1> uc4;
54     dowork(uc4.grid());
55
56 #if HAVE_ALUGRID
57       UnitCube<Dune::ALUSimplexGrid<3,3> ,1> uc8;
58       dowork(uc8.grid());
59 #endif
60   }
61   catch (std::exception & e) {
62     std::cout << "STL ERROR: " << e.what() << std::endl;
63     return 1;
64   }
65   catch (Dune::Exception & e) {
66     std::cout << "DUNE ERROR: " << e.what() << std::endl;
67     return 1;
68   }
69   catch (...) {
70     std::cout << "Unknown ERROR" << std::endl;
71     return 1;
72   }
73
74   // done
75   return 0;
```

```
76 }
```

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

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

where $\Omega \subset \mathbb{R}^d$ is a domain, $T = (0, t_{\text{end}})$ is a time interval, $c : \Omega \times T \to \mathbb{R}$ is the unknown concentration and $u : \Omega \times T \to \mathbb{R}^d$ 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

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

and the boundary condition

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

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

6.3.1 Numerical Scheme

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

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

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

Using integration by parts we arrive at

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

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}:

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

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

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

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:

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

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

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

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

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

When we rewrite 6.12 in the form

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

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 18 (File dune-grid-howto/transportproblem.hh)

```
1  // the initial condition c0
2  template<int dimworld, class ct>
3  double c0 (const Dune::FieldVector<ct,dimworld>& x)
4  {
5    Dune::FieldVector<ct,dimworld> y(0.25);
6    y -= x;
7    if (y.two_norm()<0.125)
8      return 1.0;
9    else
10     return 0.0;
```

```
11 }
12
13 // the boundary condition b on inflow boundary
14 template<int dimworld, class ct>
15 double b (const Dune::FieldVector<ct,dimworld>& x, double t)
16 {
17   return 0.0;
18 }
19
20 // the vector field u is returned in r
21 template<int dimworld, class ct>
22 Dune::FieldVector<double,dimworld> u (const Dune::FieldVector<ct,dimworld>& x, double t)
23 {
24   Dune::FieldVector<double,dimworld> r(0.5);
25   r[0] = 1.0;
26   return r;
27 }
```

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 passed as well.

Listing 19 (File dune-grid-howto/initialize.hh)

```
1  #include<dune/grid/common/referenceelements.hh>
2
3  //! initialize the vector of unknowns with initial value
4  template<class G, class M, class V>
5  void initialize (const G& grid, const M& mapper, V& c)
6  {
7    // first we extract the dimensions of the grid
8    const int dim = G::dimension;
9    const int dimworld = G::dimensionworld;
10
11   // type used for coordinates in the grid
12   typedef typename G::ctype ct;
13
14   // leaf iterator type
15   typedef typename G::template Codim<0>::LeafIterator LeafIterator;
16
17   // iterate through leaf grid an evaluate c0 at cell center
18   LeafIterator endit = grid.template leafend<0>();
19   for (LeafIterator it = grid.template leafbegin<0>(); it!=endit; ++it)
20     {
21       // get geometry type
22       Dune::GeometryType gt = it->type();
23
24       // get cell center in reference element
25       const Dune::FieldVector<ct,dim>&
26         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
27
28       // get global coordinate of cell center
29       Dune::FieldVector<ct,dimworld> global =
30         it->geometry().global(local);
31
32       // initialize cell concentration
33       c[mapper.map(*it)] = c0(global);
34     }
35 }
```

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 20 (File dune-grid-howto/evolve.hh)

```
1  #include <dune/grid/common/referenceelements.hh>
2
3  template<class G, class M, class V>
4  void evolve (const G& grid, const M& mapper, V& c, double t, double& dt)
5  {
6    // first we extract the dimensions of the grid
7    const int dim = G::dimension;
8    const int dimworld = G::dimensionworld;
9
10   // type used for coordinates in the grid
11   typedef typename G::ctype ct;
12
13   // iterator type
14   typedef typename G::template Codim<0>::LeafIterator LeafIterator;
15
16   // intersection iterator type
17   typedef typename G::template Codim<0>::LeafIntersectionIterator IntersectionIterator;
18
19   // entity pointer type
20   typedef typename G::template Codim<0>::EntityPointer EntityPointer;
21
22   // allocate a temporary vector for the update
23   V update(c.size());
24   for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
25
26   // initialize dt very large
27   dt = 1E100;
28
29   // compute update vector and optimum dt in one grid traversal
30   LeafIterator endit = grid.template leafend<0>();
31   for (LeafIterator it = grid.template leafbegin<0>(); it!=endit; ++it)
32     {
33       // cell geometry type
34       Dune::GeometryType gt = it->type();
35
36       // cell center in reference element
37       const Dune::FieldVector<ct,dim>&
38         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
39
40       // cell center in global coordinates
41       Dune::FieldVector<ct,dimworld>
42         global = it->geometry().global(local);
43
44       // cell volume, assume linear map here
45       double volume = it->geometry().integrationElement(local)
46         *Dune::ReferenceElements<ct,dim>::general(gt).volume();
47
48       // cell index
49       int indexi = mapper.map(*it);
50
51       // variable to compute sum of positive factors
52       double sumfactor = 0.0;
53
54       // run through all intersections with neighbors and boundary
55       IntersectionIterator isend = it->ileafend();
56       for (IntersectionIterator is = it->ileafbegin(); is!=isend; ++is)
57         {
```

