Getting started with DUNE

Oliver Sander*

May 10, 2016

This document describes how you can get started with using the Distributed and Unified Numerics Environment (DUNE). DUNE is a set of open-source C++ libraries that help you implement finite element and finite volume methods. The intended audience consists of people that roughly know how such methods work, know a bit of C++ programming, and want to start using DUNE as their basis for their own simulation codes. The text explains how to install DUNE, and how to set up your first own project. It then presents two complete example simulation programs, one using the finite element method to solve the Poisson equation, and another one using the finite volume method for a simple transport equation.

This document is actually one chapter of an entire book on the DUNE system. This book is still in the process of being written, and will be published by Springer when it is ready. Springer kindly consented to having this chapter released to the public before the release of the entire book.

*Institut für Numerische Mathematik, Technische Universität Dresden, Germany, email: oliver.sander@tu-dresden.de

Contents

1 Installation of Dune **3**
- 1.1 Installation from Binary Packages . . . 3
- 1.2 Installation from Source . . . 3

2 A First Dune Application **5**
- 2.1 Creating a New Module . . . 6
- 2.2 Testing the New Module . . . 7

3 Example: Solving the Poisson Equation Using Finite Elements **8**
- 3.1 The Main Program . . . 9
  - 3.1.1 Creating a Grid . . . 10
  - 3.1.2 Assembling the Stiffness Matrix and Load Vector . . . 11
  - 3.1.3 Incorporating the Dirichlet Boundary Conditions . . . 12
  - 3.1.4 Solving the Algebraic Problem . . . 13
  - 3.1.5 Outputting the Result . . . 14
  - 3.1.6 Running the Program . . . 14
- 3.2 Assembling the Stiffness Matrix . . . 15
  - 3.2.1 The Global Assembler . . . 16
  - 3.2.2 The Element Assembler . . . 17

4 Example: Solving the Transport Equation with a Finite Volume Method **20**
- 4.1 Discrete linear transport equation . . . 20
- 4.2 The Main Program . . . 22
- 4.3 The method `evolve` . . . 25

5 Complete source code of the finite element example **29**

6 Complete source code of the finite volume example **33**

GNU Free Documentation License **37**

References **44**

Copyright © 2012–2016 Oliver Sander
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Dune is a set of C++ libraries for the implementation of finite element and finite volume methods. It is available under the GNU General Public License (Version 2) with the linking exception,[1] which is the same license as the GNU C++ standard library.

The main public representation of Dune is its project homepage at `www.dune-project.org`. There you will find the latest releases, class documentations, general information, and ways to contact the Dune developers and users.

Many aspects of Dune have been published in scientific articles. We mention [4, 5] on the Dune grid interface, [2, 6, 7] on linear algebra, [10] on discrete functions, and [12] on the `dune-typetree` library. The most recent release at the time of writing, Dune 2.4, has been presented in [8].

1 Installation of Dune

As the first step into the Dune world you will have to install it. We have tried to make this as painless as possible. The following instructions assume that you have a Unix-style command shell and toolchain installed on your computer. This will be easy to obtain on all flavors of Linux, Apple OS X, and even on Windows if you have the cygwin environment[2] installed.

To run the examples in this text you will need seven Dune modules: `dune-common`, `dune-geometry`, `dune-grid`, `dune-istl`, `dune-localfunctions`, `dune-typetree`, and `dune-functions`.

1.1 Installation from Binary Packages

Installation is easiest if there are precompiled packages for your system. At the time of writing this is the case for Debian, Ubuntu and openSUSE, but there may be more. Check the current status for your system. On a Debian-type system, type

```
apt-get install libdune-functions-dev libdune-istl-dev
```

as root, to install all those seven Dune libraries. The ones not listed here are pulled in automatically.

If you are on openSUSE, type

```
zypper install dune-functions-devel dune-istl-devel
```

instead (again as root). The Dune modules are then installed globally on your machine, and the Dune build system will find them.

