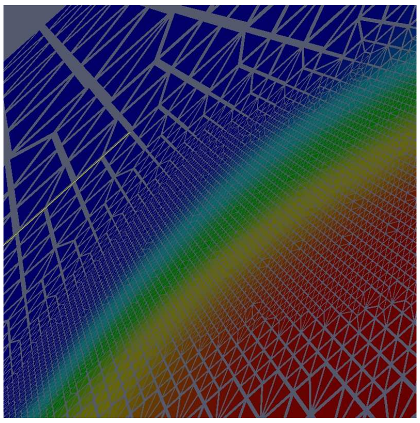
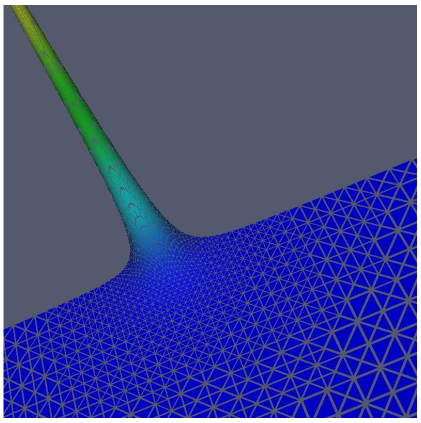
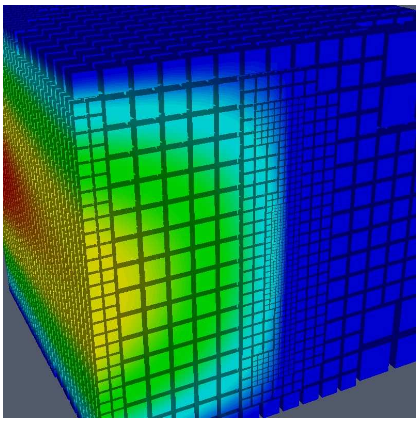
The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO

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

Version 1.2beta1, March 20, 2009

[*]Abteilung 'Simulation großer Systeme', Universität Stuttgart,
Universitätsstr. 38, D-70569 Stuttgart, Germany

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

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

[‡]Institut für Mathematik II,
Freie Universität Berlin, Arnimallee 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!

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

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)

```
// $Id: gettingstarted.cc 198 2008-01-23 16:12:41Z sander $

// Dune includes
#include"config.h"           // file constructed by ./configure script
#include<dune/grid/sgrid.hh> // load sgrid definition
#include<dune/grid/common/gridinfo.hh> // definition of gridinfo
#include <dune/common/mpihelper.hh> // include mpi helper class


int main(int argc, char **argv)
{
  // initialize MPI, finalize is done automatically on exit
  Dune::MPIHelper::instance(argc,argv);

  // start try/catch block to get error messages from dune
  try{
    // make a grid
    const int dim=3;
    typedef Dune::SGrid<dim,dim> GridType;
    Dune::FieldVector<int,dim> N(3);
    Dune::FieldVector<GridType::ctype,dim> L(-1.0);
    Dune::FieldVector<GridType::ctype,dim> H(1.0);
    GridType grid(N,L,H);

    // print some information about the grid
    Dune::gridinfo(grid);
  }
  catch (std::exception & e) {
    std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
    return 1;
  }
  catch (Dune::Exception & e) {
    std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
    return 1;
  }
  catch (...) {
    std::cout << "Unknown␣ERROR" << std::endl;
    return 1;
  }

  // done
  return 0;
}
```

This program is quite simple. It starts with some includes in lines 4-6. The file `config.h` has been produced by the `configure` script in the application's build system. It contains the current configuration and can be used to compile different versions of your code depending on the configuration selected. It is important that this file is included before any other **DUNE** header files. The next file `dune/grid/sgrid.hh` includes the headers for the `SGrid` class which provides a special implementation of the **DUNE** grid interface with a structured mesh of arbitrary dimension. Then `dune/grid/common/gridinfo.hh` loads the headers of some functions which print useful information about a grid.

Since the dimension will be used as a template parameter in many places below we define it as a constant in line number 18. The `SGrid` class template takes two template parameters which are the dimension of the grid and the dimension of the space where the grid is embedded in (its world dimension). If the world dimension is strictly greater than the grid dimension the surplus coordinates of each grid vertex are set to zero. For ease of writing we define in line 19 the type `GridType` using the selected value for the dimension. All identifiers of the **DUNE** framework are within the `Dune` namespace.

Lines 20-22 prepare the arguments for the construction of an `SGrid` object. These arguments use the class template `FieldVector<T,n>` which is a vector with `n` components of type `T`. You can either assign the same value to all components in the constructor (as is done here) or you could use `operator[]` to assign values to individual components. The variable `N` defines the number of cells or elements to be used in the respective dimension of the grid. `L` defines the coordinates of the lower left corner of the cube and `H` defines the coordinates of the upper right corner of the cube. Finally in line 23 we are now able to instantiate the `SGrid` object.

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

Listing 2 (Output of gettingstarted)

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

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

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

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

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

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

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

```
// $Id: traversal.cc 243 2009-03-10 07:05:52Z mnolte $

// C/C++ includes
#include<iostream>           // for standard I/O

// Dune includes
#include"config.h"           // file constructed by ./configure script
#include<dune/grid/sgrid.hh> // load sgrid definition
#include <dune/common/mpihelper.hh> // include mpi helper class


// example for a generic algorithm that traverses
// the entities of a given mesh in various ways
template<class G>
void traversal (G& grid)
{
  // first we extract the dimensions of the grid
  const int dim = G::dimension;

  // type used for coordinates in the grid
  // such a type is exported by every grid implementation
  typedef typename G::ctype ct;

  // Leaf Traversal
  std::cout << "***␣Traverse␣codim␣0␣leaves" << std::endl;

  // type of the GridView used for traversal
  // every grid exports a LeafGridView and a LevelGridView
  typedef typename G :: LeafGridView LeafGridView;

  // get the instance of the LeafGridView
  LeafGridView leafView = grid.leafView();

  // Get the iterator type
  // Note the use of the typename and template keywords
  typedef typename LeafGridView::template Codim<0>::Iterator ElementLeafIterator;

  // iterate through all entities of codim 0 at the leafs
  int count = 0;
  for (ElementLeafIterator it = leafView.template begin<0>();
       it!=leafView.template end<0>(); ++it)
    {
      Dune::GeometryType gt = it->type();
      std::cout << "visiting␣leaf␣" << gt
                << "␣with␣first␣vertex␣at␣" << it->geometry().corner(0)
                << std::endl;
      count++;
    }

  std::cout << "there␣are/is␣" << count << "␣leaf␣element(s)" << std::endl;

  // Leafwise traversal of codim dim
  std::cout << std::endl;
  std::cout << "***␣Traverse␣codim␣" << dim << "␣leaves" << std::endl;

  // Get the iterator type
  // Note the use of the typename and template keywords
  typedef typename LeafGridView :: template Codim<dim>
            :: Iterator VertexLeafIterator;

  // iterate through all entities of codim 0 on the given level
  count = 0;
  for (VertexLeafIterator it = leafView.template begin<dim>();
       it!=leafView.template end<dim>(); ++it)
    {
      Dune::GeometryType gt = it->type();
      std::cout << "visiting␣" << gt
                << "␣at␣" << it->geometry().corner(0)
                << std::endl;
      count++;
    }
  std::cout << "there␣are/is␣" << count << "␣leaf␣vertices(s)"
            << std::endl;

  // Levelwise traversal of codim 0
  std::cout << std::endl;
  std::cout << "***␣Traverse␣codim␣0␣level-wise" << std::endl;

  // type of the GridView used for traversal
  // every grid exports a LeafGridView and a LevelGridView
  typedef typename G :: LevelGridView LevelGridView;

  // Get the iterator type
  // Note the use of the typename and template keywords
  typedef typename LevelGridView :: template Codim<0>
            :: Iterator ElementLevelIterator;

  // iterate through all entities of codim 0 on the given level
  for (int level=0; level<=grid.maxLevel(); level++)
    {
      // get the instance of the LeafGridView
      LevelGridView levelView = grid.levelView(level);

      count = 0;
      for (ElementLevelIterator it = levelView.template begin<0>();
           it!=levelView.template end<0>(); ++it)
        {
          Dune::GeometryType gt = it->type();
          std::cout << "visiting␣" << gt
                    << "␣with␣first␣vertex␣at␣" << it->geometry().corner(0)
                    << std::endl;
          count++;
        }
      std::cout << "there␣are/is␣" << count << "␣element(s)␣on␣level␣"
                << level << std::endl;
      std::cout << std::endl;
    }
}


int main(int argc, char **argv)
{
```

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

The `main` function near the end of the listing is pretty similar to the previous one except that we use a 2d grid for the unit square that just consists of one cell. In line 127 this cell is refined once using the standard method of grid refinement of the implementation. Here, the cell is refined into four smaller cells. The main work is done in a call to the function `traversal` in line 130. This function is given in lines 14-108.

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

The algorithm should work in any dimension so we extract the grid's dimension in line 18. Next, each **DUNE** grid defines a type that it uses to represent positions. This type is extracted in line 22 for later use.

A grid is considered to be a container of "entities" which are abstractions for geometric objects like vertices, edges, quadrilaterals, tetrahedra, and so on. This is very similar to the standard template library (STL), see e. g. [9], which is part of any C++ system. A key difference is, however, that there is not just one type of entity but several. As in the STL the elements of any container can be accessed with iterators which are generalized pointers. Again, a **DUNE** grid knows several different iterators which provide access to the different kinds of entities and which also provide different patterns of access.

As we usually do not want to use the entire hierarchy of the grid, we first define a view on that part of the grid we are interested in. This can be a level or the leaf part of the grid. In line 29 a type for a `GridView` on the leaf grid is defined.

Line 36 extracts the type of an iterator from this view class. `Codim` is a `struct` within the grid class that takes an integer template parameter specifying the codimension over which to iterate. Within the `Codim` structure the type `Iterator` is defined. Since we specified codimension 0 this iterator is used to iterate over the elements which are not refined any further, i.e. which are the leaves of the refinement trees.

The `for`-loop in line 40 now visits every such element. The `begin` and `end` on the `LeafGridView` class deliver the first leaf element and one past the last leaf element. Note that the `template` keyword must be used and template parameters are passed explicitely. Within the loop body in lines 42-48 the iterator `it` acts like a pointer to an entity of dimension `dim` and codimension 0. The exact type would be `typename G::template Codim<0>::Entity` just to mention it.

An important part of an entity is its geometrical shape and position. All geometrical information is factored out into a sub-object that can be accessed via the `geometry()` method. The geometry object is in general a mapping from a d-dimensional polyhedral reference element to w dimensional space. Here we have $d =$ `G::dimension` and $w =$ `G::dimensionworld`. This mapping is also called the "local to global" mapping. The corresponding reference element has a certain type which is extracted in line 43. Since the reference elements are polyhedra they consist of a finite number of corners. The images of the corners under the local to global map can be accessed via the `corner(int n)` method. Line 44 prints the geometry type and the position of the first corner of the element. Then line 47 just counts the number of elements visited.

