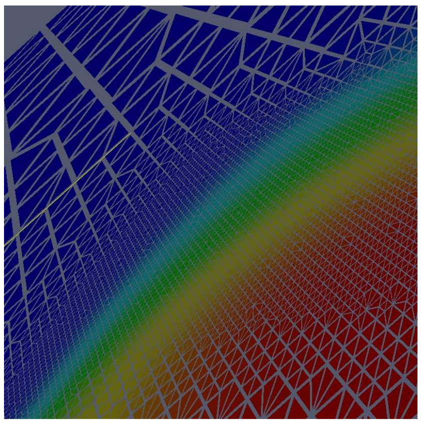
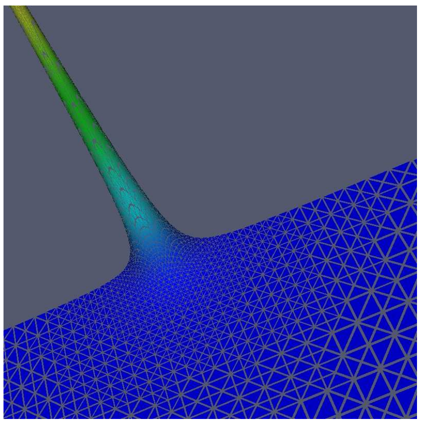
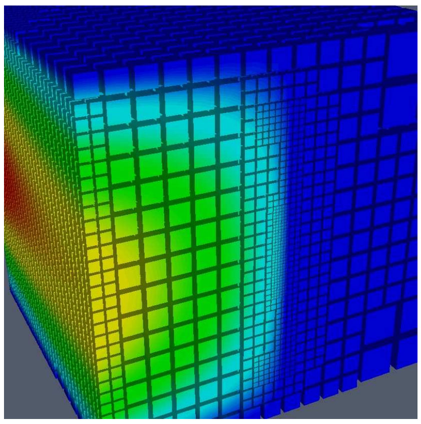
The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO

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

Version 2.4, September 25, 2015

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

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

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

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

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

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

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

Contents

1 Introduction

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-geometry` provides element geometry classes and quadrature rules.
- `dune-grid` is the grid interface 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 VTK (parallel XML format for unstructured grids). The graphics package Grape, [5] has been integrated in interactive mode.
- `dune-istl` – *Iterative Solver Template Library.* Provides generic sparse matrix/vector classes and a variety of solvers based on these classes. A special feature is the use of templates to exploit the recursive block structure of finite element matrices at compile time. Available solvers include Krylov methods, (block-) incomplete decompositions and aggregation-based algebraic multigrid.
- `dune-localfunctions` – *Library of local base functions.* Provides classes for base functions on reference elements from which global discrete function spaces can be constructed.

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

1.2 Download

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

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

Alternatively, you can download the latest development version via anonymous Git. 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 Git, see the web page for further information. Moreover, we assume that you use a UNIX system (usually Linux or OS X). If you have the Redmond system then you have to explore new horizons.

In order to build the **DUNE** framework, you need a standards compliant C++ compiler. We tested compiling with GNU `g++` in version 4.4 or newer and `clang` in version 3.4 or newer. Recent versions of Intel `icc` (≥ 15) should work, too. For older versions of `icc`, like 14.0.3, several issues are known and usage of these is discouraged.

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

```
> dune-common-2.3/bin/dunecontrol all
```

Replace "`2.3`" 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://www.iwr.uni-heidelberg.de/frame/iwrwikiequipment/software/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 Open MPI. For the **DUNE** build system to find these libraries, the `configure` scripts of the particular **DUNE** modules must be passed the locations of the respective installations. The `dunecontrol` script facilitates to pass options to the `configure` via a configuration file. Such a configuration file might look like this:

```
CONFIGURE_FLAGS="--with-alugrid=/path/to/alugrid/␣"\
"--with-alberta=/path/to/alberta␣"\
"--with-ug=/path/to/ug␣--enable-parallel"
MAKE_FLAGS="-j␣2"
```

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

```
> dune-common-2.3/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/doxygen.html`

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 similar to the GNU Lesser General Public License, which by itself isn’t suitable for a C++ template library.

The exact wording of the exception reads as follows:

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

[1] `http://www.doxygen.org/`
[2] `http://www.gnu.org/licenses/gpl-2.0.html`

2 Getting started

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

2.1 Creating your first grid

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:

// Dune includes
#include <config.h>           // file constructed by ./configure script
#include <array>
#include <memory>
#include <dune/common/parallel/mpihelper.hh> // include mpi helper class
#include <dune/grid/yaspgrid.hh> // load Yasp grid definition
#include <dune/grid/common/gridinfo.hh> // definition of gridinfo

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::YaspGrid<dim> GridType;
    Dune::FieldVector<double,dim> length(1.0);
    std::array<int,dim> elements;
    std::fill(elements.begin(), elements.end(), 4);
    GridType grid(length,elements);

    // 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
44   return 0;
45 }
```

This program is quite simple. It starts with some includes in lines 5-10. 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 file `dune/grid/yaspgrid.hh` includes the headers for the `YaspGrid` class which provides a special implementation of the **DUNE** grid interface with a structured mesh of arbitrary dimension. Then `dune/grid/common/gridinfo.hh` loads the headers of some functions which print useful information about a grid.

Since the dimension will be used as a template parameter in many places below we define it as a constant in line number 20. The `YaspGrid` class template takes its dimension as a template parameter. Technically, `YaspGrid` has a second template parameter, that defines how it stores coordinates. In this example, we use an equidistant grid, which allows us to stay with the default value `EquidistantCoordinates`. For ease of writing we define in line 21 the type `GridType` using the selected value for the dimension. All identifiers of the **DUNE** framework are within the `Dune` namespace.

Lines 22-24 prepare the arguments for the construction of a `YaspGrid` object. The first argument 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 `length` defines the lengths of the cube. The variable `elements` defines the number of cells or elements to be used in the respective dimension of the grid. Finally in line 25 we are now able to instantiate the `YaspGrid` 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)

```
=> Dune::YaspGrid<3, Dune::EquidistantCoordinates<double, 3> > (dim=3, dimworld=3)
level 0 codim[0]=64 codim[1]=240 codim[2]=300 codim[3]=125
leaf    codim[0]=64 codim[1]=240 codim[2]=300 codim[3]=125
leaf dim=3 geomTypes=((cube, 3)[0]=64,(cube, 2)[1]=240,(simplex, 1)[2]=300,(simplex, 0)[3]=125)
```

The first line tells you that you are looking at an `YaspGrid` object of the given dimensions. The **DUNE** grid interface supports unstructured, locally refined, logically nested grids. The coarsest grid is called level-0-grid or macro grid. Elements can be individually refined into a number of smaller elements. Each element of the macro grid and all its descendants obtained from refinement form a tree structure. All elements at depth n of a refinement tree form the level-n-grid. All elements that are leaves of a refinement tree together form the so-called leaf grid. The second line of the output tells us that this grid object consists only of a single level (level 0) while the next line tells us that that level 0 coincides also with the leaf grid in this case. Each line reports about the number of grid entities which make up the grid. We see that there are 64 elements (codimension 0), 240 faces (codimension 1), 300 edges (codimension 2) and 125 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*, as a line and a point are both *cube* and *simplex*.

Exercise 2.1 Try to play around with different grid sizes by assigning different values to the `element` 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)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#include <config.h>           // file constructed by ./configure script

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

// Dune includes
#include <dune/grid/yaspgrid.hh> // load yaspgrid definition
#include <dune/common/parallel/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.leafGridView();

  // 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 leaves
  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.levelGridView(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;
```

```
107   }
108 }
109
110
111 int main(int argc, char **argv)
112 {
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::YaspGrid<dim> GridType;
121     Dune::array<int,dim> N;
122     std::fill(N.begin(), N.end(), 1);
123     Dune::FieldVector<GridType::ctype,dim> H(1.0);
124     GridType grid(H,N);
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 parametrized by a class `G` that is assumed to implement the **DUNE** grid interface. Thus, it will work on *any* grid available in **DUNE** without any changes. We now go into the details of this function.

The algorithm should work in any dimension so we extract the grid's dimension in line 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 explicitly. Within the loop body in lines 42-48 the iterator `it` acts like a pointer to an entity of dimension `dim` and codimension 0. The exact type would be `typename G::template Codim<0>::Entity` just to mention it.

Please note, that from Dune 2.4 on, C++11 range based for statements will be used to iterate over entities of a grid. To enable this feature, you will need `g++` version 4.6 or higher. There are no issues with the mentioned version of `icc` or `clang`. The entire iteration loop, will then boil down to

```
for (auto&& e : elements(leafView))
{
  // the iteration loop
}
```

There is no more need of getting the iterator type by hand. Note that you should always write `auto&&` !In contrast to the above procedure you don't get an `EntityPointer`, but a reference to the `Entity` itself.