```
58            // get geometry type of face
59            Dune::GeometryType gtf = is.intersectionSelfLocal().type();
60
61            // center in face's reference element
62            const Dune::FieldVector<ct,dim-1>&
63              facelocal = Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);
64
65            // get normal vector scaled with volume
66            Dune::FieldVector<ct,dimworld> integrationOuterNormal
67              = is.integrationOuterNormal(facelocal);
68            integrationOuterNormal
69              *= Dune::ReferenceElements<ct,dim-1>::general(gtf).volume();
70
71            // center of face in global coordinates
72            Dune::FieldVector<ct,dimworld>
73              faceglobal = is.intersectionGlobal().global(facelocal);
74
75            // evaluate velocity at face center
76            Dune::FieldVector<double,dim> velocity = u(faceglobal,t);
77
78            // compute factor occuring in flux formula
79            double factor = velocity*integrationOuterNormal/volume;
80
81            // for time step calculation
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                    Dune::GeometryType nbgt = outside->type();
98                    const Dune::FieldVector<ct,dim>&
99                      nblocal = Dune::ReferenceElements<ct,dim>::general(nbgt).position(0,0);
100                   double nbvolume = outside->geometry().integrationElement(nblocal)
101                     *Dune::ReferenceElements<ct,dim>::general(nbgt).volume();
102                   double nbfactor = velocity*integrationOuterNormal/nbvolume;
103
104                   if (factor<0) // inflow
105                     {
106                       update[indexi] -= c[indexj]*factor;
107                       update[indexj] += c[indexj]*nbfactor;
108                     }
109                   else // outflow
110                     {
111                       update[indexi] -= c[indexi]*factor;
112                       update[indexj] += c[indexi]*nbfactor;
113                     }
114               }
115             }
116
117           // handle boundary face
118           if (is.boundary())
119             if (factor<0) // inflow, apply boundary condition
120               update[indexi] -= b(faceglobal,t)*factor;
```

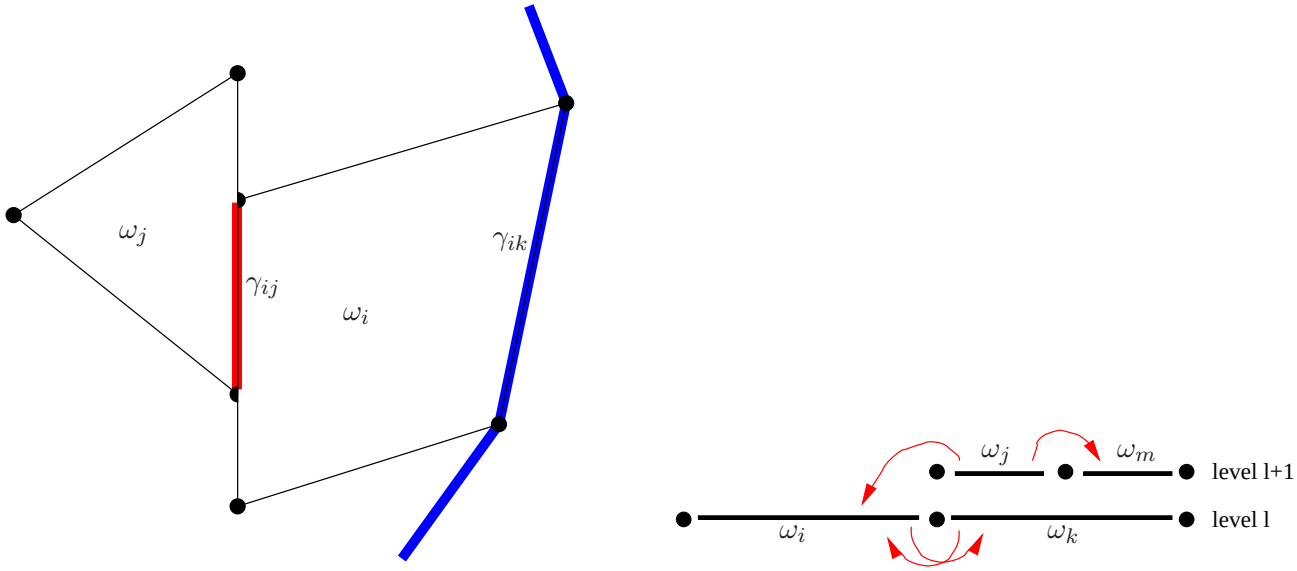


Figure 6.1: Left: inntersection with other elements and the boundary, right: intersections in the case of locally refined grids.

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

Lines 30-128 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 23, where we assume that `V` is a container with copy constructor and size method.

The computation of the fluxes is done in lines 55-123. An `IntersectionIterator` is used to access all intersections γ_{ij} of a leaf element ω_i. For a full documentation of the `IntersectionIterator` we refer to

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

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 118), 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 accessibly. 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 the the situation of ω_j refering 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. The latter condition with the indices just ensures that the flux is only computed once.

The Δt^n calculation is done in line 126 where the minimum over all cells is taken. Then, line 131 multiplies the optimum Δt^n with a safety factor to avoid any instability due to round-off errors.

Finally, line 135 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 21 (File dune-grid-howto/vtkout.hh)

```
1  #include <dune/grid/io/file/vtk/vtkwriter.hh>
2  #include <stdio.h>
3
4  template<class G, class V>
5  void vtkout (const G& grid, const V& c, char* name, int k)
6  {
7    Dune::VTKWriter<G> vtkwriter(grid);
8    char fname[128];
9    sprintf(fname,"%s-%05d",name,k);
10   vtkwriter.addCellData(c,"celldata");
11   vtkwriter.write(fname,Dune::VTKOptions::ascii);
12 }
```

Finally, the main program:

Listing 22 (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  // checks for defined gridtype and inlcudes appropriate dgfparser implementation
```

```
9  #include <dune/grid/io/file/dgfparser/dgfgridtype.hh>
10
11 #include"vtkout.hh"
12 #include"unitcube.hh"
13 #include"transportproblem2.hh"
14 #include"initialize.hh"
15 #include"evolve.hh"
16
17 //===============================================================
18 // the time loop function working for all types of grids
19 //===============================================================
20
21 //! Parameter for mapper class
22 template<int dim>
23 struct P0Layout
24 {
25   bool contains (Dune::GeometryType gt)
26   {
27     if (gt.dim()==dim) return true;
28     return false;
29   }
30 };
31
32 template<class G>
33 void timeloop (const G& grid, double tend)
34 {
35   // make a mapper for codim 0 entities in the leaf grid
36   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
37     mapper(grid);
38
39   // allocate a vector for the concentration
40   std::vector<double> c(mapper.size());
41
42   // initialize concentration with initial values
43   initialize(grid,mapper,c);
44   vtkout(grid,c,"concentration",0);
45
46   // now do the time steps
47   double t=0,dt;
48   int k=0;
49   const double saveInterval = 0.1;
50   double saveStep = 0.1;
51   int counter = 0;
52
53   while (t<tend)
54     {
55     // augment time step counter
56     ++k;
57
58     // apply finite volume scheme
59     evolve(grid,mapper,c,t,dt);
60
61     // augment time
62     t += dt;
63
64     // check if data should be written
65     if (t >= saveStep)
66     {
67       // write data
68       vtkout(grid,c,"concentration",counter);
69
70       // increase counter and saveStep for next interval
71       saveStep += saveInterval;
```

```
72        ++counter;
73      }
74
75      // print info about time, timestep size and counter
76      std::cout << "s=" << grid.size(0) << " k=" << k << " t=" << t << " dt=" << dt << std::endl;
77      }
78
79    // output results
80    vtkout(grid,c,"concentration",counter);
81  }
82
83  //=============================================================
84  // The main function creates objects and does the time loop
85  //=============================================================
86
87  int main (int argc , char ** argv)
88  {
89    // initialize MPI, finalize is done automatically on exit
90    Dune::MPIHelper::instance(argc,argv);
91
92    // start try/catch block to get error messages from dune
93    try {
94      using namespace Dune;
95
96      // use unitcube from grids
97      std::stringstream dgfFileName;
98      dgfFileName << "grids/unitcube" << GridType :: dimension << ".dgf";
99
100     // create grid pointer, GridType is defined by gridtype.hh
101     GridPtr<GridType> gridPtr( dgfFileName.str() );
102
103     // grid reference
104     GridType& grid = *gridPtr;
105
106     // half grid width 4 times
107     int level = 4 * DGFGridInfo<GridType>::refineStepsForHalf();
108
109     // refine grid until upper limit of level
110     grid.globalRefine(level);
111
112     // do time loop until end time 0.5
113     timeloop(grid, 0.5);
114   }
115   catch (std::exception & e) {
116     std::cout << "STL ERROR: " << e.what() << std::endl;
117     return 1;
118   }
119   catch (Dune::Exception & e) {
120     std::cout << "DUNE ERROR: " << e.what() << std::endl;
121     return 1;
122   }
123   catch (...) {
124     std::cout << "Unknown ERROR" << std::endl;
125     return 1;
126   }
127
128   // done
129   return 0;
130 }
```