Suppose now that we wanted to iterate over the vertices of the leaf grid instead of the elements. Now vertices have the codimension `dim` in a `dim`-dimensional grid and a corresponding iterator is provided by each grid class. It is extracted in line 59 for later use. The `for`-loop starting in line 63 is very similar to the first one except that it now uses the `VertexLeafIterator`. As you can see the different entities can be accessed with the same methods. We will see later that codimensions 0 and `dim` are specializations with an extended interface compared to all other codimensions. You can also access the codimensions between 0 and `dim`. However, currently not all implementations of the grid interface support these intermediate codimensions (though this does not restrict the implementation of finite element methods with degrees of freedom associated to, say, faces).

Finally, we show in lines 81-107 how the hierarchic structure of the mesh can be accessed. To that end a `LevelGridView` is used. It provides via an `Iterator` access to all entities of a given codimension (here 0) on a given grid level. The coarsest grid level (the initial macro grid) has number zero and the number of the finest grid level is returned by the `maxLevel()` method of the grid. The methods `begin()` and `end()` on the view deliver iterators to the first and one-past-the-last entity of a given grid level supplied as an integer argument to these methods.

The following listing shows the output of the program.

Listing 4 (Output of traversal)

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

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

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

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

Remark 2.3 Define the end iterator for efficiency.

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

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

3 The DUNE grid interface

3.1 Grid definition

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

Reference elements

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

Dimension and world dimension

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

Faces, entities and codimension

The intersection of a d-dimensional convex polytope (in d-dimensional space) with a tangent plane is called a face (note that there are faces of dimensionality $0, \ldots, d-1$). 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.

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

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

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

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

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

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

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

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

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

3.3 Propagation of type information

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

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

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

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

4 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)

```
#ifndef UNITCUBE_HH
#define UNITCUBE_HH

#include <dune/common/exceptions.hh>
#include <dune/common/fvector.hh>

// default implementation for any template parameter
template<typename T, int variant>
class UnitCube
{
public:
  typedef T GridType;

  // constructor throwing exception
  UnitCube ()
  {
    DUNE_THROW(Dune::Exception,"no␣specialization␣for␣this␣grid␣available");
  }

  T& grid ()
  {
    return grid_;
  }

private:
  // the constructed grid object
  T grid_;
};

// include basic unitcube using GridFactory concept
#include "basicunitcube.hh"

// include specializations
#include"unitcube_onedgrid.hh"
#include"unitcube_sgrid.hh"
```

```
#include"unitcube_yaspgrid.hh"
#include"unitcube_uggrid.hh"
#include"unitcube_albertagrid.hh"
#include"unitcube_alugrid.hh"

#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)

```
#ifndef UNITCUBE_ONEDGRID_HH
#define UNITCUBE_ONEDGRID_HH

#include<dune/grid/onedgrid.hh>

// OneDGrid specialization
template<>
class UnitCube<Dune::OneDGrid,1>
{
public:
  typedef Dune::OneDGrid GridType;

  UnitCube () : grid_(1,0.0,1.0)
  {}

  Dune::OneDGrid& grid ()
  {
    return grid_;
  }

private:
  Dune::OneDGrid grid_;
};

#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)

```
#ifndef UNITCUBE_SGRID_HH
#define UNITCUBE_SGRID_HH

#include<dune/grid/sgrid.hh>

// SGrid specialization
template<int dim>
class UnitCube<Dune::SGrid<dim,dim>,1>
{
public:
  typedef Dune::SGrid<dim,dim> GridType;

  Dune::SGrid<dim,dim>& grid ()
  {
    return grid_;
  }

private:
  Dune::SGrid<dim,dim> grid_;
};

#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)

```
#ifndef UNITCUBE_YASPGRID_HH
#define UNITCUBE_YASPGRID_HH

#include <dune/grid/yaspgrid.hh>

// YaspGrid specialization
template<int dim, int size>
class UnitCube<Dune::YaspGrid<dim>,size>
{
public:
  typedef Dune::YaspGrid<dim> GridType;

  UnitCube () : Len(1.0), s(size), p(false),
#if HAVE_MPI
  grid_(MPI_COMM_WORLD,Len,s,p,1)
#else
  grid_(Len,s,p,1)
#endif
  {  }

  Dune::YaspGrid<dim>& grid ()
  {
    return grid_;
  }

private:
  Dune::FieldVector<double,dim> Len;
  Dune::FieldVector<int,dim> s;
  Dune::FieldVector<bool,dim> p;
  Dune::YaspGrid<dim> grid_;
};

#endif
```

GridFactory

The file `basicunitcube.hh` provides an interface class for the set-up of a unit cube in two or three dimensions with simplicial or cubic elements with help of the dune-grid class `GridFactory`. This class hides grid specific methods for insertion of vertices and elements in the macro grid. Specializations of the `GridFactory` exist for `UGGrid`, `AlbertaGrid` and the three dimensional `ALUGrid` objects.

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

```
#ifndef  BASICUNITCUBE_HH
#define  BASICUNITCUBE_HH

#include <dune/grid/common/gridfactory.hh>

// declaration of a basic unit cube that uses the GridFactory
template< int dim >
class BasicUnitCube;

// unit cube in two dimensions with 2 variants: triangle and rectangle elements
template<>
class BasicUnitCube< 2 >
{
protected:
  template< class Grid >
  static void insertVertices ( Dune::GridFactory< Grid > &factory )
  {
    Dune::FieldVector<double,2> pos;

    pos[0] = 0;  pos[1] = 0;
    factory.insertVertex(pos);

    pos[0] = 1;  pos[1] = 0;
    factory.insertVertex(pos);

    pos[0] = 0;  pos[1] = 1;
    factory.insertVertex(pos);

    pos[0] = 1;  pos[1] = 1;
    factory.insertVertex(pos);
  }

  template< class Grid >
  static void insertSimplices ( Dune::GridFactory< Grid > &factory )
  {
    const Dune::GeometryType type( Dune::GeometryType::simplex, 2 );
    std::vector< unsigned int > cornerIDs( 3 );

    cornerIDs[0] = 0;  cornerIDs[1] = 1;  cornerIDs[2] = 2;
    factory.insertElement( type, cornerIDs );

    cornerIDs[0] = 2;  cornerIDs[1] = 1;  cornerIDs[2] = 3;
    factory.insertElement( type, cornerIDs );
  }

  template< class Grid >
  static void insertCubes ( Dune::GridFactory< Grid > &factory )
  {
    const Dune::GeometryType type( Dune::GeometryType::cube, 2 );
    std::vector< unsigned int > cornerIDs( 4 );
    for( int i = 0; i < 4; ++i )
      cornerIDs[ i ] = i;
    factory.insertElement( type, cornerIDs );
  }
};

// unit cube in 3 dimensions with two variants: tetraheda and hexahedra
template<>
```

```
class BasicUnitCube< 3 >
{
protected:
  template< class Grid >
  static void insertVertices ( Dune::GridFactory< Grid > &factory )
  {
    Dune::FieldVector< double, 3 > pos;

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

  template< class Grid >
  static void insertSimplices ( Dune::GridFactory< Grid > &factory )
  {
    const Dune::GeometryType type( Dune::GeometryType::simplex, 3 );
    std::vector< unsigned int > cornerIDs( 4 );

    cornerIDs[0] = 0;  cornerIDs[1] = 1;  cornerIDs[2] = 2;  cornerIDs[3] = 4;
    factory.insertElement( type, cornerIDs );

    cornerIDs[0] = 1;  cornerIDs[1] = 3;  cornerIDs[2] = 2;  cornerIDs[3] = 7;
    factory.insertElement( type, cornerIDs );

    cornerIDs[0] = 1;  cornerIDs[1] = 7;  cornerIDs[2] = 2;  cornerIDs[3] = 4;
    factory.insertElement( type, cornerIDs );

    cornerIDs[0] = 1;  cornerIDs[1] = 7;  cornerIDs[2] = 4;  cornerIDs[3] = 5;
    factory.insertElement( type, cornerIDs );

    cornerIDs[0] = 4;  cornerIDs[1] = 7;  cornerIDs[2] = 2;  cornerIDs[3] = 6;
    factory.insertElement( type, cornerIDs );
  }

  template< class Grid >
  static void insertCubes ( Dune::GridFactory< Grid > &factory )
  {
    const Dune::GeometryType type( Dune::GeometryType::cube, 3 );
    std::vector< unsigned int > cornerIDs( 8 );
    for( int i = 0; i < 8; ++i )
      cornerIDs[ i ] = i;
    factory.insertElement( type, cornerIDs );
  }
};

#endif  /*BASICUNITCUBE_HH*/
```

UGGrid

The following listing shows how to create `UGGrid` objects based on the `BasicUnitCube`. 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 10 (File dune-grid-howto/unitcube_uggrid.hh)

```
#ifndef UNITCUBE_UGGRID_HH
#define UNITCUBE_UGGRID_HH

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

template< int dim, int variant >
class UnitCube< Dune::UGGrid< dim >, variant >
: public BasicUnitCube< dim >
{
public:
  typedef Dune::UGGrid< dim > GridType;

private:
  GridType* grid_;

public:
  UnitCube ()
  {
    Dune::GridFactory< GridType > factory;
    BasicUnitCube< dim >::insertVertices( factory );
    if( variant == 1 )
      BasicUnitCube< dim >::insertCubes( factory );
    else if( variant == 2 )
      BasicUnitCube< dim >::insertSimplices( factory );
    else
      DUNE_THROW( Dune::NotImplemented, "Variant␣"
                  << variant << "␣of␣UG␣unit␣cube␣not␣implemented." );
    grid_ = factory.createGrid();
  }

  ~UnitCube ()
  {
    delete grid_;
  }

  GridType &grid ()
  {
    return *grid_;
  }
};

#endif // #if HAVE_UG

#endif
```

AlbertaGrid

The following listing contains specializations of the `UnitCube` template for Alberta in two and three dimensions. When using Alberta versions less than 2.0 the **DUNE** framework has to be configured with a dimension (`--with-alberta-dim=2`, `--with-alberta-world-dim=2`) and only this dimension can then be used. The dimension from the configure run is available in the macro `ALBERTA_DIM` and `ALBERTA_WORLD_DIM` in the file `config.h` (see next section). The `variant` parameter must be 1. Like in the `UGGrid` implementation the grid factory concept is used as the `UnitCube` specializations are based on the `BasicUnitCube`.

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

```
#ifndef UNITCUBE_ALBERTAGRID_HH
#define UNITCUBE_ALBERTAGRID_HH
```

```

#if HAVE_ALBERTA
#include <dune/grid/albertagrid.hh>
#include <dune/grid/albertagrid/gridfactory.hh>

template< int dim >
class UnitCube< Dune::AlbertaGrid< dim, dim >, 1 >
: public BasicUnitCube< dim >
{
public:
  typedef Dune::AlbertaGrid< dim, dim > GridType;

private:
  GridType *grid_;

public:
  UnitCube ()
  {
    Dune::GridFactory< GridType > factory;
    BasicUnitCube< dim >::insertVertices( factory );
    BasicUnitCube< dim >::insertSimplices( factory );
    grid_ = factory.createGrid( "UnitCube", true );
  }

  ~UnitCube ()
  {
    Dune::GridFactory< GridType >::destroyGrid( grid_ );
  }

  GridType &grid ()
  {
    return *grid_;
  }
};

#endif // #if HAVE_ALBERTA

#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. For three dimensional versions the grid objects are set up with help of the `BasicUnitCube` class.