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).

Again, above task, can be done with a range based for statement by using the function `vertices` instead of `elements` in the above code snippet.

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

Dimension and world dimension

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

Faces, entities and codimension

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

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

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

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

4.1 Creating Structured Grids

Creating structured grids is comparatively simple, as little information needs to be provided. In general, for uniform structured grids, the grid dimension, bounding box, and number of elements in each direction suffices. Such information can be given directly with the constructor of the grid object. **DUNE** does not currently specify the signature of grid constructors, and hence they are all slightly different for different grid implementations.

As already seen int he introduction, the code to construct a sequential `YaspGrid` with 10 cells in each direction is

```
    Dune::array<int,dim> n;
    std::fill(n.begin(). n.end(), 10);

    Dune::FieldVector<double,dim> upper(1.0);

    YaspGrid<dim> grid(upper, n);
```

Note that for an equidistant grid, you do not have to specify the lower left corner as `YaspGrid` hardwires it to zero. This limitation can be overcome by using another feature of `YaspGrid`:

`YaspGrid` can also be used a tensorproduct grid. A tensorproduct grid is defined by a set of coordinates $\{x_{i,0}, \dots, x_{i,n_i}\}$ for each direction i. The vertices of the grid are then given by the tuples

$$\{(x_{0,i_0}, \dots, x_{d-1,i_{d-1}}) \;\; with \;\; 0 \leq i_j \leq n_j \;\; \forall j\}$$

Such tensorproduct grid is a structured non-equidistant grid. Having a non-zero lower left corner is possible by specifying coordinates accordingly. `YaspGrid`s tensorproduct features are enabled by using

TensorProductCoordinates<ctype,dim> as the second template paramter. Coordinates are given as as a Dune::array<std::vector<ctype>,dim>. See the following example which initializes a grid is one cell, that covers $[-1,1]^2$:

```
Dune::array<std::vector<ctype,2> > coords;
coords[0] = {-1., 1.};
coords[1] = {-1., 1.};

YaspGrid<dim, TensorProductCoordinates<ctype,dim> > grid(coords);
```

Another unstructured is the one-dimensional OneDGrid, which has a constructor

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

for uniform grids.

4.2 Reading Unstructured Grids from Files

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

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

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

A second format is AmiraMesh, which is the native format of the Amira.[2] To read AmiraMesh files you need to have the library libamiramesh[3] installed. Then

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

reads the grid in filename into the FooGrid.

[1]http://geuz.org/gmsh/
[2]http://www.amira.com/
[3]http://amira.com/downloads/patch-archive/patches412/81.html

Further available formats are StarCD and the native Alberta format. See the class documentation of dune-grid for an up-to-date list. Demo grids for each format can be found in `dune-grid/doc/grids`. They exist for documentation and also as example grids for the unit tests of the file readers. The unit tests should not hardwire the path to the example grids. Instead, the path should be provided in the preprocessor variable `DUNE_GRID_EXAMPLE_GRIDS_PATH`.

4.3 The DUNE Grid Format (DGF)

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

Here we only give a short introduction. Basically one can choose the grid manager during the make process of your program: `make GRIDTYPE=MYGRID GRIDDIM=d GRIDWORLD=w` Including `config.h` will then introduce the namespace `GridSelector` into the `Dune` namespace. This contains the typedef `GridType` which is the type of the grid. Furthermore the required header files for the grid manager are included. Through the module **DUNE**-grid the following grid managers can be used (for `MYGROD` in the example above):
`ALBERTAGRID,ALUGRID_CUBE,ALUGRID_SIMPLEX,ALUGRID_CONFORM,ONEDGRID,UGGRID`,
and `YASPGRID`.
The following example shows how an instance of the defined grid is generated. Given a DGF file, for example `unitcube2.dgf`, a grid pointer is created as follows

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

The grid is accessed by dereferencing the grid pointer.

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

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

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

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

4.4 The Grid Factory

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

[4]`http://www.dune-project.org/doc/doxygen/html/classDune_1_1DuneGridFormatParser.html`

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

1. Create a GridFactory Object

   Get a new GridFactory object by calling

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

2. Enter the Vertices

   Insert the grid vertices by calling

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

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

3. Enter the elements

   For each element call

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

   The parameters are

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

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

4. Parametrized Domains

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

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

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

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

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

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

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

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

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

5. Finish construction

   To finish off the construction of the FooGrid object call

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

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

4.5 Attaching Data to a New Grid

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

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

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

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

4.6 Example: The UnitCube class

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

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

The definition of the general template is as follows.

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef UNITCUBE_HH
#define UNITCUBE_HH

#include <array>
#include <memory>

#include <dune/common/exceptions.hh>
#include <dune/common/fvector.hh>
#include <dune/grid/utility/structuredgridfactory.hh>

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

  static const int dim = GridType::dimension;

  // constructor throwing exception
  UnitCube ()
  {
    Dune::FieldVector<typename GridType::ctype,dim> lowerLeft(0);
    Dune::FieldVector<typename GridType::ctype,dim> upperRight(1);
    std::array<unsigned int,dim> elements;
    std::fill(elements.begin(), elements.end(), 1);

    switch (variant) {
    case 1 :
      grid_ = Dune::StructuredGridFactory<GridType>::createCubeGrid(lowerLeft, upperRight,
          elements);
      break;
    case 2 :
      grid_ = Dune::StructuredGridFactory<GridType>::createSimplexGrid(lowerLeft, upperRight,
          elements);
      break;
    default :
      DUNE_THROW( Dune::NotImplemented, "Variant␣"
                  << variant << "␣of␣unit␣cube␣not␣implemented." );
    }
  }

  T& grid ()
  {
    return *grid_;
  }

private:
  // the constructed grid object
  std::shared_ptr<T> grid_;
};


// include specializations
#include "unitcube_yaspgrid.hh"
#include "unitcube_albertagrid.hh"
#include "unitcube_alugrid.hh"

#endif
```

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

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

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

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 6 (File dune-grid-howto/unitcube_yaspgrid.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef UNITCUBE_YASPGRID_HH
#define UNITCUBE_YASPGRID_HH

#include <array>
#include <memory>

#include <dune/grid/yaspgrid.hh>

#include "unitcube.hh"

// YaspGrid specialization
```

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

  UnitCube ()
  {
    Dune::FieldVector<double,dim> length(1.0);
    std::array<int,dim> elements;
    std::fill(elements.begin(), elements.end(), size);

    grid_ = std::unique_ptr<Dune::YaspGrid<dim> >(new Dune::YaspGrid<dim>(length,elements));
  }

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

private:
  std::unique_ptr<Dune::YaspGrid<dim> > grid_;
};

#endif
```

AlbertaGrid

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

Listing 7 (File dune-grid-howto/unitcube_albertagrid.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef UNITCUBE_ALBERTAGRID_HH
#define UNITCUBE_ALBERTAGRID_HH

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

Listing 8 (File dune-grid-howto/unitcube_alugrid.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef UNITCUBE_ALUGRID_HH
#define UNITCUBE_ALUGRID_HH

#include <array>
#include <memory>

#include "unitcube.hh"

#if HAVE_ALUGRID
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/3d/alu3dgridfactory.hh>
#elif HAVE_DUNE_ALUGRID
#include <dune/alugrid/grid.hh>
#include <dune/alugrid/3d/gridfactory.hh>
#endif

#if HAVE_ALUGRID || HAVE_DUNE_ALUGRID
// ALU3dGrid and ALU2dGrid simplex specialization.
// Note: element type determined by type
template<int dim>
class UnitCube<Dune::ALUGrid<dim,dim,Dune::simplex,Dune::nonconforming>,1>
{
public:
  typedef Dune::ALUGrid<dim,dim,Dune::simplex,Dune::nonconforming> GridType;

private:
  std::shared_ptr<GridType> grid_;

public:
  UnitCube ()
  {
    Dune::FieldVector<typename GridType::ctype,dim> lowerLeft(0);
    Dune::FieldVector<typename GridType::ctype,dim> upperRight(1);
    std::array<unsigned int,dim> elements;
    std::fill(elements.begin(), elements.end(), 1);

    grid_ = Dune::StructuredGridFactory<GridType>::createSimplexGrid(lowerLeft, upperRight,
        elements);
  }

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

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

private:
  std::shared_ptr<GridType> grid_;

public:
  UnitCube ()
  {
    Dune::FieldVector<GridType::ctype,3> lowerLeft(0);
    Dune::FieldVector<GridType::ctype,3> upperRight(1);
    std::array<unsigned int,3> elements = { {1,1,1} };

    grid_ = Dune::StructuredGridFactory<GridType>::createCubeGrid(lowerLeft, upperRight,
        elements);
  }

  GridType &grid ()
  {
    return *grid_;
  }
};
#endif // HAVE_ALUGRID || HAVE_DUNE_ALUGRID

#endif // UNITCUBE_ALUGRID_HH
```

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 `dim` is its dimension.