The function `timeloop` constructs a mapper and allocates the concentration vector with one entry per element in the leaf grid. In line 43 this vector is initialized with the initial concentration and the

loop in line 53-77 evolves the concentration in time. Finally, the simulation result is written to a file in line 80.

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 intgral 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

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

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 knew the maximum local error on all the new elements that resulted from refining the current mesh (without actually doing so). Then it would be a good idea to refine only those elements in the mesh where the local error is not already below that maximum local error that will be attained anyway. In [4] it is shown that the local error after mesh refinement can be effectively computed without actually doing the refinement. Consider an element 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

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

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:

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

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: adaptiveintegration.cc 183 2007-10-16 11:48:01Z robertk $
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  // checks for defined gridtype and inlcudes appropriate dgfparser implementation
10 #include <dune/grid/io/file/dgfparser/dgfgridtype.hh>
11
12
13 #include"unitcube.hh"
14 #include"functors.hh"
15 #include"integrateentity.hh"
16
17
18 //! adaptive refinement test
19 template<class Grid, class Functor>
20 void adaptiveintegration (Grid& grid, const Functor& f)
21 {
22   // get iterator type
23   typedef typename Grid::template Codim<0>::LeafIterator ElementLeafIterator;
24
25   // algorithm parameters
26   const double tol=1E-8;
27   const int loworder=1;
28   const int highorder=3;
29
30   // loop over grid sequence
31   double oldvalue=1E100;
32   for (int k=0; k<100; k++)
33     {
34       // compute integral on current mesh
35           double value=0;
36       for (ElementLeafIterator it = grid.template leafbegin<0>();
37            it!=grid.template leafend<0>(); ++it)
38         value += integrateentity(it,f,highorder);
39
40       // print result
```

```
41        double estimated_error = std::abs(value-oldvalue);
42        oldvalue=value; // save value for next estimate
43        std::cout << "elements="
44                  << std::setw(8) << std::right
45                  << grid.size(0)
46                  << "␣integral="
47                  << std::scientific << std::setprecision(8)
48                  << value
49                  << "␣error=" << estimated_error
50                  << std::endl;
51
52        // check convergence
53        if (estimated_error <= tol*value)
54          break;
55
56        // refine grid globally in first step to ensure
57        // that every element has a father
58        if (k==0)
59          {
60            grid.globalRefine(1);
61            continue;
62          }
63
64        // compute threshold for subsequent refinement
65        double maxerror=-1E100;
66        double maxextrapolatederror=-1E100;
67        for (ElementLeafIterator it = grid.template leafbegin<0>();
68             it!=grid.template leafend<0>(); ++it)
69          {
70            // error on this entity
71            double lowresult=integrateentity(it,f,loworder);
72            double highresult=integrateentity(it,f,highorder);
73            double error = std::abs(lowresult-highresult);
74
75            // max over whole grid
76            maxerror = std::max(maxerror,error);
77
78            // error on father entity
79            double fatherlowresult=integrateentity(it->father(),f,loworder);
80            double fatherhighresult=integrateentity(it->father(),f,highorder);
81            double fathererror = std::abs(fatherlowresult-fatherhighresult);
82
83            // local extrapolation
84            double extrapolatederror = error*error/(fathererror+1E-30);
85            maxextrapolatederror = std::max(maxextrapolatederror,extrapolatederror);
86          }
87        double kappa = std::min(maxextrapolatederror,0.5*maxerror);
88
89        // mark elements for refinement
90        for (ElementLeafIterator it = grid.template leafbegin<0>();
91             it!=grid.template leafend<0>(); ++it)
92          {
93            double lowresult=integrateentity(it,f,loworder);
94            double highresult=integrateentity(it,f,highorder);
95            double error = std::abs(lowresult-highresult);
96            if (error>kappa) grid.mark(1,it);
97          }
98
99        // adapt the mesh
100       grid.preAdapt();
101       grid.adapt();
102       grid.postAdapt();
103     }
```

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

The work is done in the function `adaptiveintegration`. Lines 35-38 compute the value of the integral on the current mesh. After printing the result the decission whether to continue or not is done in line 53. 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 60). Now the refinement threshold κ can be computed in lines 65-87. Finally the last loop in lines 90-97 marks elements for refinement and lines 100-102 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.

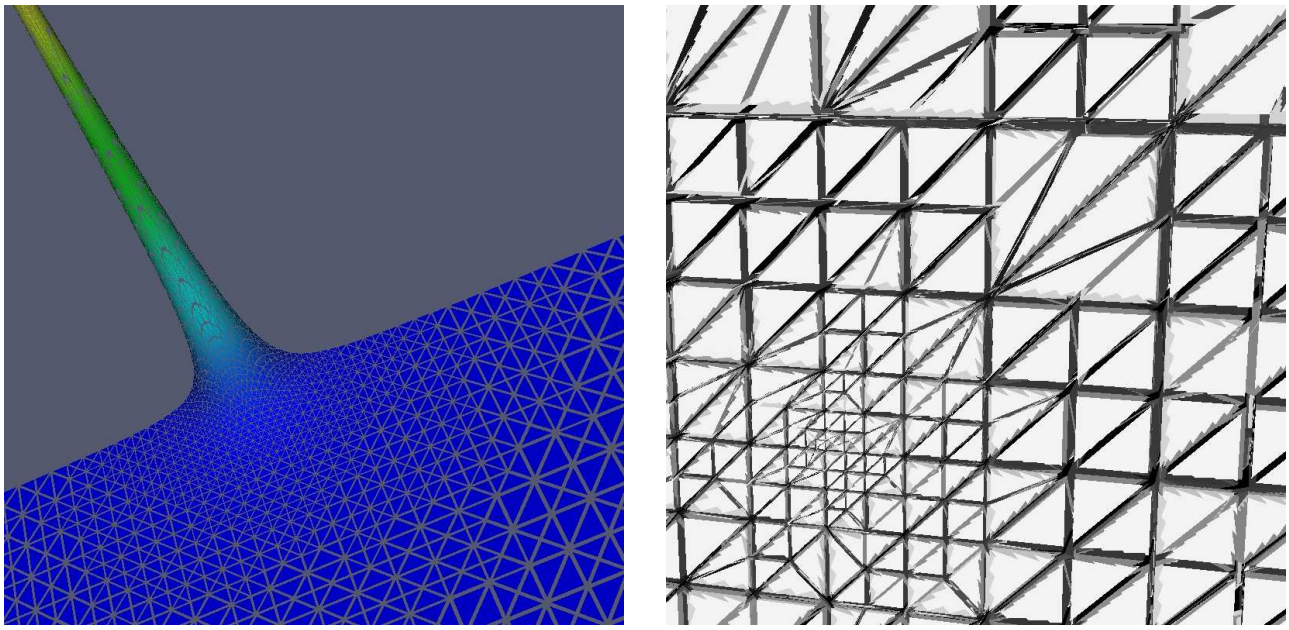


Figure 7.1: Two and three-dimensional grids generated by the adaptive integration algorithm applied to the needle pulse. Left grid is generated using Alberta, right grid is generated using UG.

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

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

. 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 refined $\underline{M}$ times and at most $\overline{M}$ times.

After mesh modification the solution from the previous grid must be transfered 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 from through refining some element in the old mesh: Copy the value from this element in the old mesh.

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

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