Listing 12 (File dune-grid-howto/unitcube_alugrid.hh)

```
#ifndef UNITCUBE_ALU3DGRID_HH
#define UNITCUBE_ALU3DGRID_HH

#if HAVE_ALUGRID
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/3d/alu3dgridfactory.hh>

// ALU3dGrid and ALU2dGrid simplex specialization.
// Note: element type determined by type
template<>
class UnitCube<Dune::ALUSimplexGrid<3,3>,1>
: public BasicUnitCube< 3 >
```

```
{
public:
  typedef Dune::ALUSimplexGrid<3,3> GridType;

private:
  GridType * grid_;

public:
  UnitCube ()
  {
    Dune::GridFactory< GridType > factory;
    BasicUnitCube< 3 >::insertVertices( factory );
    BasicUnitCube< 3 >::insertSimplices( factory );
    grid_ = factory.createGrid( );
  }

  ~UnitCube()
  {
    delete grid_;
  }

  GridType &grid ()
  {
    return *grid_;
  }
};

// ALU2SimplexGrid 2d specialization. Note: element type determined by type
template<>
class UnitCube<Dune::ALUSimplexGrid<2,2>,1>
{
public:
  typedef Dune::ALUSimplexGrid<2,2> GridType;

  UnitCube () : filename("grids/2dsimplex.alu"), grid_(filename.c_str())
  {}

  GridType& grid ()
  {
    return grid_;
  }

private:
  std::string filename;
  GridType grid_;
};

// ALU3dGrid hexahedra specialization. Note: element type determined by type
template<>
class UnitCube<Dune::ALUCubeGrid<3,3>,1>
: public BasicUnitCube< 3 >
{
public:
  typedef Dune::ALUCubeGrid<3,3> GridType;

private:
  GridType * grid_;

public:
  UnitCube ()
  {
    Dune::GridFactory< GridType > factory;
    BasicUnitCube< 3 >::insertVertices( factory );
    BasicUnitCube< 3 >::insertCubes( factory );
    grid_ = factory.createGrid( );
  }

  ~UnitCube()
  {
    delete grid_;
  }

  GridType &grid ()
  {
    return *grid_;
  }
};
#endif

#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"
```

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 its own macro grid format, the Dune Grid Format. A detailed description of the DGF and how to use it can be found on the homepage of Dune under the documentation section (see DuneGridFormatParser[1]).

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

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

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

[1] http://www.dune-project.org/doc/doxygen/dune-grid-html/group__DuneGridFormatParser.html

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

```
// a smooth function
template<typename ct, int dim>
class Exp {
public:
  Exp () {midpoint = 0.5;}
  double operator() (const Dune::FieldVector<ct,dim>& x) const
  {
    Dune::FieldVector<ct,dim> y(x);
    y -= midpoint;
    return exp(-3.234*(y*y));
  }
private:
  Dune::FieldVector<ct,dim> midpoint;
};

// a function with a local feature
template<typename ct, int dim>
class Needle {
public:
  Needle ()
  {
    midpoint = 0.5;
    midpoint[dim-1] = 1;
  }
  double operator() (const Dune::FieldVector<ct,dim>& x) const
  {
    Dune::FieldVector<ct,dim> y(x);
    y -= midpoint;
    return 1.0/(1E-4+y*y);
  }
private:
  Dune::FieldVector<ct,dim> midpoint;
};
```

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

```
#ifndef DUNE_INTEGRATE_ENTITY_HH
#define DUNE_INTEGRATE_ENTITY_HH

#include<dune/common/exceptions.hh>
#include<dune/grid/common/quadraturerules.hh>

//! compute integral of function over entity with given order
template<class Iterator, class Functor>
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 bisected in every coordinate direction, the function to be integrated is sufficiently smooth and the order of the quadrature rule is $p+1$, then the error should be reduced by a factor of $(1/2)^p$ after each mesh refinement.

Listing 15 (dune-grid-howto/integration.cc)

```cpp
// $Id: integration.cc 243 2009-03-10 07:05:52Z mnolte $

// Dune includes
#include"config.h"           // file constructed by ./configure script
#include<dune/grid/sgrid.hh> // load sgrid definition
#include <dune/common/mpihelper.hh> // include mpi helper class

// checks for defined gridtype and includes appropriate dgfparser implementation
#include <dune/grid/io/file/dgfparser/dgfgridtype.hh>

#include"functors.hh"
#include"integrateentity.hh"

//! uniform refinement test
template<class Grid>
void uniformintegration (Grid& grid)
{
  // function to integrate
  Exp<typename Grid::ctype,Grid::dimension> f;

  // get GridView on leaf grid - type
  typedef typename Grid :: LeafGridView GridView;

  // get GridView instance
  GridView gridView = grid.leafView();

  // get iterator type
  typedef typename GridView :: template Codim<0> :: Iterator LeafIterator;

  // loop over grid sequence
  double oldvalue=1E100;
  for (int k=0; k<10; k++)
    {
      // compute integral with some order
      double value = 0.0;
      LeafIterator eendit = gridView.template end<0>();
      for (LeafIterator it = gridView.template begin<0>(); it!=eendit; ++it)
              value += integrateentity(it,f,1);

      // print result and error estimate
      std::cout << "elements="
                << std::setw(8) << std::right
                << grid.size(0)
                << "␣integral="
                << std::scientific << std::setprecision(12)
                << value
                << "␣error=" << std::abs(value-oldvalue)
                << std::endl;

      // save value of integral
      oldvalue=value;

      // refine all elements
      grid.globalRefine(1);
    }
}

int main(int argc, char **argv)
{
  // initialize MPI, finalize is done automatically on exit
  Dune::MPIHelper::instance(argc,argv);

63   // start try/catch block to get error messages from dune
64   try {
65     using namespace Dune;
66
67     // use unitcube from grids
68     std::stringstream dgfFileName;
69     dgfFileName << "grids/unitcube" << GridType :: dimension << ".dgf";
70
71     // create grid pointer, GridType is defined by gridtype.hh
72     GridPtr<GridType> gridPtr( dgfFileName.str() );
73
74     // integrate and compute error with extrapolation
75     uniformintegration( *gridPtr );
76   }
77   catch (std::exception & e) {
78     std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
79     return 1;
80   }
81   catch (Dune::Exception & e) {
82     std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
83     return 1;
84   }
85   catch (...) {
86     std::cout << "Unknown␣ERROR" << std::endl;
87     return 1;
88   }
89
90   // done
91   return 0;
92 }
```

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

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

6.2 Visualization of discrete functions

Let us use mappers to evaluate a function $f : \Omega \to \mathbb{R}$ 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 16 (File dune-grid-howto/elementdata.hh)

```
#include<dune/grid/common/referenceelements.hh>
#include<dune/grid/common/mcmgmapper.hh>
#include<dune/grid/io/file/vtk/vtkwriter.hh>
#if HAVE_GRAPE
#include<dune/grid/io/visual/grapedatadisplay.hh>
#endif

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

// demonstrate attaching data to elements
template<class G, class F>
void elementdata (const G& grid, const F& f)
{
  // the usual stuff
  const int dim = G::dimension;
  const int dimworld = G::dimensionworld;
  typedef typename G::ctype ct;
  typedef typename G::LeafGridView GridView;
  typedef typename GridView::template Codim<0>::Iterator ElementLeafIterator;

  // get grid view on leaf part
  GridView gridView = grid.leafView();

  // make a mapper for codim 0 entities in the leaf grid
  Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
      mapper(grid);

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

  // iterate through all entities of codim 0 at the leafs
  for (ElementLeafIterator it = gridView.template begin<0>();
       it!=gridView.template end<0>(); ++it)
    {
      // cell geometry type
      Dune::GeometryType gt = it->type();

      // cell center in reference element
      const Dune::FieldVector<ct,dim>&
        local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);

      // get global coordinate of cell center
      Dune::FieldVector<ct,dimworld> global = it->geometry().global(local);

      // evaluate functor and store value
      c[mapper.map(*it)] = f(global);
    }

  // generate a VTK file
  // Dune::LeafP0Function<G,double> cc(grid,c);
  Dune::VTKWriter<typename G::LeafGridView> vtkwriter(gridView);
  vtkwriter.addCellData(c,"data");
  vtkwriter.write("elementdata",Dune::VTKOptions::binaryappended);

  // online visualization with Grape
#if HAVE_GRAPE
  {
    const int polynomialOrder = 0; // we piecewise constant data
    const int dimRange = 1; // we have scalar data here
    // create instance of data display
    Dune::GrapeDataDisplay<G> grape(grid);
    // display data
    grape.displayVector("concentration", // name of data that appears in grape
                        c,  // data vector
                        gridView.indexSet(), // used index set
                        polynomialOrder, // polynomial order of data
                        dimRange); // dimRange of data
  }
#endif
}
```

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

The data vector is allocated in line 38. 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 41-56 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 55 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 60-62 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 65-78 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 17 (File dune-grid-howto/vertexdata.hh)

```cpp
#include <dune/grid/common/referenceelements.hh>
#include <dune/grid/common/mcmgmapper.hh>
#include <dune/grid/io/file/vtk/vtkwriter.hh>
#if HAVE_GRAPE
#include <dune/grid/io/visual/grapedatadisplay.hh>
#endif

//! Parameter for mapper class
template<int dim>
struct P1Layout
{
  bool contains (Dune::GeometryType gt)
  {
    if (gt.dim()==0) return true;
    return false;
  }
};

// demonstrate attaching data to elements
template<class G, class F>
void vertexdata (const G& grid, const F& f)
{
  // get dimension and coordinate type from Grid
  const int dim = G::dimension;
  typedef typename G::ctype ct;
  typedef typename G::LeafGridView GridView;
  // dertermine type of LeafIterator for codimension = dimension
  typedef typename GridView::template Codim<dim>::Iterator VertexLeafIterator;

  // get grid view on the leaf part
  GridView gridView = grid.leafView();

  // make a mapper for codim 0 entities in the leaf grid
  Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P1Layout>
    mapper(grid);

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

  // iterate through all entities of codim 0 at the leafs
  for (VertexLeafIterator it = gridView.template begin<dim>();
       it!=gridView.template end<dim>(); ++it)
    {
      // evaluate functor and store value
            c[mapper.map(*it)] = f(it->geometry().corner(0));
    }

  // generate a VTK file
  //   Dune::LeafP1Function<G,double> cc(grid,c);
  Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafView());
  vtkwriter.addVertexData(c,"data");
  vtkwriter.write("vertexdata",Dune::VTKOptions::binaryappended);

  // online visualization with Grape
#if HAVE_GRAPE
  {
    const int polynomialOrder = 1; // we piecewise linear data
    const int dimRange = 1; // we have scalar data here
    // create instance of data display
    Dune::GrapeDataDisplay<G> grape(grid);
    // display data
    grape.displayVector("concentration", // name of data that appears in grape
                        c,  // data vector
                        gridView.indexSet(), // used index set
                        polynomialOrder, // polynomial order of data
                        dimRange); // dimRange of data
  }
#endif
}
```

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

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

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

Listing 18 (File dune-grid-howto/visualization.cc)