Listing 9 (dune-grid-howto/functors.hh) Here are some examples for functors.

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_FUNCTORS_HH__
#define __DUNE_GRID_HOWTO_FUNCTORS_HH__

#include <dune/common/fvector.hh>
// 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;
};

#endif // __DUNE_GRID_HOWTO_FUNCTORS_HH__
```

5.3 Integration over a single element

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

Listing 10 (dune-grid-howto/integrateentity.hh)

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

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

Line 27 extracts a reference to a `Dune::QuadratureRule` from the `Dune::QuadratureRules` singleton which is a container containing quadrature rules for all the different reference element types and different orders of approximation. A rule of order p is guaranteed to integrate polynomials of order p exactly, up to floating point errors.

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

5.4 Integration with global error estimation

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

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

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, then the error should be reduced by a factor of $(1/2)^{p+1}$ after each mesh refinement.

Listing 11 (dune-grid-howto/integration.cc)

```cpp
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:

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

#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.leafGridView();

  // 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);

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

    // the GridSelector :: GridType is defined in gridtype.hh and is
    // set during compilation
    typedef GridSelector :: GridType Grid;

    // use unitcube from grids
    std::stringstream dgfFileName;
    dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
      << "unitcube" << Grid::dimension << ".dgf";

    // create grid pointer
    GridPtr<Grid> gridPtr( dgfFileName.str() );

    // integrate and compute error with extrapolation
    uniformintegration( *gridPtr );
  }
  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;
}
```

Running the executable **integration** on a YaspGrid in two space dimensions with a quadrature rule of order one 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, which corresponds to an error decay of order $p + 1$.

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

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

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 storage and run-time). Basically there are two different kinds of mappers.

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

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

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

6.2 Visualization of discrete functions

Let us use mappers to evaluate a function $f : \Omega \to \mathbb{R}$ 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 12 (File dune-grid-howto/elementdata.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_ELEMENT_DATA_HH
#define __DUNE_GRID_HOWTO_ELEMENT_DATA_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
/** This class is only here to show what such a class looks like -- it does
    exactly the same as Dune::MCMGElementLayout. */
template<int dimgrid>
struct P0Layout
{
  bool contains (Dune::GeometryType gt)
  {
    if (gt.dim()==dimgrid) 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;
  typedef typename ElementLeafIterator::Entity::Geometry LeafGeometry;

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

  // 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 leaves
  for (ElementLeafIterator it = gridView.template begin<0>();
       it!=gridView.template end<0>(); ++it)
  {
    // cell geometry
    const LeafGeometry geo = it->geometry();

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

    // evaluate functor and store value
    c[mapper.index(*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::VTK::appendedraw );

  // 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
}

#endif //__DUNE_GRID_HOWTO_ELEMENT_DATA_HH
```

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

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

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

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

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

Listing 13 (File dune-grid-howto/vertexdata.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_VERTEXDATA_HH__
#define __DUNE_GRID_HOWTO_VERTEXDATA_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
/** This class is only here to show what such a class looks like -- it does
    exactly the same as Dune::MCMGVertexLayout. */
template<int dimgrid>
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 from Grid
  const int dim = G::dimension;
  typedef typename G::LeafGridView GridView;
  // determine type of LeafIterator for codimension = dimension
  typedef typename GridView::template Codim<dim>::Iterator VertexLeafIterator;

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

  // 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 leaves
  for (VertexLeafIterator it = gridView.template begin<dim>();
       it!=gridView.template end<dim>(); ++it)
  {
    // evaluate functor and store value
    c[mapper.index(*it)] = f(it->geometry().corner(0));
  }

  // generate a VTK file
  Dune::VTKWriter<typename G::LeafGridView> vtkwriter(grid.leafGridView());
  vtkwriter.addVertexData(c,"data");
  vtkwriter.write( "vertexdata", Dune::VTK::appendedraw );

  // 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
}
#endif // __DUNE_GRID_HOWTO_VERTEXDATA_HH__
```

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

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

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

Listing 14 (File dune-grid-howto/visualization.cc)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:

#include <config.h>
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <dune/common/parallel/mpihelper.hh> // include mpi helper class
#include <dune/grid/io/file/dgfparser/dgfparser.hh>


#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
  {
    if( argc > 1 )
    {
      typedef Dune::GridSelector::GridType DGFGridType;
      // create grid pointer
      Dune :: GridPtr< DGFGridType > gridPtr( argv[ 1 ] );
      dowork( *gridPtr, 3 );
    }

    /*
       UnitCube<Dune::OneDGrid,1> uc0;
     */

       UnitCube<Dune::YaspGrid<dimGrid>,1> uc1;
```

```
        dowork( uc1.grid(), 3 );

     /*
        #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
      */

#if HAVE_ALUGRID || HAVE_DUNE_ALUGRID
    UnitCube< Dune::ALUGrid< dimGrid, dimGrid, Dune::simplex,
            Dune::nonconforming > , 1 > uc5;
    dowork( uc5.grid(), 3 );

#if GRIDDIM == 2 || GRIDDIM == 3
    UnitCube< Dune::ALUGrid< dimGrid, dimGrid, Dune::cube,
            Dune::nonconforming > , 1 > uc6;
    dowork( uc6.grid(), 3 );
#endif // #if GRIDDIM == 2 || 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}):

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

Using integration by parts we arrive at

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

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

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

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

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

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

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

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

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_TRANSPORTPROBLEM_HH__
#define __DUNE_GRID_HOWTO_TRANSPORTPROBLEM_HH__

#include <dune/common/fvector.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;
}
#endif // __DUNE_GRID_HOWTO_TRANSPORTPROBLEM2_HH__
```

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_INITIALIZE_HH__
#define __DUNE_GRID_HOWTO_INITIALIZE_HH__

#include <dune/common/fvector.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;

  // geometry type
  typedef typename LeafIterator::Entity::Geometry Geometry;

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

  // 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
    const Geometry geo = it->geometry();

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

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

#endif // __DUNE_GRID_HOWTO_INITIALIZE_HH__
```

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

Listing 17 (File dune-grid-howto/evolve.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_EVOLVE_HH__
#define __DUNE_GRID_HOWTO_EVOLVE_HH__

#include <dune/common/fvector.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 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;

  // leaf entity geometry
  typedef typename LeafIterator::Entity::Geometry LeafGeometry;

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

  // intersection geometry
  typedef typename IntersectionIterator::Intersection::Geometry IntersectionGeometry;

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

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

  // 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
    const LeafGeometry geo = it->geometry();


    // cell volume, assume linear map here
    double volume = geo.volume();

    // cell index
    int indexi = mapper.index(*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
      const IntersectionGeometry igeo = is->geometry();

      // get normal vector scaled with volume
      Dune::FieldVector<ct,dimworld> integrationOuterNormal
        = is->centerUnitOuterNormal();
      integrationOuterNormal *= igeo.volume();

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

      // evaluate velocity at face center
      Dune::FieldVector<double,dimworld> 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.index(*outside);

        // compute flux from one side only
        if (indexi<indexj)
        {
          // compute factor in neighbor
          const LeafGeometry nbgeo = outside->geometry();
          double nbvolume = nbgeo.volume();
          double nbfactor = velocity*integrationOuterNormal/nbvolume;

          if (factor<0)                           // inflow
          {
            update[indexi] -= c[indexj]*factor;
            update[indexj] += c[indexj]*nbfactor;
          }
          else                           // outflow
          {
            update[indexi] -= c[indexi]*factor;
            update[indexj] += c[indexi]*nbfactor;
          }
        }
      }

      // handle boundary face
      if (is->boundary())
      {
        if (factor<0)                  // inflow, apply boundary condition
          update[indexi] -= b(faceglobal,t)*factor;
        else                  // outflow
          update[indexi] -= c[indexi]*factor;
```

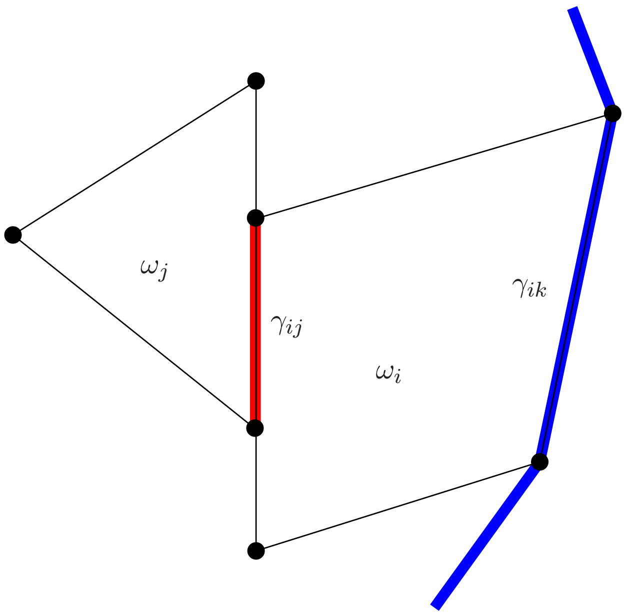


Figure 6.1: Intersection with other elements and the boundary

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

Lines 46-127 contain the loop over all leaf elements where the optimum Δt^n and the cell updates δ_i are computed. The update vector is allocated in line 39, where we assume that `V` is a container with copy constructor and size method.