```
1  #include <map>
2
3  struct RestrictedValue
4  {
5    double value;
6    int count;
7    RestrictedValue ()
8    {
9      value = 0;
10     count = 0;
11   }
12 };
13
14 template<class G, class M, class V>
15 bool finitevolumeadapt (G& grid, M& mapper, V& c, int lmin, int lmax, int k)
16 {
17   // tol value for refinement strategy
18   const double refinetol  = 0.05;
19   const double coarsentol = 0.001;
20
21   // type used for coordinates in the grid
22   typedef typename G::ctype ct;
23
24   // iterator types
25   typedef typename G::template Codim<0>::LeafIterator LeafIterator;
26   typedef typename G::template Codim<0>::LevelIterator LevelIterator;
27
28   // entity pointer
29   typedef typename G::template Codim<0>::EntityPointer EntityPointer;
30
31   // intersection iterator type
32   typedef typename G::template Codim<0>::LeafIntersectionIterator IntersectionIterator;
```

```
33
34  // global id set types
35  typedef typename G::template Codim<0>::LocalIdSet IdSet;
36  typedef typename IdSet::IdType IdType;
37
38  // compute cell indicators
39  V indicator(c.size(),-1E100);
40  double globalmax = -1E100;
41  double globalmin =  1E100;
42  for (LeafIterator it = grid.template leafbegin<0>();
43       it!=grid.template leafend<0>(); ++it)
44    {
45      // my index
46      int indexi = mapper.map(*it);
47
48      // global min/max
49      globalmax = std::max(globalmax,c[indexi]);
50      globalmin = std::min(globalmin,c[indexi]);
51
52      IntersectionIterator isend = it->ileafend();
53      for (IntersectionIterator is = it->ileafbegin(); is!=isend; ++is)
54        if (is.neighbor())
55          {
56            // access neighbor
57            EntityPointer outside = is.outside();
58            int indexj = mapper.map(*outside);
59
60            // handle face from one side only
61            if ( it.level()>outside->level() ||
62                 (it.level()==outside->level() && indexi<indexj) )
63              {
64                double localdelta = std::abs(c[indexj]-c[indexi]);
65                indicator[indexi] = std::max(indicator[indexi],localdelta);
66                indicator[indexj] = std::max(indicator[indexj],localdelta);
67              }
68          }
69    }
70
71  // mark cells for refinement/coarsening
72  double globaldelta = globalmax-globalmin;
73  int marked=0;
74  for (LeafIterator it = grid.template leafbegin<0>();
75       it!=grid.template leafend<0>(); ++it)
76    {
77      if (indicator[mapper.map(*it)]>refinetol*globaldelta
78          && (it.level()<lmax || !it->isRegular()))
79        {
80          grid.mark(1,it);
81          marked++;
82          IntersectionIterator isend = it->ileafend();
83          for (IntersectionIterator is = it->ileafbegin(); is!=isend; ++is)
84            if (is.neighbor())
85              if (is.outside().level()<lmax || !is.outside()->isRegular())
86                grid.mark(1,is.outside());
87        }
88      if (indicator[mapper.map(*it)]<coarsentol*globaldelta && it.level()>lmin)
89        {
90          grid.mark(-1,it);
91          marked++;
92        }
93    }
94  if (marked==0) return false;
95
```