```
// $Id: visualization.cc 247 2009-03-18 09:47:23Z sander $

#include"config.h"
#include<iostream>
#include<iomanip>
```

```
#include<stdio.h>
#include <dune/common/mpihelper.hh> // include mpi helper class


#include"elementdata.hh"
#include"vertexdata.hh"
#include"functors.hh"
#include"unitcube.hh"


#ifdef GRIDDIM
const int dimGrid = GRIDDIM;
#else
const int dimGrid = 2;
#endif


//! supply functor
template<class Grid>
void dowork ( Grid &grid, int refSteps = 5 )
{
  // make function object
  Exp<typename Grid::ctype,Grid::dimension> f;

  // refine the grid
  grid.globalRefine( refSteps );

  // call the visualization functions
  elementdata(grid,f);
  vertexdata(grid,f);
}

int main(int argc, char **argv)
{
  // initialize MPI, finalize is done automatically on exit
  Dune::MPIHelper::instance(argc,argv);

  // start try/catch block to get error messages from dune
  try
  {
    /*
    UnitCube<Dune::OneDGrid,1> uc0;
    UnitCube<Dune::YaspGrid<dimGrid>,1> uc1;

#if HAVE_UG
    UnitCube< Dune::UGGrid< dimGrid >, 2 > uc2;
    dowork( uc2.grid(), 3 );
#endif

#if HAVE_ALBERTA
    {
      UnitCube< Dune::AlbertaGrid< dimGrid, dimGrid >, 1 > unitcube;
      // note: The 3d cube cannot be bisected recursively
      dowork( unitcube.grid(), (dimGrid < 3 ? 6 : 0) );
    }
#endif // #if HAVE_ALBERTA
    */

    UnitCube< Dune::SGrid< dimGrid, dimGrid >, 1 > uc4;
    dowork( uc4.grid(), 3 );

#if HAVE_ALUGRID
    UnitCube< Dune::ALUSimplexGrid< dimGrid, dimGrid > , 1 > uc5;
    dowork( uc5.grid(), 3 );

#if GRIDDIM == 3
    UnitCube< Dune::ALUCubeGrid< dimGrid, dimGrid > , 1 > uc6;
    dowork( uc6.grid(), 3 );
#endif // #if GRIDDIM == 3
#endif // #if HAVE_ALUGRID
  }
  catch (std::exception & e) {
    std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
    return 1;
  }
  catch (Dune::Exception & e) {
    std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
    return 1;
  }
  catch (...) {
    std::cout << "Unknown␣ERROR" << std::endl;
    return 1;
  }

  // done
  return 0;
}
```

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}):

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

Using integration by parts we arrive at

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

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}:

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

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

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

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

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

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

```
// the initial condition c0
template<int dimworld, class ct>
double c0 (const Dune::FieldVector<ct,dimworld>& x)
{
  Dune::FieldVector<ct,dimworld> y(0.25);
  y -= x;
  if (y.two_norm()<0.125)
    return 1.0;
  else
    return 0.0;
}

// the boundary condition b on inflow boundary
template<int dimworld, class ct>
double b (const Dune::FieldVector<ct,dimworld>& x, double t)
{
  return 0.0;
}

// the vector field u is returned in r
template<int dimworld, class ct>
Dune::FieldVector<double,dimworld> u (const Dune::FieldVector<ct,dimworld>& x, double t)
{
  Dune::FieldVector<double,dimworld> r(0.5);
  r[0] = 1.0;
  return r;
}
```

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

Listing 20 (File dune-grid-howto/initialize.hh)

```
#include<dune/grid/common/referenceelements.hh>

//! initialize the vector of unknowns with initial value
template<class G, class M, class V>
void initialize (const G& grid, const M& mapper, V& c)
{
  // first we extract the dimensions of the grid
  const int dim = G::dimension;
  const int dimworld = G::dimensionworld;

  // type used for coordinates in the grid
  typedef typename G::ctype ct;

  // type of grid view on leaf part
  typedef typename G::LeafGridView GridView;

  // leaf iterator type
  typedef typename GridView::template Codim<0>::Iterator LeafIterator;

  // get grid view on leaf part
  GridView gridView = grid.leafView();

  // iterate through leaf grid an evaluate c0 at cell center
  LeafIterator endit = gridView.template end<0>();
  for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
    {
      // get geometry type
      Dune::GeometryType gt = it->type();

      // get cell center in reference element
      const Dune::FieldVector<ct,dim>&
        local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);

      // get global coordinate of cell center
      Dune::FieldVector<ct,dimworld> global =
        it->geometry().global(local);

      // initialize cell concentration
      c[mapper.map(*it)] = c0(global);
    }
}
```

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

```
#include<dune/grid/common/referenceelements.hh>

template<class G, class M, class V>
void evolve (const G& grid, const M& mapper, V& c, double t, double& dt)
{
  // first we extract the dimensions of the grid
  const int dim = G::dimension;
  const int dimworld = G::dimensionworld;

  // type used for coordinates in the grid
  typedef typename G::ctype ct;

  // type of grid view on leaf part
  typedef typename G::LeafGridView GridView;

  // element iterator type
  typedef typename GridView::template Codim<0>::Iterator LeafIterator;

  // intersection iterator type
  typedef typename GridView::IntersectionIterator IntersectionIterator;

  // entity pointer type
  typedef typename G::template Codim<0>::EntityPointer EntityPointer;

  // get grid view on leaf part
  GridView gridView = grid.leafView();

  // allocate a temporary vector for the update
  V update(c.size());
  for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;

  // initialize dt very large
```

```
dt = 1E100;

// compute update vector and optimum dt in one grid traversal
LeafIterator endit = gridView.template end<0>();
for (LeafIterator it = gridView.template begin<0>(); it!=endit; ++it)
  {
    // cell geometry type
    Dune::GeometryType gt = it->type();

    // cell center in reference element
    const Dune::FieldVector<ct,dim>&
      local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);

    // cell center in global coordinates
    Dune::FieldVector<ct,dimworld>
      global = it->geometry().global(local);

    // cell volume, assume linear map here
    double volume = it->geometry().integrationElement(local)
      *Dune::ReferenceElements<ct,dim>::general(gt).volume();

    // cell index
    int indexi = mapper.map(*it);

    // variable to compute sum of positive factors
    double sumfactor = 0.0;

    // run through all intersections with neighbors and boundary
    IntersectionIterator isend = gridView.iend(*it);
    for (IntersectionIterator is = gridView.ibegin(*it); is!=isend; ++is)
      {
        // get geometry type of face
        Dune::GeometryType gtf = is->intersectionSelfLocal().type();

        // center in face's reference element
        const Dune::FieldVector<ct,dim-1>&
          facelocal = Dune::ReferenceElements<ct,dim-1>::general(gtf).position(0,0);

        // get normal vector scaled with volume
        Dune::FieldVector<ct,dimworld> integrationOuterNormal
          = is->integrationOuterNormal(facelocal);
        integrationOuterNormal
          *= Dune::ReferenceElements<ct,dim-1>::general(gtf).volume();

        // center of face in global coordinates
        Dune::FieldVector<ct,dimworld>
          faceglobal = is->intersectionGlobal().global(facelocal);

        // evaluate velocity at face center
        Dune::FieldVector<double,dim> velocity = u(faceglobal,t);

        // compute factor occuring in flux formula
        double factor = velocity*integrationOuterNormal/volume;

        // for time step calculation
        if (factor>=0) sumfactor += factor;

        // handle interior face
        if (is->neighbor()) // "correct" version
          {
            // access neighbor
            EntityPointer outside = is->outside();
            int indexj = mapper.map(*outside);
```

```
 96
 97               // compute flux from one side only
 98               // this should become easier with the new IntersectionIterator functionality!
 99               if ( it->level()>outside->level() ||
100                    (it->level()==outside->level() && indexi<indexj) )
101                 {
102                   // compute factor in neighbor
103                   Dune::GeometryType nbgt = outside->type();
104                   const Dune::FieldVector<ct,dim>&
105                     nblocal = Dune::ReferenceElements<ct,dim>::general(nbgt).position(0,0);
106                   double nbvolume = outside->geometry().integrationElement(nblocal)
107                     *Dune::ReferenceElements<ct,dim>::general(nbgt).volume();
108                   double nbfactor = velocity*integrationOuterNormal/nbvolume;
109
110                   if (factor<0) // inflow
111                     {
112                       update[indexi] -= c[indexj]*factor;
113                       update[indexj] += c[indexj]*nbfactor;
114                     }
115                   else // outflow
116                     {
117                       update[indexi] -= c[indexi]*factor;
118                       update[indexj] += c[indexi]*nbfactor;
119                     }
120                 }
121             }
122
123           // handle boundary face
124           if (is->boundary())
125                   {
126             if (factor<0) // inflow, apply boundary condition
127               update[indexi] -= b(faceglobal,t)*factor;
128             else // outflow
129               update[indexi] -= c[indexi]*factor;
130                   }
131         } // end all intersections
132
133       // compute dt restriction
134       dt = std::min(dt,1.0/sumfactor);
135
136     } // end grid traversal
137
138   // scale dt with safety factor
139   dt *= 0.99;
140
141   // update the concentration vector
142   for (unsigned int i=0; i<c.size(); ++i)
143     c[i] += dt*update[i];
144
145   return;
146 }
```

Lines 36-136 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 29, where we assume that `V` is a container with copy constructor and size method.

The computation of the fluxes is done in lines 61-131. An `IntersectionIterator` is used to access all intersections γ_{ij} of a leaf element ω_i. For a full documentation on the `Intersection` class, we refer to the doxygen module page on Intersections[1] An Intersection is with another element ω_j if the

[1]`http://www.dune-project.org/doc/doxygen/dune-grid-html/group__GIIntersectionIterator.html`

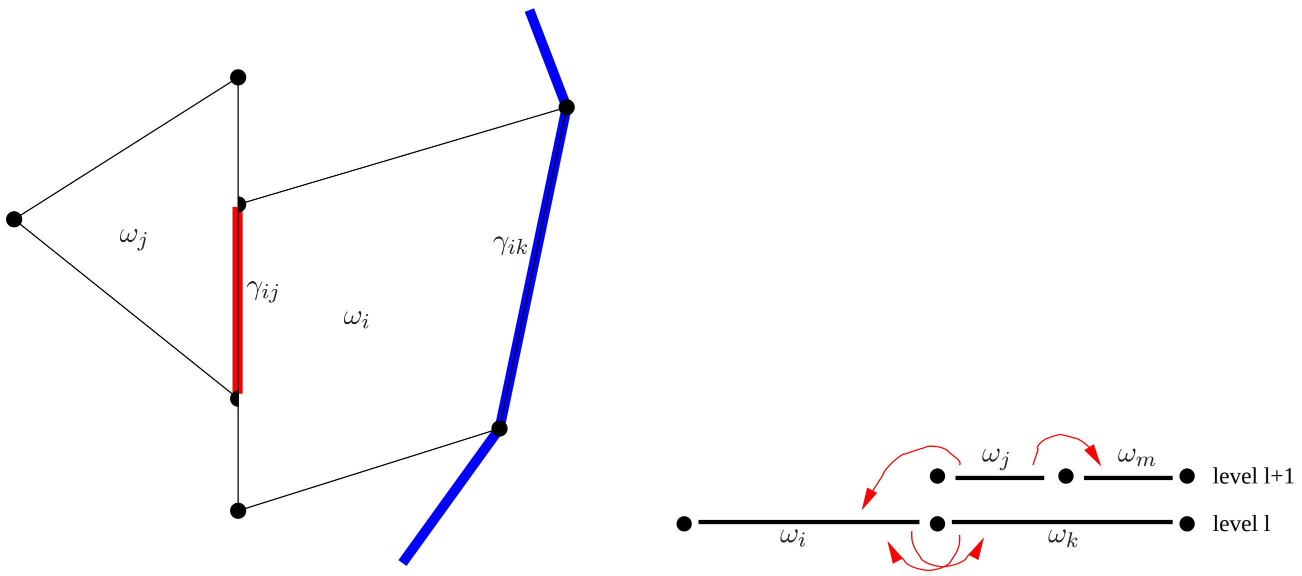


Figure 6.1: Left: inntersection with other elements and the boundary, right: intersections in the case of locally refined grids.

`neighbor()` method of the iterator returns true (line 91) or with the external boundary if `boundary()` returns true (line 124), 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 134 where the minimum over all cells is taken. Then, line 139 multiplies the optimum Δt^n with a safety factor to avoid any instability due to round-off errors.

Finally, line 143 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 22 (File dune-grid-howto/vtkout.hh)