The computation of the fluxes is done in lines 63-122. An `IntersectionIterator` is used to access all intersections γ_{ij} of a leaf element ω_i. For a full documentation on the `Intersection` class, we refer to the doxygen module page on Intersections[1] An Intersection is with another element ω_j if the `neighbor()` method of the iterator returns true (line 87) or with the external boundary if `boundary()` returns true (line 115), see also Figure 6.1. An intersection γ_{ij} is described by several mappings: (i)

[1]`http://www.dune-project.org/doc/doxygen/html/classDune_1_1IntersectionIterator.html`

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.

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

Finally, line 134 computes the new concentration by adding the scaled update to the current concentration.

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_VTKOUT_HH__
#define __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.leafGridView());
  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::VTK::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();
  }
}

#endif // __DUNE_GRID_HOWTO_VTKOUT_HH__
```

In addition to the snapshots that are produced at each timestep, this function also generates a series file which stores the actual `time` of an evolution scheme together with the snapshots' filenames. After executing the shell script `writePVD` on this series file, we get a <u>P</u>ara<u>v</u>iew 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 19 (File dune-grid-howto/finitevolume.cc)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#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/parallel/mpihelper.hh> // include mpi helper class

#include "vtkout.hh"
#include "transportproblem2.hh"
#include "initialize.hh"
#include "evolve.hh"

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

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

```

```
68 //===============================================================
69 // The main function creates objects and does the time loop
70 //===============================================================
71
72 int main (int argc , char ** argv)
73 {
74   // initialize MPI, finalize is done automatically on exit
75   Dune::MPIHelper::instance(argc,argv);
76
77   // start try/catch block to get error messages from dune
78   try {
79     using namespace Dune;
80
81     // the GridSelector :: GridType is defined in gridtype.hh and is
82     // set during compilation
83     typedef GridSelector :: GridType Grid;
84
85     // use unitcube from dgf grids
86     std::stringstream dgfFileName;
87     dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
88       << "unitcube" << Grid::dimension << ".dgf";
89
90     // create grid pointer
91     GridPtr<Grid> gridPtr( dgfFileName.str() );
92
93     // grid reference
94     Grid& grid = *gridPtr;
95
96     // half grid width 4 times
97     int level = 4 * DGFGridInfo<Grid>::refineStepsForHalf();
98
99     // refine grid until upper limit of level
100    grid.globalRefine(level);
101
102    // do time loop until end time 0.5
103    timeloop(grid, 0.5);
104  }
105  catch (std::exception & e) {
106    std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
107    return 1;
108  }
109  catch (Dune::Exception & e) {
110    std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
111    return 1;
112  }
113  catch (...) {
114    std::cout << "Unknown␣ERROR" << std::endl;
115    return 1;
116  }
117
118  // done
119  return 0;
120 }
```

The function `timeloop` constructs a mapper and allocates the concentration vector with one entry per element in the leaf grid. In line 30 this vector is initialized with the initial concentration and the loop in line 40-65 evolves the concentration in time. Every time the current time crosses a multiple of `saveStep` = 0.1, the simulation result is written to a file in line 55.

6.4 A FEM example: The Poisson equation

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

$$-\Delta u = f \quad in\ \Omega \tag{6.15}$$
$$u = 0 \quad on\ \partial\Omega \tag{6.16}$$

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

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

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

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

(i) $\bigcup_{\Delta\in\mathcal{T}} \overline{\Delta} = \overline{\Omega}$

(ii) $\Delta_i \cap \Delta_j\ i \neq j$ is an entity of higher codimension of the elements Δ_i, Δ_j

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

$$V_h = \{u \in C(\bar{\Omega}) \;\Big|\; u\big|_\Delta \in \mathcal{P}_1(\Delta)\ \forall \Delta \in \mathcal{T}\} \tag{6.18}$$

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

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

$$N_0(x) = 1 - \sum_{i=1}^{d} x_i \tag{6.19}$$
$$N_i(x) = x_i \quad i = 1, ..., d \tag{6.20}$$

to evaluate the basis functions and their gradients.

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

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

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

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

$$\begin{array}{cc} & \begin{array}{ccccccc} & & & i\downarrow & & & \end{array} \\ \to i & \begin{pmatrix} \star & \star & \star & \star & \star & \star & \star \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ \star & \star & \star & \star & \star & \star & \star \end{pmatrix} \end{array} \begin{pmatrix} \star \\ u_i \\ \star \end{pmatrix} = \begin{pmatrix} \star \\ 0 \\ \star \end{pmatrix}$$

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

$$\begin{aligned} A_{ij} &= \int\limits_{\Omega} \nabla\phi_i \cdot \nabla\phi_j \; dx && (6.22) \\ &= \sum_{\Delta\in\mathcal{T}} \int\limits_{\Delta} \nabla\phi_i \cdot \nabla\phi_j \; dx && (6.23) \\ &= \sum_{\Delta\in\mathcal{T}} \int\limits_{\hat\Delta} \nabla\phi_i(g(\hat x)) \cdot \nabla\phi_j(g(\hat x))\mu(\hat x) d\hat x && (6.24) \\ &= \sum_{\Delta\in\mathcal{T}} \int\limits_{\hat\Delta} (J_g^{-T}\hat\nabla\hat\phi_i)(\hat x) \cdot (J_g^{-T}\hat\nabla\hat\phi_j)(\hat x)\mu(\hat x) d\hat x && (6.25) \end{aligned}$$

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

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

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

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

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

After assembling the matrix we implement the Dirichlet boundary conditions by overwriting the lines of the equation system associated with boundary nodes with trivial lines, see figure 6.2.

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

6.4.1 Implementation

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

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef SHAPEFUNCTIONS_HH
#define SHAPEFUNCTIONS_HH

#include <dune/common/fvector.hh>

// LinearShapeFunction:
// represents a shape function and provides methods to evaluate the function
// and its gradient
template<class ctype, class rtype, int dim>
class LinearShapeFunction
{
public:
  enum { dimension = dim };

  LinearShapeFunction() : coeff0(0.0), coeff1(0.0) {}

  LinearShapeFunction(rtype coeff0_, const Dune::FieldVector<rtype,dim>& coeff1_)
    : coeff0(coeff0_), coeff1(coeff1_) {}

  void setCoeff(rtype coeff0_, const Dune::FieldVector<rtype,dim>& coeff1_)
  {
    coeff0 = coeff0_;
    coeff1 = coeff1_;
  }

  rtype evaluateFunction(const Dune::FieldVector<ctype,dim>& local) const
  {
    rtype result = coeff0;
    for (int i = 0; i < dim; ++i)
      result += coeff1[i] * local[i];
    return result;
  }

  Dune::FieldVector<rtype,dim>
  evaluateGradient(const Dune::FieldVector<ctype,dim>& local) const
  {
    return coeff1;
  }

private:
  rtype coeff0;
  Dune::FieldVector<rtype,dim> coeff1;
};

// P1ShapeFunctionSet
// initializes one and only one set of LinearShapeFunction
template<class ctype, class rtype, int dim>
class P1ShapeFunctionSet
{
public:
  enum { n = dim + 1 };

  typedef LinearShapeFunction<ctype,rtype,dim> ShapeFunction;
  typedef rtype resulttype;

  // get the only instance of this class
  static const P1ShapeFunctionSet& instance()
  {
    static const P1ShapeFunctionSet sfs;
    return sfs;
  }

  const ShapeFunction& operator[](int i) const
  {
    if (!i)
      return f0;
    else
      return f1[i - 1];
  }

private:
  // private constructor prevents additional instances
  P1ShapeFunctionSet()
  {
    Dune::FieldVector<rtype,dim> e(-1.0);
    f0.setCoeff(1.0, e);
    for (int i = 0; i < dim; ++i)
    {
      Dune::FieldVector<rtype,dim> e(0.0);
      e[i] = 1.0;
      f1[i].setCoeff(0.0, e);
    }
  }

  P1ShapeFunctionSet(const P1ShapeFunctionSet& other)
  {}

  ShapeFunction f0;
  ShapeFunction f1[dim];
};