1.2 Installation from Source

If there are no precompiled packages available for your system, or if you do not have root access on your machine, then you need to install Dune from the source code. First

[1] `www.dune-project.org/license.html`
[2] `http://cygwin.com/`

you need to download the code from the DUNE website at www.dune-project.org. We recommend downloading the release tarballs, named

```
dune-modulename-X.Y.Z.tar.gz
```

where modulename is one of common, geometry, grid, localfunctions, istl, and X.Y.Z is the release version number. The code examples in this book require at least version 2.4.1. Release tarballs for dune-typetree are available from http://www.dune-project.org/pdelab/. Check for release tarballs for dune-functions at http://www.dune-project.org/modules/dune-functions/. At the time of writing there was no 2.4 release of dune-functions, but you can use the 2.4-compatible branch from the git tree instead.

If you are brave and/or you need very recent features you can also get the development branches of the DUNE modules by anonymous git access. The repositories are at https://gitlab.dune-project.org. To clone (i.e., download) the source code for one module type

```
git clone https://gitlab.dune-project.org/core/dune-modulename.git
```

where again you need to replace modulename by common, geometry, grid, localfunctions, or istl. The module dune-typetree is available by[3]

```
git clone https://gitlab.dune-project.org/pdelab/dune-typetree.git
```

and dune-functions by

```
git clone https://gitlab.dune-project.org/staging/dune-functions.git
```

Out-of-the-box, DUNE currently only provides structured grids. To get all the power of DUNE with an unstructured grid you need to install the UG-3 library [3]. First check whether it is available from your system package manager (for example, it is called libug-dev on Debian systems). Otherwise, download the code from www.iwr.uni-heidelberg.de/frame/iwrwikiequipment/software/ug, and build and install it following the instructions given there. If you have it installed in a standard location, the DUNE build system will find it automatically. If you are on a Debian-based system then UG will be installed together with dune-grid.

As an alternative to UG you may also install the ALBERTA [14] or ALUGRID grid managers [9, 15]. UG, ALBERTA and ALUGRID all have their strength and weaknesses.

Let us assume that you have created an empty directory called dune, and downloaded the sources of the seven DUNE modules into this directory. Suppose further that you have unpacked the tarballs (provided that you are using the tarballs) by

[3]The sources for the dune-typetree module are scheduled to move into the staging namespace eventually. When that has happened, the git path for dune-typetree will be the same as the one for dune-functions.

```
tar -zxvf dune-modulename-X.Y.Z.tar.gz
```

for each DUNE module. To build them all, go into the `dune` directory and type[4]

```
./dune-common/bin/dunecontrol all
```

Note that this may take several minutes. Once the process has completed, you can install DUNE by saying

```
./dune-common/bin/dunecontrol make install
```

as root. This will install the DUNE core modules to `/usr/local`.

If you want to install DUNE to a non-standard location (because maybe you do not have root access, or you want to have several versions of DUNE installed in parallel), then you can set an installation path. For this, create a text file `dune.opts`, which should look like this:

```
CMAKE_FLAGS=" -DCMAKE_INSTALL_PREFIX=your/desired/installation/path "
```

The command `dunecontrol` will pickup the variable `CMAKE_FLAGS` from this file and use it as a command line option for any call to CMAKE. This particular options file will tell CMAKE that all modules should be installed to `your/desired/installation/path`. Then call

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

and

```
./dune-common/bin/dunecontrol make install
```

This should install all DUNE modules into `your/desired/installation/path`.

If you have installed UG in a nonstandard location you also have to add

```
CMAKE_FLAGS+=" -DCMAKE_PREFIX_PATH=your/ug/installation/path "
```

to the options file.

The `dunecontrol` program will also manage your own code. If you have installed DUNE locally you have to tell it where to find the DUNE core modules. You can do that by prepending the path `your/desired/installation/path` to the environment variable `DUNE_CONTROL_PATH`. Note, however, that if `DUNE_CONTROL_PATH` is set to anything, then the current directory is *not* searched automatically for DUNE modules. If you do want the current directory to be searched then the `DUNE_CONTROL_PATH` variable has to contain `:.:` somewhere.[5]

2 A First Dune Application

DUNE is organized in modules, where each module is roughly a directory with a predefined structure.At this stage we suggest you write your first own DUNE program into a module, but later you do not have to do that. So the first thing you have to do is to create a new empty DUNE module.