```
#include <dune/grid/io/file/vtk/vtkwriter.hh>
#include <stdio.h>

template<class G, class V>
void vtkout (const G& grid, const V& c, const char* name, int k, double time=0.0, int rank=0)
{
  Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafView());
  char fname[128];
  char sername[128];
  sprintf(fname,"%s-%05d",name,k);
  sprintf(sername,"%s.series",name);
  vtkwriter.addCellData(c,"celldata");
  vtkwriter.write(fname,Dune::VTKOptions::ascii);

  if ( rank == 0)
  {
    std::ofstream serstream(sername, (k==0 ? std::ios_base::out : std::ios_base::app));
    serstream << k << "␣" << fname << ".vtu␣" << time << std::endl;
    serstream.close();
  }
}
```

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

Finally, the main program:

Listing 23 (File dune-grid-howto/finitevolume.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

// checks for defined gridtype and inlcudes appropriate dgfparser implementation
#include <dune/grid/io/file/dgfparser/dgfgridtype.hh>

#include"vtkout.hh"
#include"unitcube.hh"
#include"transportproblem2.hh"
#include"initialize.hh"
#include"evolve.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 timeloop (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,"concentration",0,0.0);

  // now do the time steps
  double t=0,dt;
  int k=0;
  const double saveInterval = 0.1;
  double saveStep = 0.1;
  int counter = 1;

  while (t<tend)
    {
    // augment time step counter
    ++k;

    // apply finite volume scheme
    evolve(grid,mapper,c,t,dt);

    // augment time
    t += dt;

    // check if data should be written
    if (t >= saveStep)
    {
      // write data
        vtkout(grid,c,"concentration",counter,t);

      // increase counter and saveStep for next interval
      saveStep += saveInterval;
      ++counter;
    }

    // print info about time, timestep size and counter
      std::cout << "s=" << grid.size(0)
           << "␣k=" << k << "␣t=" << t << "␣dt=" << dt << std::endl;
    }

  // output results
  vtkout(grid,c,"concentration",counter,tend);
}

//===============================================================
 85 // The main function creates objects and does the time loop
 86 //===============================================================
 87 
 88 int main (int argc , char ** argv)
 89 {
 90   // initialize MPI, finalize is done automatically on exit
 91   Dune::MPIHelper::instance(argc,argv);
 92 
 93   // start try/catch block to get error messages from dune
 94   try {
 95     using namespace Dune;
 96 
 97     // use unitcube from dgf grids
 98     std::stringstream dgfFileName;
 99     dgfFileName << "grids/unitcube" << GridType :: dimension << ".dgf";
100 
101     // create grid pointer, GridType is defined by gridtype.hh
102     GridPtr<GridType> gridPtr( dgfFileName.str() );
103 
104     // grid reference
105     GridType& grid = *gridPtr;
106 
107     // half grid width 4 times
108     int level = 4 * DGFGridInfo<GridType>::refineStepsForHalf();
109 
110     // refine grid until upper limit of level
111     grid.globalRefine(level);
112 
113     // do time loop until end time 0.5
114     timeloop(grid, 0.5);
115   }
116   catch (std::exception & e) {
117     std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
118     return 1;
119   }
120   catch (Dune::Exception & e) {
121     std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
122     return 1;
123   }
124   catch (...) {
125     std::cout << "Unknown␣ERROR" << std::endl;
126     return 1;
127   }
128 
129   // done
130   return 0;
131 }
```

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 53-78 evolves the concentration in time. Finally, the simulation result is written to a file in line 81.

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 24 (File dune-grid-howto/adaptiveintegration.cc)

```
// $Id: adaptiveintegration.cc 243 2009-03-10 07:05:52Z mnolte $

#include"config.h"
#include<iostream>
#include<iomanip>
#include<dune/grid/io/file/vtk/vtkwriter.hh> // VTK output routines
#include <dune/common/mpihelper.hh> // include mpi helper class

// checks for defined gridtype and inlcudes appropriate dgfparser implementation
#include <dune/grid/io/file/dgfparser/dgfgridtype.hh>


#include"unitcube.hh"
#include"functors.hh"
#include"integrateentity.hh"


//! adaptive refinement test
template<class Grid, class Functor>
void adaptiveintegration (Grid& grid, const Functor& f)
{
  // get grid view type for leaf grid part
  typedef typename Grid::LeafGridView GridView;
  // get iterator type
  typedef typename GridView::template Codim<0>::Iterator ElementLeafIterator;

  // get grid view on leaf part
  GridView gridView = grid.leafView();

  // algorithm parameters
  const double tol=1E-8;
  const int loworder=1;
  const int highorder=3;

  // loop over grid sequence
  double oldvalue=1E100;
  for (int k=0; k<100; k++)
    {
      // compute integral on current mesh
          double value=0;
      for (ElementLeafIterator it = gridView.template begin<0>();
           it!=gridView.template end<0>(); ++it)
        value += integrateentity(it,f,highorder);

      // print result
      double estimated_error = std::abs(value-oldvalue);
      oldvalue=value; // save value for next estimate
      std::cout << "elements="
                << std::setw(8) << std::right
                << grid.size(0)
                << "␣integral="
                << std::scientific << std::setprecision(8)
                << value
                << "␣error=" << estimated_error
                << std::endl;

      // check convergence
      if (estimated_error <= tol*value)
        break;

      // refine grid globally in first step to ensure
      // that every element has a father
      if (k==0)
        {
          grid.globalRefine(1);
          continue;
        }

      // compute threshold for subsequent refinement
      double maxerror=-1E100;
      double maxextrapolatederror=-1E100;
      for (ElementLeafIterator it = grid.template leafbegin<0>();
           it!=grid.template leafend<0>(); ++it)
        {
          // error on this entity
          double lowresult=integrateentity(it,f,loworder);
          double highresult=integrateentity(it,f,highorder);
          double error = std::abs(lowresult-highresult);

          // max over whole grid
          maxerror = std::max(maxerror,error);

          // error on father entity
          double fatherlowresult=integrateentity(it->father(),f,loworder);
          double fatherhighresult=integrateentity(it->father(),f,highorder);
          double fathererror = std::abs(fatherlowresult-fatherhighresult);

          // local extrapolation
          double extrapolatederror = error*error/(fathererror+1E-30);
          maxextrapolatederror = std::max(maxextrapolatederror,extrapolatederror);
        }
      double kappa = std::min(maxextrapolatederror,0.5*maxerror);

      // mark elements for refinement
      for (ElementLeafIterator it = gridView.template begin<0>();
           it!=gridView.template end<0>(); ++it)
        {
          double lowresult=integrateentity(it,f,loworder);
          double highresult=integrateentity(it,f,highorder);
          double error = std::abs(lowresult-highresult);
          if (error>kappa) grid.mark(1,*it);
        }

```

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

The work is done in the function `adaptiveintegration`. Lines 40-43 compute the value of the integral on the current mesh. After printing the result the decision whether to continue or not is done in line 58. 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 65). Now the refinement threshold κ can be computed in lines 70-92. Finally the last loop in lines 95-102 marks elements for refinement and lines 105-107 actually do the refinement. The reason for dividing refinement into three functions

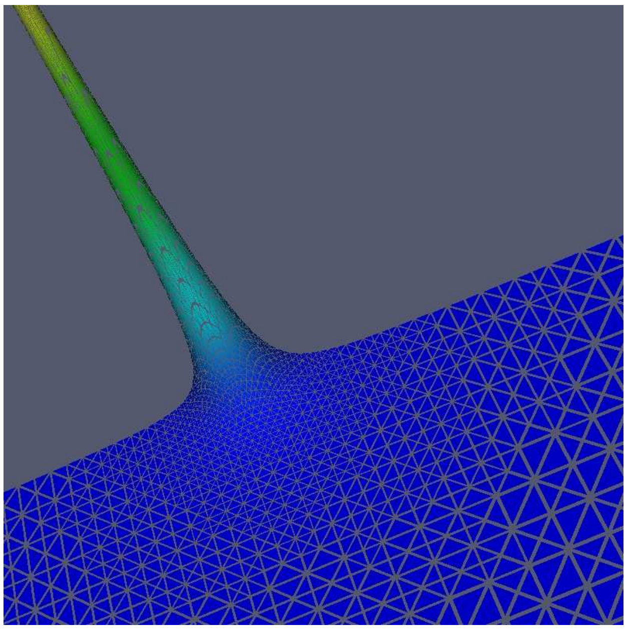
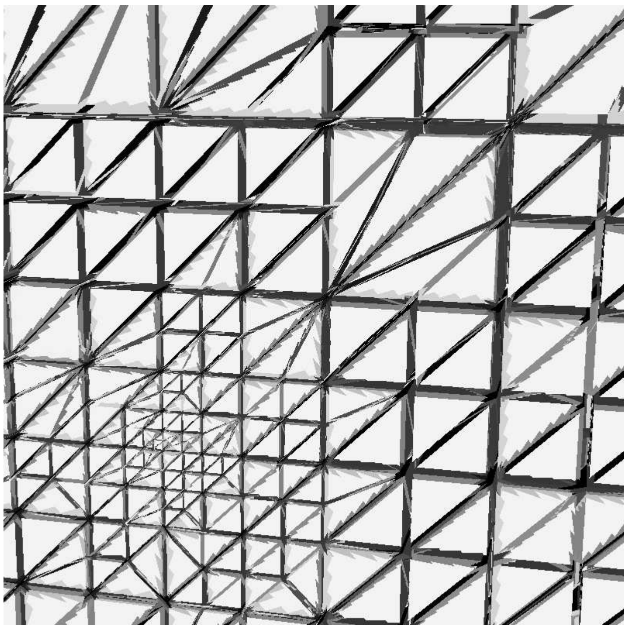



Figure 7.1: Two and three-dimensional grids generated by the adaptive integration algorithm applied to the needle pulse. Left grid is generated using Alberta, right grid is generated using UG.