#endif
```

And now the actual FEM code:

Listing 21 (File dune-grid-howto/finiteelements.cc)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#include <config.h>
#include <iostream>
#include <vector>
#include <set>
#include <dune/common/fvector.hh>
#include <dune/common/fmatrix.hh>
#include <dune/geometry/quadraturerules.hh>
#include <dune/grid/io/file/vtk/vtkwriter.hh>
#include <dune/grid/albertagrid.hh>

#if HAVE_DUNE_ISTL
#include <dune/istl/bvector.hh>
#include <dune/istl/bcrsmatrix.hh>
#include <dune/istl/ilu.hh>
#include <dune/istl/operators.hh>
#include <dune/istl/solvers.hh>
#include <dune/istl/preconditioners.hh>
#include <dune/istl/io.hh>
#else
#include <dune/common/dynvector.hh>
```

```
#include <dune/common/dynmatrix.hh>
#endif // HAVE_DUNE_ISTL

#include "shapefunctions.hh"

// P1Elements:
// a P1 finite element discretization for elliptic problems Dirichlet
// boundary conditions on simplicial conforming grids
template<class GV, class F>
class P1Elements
{
public:
  static const int dim = GV::dimension;

  typedef typename GV::ctype ctype;
#if HAVE_DUNE_ISTL
  typedef Dune::BCRSMatrix<Dune::FieldMatrix<ctype,1,1> > Matrix;
  typedef Dune::BlockVector<Dune::FieldVector<ctype,1> > ScalarField;
#else
  typedef Dune::DynamicMatrix<ctype> Matrix;
  typedef Dune::DynamicVector<ctype> ScalarField;
#endif // HAVE_DUNE_ISTL

private:
  typedef typename GV::template Codim<0>::Iterator LeafIterator;
  typedef typename GV::template Codim<0>::Geometry::JacobianInverseTransposed
       JacobianInverseTransposed;
  typedef typename GV::IntersectionIterator IntersectionIterator;
  typedef typename GV::IndexSet LeafIndexSet;

  const GV& gv;
  const F& f;

public:
  Matrix A;
  ScalarField b;
  ScalarField u;
  std::vector< std::set<int> > adjacencyPattern;

  P1Elements(const GV& gv_, const F& f_) : gv(gv_), f(f_) {}

  // store adjacency information in a vector of sets
  void determineAdjacencyPattern();

  // assemble stiffness matrix A and right side b
  void assemble();

  // finally solve Au = b for u
  void solve();
};

template<class GV, class F>
void P1Elements<GV, F>::determineAdjacencyPattern()
{
  const int N = gv.size(dim);
  adjacencyPattern.resize(N);

  const LeafIndexSet& set = gv.indexSet();
  const LeafIterator itend = gv.template end<0>();

  for (LeafIterator it = gv.template begin<0>(); it != itend; ++it)
  {
    Dune::GeometryType gt = it->type();
    const Dune::template ReferenceElement<ctype,dim> &ref =
      Dune::ReferenceElements<ctype,dim>::general(gt);

    // traverse all codim-1-entities of the current element and store all
    // pairs of vertices in adjacencyPattern
    const IntersectionIterator isend = gv.iend(*it);
    for (IntersectionIterator is = gv.ibegin(*it) ; is != isend ; ++is)
    {
      int vertexsize = ref.size(is->indexInInside(),1,dim);
      for (int i=0; i < vertexsize; i++)
      {
        int indexi = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,i,dim),dim);
        for (int j=0; j < vertexsize; j++)
        {
          int indexj = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,j,dim),dim);
          adjacencyPattern[indexi].insert(indexj);
        }
      }
    }
  }
}

template<class GV, class F>
void P1Elements<GV, F>::assemble()
{
  const int N = gv.size(dim);

  const LeafIndexSet& set = gv.indexSet();
  const LeafIterator itend = gv.template end<0>();

  // set sizes of A and b
#if HAVE_DUNE_ISTL
  A.setSize(N, N, N + 2*gv.size(dim-1));
  A.setBuildMode(Matrix::random);
  b.resize(N, false);

  for (int i = 0; i < N; i++)
    A.setrowsize(i,adjacencyPattern[i].size());
  A.endrowsizes();

  // set sparsity pattern of A with the information gained in determineAdjacencyPattern
  for (int i = 0; i < N; i++)
  {
    std::template set<int>::iterator setend = adjacencyPattern[i].end();
    for (std::template set<int>::iterator setit = adjacencyPattern[i].begin();
         setit != setend; ++setit)
      A.addindex(i,*setit);
  }

  A.endindices();
#else
  A.resize(N, N);
  b.resize(N);
#endif // HAVE_DUNE_ISTL

  // initialize A and b
  A = 0.0;
  b = 0.0;

  // get a set of P1 shape functions
  const P1ShapeFunctionSet<ctype,ctype,dim>& basis = P1ShapeFunctionSet<ctype,ctype,dim>::
      instance();

```

```
for (LeafIterator it = gv.template begin<0>(); it != itend; ++it)
{
  // determine geometry type of the current element and get the matching reference element
  Dune::GeometryType gt = it->type();
  const Dune::template ReferenceElement<ctype,dim> &ref =
    Dune::ReferenceElements<ctype,dim>::general(gt);
  int vertexsize = ref.size(dim);

  // get a quadrature rule of order one for the given geometry type
  const Dune::QuadratureRule<ctype,dim>& rule = Dune::QuadratureRules<ctype,dim>::rule(gt,1);
  for (typename Dune::QuadratureRule<ctype,dim>::const_iterator r = rule.begin();
       r != rule.end() ; ++r)
  {
    // compute the jacobian inverse transposed to transform the gradients
    JacobianInverseTransposed jacInvTra =
      it->geometry().jacobianInverseTransposed(r->position());

    // get the weight at the current quadrature point
    ctype weight = r->weight();

    // compute Jacobian determinant for the transformation formula
    ctype detjac = it->geometry().integrationElement(r->position());
    for (int i = 0; i < vertexsize; i++)
    {
      // compute transformed gradients
      Dune::FieldVector<ctype,dim> grad1;
      jacInvTra.mv(basis[i].evaluateGradient(r->position()),grad1);
      for (int j = 0; j < vertexsize; j++)
      {
        Dune::FieldVector<ctype,dim> grad2;
        jacInvTra.mv(basis[j].evaluateGradient(r->position()),grad2);

        // gain global inidices of vertices i and j and update associated matrix entry
        A[set.subIndex(*it,i,dim)][set.subIndex(*it,j,dim)]
          += (grad1*grad2) * weight * detjac;
      }
    }
  }

  // get a quadrature rule of order two for the given geometry type
  const Dune::QuadratureRule<ctype,dim>& rule2 = Dune::QuadratureRules<ctype,dim>::rule(gt,2)
      ;
  for (typename Dune::QuadratureRule<ctype,dim>::const_iterator r = rule2.begin();
       r != rule2.end() ; ++r)
  {
    ctype weight = r->weight();
    ctype detjac = it->geometry().integrationElement(r->position());
    for (int i = 0 ; i<vertexsize; i++)
    {
      // evaluate the integrand of the right side
      ctype fval = basis[i].evaluateFunction(r->position())
                   * f(it->geometry().global(r->position())) ;
      b[set.subIndex(*it,i,dim)] += fval * weight * detjac;
    }
  }
}

// Dirichlet boundary conditions:
// replace lines in A related to Dirichlet vertices by trivial lines
for ( LeafIterator it = gv.template begin<0>() ; it != itend ; ++it)
{
  const IntersectionIterator isend = gv.iend(*it);
  for (IntersectionIterator is = gv.ibegin(*it) ; is != isend ; ++is)
```

```
    {
      // determine geometry type of the current element and get the matching reference element
      Dune::GeometryType gt = it->type();
      const Dune::template ReferenceElement<ctype,dim> &ref =
        Dune::ReferenceElements<ctype,dim>::general(gt);

      // check whether current intersection is on the boundary
      if ( is->boundary() )
      {
        // traverse all vertices the intersection consists of
        for (int i=0; i < ref.size(is->indexInInside(),1,dim); i++)
        {
          // and replace the associated line of A and b with a trivial one
          int indexi = set.subIndex(*it,ref.subEntity(is->indexInInside(),1,i,dim),dim);

          A[indexi] = 0.0;
          A[indexi][indexi] = 1.0;
          b[indexi] = 0.0;
        }
      }
    }
  }
}

#if HAVE_DUNE_ISTL
template<class GV, class E>
void P1Elements<GV, E>::solve()
{
  // make linear operator from A
  Dune::MatrixAdapter<Matrix,ScalarField,ScalarField> op(A);

  // initialize preconditioner
  Dune::SeqILUn<Matrix,ScalarField,ScalarField> ilu1(A, 1, 0.92);

  // the inverse operator
  Dune::BiCGSTABSolver<ScalarField> bcgs(op, ilu1, 1e-15, 5000, 0);
  Dune::InverseOperatorResult r;

  // initialize u to some arbitrary value to avoid u being the exact
  // solution
  u.resize(b.N(), false);
  u = 2.0;

  // finally solve the system
  bcgs.apply(u, b, r);
}
#endif // HAVE_DUNE_ISTL

// an example right hand side function
template<class ctype, int dim>
class Bump {
public:
  ctype operator() (Dune::FieldVector<ctype,dim> x) const
  {
    ctype result = 0;
    for (int i=0 ; i < dim ; i++)
      result += 2.0 * x[i]* (1-x[i]);
    return result;
  }
};

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