```
96  // restrict to coarse elements
97  std::map<IdType,RestrictedValue> restrictionmap; // restricted concentration
98  const IdSet& idset = grid.localIdSet();
99  for (int level=grid.maxLevel(); level>=0; level--)
100   for (LevelIterator it = grid.template lbegin<0>(level);
101        it!=grid.template lend<0>(level); ++it)
102     {
103       // get your map entry
104       IdType idi = idset.id(*it);
105       RestrictedValue& rv = restrictionmap[idi];
106
107       // put your value in the map
108       if (it->isLeaf())
109         {
110           int indexi = mapper.map(*it);
111           rv.value = c[indexi];
112           rv.count = 1;
113         }
114
115       // average in father
116       if (it.level()>0)
117         {
118           EntityPointer ep = it->father();
119           IdType idf = idset.id(*ep);
120           RestrictedValue& rvf = restrictionmap[idf];
121           rvf.value += rv.value/rv.count;
122           rvf.count += 1;
123         }
124     }
125 grid.preAdapt();
126
127 // adapt mesh and mapper
128 bool rv=grid.adapt();
129 mapper.update();
130 c.resize(mapper.size());
131
132 // interpolate new cells, restrict coarsened cells
133 for (int level=0; level<=grid.maxLevel(); level++)
134   for (LevelIterator it = grid.template lbegin<0>(level);
135        it!=grid.template lend<0>(level); ++it)
136     {
137       // get your id
138       IdType idi = idset.id(*it);
139
140       // check map entry
141       typename std::map<IdType,RestrictedValue>::iterator rit = restrictionmap.find(idi);
142       if (rit!=restrictionmap.end())
143         {
144           // entry is in map, write in leaf
145           if (it->isLeaf())
146             {
147               int indexi = mapper.map(*it);
148               c[indexi] = rit->second.value/rit->second.count;
149             }
150         }
151       else
152         {
153           // value is not in map, interpolate
154           if (it.level()>0)
155             {
156               EntityPointer ep = it->father();
157               IdType idf = idset.id(*ep);
158               RestrictedValue& rvf = restrictionmap[idf];
```

```
159                 if (it->isLeaf())
160                   {
161                     int indexi = mapper.map(*it);
162                     c[indexi] = rvf.value/rvf.count;
163                   }
164                 else
165                   {
166                     // create new entry
167                     RestrictedValue& rv = restrictionmap[idi];
168                     rv.value = rvf.value/rvf.count;
169                     rv.count = 1;
170                   }
171               }
172           }
173       }
174   grid.postAdapt();
175
176   return rv;
177 }
```

The loop in lines 42-69 computes the indicator values η_i as well as the global minimum and maximum $\overline{C}, \underline{C}$. Then the next loop in lines 74-93 marks the elements for refinement. Lines 97-124 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 grid can really be modified in line 128 by calling the `adapt()` method on the grid object. The mapper is updated to reflect the changes in the grid in line 129 and the concentration vector is resized to the new size in line 130. 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 133-173.

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  // checks for defined gridtype and inlcudes appropriate dgfparser implementation
7  #include <dune/grid/io/file/dgfparser/dgfgridtype.hh>
8
9  #include <dune/grid/common/mcmgmapper.hh> // mapper class
10 #include <dune/common/mpihelper.hh> // include mpi helper class
11
12 #include "vtkout.hh"
13 #include "unitcube.hh"
14 #include "transportproblem2.hh"
15 #include "initialize.hh"
16 #include "evolve.hh"
17 #include "finitevolumeadapt.hh"
18
19 //===============================================================
20 // the time loop function working for all types of grids
21 //===============================================================
```

```
22
23 //! Parameter for mapper class
24 template<int dim>
25 struct P0Layout
26 {
27   bool contains (Dune::GeometryType gt)
28   {
29     if (gt.dim()==dim) return true;
30     return false;
31   }
32 };
33
34 template<class G>
35 void gnuplot (G& grid, std::vector<double>& c)
36 {
37   // first we extract the dimensions of the grid
38   const int dim = G::dimension;
39   const int dimworld = G::dimensionworld;
40
41   // type used for coordinates in the grid
42   // such a type is exported by every grid implementation
43   typedef typename G::ctype ct;
44
45   // the grid has an iterator providing the access to
46   // all elements (better codim 0 entities) which are leafs
47   // of the refinement tree.
48   // Note the use of the typename keyword and the traits class
49   typedef typename G::template Codim<0>::LeafIterator ElementLeafIterator;
50
51   // make a mapper for codim 0 entities in the leaf grid
52   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
53     mapper(grid);
54
55   // iterate through all entities of codim 0 at the leafs
56   int count = 0;
57   for (ElementLeafIterator it = grid.template leafbegin<0>();
58        it!=grid.template leafend<0>(); ++it)
59     {
60       Dune::GeometryType gt = it->type();
61       const Dune::FieldVector<ct,dim>&
62         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
63       Dune::FieldVector<ct,dimworld>
64         global = it->geometry().global(local);
65       std::cout << global[0] << "␣" << c[mapper.map(*it)] << std::endl;
66       count++;
67     }
68 }
69
70
71 template<class G>
72 void timeloop (G& grid, double tend, int lmin, int lmax)
73 {
74   // make a mapper for codim 0 entities in the leaf grid
75   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
76     mapper(grid);
77
78   // allocate a vector for the concentration
79   std::vector<double> c(mapper.size());
80
81   // initialize concentration with initial values
82   initialize(grid,mapper,c);
83   for (int i=grid.maxLevel(); i<lmax; i++)
84     {
```

```
85        if (grid.maxLevel()>=lmax) break;
86        finitevolumeadapt(grid,mapper,c,lmin,lmax,0);
87        initialize(grid,mapper,c);
88      }
89
90    // write initial data
91    vtkout(grid,c,"concentration",0);
92
93    // variables for time, timestep etc.
94    double dt, t=0;
95    double saveStep = 0.1;
96    const double saveInterval = 0.1;
97    int counter = 0;
98    int k = 0;
99
100   std::cout << "s=" << grid.size(0) << " k=" << k << " t=" << t << std::endl;
101   while (t<tend)
102     {
103     // augment time step counter
104     ++k;
105
106     // apply finite volume scheme
107       evolve(grid,mapper,c,t,dt);
108
109     // augment time
110       t += dt;
111
112     // check if data should be written
113     if (t >= saveStep)
114     {
115       // write data
116       vtkout(grid,c,"concentration",counter);
117
118       // increase counter and saveStep for next interval
119       saveStep += saveInterval;
120       ++counter;
121     }
122
123     // print info about time, timestep size and counter
124     std::cout << "s=" << grid.size(0) << " k=" << k << " t=" << t << " dt=" << dt << std::endl;
125
126     // for unstructured grids call adaptation algorithm
127     if( Dune :: Capabilities :: IsUnstructured<G> :: v )
128     {
129         finitevolumeadapt(grid,mapper,c,lmin,lmax,k);
130     }
131     }
132
133   // write last time step
134   vtkout(grid,c,"concentration",counter);
135 }
136
137 //===============================================================
138 // The main function creates objects and does the time loop
139 //===============================================================
140
141 int main (int argc , char ** argv)
142 {
143   // initialize MPI, finalize is done automatically on exit
144   Dune::MPIHelper::instance(argc,argv);
145
146   // start try/catch block to get error messages from dune
147   try {
```

```
148     using namespace Dune;
149
150     // use unitcube from grids
151     std::stringstream dgfFileName;
152     dgfFileName << "grids/unitcube" << GridType :: dimension << ".dgf";
153
154     // create grid pointer, GridType is defined by gridtype.hh
155     GridPtr<GridType> gridPtr( dgfFileName.str() );
156
157     // grid reference
158     GridType& grid = *gridPtr;
159
160     // minimal allowed level during refinement
161     int minLevel = 2 * DGFGridInfo<GridType>::refineStepsForHalf();
162
163     // refine grid until upper limit of level
164     grid.globalRefine(minLevel);
165
166     // maximal allowed level during refinement
167     int maxLevel = minLevel + 3 * DGFGridInfo<GridType>::refineStepsForHalf();
168
169     // do time loop until end time 0.5
170     timeloop(grid, 0.5, minLevel, maxLevel);
171   }
172   catch (std::exception & e) {
173     std::cout << "STL ERROR: " << e.what() << std::endl;
174     return 1;
175   }
176   catch (Dune::Exception & e) {
177     std::cout << "DUNE ERROR: " << e.what() << std::endl;
178     return 1;
179   }
180   catch (...) {
181     std::cout << "Unknown ERROR" << std::endl;
182     return 1;
183   }
184
185   // done
186   return 0;
187 }
```