`preAdapt()`, `adapt()` and `postAdapt()` will be explained with the next example. Note the flexibility of this algorithm: It runs in any space dimension on any kind of grid and different integration orders can easily be incorporated. And that with just about 100 lines of code including comments.

Figure 7.1.2 shows two grids generated by the adaptive integration algorithm.

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

7.2 Adaptive cell centered finite volumes

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

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

- Compute global maximum and minimum of element concentrations:

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

- As the local indicator in cell ω_i we define

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

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 through refining some element in the old mesh: Copy the value from the element in the old mesh to the new mesh.

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

Listing 25 (File dune-grid-howto/finitevolumeadapt.hh)

```
#include <map>

struct RestrictedValue
{
  double value;
  int count;
  RestrictedValue ()
  {
    value = 0;
    count = 0;
  }
};

template<class G, class M, class V>
bool finitevolumeadapt (G& grid, M& mapper, V& c, int lmin, int lmax, int k)
{
  // tol value for refinement strategy
  const double refinetol  = 0.05;
  const double coarsentol = 0.001;

  // type used for coordinates in the grid
  typedef typename G::ctype ct;

  // grid view types
  typedef typename G::LeafGridView LeafGridView;
  typedef typename G::LevelGridView LevelGridView;

  // iterator types
  typedef typename LeafGridView::template Codim<0>::Iterator LeafIterator;
  typedef typename LevelGridView::template Codim<0>::Iterator LevelIterator;
```

```

// entity and entity pointer
typedef typename G::template Codim<0>::Entity Entity;
typedef typename G::template Codim<0>::EntityPointer EntityPointer;

// intersection iterator type
typedef typename LeafGridView::IntersectionIterator LeafIntersectionIterator;

// global id set types, local means that the numbering is unique in a single process only.
typedef typename G::template Codim<0>::LocalIdSet IdSet;
// type for the index set, note that this is _not_ an integer
typedef typename IdSet::IdType IdType;

// get grid view on leaf grid
LeafGridView leafView = grid.leafView();

// compute cell indicators
V indicator(c.size(),-1E100);
double globalmax = -1E100;
double globalmin =  1E100;
for (LeafIterator it = leafView.template begin<0>();
     it!=leafView.template end<0>(); ++it)
{
  // my index
  int indexi = mapper.map(*it);

  // global min/max
  globalmax = std::max(globalmax,c[indexi]);
  globalmin = std::min(globalmin,c[indexi]);

  LeafIntersectionIterator isend = leafView.iend(*it);
  for (LeafIntersectionIterator is = leafView.ibegin(*it); is!=isend; ++is)
  {
    const typename LeafIntersectionIterator::Intersection &intersection = *is;
    if( !intersection.neighbor() )
      continue;

    // access neighbor
    const EntityPointer pOutside = intersection.outside();
    const Entity &outside = *pOutside;
    int indexj = mapper.map( outside );

    // handle face from one side only
    if ( it.level() > outside.level() ||
            (it.level() == outside.level() && indexi<indexj) )
    {
      double localdelta = std::abs(c[indexj]-c[indexi]);
      indicator[indexi] = std::max(indicator[indexi],localdelta);
      indicator[indexj] = std::max(indicator[indexj],localdelta);
    }
  }
}

// mark cells for refinement/coarsening
double globaldelta = globalmax-globalmin;
int marked=0;
for (LeafIterator it = leafView.template begin<0>();
     it!=leafView.template end<0>(); ++it)
{
  if (indicator[mapper.map(*it)]>refinetol*globaldelta
          && (it.level()<lmax || !it->isRegular()))
  {
    const Entity &entity = *it;
```

```
      grid.mark( 1, entity );
      ++marked;
      LeafIntersectionIterator isend = leafView.iend(entity);
      for( LeafIntersectionIterator is = leafView.ibegin(entity); is != isend; ++is )
      {
        const typename LeafIntersectionIterator::Intersection intersection = *is;
        if( !intersection.neighbor() )
          continue;

        const EntityPointer pOutside = intersection.outside();
        const Entity &outside = *pOutside;
        if( (outside.level() < lmax) || !outside.isRegular() )
          grid.mark( 1, outside );
      }
    }
    if (indicator[mapper.map(*it)]<coarsentol*globaldelta && it.level()>lmin)
    {
      grid.mark( -1, *it );
      ++marked;
    }
  }
  if( marked==0 )
    return false;

  // restrict to coarse elements
  std::map<IdType,RestrictedValue> restrictionmap; // restricted concentration
  const IdSet& idset = grid.localIdSet();
  for (int level=grid.maxLevel(); level>=0; level--)
    {
      // get grid view on level grid
      LevelGridView levelView = grid.levelView(level);
      for (LevelIterator it = levelView.template begin<0>();
           it!=levelView.template end<0>(); ++it)
      {
        // get your map entry
        IdType idi = idset.id(*it);
        RestrictedValue& rv = restrictionmap[idi];

        // put your value in the map
        if (it->isLeaf())
          {
            int indexi = mapper.map(*it);
            rv.value = c[indexi];
            rv.count = 1;
          }

        // average in father
        if (it.level()>0)
          {
            EntityPointer ep = it->father();
            IdType idf = idset.id(*ep);
            RestrictedValue& rvf = restrictionmap[idf];
            rvf.value += rv.value/rv.count;
            rvf.count += 1;
          }
      }
      }
  grid.preAdapt();

  // adapt mesh and mapper
  bool rv=grid.adapt();
  mapper.update();
  c.resize(mapper.size());
```

```
157
158   // interpolate new cells, restrict coarsened cells
159   for (int level=0; level<=grid.maxLevel(); level++)
160     {
161       LevelGridView levelView = grid.levelView(level);
162       for (LevelIterator it = levelView.template begin<0>();
163            it!=levelView.template end<0>(); ++it)
164       {
165         // get your id
166         IdType idi = idset.id(*it);
167
168         // check map entry
169           typename std::map<IdType,RestrictedValue>::iterator rit
170               = restrictionmap.find(idi);
171         if (rit!=restrictionmap.end())
172           {
173             // entry is in map, write in leaf
174             if (it->isLeaf())
175               {
176                 int indexi = mapper.map(*it);
177                 c[indexi] = rit->second.value/rit->second.count;
178               }
179           }
180         else
181           {
182               // value is not in map, interpolate from father element
183             if (it.level()>0)
184               {
185                 EntityPointer ep = it->father();
186                 IdType idf = idset.id(*ep);
187                 RestrictedValue& rvf = restrictionmap[idf];
188                 if (it->isLeaf())
189                   {
190                     int indexi = mapper.map(*it);
191                     c[indexi] = rvf.value/rvf.count;
192                   }
193                 else
194                   {
195                     // create new entry
196                     RestrictedValue& rv = restrictionmap[idi];
197                     rv.value = rvf.value/rvf.count;
198                     rv.count = 1;
199                   }
200               }
201           }
202       }
203     }
204   grid.postAdapt();
205
206   return rv;
207 }
```

The loop in lines 51-82 computes the indicator values η_i as well as the global minimum and maximum $\overline{C}, \underline{C}$. Then the next loop in lines 87-114 marks the elements for refinement. Lines 119-149 construct a map that stores for each element in the mesh (on all levels) the average of the element values in the leaf elements of the subtree of the given element. This is accomplished by descending from the fine grid levels to the coarse grid levels and thereby adding the value in an element to the father element. The key into the map is the global id of an element. Thus the value is accessible also after mesh modification.

Now the grid can really be modified in line 154 by calling the `adapt()` method on the grid object.

The mapper is updated to reflect the changes in the grid in line 155 and the concentration vector is resized to the new size in line 156. Then the values have to be interpolated to the new elements in the mesh using the map and finally to be transferred to the resized concentration vector. This is done in the loop in lines 159-202.

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

Listing 26 (File dune-grid-howto/adativefinitevolume.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

// checks for defined gridtype and inlcudes appropriate dgfparser implementation
#include <dune/grid/io/file/dgfparser/dgfgridtype.hh>

#include <dune/grid/common/mcmgmapper.hh> // mapper class
#include <dune/common/mpihelper.hh> // include mpi helper class

#include "vtkout.hh"
#include "unitcube.hh"
#include "transportproblem2.hh"
#include "initialize.hh"
#include "evolve.hh"
#include "finitevolumeadapt.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 timeloop (G& grid, double tend, int lmin, int lmax)
{
  // 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);
  for (int i=grid.maxLevel(); i<lmax; i++)
    {
      if (grid.maxLevel()>=lmax) break;
      finitevolumeadapt(grid,mapper,c,lmin,lmax,0);
      initialize(grid,mapper,c);
    }

  // write initial data
  vtkout(grid,c,"concentration",0,0);

  // variables for time, timestep etc.
  double dt, t=0;
  double saveStep = 0.1;
  const double saveInterval = 0.1;
  int counter = 0;
  int k = 0;

  std::cout << "s=" << grid.size(0) << "␣k=" << k << "␣t=" << t << std::endl;
  while (t<tend)
    {
    // augment time step counter
    ++k;

    // apply finite volume scheme
      evolve(grid,mapper,c,t,dt);

    // augment time
      t += dt;

    // check if data should be written
    if (t >= saveStep)
    {
      // write data
        vtkout(grid,c,"concentration",counter,t);

      // increase counter and saveStep for next interval
      saveStep += saveInterval;
      ++counter;
    }

    // print info about time, timestep size and counter
      std::cout << "s=" << grid.size(0)
          << "␣k=" << k << "␣t=" << t << "␣dt=" << dt << std::endl;

    // for unstructured grids call adaptation algorithm
    if( Dune :: Capabilities :: IsUnstructured<G> :: v )
    {
        finitevolumeadapt(grid,mapper,c,lmin,lmax,k);
    }
    }

  // write last time step
  vtkout(grid,c,"concentration",counter,tend);

  // write
}

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

int main (int argc , char ** argv)
{
  // initialize MPI, finalize is done automatically on exit
  Dune::MPIHelper::instance(argc,argv);

  // start try/catch block to get error messages from dune
  try {
    using namespace Dune;