```
272 #if HAVE_ALBERTA && ALBERTA_DIM==2
273   static const int dim = 2;
274   const char* gridfile = "grids/2dgrid.al";
275 
276   typedef Dune::AlbertaGrid<dim,dim> GridType;
277   typedef GridType::LeafGridView GV;
278 
279   typedef GridType::ctype ctype;
280   typedef Bump<ctype,dim> Func;
281 
282   GridType grid(gridfile);
283   const GV& gv = grid.leafGridView();
284 
285   Func f;
286   P1Elements<GV,Func> p1(gv, f);
287 
288 #if HAVE_DUNE_ISTL
289   grid.globalRefine(16);
290 #else
291   grid.globalRefine(10);
292 #endif // HAVE_DUNE_ISTL
293 
294   std::cout << "-----------------------------------" << "\n";
295   std::cout << "number␣of␣unknowns:␣" << grid.size(dim) << "\n";
296 
297   std::cout << "determine␣adjacency␣pattern..." << "\n";
298   p1.determineAdjacencyPattern();
299 
300   std::cout << "assembling..." << "\n";
301   p1.assemble();
302 
303 #if HAVE_DUNE_ISTL
304   std::cout << "solving..." << "\n";
305   p1.solve();
306 
307   std::cout << "visualizing..." << "\n";
308   Dune::VTKWriter<GridType::LeafGridView> vtkwriter(grid.leafGridView());
309   vtkwriter.addVertexData(p1.u, "u");
310   vtkwriter.write("fem2d", Dune::VTK::appendedraw);
311 #else
312   std::cout << "for␣solving␣and␣visualizing␣dune-istl␣is␣necessary." << "\n";
313 #endif // HAVE_DUNE_ISTL
314 #else
315   std::cerr << "You␣need␣Alberta␣in␣2d␣for␣this␣program." << std::endl;
316 #endif // HAVE_ALBERTA && ALBERTA_DIM==2
317 }
```

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

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

As already said above, we do directly implement Dirichlet boundaries into our matrix. This is done

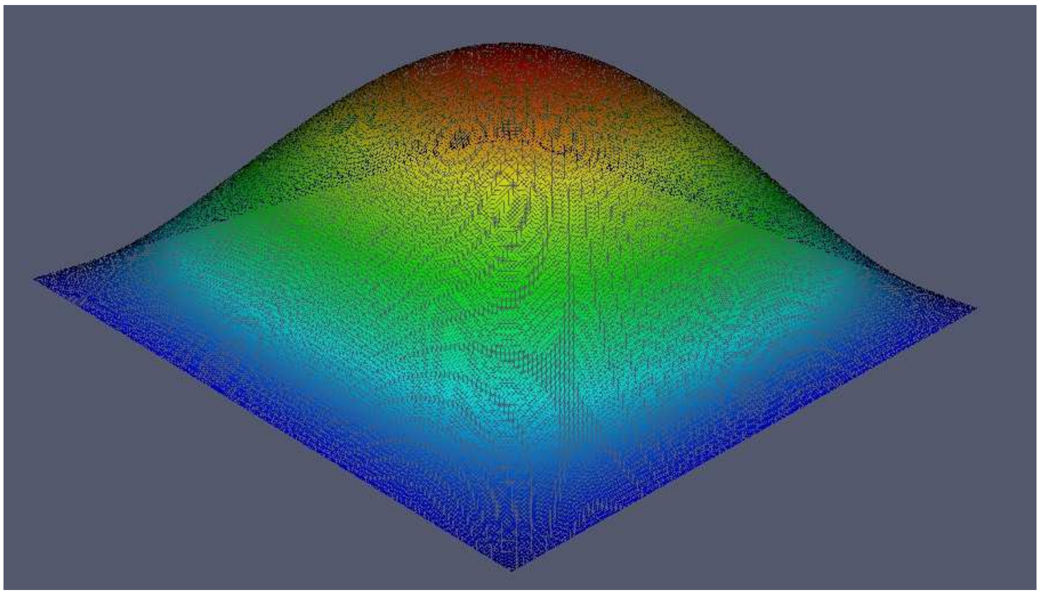


Figure 6.3: Solution in 2D

in lines 205 to 230. We have to traverse the whole grid once again and check for each intersection of elements whether it is on the boundary. In line 224 we overwrite the line corresponding to a node on the boundary as shown in figure 6.2.

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

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

```
make ALBERTA_DIM=3
```

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

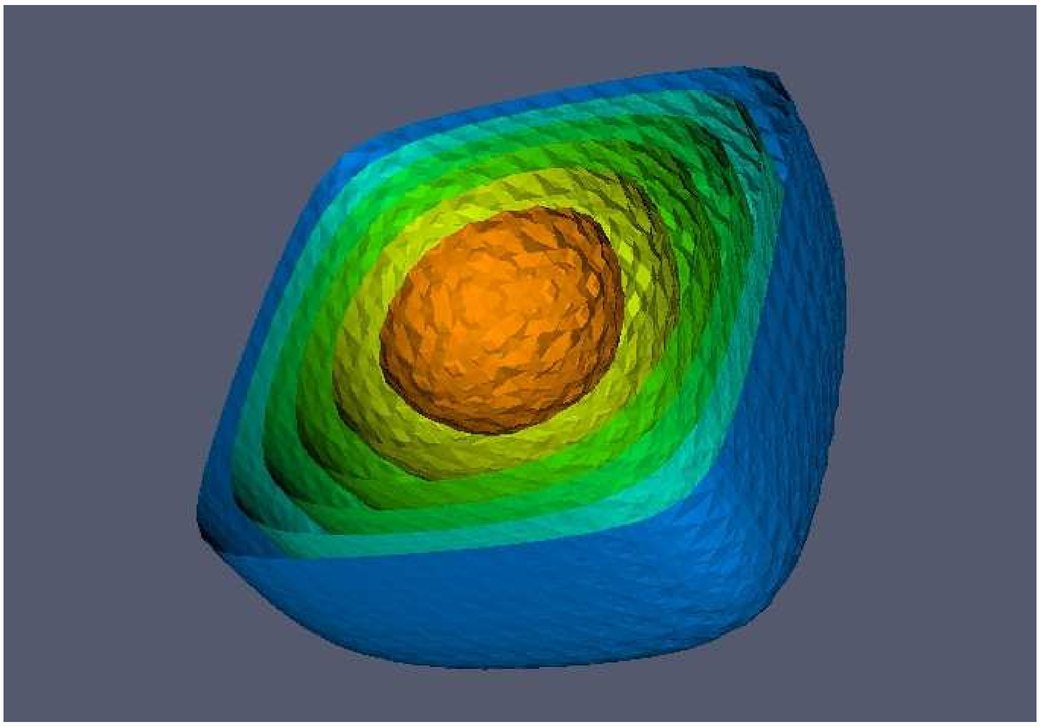


Figure 6.4: Solution in 3D

7 Adaptivity

7.1 Adaptive integration

7.1.1 Adaptive multigrid integration

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

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

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

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:

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

#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.leafGridView();

  // 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);
    }

    // adapt the mesh
    grid.preAdapt();
```

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

The work is done in the function `adaptiveintegration`. Lines 37-40 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 55. 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 62). Now the refinement threshold