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 run 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* for 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
$B_i = \overline{\partial\Omega_i \setminus \partial\Omega}$	*border*
$\overline{\Omega_i} \setminus B_i$	*interior*
$F_i = \overline{\partial\hat{\Omega}_i \setminus \partial\Omega} \setminus B_i$	*front*
$\overline{\hat{\Omega}_i} \setminus (B_i \cup F_i)$	*overlap*
Rest	*ghost*

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

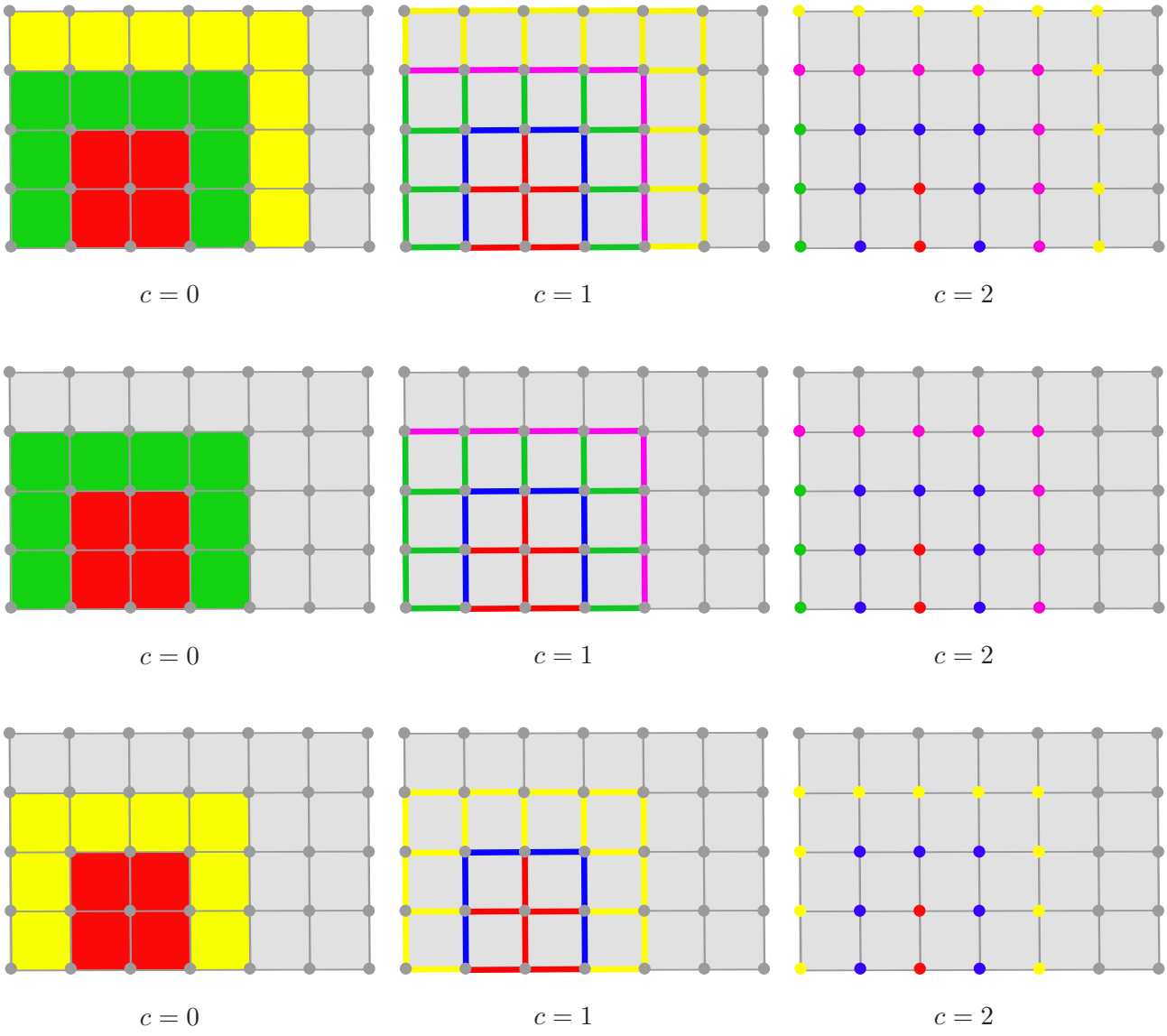


Figure 8.1: Color coded illustration of different data decompositions: interior (red), border (blue), overlap (green), front (magenta) and ghost (yellow), gray encodes entites not stored by the process. First row shows case with interior, overlap and ghost entities, second row shows a case with interior 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 $X^c_{i,j}$ 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 $X^c_{i,j} \neq \emptyset$. The message contains some data for every entity in $X^c_{i,j}$. Since all processes participate in the communication step, process i will receive data from a process j whenever $X^c_{j,i} \neq \emptyset$. This data corresponds to entities in process i that have a copy in $X^c_{j,i}$.