    // use unitcube from grids
    std::stringstream dgfFileName;
118     dgfFileName << "grids/unitcube" << GridType :: dimension << ".dgf";
119
120     // create grid pointer, GridType is defined by gridtype.hh
121     GridPtr<GridType> gridPtr( dgfFileName.str() );
122
123     // grid reference
124     GridType& grid = *gridPtr;
125
126     // minimal allowed level during refinement
127     int minLevel = 2 * DGFGridInfo<GridType>::refineStepsForHalf();
128
129     // refine grid until upper limit of level
130     grid.globalRefine(minLevel);
131
132     // maximal allowed level during refinement
133     int maxLevel = minLevel + 3 * DGFGridInfo<GridType>::refineStepsForHalf();
134
135     // do time loop until end time 0.5
136     timeloop(grid, 0.5, minLevel, maxLevel);
137   }
138   catch (std::exception & e) {
139     std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
140     return 1;
141   }
142   catch (Dune::Exception & e) {
143     std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
144     return 1;
145   }
146   catch (...) {
147     std::cout << "Unknown␣ERROR" << std::endl;
148     return 1;
149   }
150
151   // done
152   return 0;
153 }
```

The program works analogously to the non adaptive `finitevolume` version from the previous chapter. The only differences are inside the `timeloop` function. During the initialization of the concentration vector in line 49 and after each time step in line 93 the function `finitevolumeadapt` is called in order to refine the grid. The initial adaptation is repeated $\overline{M}$ times. Note that adaptation after each time steps is deactivated during the compiler phase for unstructured grids with help of the `Capabilities` class. This is because structured grids do not allow a conforming refinement and are therefore unusable for adaptive schemes. In fact, the `adapt` method on a grid of `YaspGrid` e.g. results in a *global* grid refinement.

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

8 Parallelism

8.1 DUNE Data Decomposition Model

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

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

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

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

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

Entity located in	Partition Type value
$B_i = \overline{\partial\Omega_i \setminus \partial\Omega}$	*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

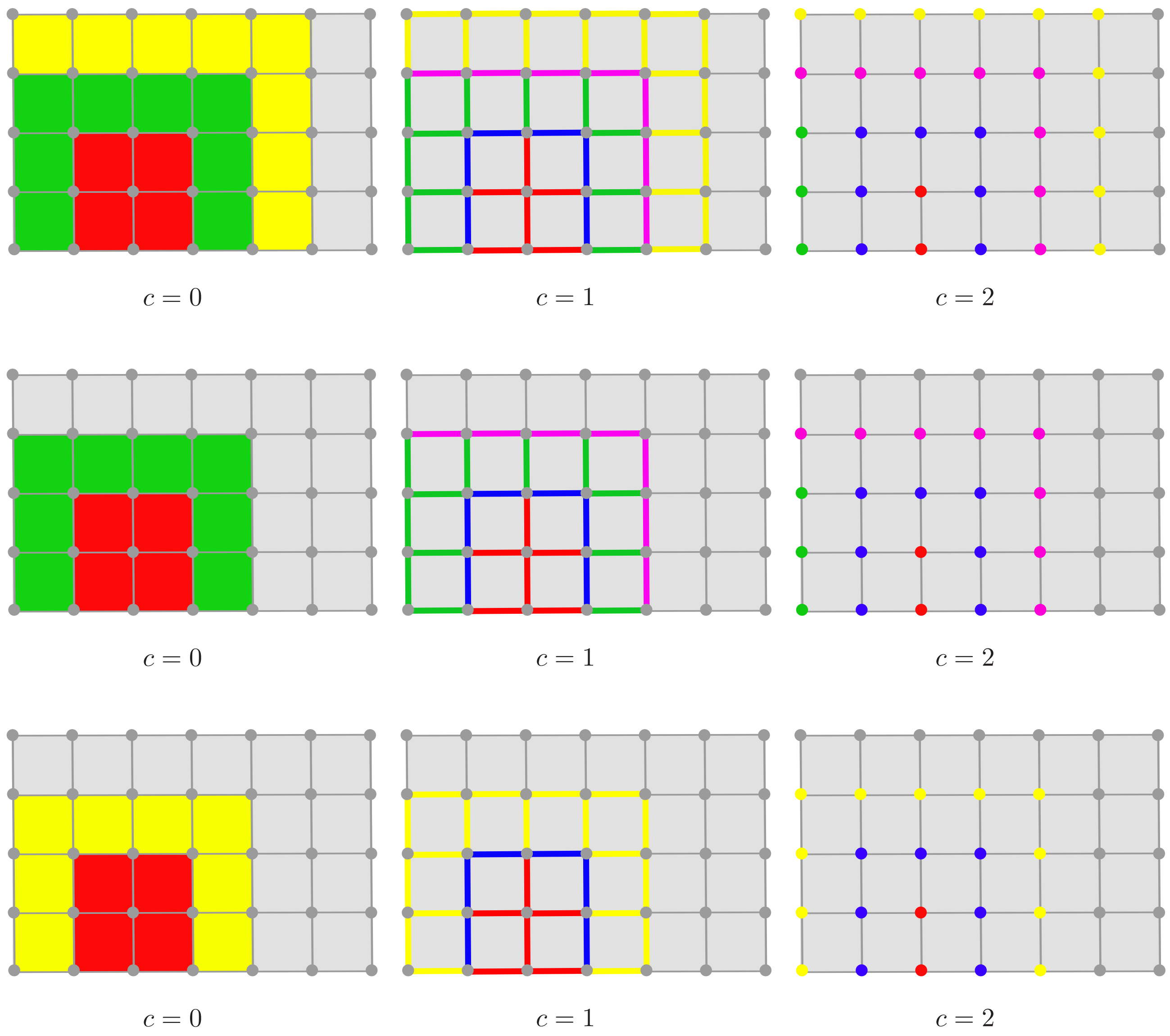


Figure 8.1: Color coded illustration of different data decompositions: interior (red), border (blue), overlap (green), front (magenta) and ghost (yellow), gray encodes entites not stored by the process. First row shows case with interior, overlap and ghost entities, second row shows a case with interior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

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

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

```
// A DataHandle class to exchange entries of a vector
template<class M, class V> // mapper type and vector type
class VectorExchange
  : public Dune::CommDataHandleIF<VectorExchange<M,V>,
                  typename V::value_type>
{
public:
  //! export type of data for message buffer
  typedef typename V::value_type DataType;

  //! returns true if data for this codim should be communicated
  bool contains (int dim, int codim) const
  {
    return (codim==0);
  }

  //! returns true if size per entity of given dim and codim is a constant
  bool fixedsize (int dim, int codim) const
  {
    return true;
  }

  /*! how many objects of type DataType have to be sent for a given entity

  Note: Only the sender side needs to know this size.
  */
  template<class EntityType>
  size_t size (EntityType& e) const
  {
    return 1;
  }

  //! pack data from user to message buffer
  template<class MessageBuffer, class EntityType>
  void gather (MessageBuffer& buff, const EntityType& e) const
  {
    buff.write(c[mapper.map(e)]);
  }

  /*! unpack data from message buffer to user

  n is the number of objects sent by the sender
  */
  template<class MessageBuffer, class EntityType>
  void scatter (MessageBuffer& buff, const EntityType& e, size_t n)
  {
    DataType x;
    buff.read(x);
    c[mapper.map(e)]=x;
  }

  //! constructor
  VectorExchange (const M& mapper_, V& c_)
    : mapper(mapper_), c(c_)
  {}

private:
  const M& mapper;
  V& c;
};
```

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 21 on page 45.

Listing 28 (File dune-grid-howto/parevolve.hh)

```
#include<dune/grid/common/referenceelements.hh>

template<class G, class M, class V>
void parevolve (const G& grid, const M& mapper, V& c, double t, double& dt)
{
  // check data partitioning
  assert(grid.overlapSize(0)>0 || (grid.ghostSize(0)>0));

  // first we extract the dimensions of the grid
  const int dim = G::dimension;
  const int dimworld = G::dimensionworld;

  // type used for coordinates in the grid
  typedef typename G::ctype ct;

  // type for grid view on leaf part
  typedef typename G::LeafGridView GridView;

  // iterator type
  typedef typename GridView::template Codim<0>::
    template Partition<Dune::All_Partition>::Iterator LeafIterator;

  // intersection iterator type
  typedef typename GridView::IntersectionIterator IntersectionIterator;

  // type of intersection
  typedef typename IntersectionIterator::Intersection Intersection;

  // entity pointer type
  typedef typename G::template Codim<0>::EntityPointer EntityPointer;

  // allocate a temporary vector for the update
  V update(c.size());
  for (typename V::size_type i=0; i<c.size(); i++) update[i] = 0;

  // initialize dt very large
  dt = 1E100;

  // get grid view instance on leaf grid
  GridView gridView = grid.leafView();

  // compute update vector and optimum dt in one grid traversal
  // iterate over all entities, but update is only used on interior entities
  LeafIterator endit = gridView.template end<0,Dune::All_Partition>();
  for (LeafIterator it = gridView.template begin<0,Dune::All_Partition>(); it!=endit; ++it)
    {
      // cell geometry type
      Dune::GeometryType gt = it->type();

      // cell center in reference element
      const Dune::FieldVector<ct,dim>&
        local = Dune::ReferenceElements<ct,dim>::general(gt).position(0,0);

      // cell center in global coordinates
```

```
Dune::FieldVector<ct,dimworld>
  global = it->geometry().global(local);

// cell volume, assume linear map here
double volume = it->geometry().integrationElement(local)
  *Dune::ReferenceElements<ct,dim>::general(gt).volume();

// cell index
int indexi = mapper.map(*it);

// variable to compute sum of positive factors
double sumfactor = 0.0;

// run through all intersections with neighbors and boundary
const IntersectionIterator isend = gridView.iend(*it);
for( IntersectionIterator is = gridView.ibegin(*it); is != isend; ++is )
    {
      const Intersection &intersection = *is;

      // get geometry type of face
      Dune::GeometryType gtf = intersection.intersectionSelfLocal().type();

      const Dune::ReferenceElement< ct, dim-1 > &refElement
        = Dune::ReferenceElements< ct, dim-1 >::general( gtf );

      // center in face's reference element
      const Dune::FieldVector< ct, dim-1 > &facelocal = refElement.position( 0, 0 );

      // get normal vector scaled with volume
      Dune::FieldVector< ct, dimworld > integrationOuterNormal
        = intersection.integrationOuterNormal( facelocal );
      integrationOuterNormal *= refElement.volume();

      // center of face in global coordinates
      Dune::FieldVector< ct, dimworld > faceglobal
        = intersection.intersectionGlobal().global( facelocal );

      // evaluate velocity at face center
      Dune::FieldVector<double,dim> velocity = u(faceglobal,t);

      // compute factor occuring in flux formula
      double factor = velocity*integrationOuterNormal/volume;

      // for time step calculation
      if (factor>=0) sumfactor += factor;

      // handle interior face
      if( intersection.neighbor() )
      {
        // access neighbor
        EntityPointer outside = intersection.outside();
        int indexj = mapper.map(*outside);

        const int insideLevel = it->level();
        const int outsideLevel = outside->level();