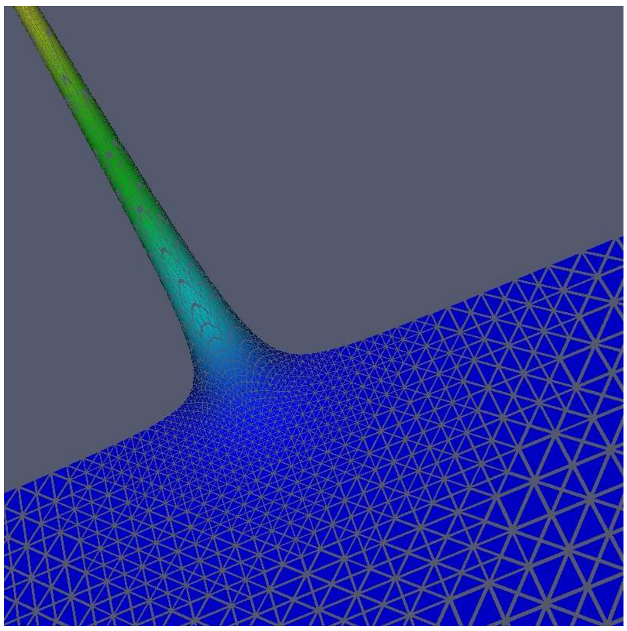
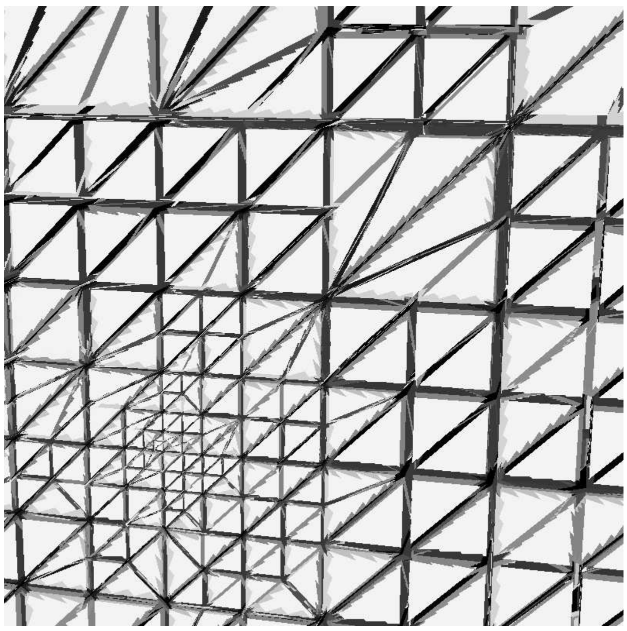



Figure 7.1: Two and three-dimensional grids generated by the adaptive integration algorithm applied to the needle pulse. Left grid is generated using Alberta, right grid is generated using UG.

κ can be computed in lines 67-89. Finally the last loop in lines 92-99 marks elements for refinement and lines 102-104 actually do the refinement. The reason for dividing refinement into three functions `preAdapt()`, `adapt()` and `postAdapt()` will be explained with the next example. Note the flexibility of this algorithm: It runs in any space dimension on any kind of grid and different integration orders can easily be incorporated. And that with just about 100 lines of code including comments.

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

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

7.2 Adaptive cell centered finite volumes

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

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

- Compute global maximum and minimum of element concentrations:

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

- 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 23 (File dune-grid-howto/finitevolumeadapt.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_FINITEVOLUMEADAPT_HH__
#define __DUNE_GRID_HOWTO_FINITEVOLUMEADAPT_HH__

#include <cmath>
#include <dune/grid/utility/persistentcontainer.hh>

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;

  // 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;

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

  // 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.index(*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.index(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.index(*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.index(*it)] < coarsentol*globaldelta && it->level() > lmin)
    {
      grid.mark( -1, *it );
      ++marked;
    }
  }
  if( marked==0 )
    return false;

  grid.preAdapt();

  typedef Dune::PersistentContainer<G,RestrictedValue> RestrictionMap;
  RestrictionMap restrictionmap(grid,0); // restricted concentration

  for (int level=grid.maxLevel(); level>=0; level--)
  {
    // get grid view on level grid
    LevelGridView levelView = grid.levelGridView(level);
    for (LevelIterator it = levelView.template begin<0>();
         it!=levelView.template end<0>(); ++it)
    {
      // get your map entry
      RestrictedValue& rv = restrictionmap[*it];
      // put your value in the map
      if (it->isLeaf())
      {
        int indexi = mapper.index(*it);
        rv.value = c[indexi];
        rv.count = 1;
      }

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

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

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

The loop in lines 49-80 computes the indicator values η_i as well as the global minimum and maximum $\overline{C}, \underline{C}$. Then the next loop in lines 85-112 marks the elements for refinement. Lines 119-146 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 150 by calling the `adapt()` method on the grid object. The mapper is updated to reflect the changes in the grid in line 151 and the concentration vector is resized to the new size in line 153. 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 156-194.

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

Listing 24 (File dune-grid-howto/adativefinitevolume.cc)

```cpp
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#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/parallel/mpihelper.hh> // include mpi helper class

#include "vtkout.hh"
#include "transportproblem2.hh"
#include "initialize.hh"
#include "evolve.hh"
#include "finitevolumeadapt.hh"

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

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,Dune::MCMGElementLayout>
  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 = t + 0.1;
  int counter = 1;
  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
    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;

    // the GridSelector :: GridType is defined in gridtype.hh and is
    // set during compilation
    typedef GridSelector :: GridType Grid;

    // use unitcube from grids
    std::stringstream dgfFileName;
    dgfFileName << DUNE_GRID_HOWTO_EXAMPLE_GRIDS_PATH
      << "unitcube" << Grid::dimension << ".dgf";

    // create grid pointer
    GridPtr<Grid> gridPtr( dgfFileName.str() );

    // grid reference
    Grid& grid = *gridPtr;

    // minimal allowed level during refinement
    int minLevel = 2 * DGFGridInfo<Grid>::refineStepsForHalf();

    // refine grid until upper limit of level
```

```
119      grid.globalRefine(minLevel);
120
121      // maximal allowed level during refinement
122      int maxLevel = minLevel + 3 * DGFGridInfo<Grid>::refineStepsForHalf();
123
124      // do time loop until end time 0.5
125      timeloop(grid, 0.5, minLevel, maxLevel);
126    }
127    catch (std::exception & e) {
128      std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
129      return 1;
130    }
131    catch (Dune::Exception & e) {
132      std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
133      return 1;
134    }
135    catch (...) {
136      std::cout << "Unknown␣ERROR" << std::endl;
137      return 1;
138    }
139
140    // done
141    return 0;
142  }
```

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

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

8 Parallelism

8.1 DUNE Data Decomposition Model

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

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

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

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

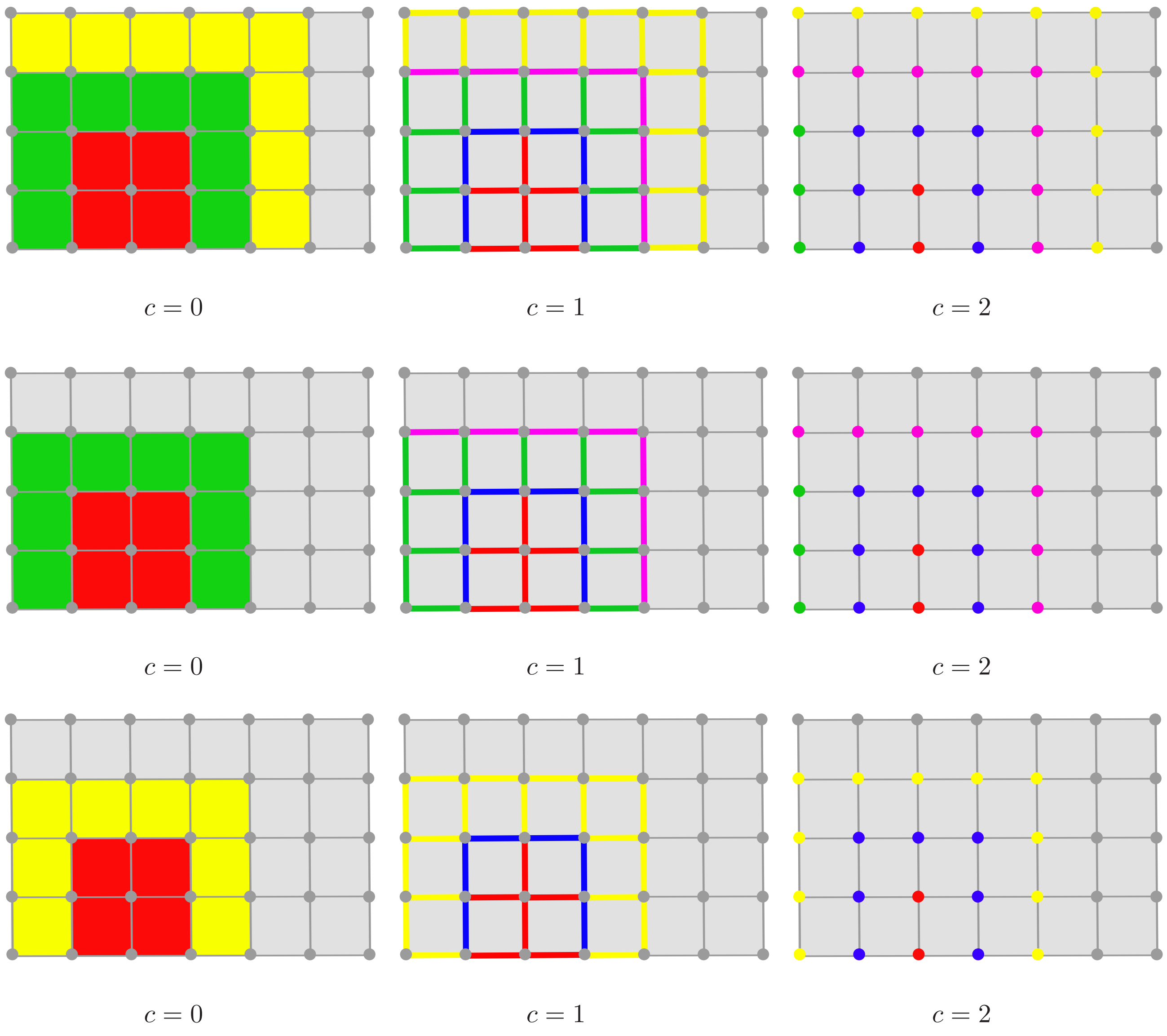


Figure 8.1: Color coded illustration of different data decompositions: interior (red), border (blue), overlap (green), front (magenta) and ghost (yellow), gray encodes entities not stored by the process. First row shows case with interior, overlap and ghost entities, second row shows a case with interior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