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  // A DataHandle class to exchange entries of a vector
2  template<class M, class V> // mapper type and vector type
3  class VectorExchange
4    : public Dune::CommDataHandleIF<VectorExchange<M,V>,
5                    typename V::value_type>
6  {
7  public:
8    //! export type of data for message buffer
9    typedef typename V::value_type DataType;
10
11   //! returns true if data for this codim should be communicated
12   bool contains (int dim, int codim) const
13   {
14     return (codim==0);
15   }
16
17   //! returns true if size per entity of given dim and codim is a constant
18   bool fixedsize (int dim, int codim) const
19   {
20     return true;
21   }
22
```

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

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 20 on page 44.

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

```
1  #include <dune/grid/common/referenceelements.hh>
2
3  template<class G, class M, class V>
4  void parevolve (const G& grid, const M& mapper, V& c, double t, double& dt)
5  {
6    // check data partitioning
7    assert(grid.overlapSize(0)>0 || (grid.ghostSize(0)>0));
8
9    // first we extract the dimensions of the grid
10   const int dim = G::dimension;
11   const int dimworld = G::dimensionworld;
12
13   // type used for coordinates in the grid
14   typedef typename G::ctype ct;
15
16   // iterator type
17   typedef typename G::template Codim<0>::
18     template Partition<Dune::All_Partition>::LeafIterator LeafIterator;
19
20   // intersection iterator type
21   typedef typename G::template Codim<0>::LeafIntersectionIterator IntersectionIterator;
22
23   // entity pointer type
24   typedef typename G::template Codim<0>::EntityPointer EntityPointer;
25
26   // allocate a temporary vector for the update
27   V update(c.size());
28   for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;
29
30   // initialize dt very large
31   dt = 1E100;
32
33   // compute update vector and optimum dt in one grid traversal
34   // iterate over all entities, but update is only used on interior entities
35   LeafIterator endit = grid.template leafend<0,Dune::All_Partition>();
36   for (LeafIterator it = grid.template leafbegin<0,Dune::All_Partition>(); it!=endit; ++it)
37     {
38       // cell geometry type
39       Dune::GeometryType gt = it->type();
40
41       // cell center in reference element
42       const Dune::FieldVector<ct,dim>&
43         local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);
44
45       // cell center in global coordinates
46       Dune::FieldVector<ct,dimworld>
47         global = it->geometry().global(local);
48
49       // cell volume, assume linear map here
50       double volume = it->geometry().integrationElement(local)
51         *Dune::ReferenceElements<ct,dim>::general(gt).volume();
52
53       // cell index
54       int indexi = mapper.map(*it);
```

```
55
56        // variable to compute sum of positive factors
57        double sumfactor = 0.0;
58
59        // run through all intersections with neighbors and boundary
60        IntersectionIterator isend = it->ileafend();
61        for (IntersectionIterator is = it->ileafbegin(); is!=isend; ++is)
62          {
63            // get geometry type of face
64            Dune::GeometryType gtf = is.intersectionSelfLocal().type();
65
66            // center in face's reference element
67            const Dune::FieldVector<ct,dim-1>&
68              facelocal = Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);
69
70            // get normal vector scaled with volume
71            Dune::FieldVector<ct,dimworld> integrationOuterNormal
72              = is.integrationOuterNormal(facelocal);
73            integrationOuterNormal
74              *= Dune::ReferenceElements<ct,dim-1>::general(gtf).volume();
75
76            // center of face in global coordinates
77            Dune::FieldVector<ct,dimworld>
78              faceglobal = is.intersectionGlobal().global(facelocal);
79
80            // evaluate velocity at face center
81            Dune::FieldVector<double,dim> velocity = u(faceglobal,t);
82
83            // compute factor occuring in flux formula
84            double factor = velocity*integrationOuterNormal/volume;
85
86            // for time step calculation
87            if (factor>=0) sumfactor += factor;
88
89            // handle interior face
90            if (is.neighbor())
91              {
92                // access neighbor
93                EntityPointer outside = is.outside();
94                int indexj = mapper.map(*outside);
95
96                // handle face from one side
97                if ( it->level()>outside->level() ||
98                     (it->level()==outside->level() && indexi<indexj) )
99                  {
100                   // compute factor in neighbor
101                   Dune::GeometryType nbgt = outside->type();
102                   const Dune::FieldVector<ct,dim>&
103                     nblocal = Dune::ReferenceElements<ct,dim>::general(nbgt).position(0,0);
104                   double nbvolume = outside->geometry().integrationElement(nblocal)
105                     *Dune::ReferenceElements<ct,dim>::general(nbgt).volume();
106                   double nbfactor = velocity*integrationOuterNormal/nbvolume;
107
108                   if (factor<0) // inflow
109                     {
110                       update[indexi] -= c[indexj]*factor;
111                       update[indexj] += c[indexj]*nbfactor;
112                     }
113                   else // outflow
114                     {
115                       update[indexi] -= c[indexi]*factor;
116                       update[indexj] += c[indexi]*nbfactor;
117                     }
```

```
118                 }
119               }
120
121             // handle boundary face
122             if (is.boundary())
123               if (factor<0) // inflow, apply boundary condition
124                 update[indexi] -= b(faceglobal,t)*factor;
125               else // outflow
126                 update[indexi] -= c[indexi]*factor;
127           } // end all intersections
128
129         // compute dt restriction
130         if (it->partitionType()==Dune::InteriorEntity)
131           dt = std::min(dt,1.0/sumfactor);
132
133       } // end grid traversal
134
135     // global min over all partitions
136     dt = grid.comm().min(dt);
137     // scale dt with safety factor
138     dt *= 0.99;
139
140     // exchange update
141     VectorExchange<M,V> dh(mapper,update);
142     grid.template
143       communicate<VectorExchange<M,V> >(dh,Dune::InteriorBorder_All_Interface,
144                                         Dune::ForwardCommunication);
145
146     // update the concentration vector
147     for (unsigned int i=0; i<c.size(); ++i)
148       c[i] += dt*update[i];
149
150     return;
151   }
```

The first difference to the sequential version is in line 7 where it is checked that the grid provides an overlap of at least one element. The overlap my 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 partiton type the leaf and level iterators can be parametrized by an additional argument `PartitionIteratorType` as shown in line 18. 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 explicitely in an iterator the same argument has also to be specified in the begin and end methods on the grid as shown in lines 35-36.

The next change is in line 130 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 136. 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 141-144 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)

```
#include"config.h"                // know what grids are present
#include<iostream>                 // for input/output to shell
#include<fstream>                  // for input/output to files
#include<vector>                   // STL vector class
#include<dune/grid/common/mcmgmapper.hh> // mapper class
#include <dune/common/mpihelper.hh> // include mpi helper class