        // handle face from one side
        if( (insideLevel > outsideLevel)
            || ((insideLevel == outsideLevel) && (indexi < indexj)) )
        {
          // compute factor in neighbor
          Dune::GeometryType nbgt = outside->type();
          const Dune::FieldVector<ct,dim>&
118                       nblocal = Dune::ReferenceElements<ct,dim>::general(nbgt).position(0,0);
119                 double nbvolume = outside->geometry().integrationElement(nblocal)
120                       *Dune::ReferenceElements<ct,dim>::general(nbgt).volume();
121                 double nbfactor = velocity*integrationOuterNormal/nbvolume;
122 
123                 if( factor < 0 ) // inflow
124                 {
125                   update[indexi] -= c[indexj]*factor;
126                   update[indexj] += c[indexj]*nbfactor;
127                 }
128                 else // outflow
129                 {
130                   update[indexi] -= c[indexi]*factor;
131                   update[indexj] += c[indexi]*nbfactor;
132                 }
133               }
134             }
135 
136             // handle boundary face
137             if( intersection.boundary() )
138             {
139               if( factor < 0 ) // inflow, apply boundary condition
140                 update[indexi] -= b(faceglobal,t)*factor;
141               else // outflow
142                 update[indexi] -= c[indexi]*factor;
143             }
144           } // end all intersections
145 
146       // compute dt restriction
147       if (it->partitionType()==Dune::InteriorEntity)
148         dt = std::min(dt,1.0/sumfactor);
149 
150     } // end grid traversal
151 
152   // global min over all partitions
153   dt = grid.comm().min(dt);
154   // scale dt with safety factor
155   dt *= 0.99;
156 
157   // exchange update
158   VectorExchange<M,V> dh(mapper,update);
159   grid.template
160     communicate<VectorExchange<M,V> >(dh,Dune::InteriorBorder_All_Interface,
161                                       Dune::ForwardCommunication);
162 
163   // update the concentration vector
164   for (unsigned int i=0; i<c.size(); ++i)
165     c[i] += dt*update[i];
166 
167   return;
168 }
```

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 may be either of partition type *overlap* or *ghost*. The finite volume scheme itself only computes the updates for the elements with partition type *interior*.

In order to iterate over entities with a specific partiton type the leaf and level iterators can be parametrized by an additional argument `PartitionIteratorType` as shown in line 21. 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 44-45.

The next change is in line 147 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 153. 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 158-161 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 29 (File dune-grid-howto/parfinitevolume.cc)

```
 1 #include"config.h"                 // know what grids are present
 2 #include<iostream>                 // for input/output to shell
 3 #include<fstream>                  // for input/output to files
 4 #include<vector>                   // STL vector class
 5 #include<dune/grid/common/mcmgmapper.hh> // mapper class
 6 #include <dune/common/mpihelper.hh> // include mpi helper class
 7 
 8 
 9 // checks for defined gridtype and inlcudes appropriate dgfparser implementation
10 #include"vtkout.hh"
11 #include"unitcube.hh"
12 #include "transportproblem2.hh"
13 #include"initialize.hh"
14 #include"parfvdatahandle.hh"
15 #include"parevolve.hh"
16 
17 
18 //===============================================================
19 // the time loop function working for all types of grids
20 //===============================================================
21 
22 //! Parameter for mapper class
23 template<int dim>
24 struct P0Layout
25 {
26   bool contains (Dune::GeometryType gt)
27   {
28     if (gt.dim()==dim) return true;
29     return false;
30   }
31 };
32 
33 template<class G>
34 void partimeloop (const G& grid, double tend)
35 {
36   // make a mapper for codim 0 entities in the leaf grid
37   Dune::LeafMultipleCodimMultipleGeomTypeMapper<G,P0Layout>
38     mapper(grid);
39 
40   // allocate a vector for the concentration
41   std::vector<double> c(mapper.size());
42 
43   // initialize concentration with initial values
44   initialize(grid,mapper,c);
45   vtkout(grid,c,"pconc",0,0.0,grid.comm().rank());
```

```

  // now do the time steps
  double t=0,dt;
  int k=0;
  const double saveInterval = 0.1;
  double saveStep = 0.1;
  int counter = 1;
  while (t<tend)
    {
      // augment time step counter
      k++;

      // apply finite volume scheme
      parevolve(grid,mapper,c,t,dt);

      // augment time
      t += dt;

      // check if data should be written
      if (t >= saveStep)
      {
        // write data
        vtkout(grid,c,"pconc",counter,t,grid.comm().rank());

        //increase counter and saveStep for next interval
        saveStep += saveInterval;
        ++counter;
      }

      // print info about time, timestep size and counter
      if (grid.comm().rank()==0)
        std::cout << "k=" << k << "␣t=" << t << "␣dt=" << dt << std::endl;
    }
  vtkout(grid,c,"pconc",counter,tend,grid.comm().rank());
}

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

int main (int argc , char ** argv)
{
  // initialize MPI, finalize is done automatically on exit
  Dune::MPIHelper::instance(argc,argv);

  // start try/catch block to get error messages from dune
  try {
    using namespace Dune;

    UnitCube<YaspGrid<2>,64> uc;
    uc.grid().globalRefine(2);
    partimeloop(uc.grid(),0.5);

    /* To use an alternative grid implementations for parallel computations,
       uncomment exactly one definition of uc2 and the line below. */
//     #define LOAD_BALANCING

// UGGrid supports parallelization in 2 or 3 dimensions
#if HAVE_UG
//     typedef UGGrid< 2 > GridType;
//     UnitCube< GridType, 2 > uc2;
#endif

109 //  ALUGRID supports parallelization in 3 dimensions only
110 #if HAVE_ALUGRID
111 //     typedef ALUCubeGrid< 3, 3 > GridType;
112 //     typedef ALUSimplexGrid< 3, 3 > GridType;
113 //     UnitCube< GridType , 1 > uc2;
114 #endif
115
116 #ifdef LOAD_BALANCING
117
118     // refine grid until upper limit of level
119     uc2.grid().globalRefine( 6 );
120
121     // re-partition grid for better load balancing
122     uc2.grid().loadBalance();
123
124     // do time loop until end time 0.5
125     partimeloop(uc2.grid(), 0.5);
126 #endif
127
128   }
129   catch (std::exception & e) {
130     std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
131     return 1;
132   }
133   catch (Dune::Exception & e) {
134     std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
135     return 1;
136   }
137   catch (...) {
138     std::cout << "Unknown␣ERROR" << std::endl;
139     return 1;
140   }
141
142   // done
143   return 0;
144 }
```

A difference to the sequential program can be found in line 76 where the printing of the data of the current time step is restricted to the process with rank 0. `YaspGrid` does not support dynamical load balancing and therefore needs to start with a sufficiently fine grid that allows a reasonable partition where each processes gets a non-empty part of grid. This is why we do not use DGF Files in the parallel example and initialize the grid by the UnitCube class instead. For `YaspGrid` this allows an easy selection of the grid's initial coarseness through the second template argument of the `UnitCube`. This argument should be chosen sufficiently high, because after each global refinement step the overlap region grows and therefore the communicaton overhead increases.

If you want to use a grid with support for dynamical load balancing, define the macro `LOAD\_BALANCING` and uncomment on of the possible definitions for such a grid in the code. In this case in line 122 the method `loadBalance` is called on the grid. This method re-partitions the grid in a way such that on every partition there is an equal amount of grid elements.

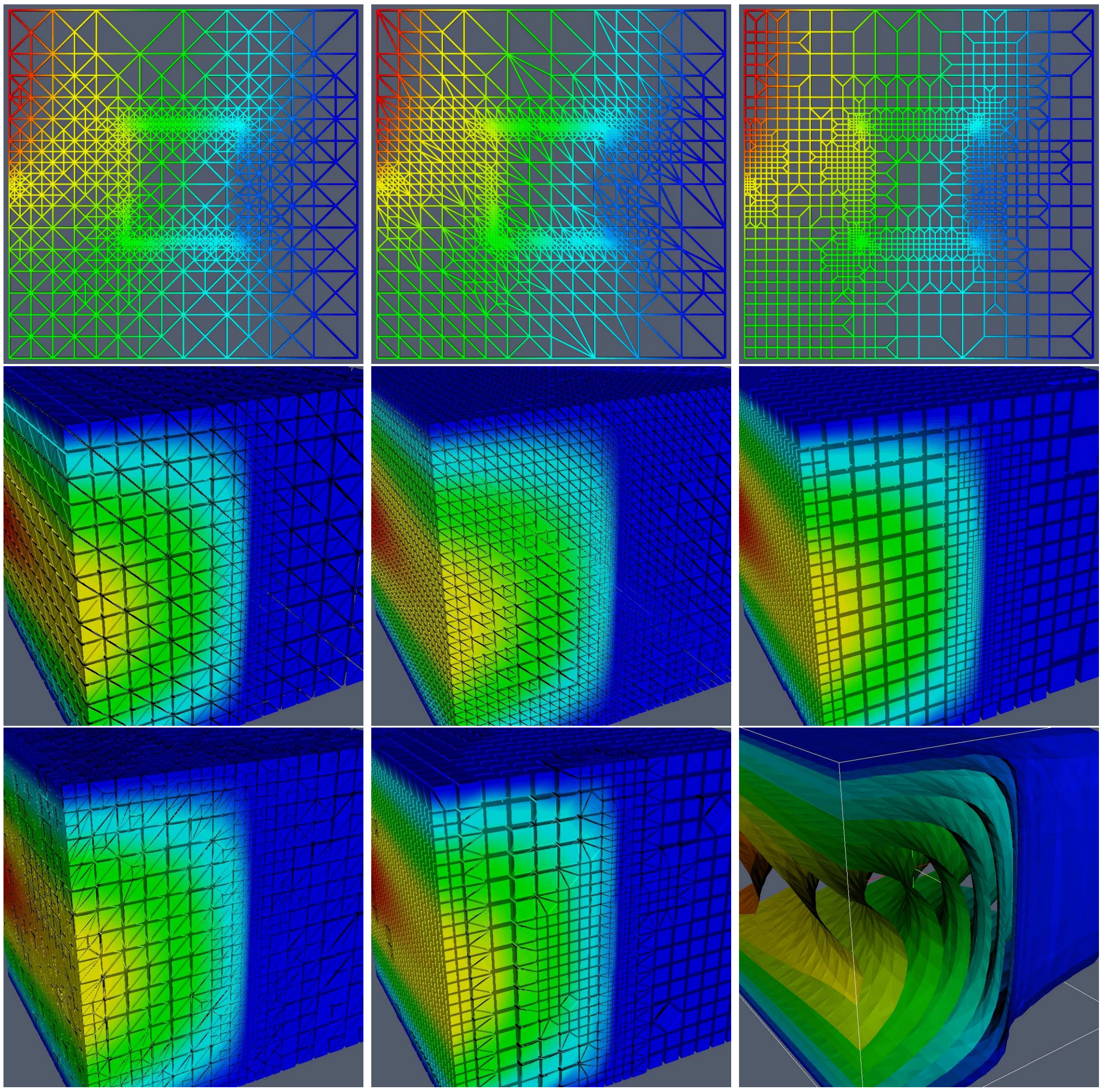


Figure 8.2: Adaptive solution of an elliptic model problem with P_1 conforming finite elements and residual based error estimator. Illustrates that adaptive finite element algorithm can be formulated independent of dimension, element type and refinement scheme. From top to bottom, left to right: Alberta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

Bibliography

[1] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[2] P. Bastian, K. Birken, S. Lang, K. Johannsen, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG: A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1:27–40, 1997.

[3] A. Dedner, C. Rohde, B. Schupp, and M. Wesenberg. A parallel, load balanced mhd code on locally adapted, unstructured grids in 3d. *Computing and Visualization in Science*, 7:79–96, 2004.

[4] P. 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.