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

8.2 Communication Interfaces

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

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

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

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

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__
#define __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__

#include <dune/grid/common/datahandleif.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.index(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.index(e)]=x;
  }

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

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

#endif // __DUNE_GRID_HOWTO_PARFVDATAHANDLE_HH__
```

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

8.3 Parallel finite volume scheme

In this section we parallelize the (nonadaptive!) cell centered finite volume scheme. Essentially only the `evolve` method has to be parallelized. The following listing shows the parallel version of this method. Compare this with listing 17 on page 47.

Listing 26 (File dune-grid-howto/parevolve.hh)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#ifndef __DUNE_GRID_HOWTO_PAREVOLVE_HH__
#define __DUNE_GRID_HOWTO_PAREVOLVE_HH__

#include "parfvdatahandle.hh"

#include <dune/grid/common/gridenums.hh>
#include <dune/common/fvector.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;

  // leaf entity geometry
  typedef typename LeafIterator::Entity::Geometry LeafGeometry;

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

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

  // intersection geometry
  typedef typename Intersection::Geometry IntersectionGeometry;

  // 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.leafGridView();

// 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
  const LeafGeometry geo = it->geometry();

  // cell volume
  double volume = geo.volume();

  // cell index
  int indexi = mapper.index(*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
    const IntersectionGeometry igeo = intersection.geometry();

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

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

    // 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.index(*outside);

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

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

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

In order to iterate over entities with a specific partiton type the leaf and level iterators can be parametrized by an additional argument `PartitionIteratorType` as shown in line 29. If the argument `All_Partition` is given then all entities are processed, regardless of their partition type. This is also the default behavior of the level and leaf iterators. If the partition iterator type is specified 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 58-59.

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

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

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

Listing 27 (File dune-grid-howto/parfinitevolume.cc)

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
// vi: set et ts=4 sw=2 sts=2:
#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/parallel/mpihelper.hh> // include mpi helper class


// checks for defined gridtype and inlcudes appropriate dgfparser implementation
#include "vtkout.hh"
#include "unitcube.hh"
#include "transportproblem2.hh"
#include "initialize.hh"
#include "parfvdatahandle.hh"
#include "parevolve.hh"


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

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

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

  // initialize concentration with initial values
  initialize(grid,mapper,c);
  vtkout(grid,c,"pconc",0,0.0,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

    //  ALUGRID supports parallelization in 3 dimensions only
#if HAVE_ALUGRID
    //     typedef Dune::ALUGrid< 3, 3, Dune::cube, Dune::nonconforming > GridType;
    //     typedef Dune::ALUGrid< 3, 3, Dune::simplex, Dune::nonconforming > GridType;
    //     UnitCube< GridType , 1 > uc2;
#endif
106
107 #ifdef LOAD_BALANCING
108
109     // refine grid until upper limit of level
110     uc2.grid().globalRefine( 6 );
111
112     // re-partition grid for better load balancing
113     uc2.grid().loadBalance();
114
115     // do time loop until end time 0.5
116     partimeloop(uc2.grid(), 0.5);
117 #endif
118
119   }
120   catch (std::exception & e) {
121     std::cout << "STL␣ERROR:␣" << e.what() << std::endl;
122     return 1;
123   }
124   catch (Dune::Exception & e) {
125     std::cout << "DUNE␣ERROR:␣" << e.what() << std::endl;
126     return 1;
127   }
128   catch (...) {
129     std::cout << "Unknown␣ERROR" << std::endl;
130     return 1;
131   }
132
133   // done
134   return 0;
135 }
```

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

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

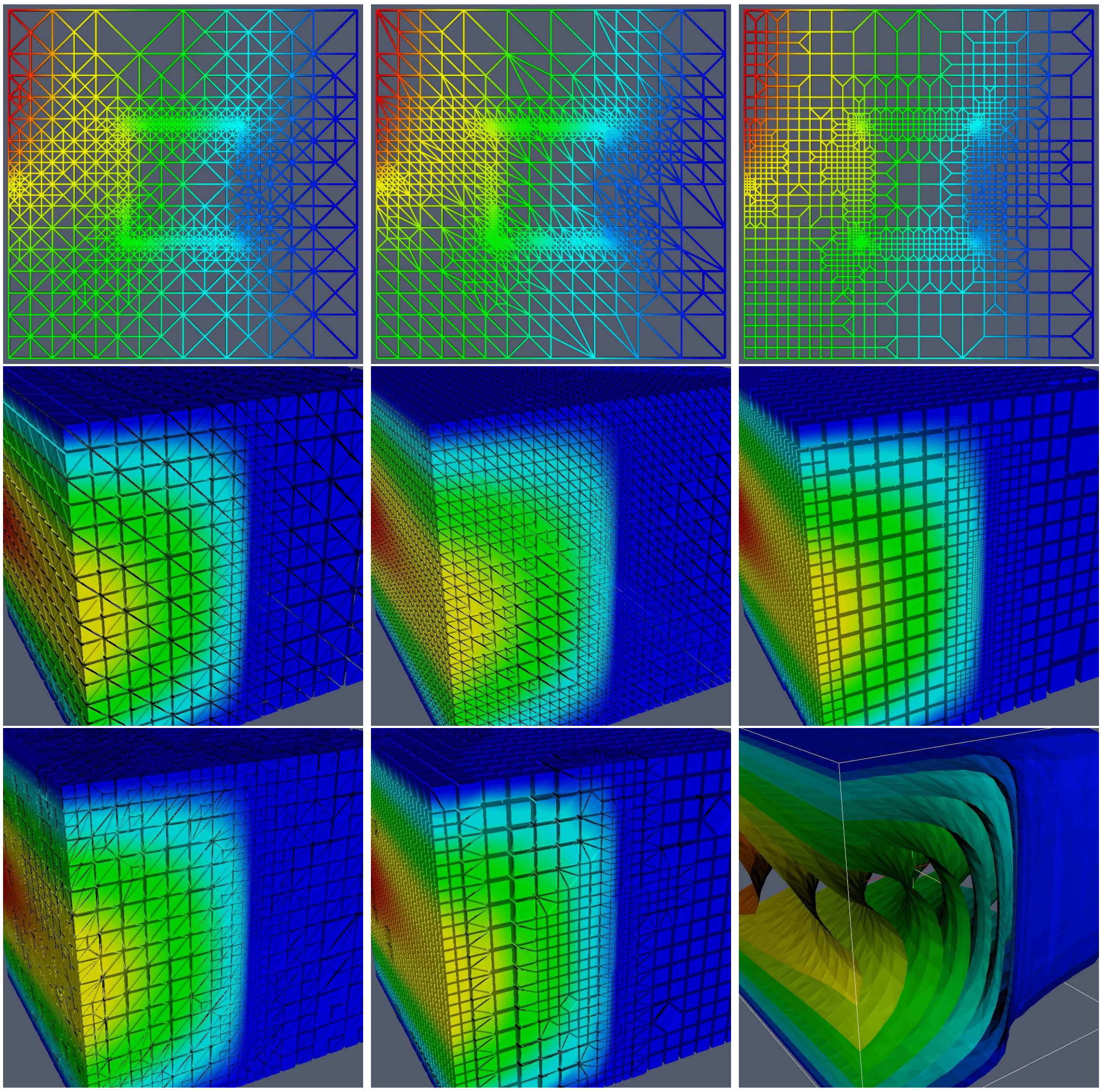


Figure 8.2: Adaptive solution of an elliptic model problem with P_1 conforming finite elements and residual based error estimator. Illustrates that adaptive finite element algorithm can be formulated independent of dimension, element type and refinement scheme. From top to bottom, left to right: Alberta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

Bibliography

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

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

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

[4] P. 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.