#include"vtkout.hh"
#include"unitcube.hh"
#include"transportproblem.hh"
#include"initialize.hh"
#include"parfvdatahandle.hh"
#include"parevolve.hh"


//===============================================================
// the time loop function working for all types of grids
//===============================================================

//! Parameter for mapper class
template<int dim>
struct P0Layout
{
  bool contains (Dune::GeometryType gt)
  {
    if (gt.dim()==dim) return true;
    return false;
  }
};

template<class G>
void partimeloop (const G& grid, double tend)
{
  // make a mapper for codim 0 entities in the leaf grid
  Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
    mapper(grid);

  // allocate a vector for the concentration
  std::vector<double> c(mapper.size());

  // initialize concentration with initial values
  initialize(grid,mapper,c);
  vtkout(grid,c,"pconc",0);

  // now do the time steps
  double t=0,dt;
  int k=0;
  while (t<tend)
    {
      k++;
      parevolve(grid,mapper,c,t,dt);
      t += dt;
      if (grid.comm().rank()==0)
        std::cout << "k=" << k << "␣t=" << t << "␣dt=" << dt << std::endl;
      if (k%20==0) vtkout(grid,c,"pconc",k/20);
    }
  vtkout(grid,c,"pconc",k/20);
}

//===============================================================
// The main function creates objects and does the time loop
//===============================================================
```

```
63
64 int main (int argc , char ** argv)
65 {
66   // initialize MPI, finalize is done automatically on exit
67   Dune::MPIHelper::instance(argc,argv);
68
69   // start try/catch block to get error messages from dune
70   try {
71     UnitCube<Dune::YaspGrid<2,2>,64> uc;
72     uc.grid().globalRefine(2);
73     partimeloop(uc.grid(),0.5);
74   }
75   catch (std::exception & e) {
76     std::cout << "STL ERROR: " << e.what() << std::endl;
77     return 1;
78   }
79   catch (Dune::Exception & e) {
80     std::cout << "DUNE ERROR: " << e.what() << std::endl;
81     return 1;
82   }
83   catch (...) {
84     std::cout << "Unknown ERROR" << std::endl;
85     return 1;
86   }
87
88   // done
89   return 0;
90 }
```

The only essential difference to the sequential program is in line 53 where the printing of the data of the current time step is restricted to the process with rank 0.

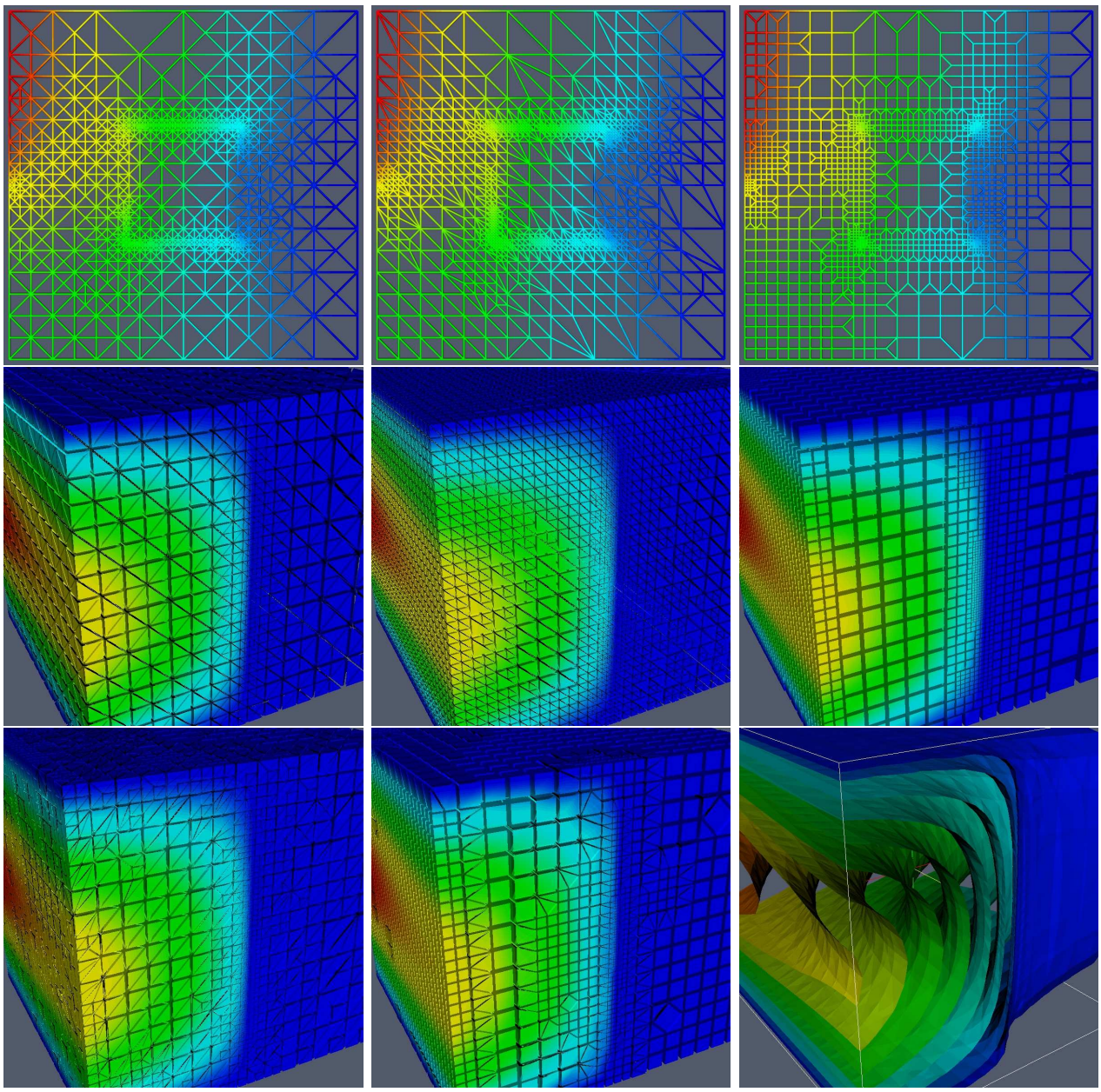


Figure 8.2: Adaptive solution of an elliptic model problem with P_1 conforming finite elements and residual based error estimator. Illustrates that adaptive finite element algorithm can be formulated independent of dimension, element type and refinement scheme. From top to bottom, left to right: Alberta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