[4]Replace `dune-common` by `dune-commmon-X.Y.Z` if you are using the tarballs.
[5]The dot denotes the current directory, and the colons are separators.

2.1 Creating a New Module

In the following we assume again that you have a Unix-type command shell on your computer, and that you have successfully installed DUNE either into the standard location, or that DUNE_CONTROL_PATH contains the installation path. To create an empty module, DUNE provides a special program called duneproject. Make a new directory, enter it, and then type

```
duneproject
```

in the shell. If the program is not installed globally you have to start it from dune-common/bin.

The duneproject program will ask you several questions before creating the module. The first is the module name. You can choose any Unix file name without whitespace, but customarily module names start with a dune- prefix. To be specific we will call your new module dune-foo.

The next question asks for other modules that your new module will depend upon. Surely the code you write will depend on other parts of DUNE, which reside in other modules. To let the build system know which modules you are going to use you have to list them here. To help you, duneproject has already collected a list of all modules it sees on your system. These are the globally installed ones, and the ones in directories listed in the DUNE_CONTROL_PATH environment variable. If you have done the installation described above correctly you should see at least dune-common, dune-geometry, dune-grid, dune-istl, dune-localfunctions, dune-typetree, and dune-functions. All you need to do is choose the ones you need. (And don't worry: if you forget one you can easily add it later.) For the purpose of this introduction please select them all.

Next is the question for a module version number. Since we are just starting out type 1.0. This is followed by a question for your email address. This address will appear in the file dune.module of the module (and nowhere else), and mark you as the responsible for it.

Finally, duneproject will ask you whether you want to "enable all available packages". Simply answer yes to this question.

After you have answered these questions, and confirmed that your answer are correct, the duneproject program exits and you now have an empty module dune-foo.

```
sander@affe:~/dune: ls
dune-foo
```

If you have the tree program installed you can see that dune-foo contains a small directory tree.

```
~/dune> tree dune-foo
dune-foo
|-- cmake
|   `-- modules
|       |-- CMakeLists.txt
|       `-- DuneFooMacros.cmake
```

```
|-- CMakeLists.txt
|-- config.h.cmake
|-- doc
|   |-- CMakeLists.txt
|   `-- doxygen
|       |-- CMakeLists.txt
|       `-- Doxylocal
|-- dune
|   |-- CMakeLists.txt
|   `-- foo
|       |-- CMakeLists.txt
|       `-- foo.hh
|-- dune-foo.pc.in
|-- dune.module
|-- README
|-- src
|   |-- CMakeLists.txt
|   `-- dune-foo.cc
`-- stamp-vc

7 directories, 16 files
```

This tree contains:

- A text file `dune.module`, which contains some meta data of the module,
- a small example program in `dune-foo.cc`,
- The CMake build system.[6]

There is nothing Dune-specific about the build system.

2.2 Testing the New Module

If you look at the contents of the new module you will see that there is a single C++ source code file `src/dune-foo.cc` already. So the module as it is already does something and this is what we are going to test now.

For this we need to build and compile the module. The `dunecontrol` program does most of the work for you. If you have already installed the Dune core modules from source you will notice that the process is hardly any different. Just type

```
dunecontrol all
```

[6] If you use Dune release 2.4 or earlier there will also be an additional AutoTools-based build system in the new directory. That build system is a fully functional alternative to the CMake build system. However, it stopped being supported after the release of Dune-2.4, and therefore we discourage you to use it.

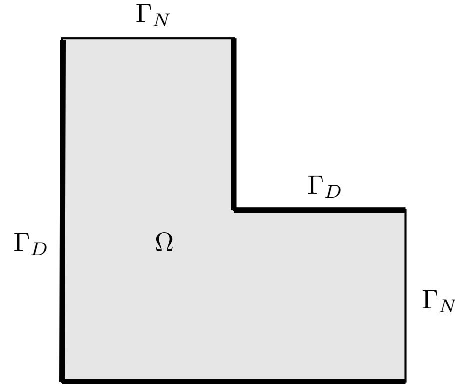


Figure 1: A simple domain Ω with Dirichlet boundary Γ_D (thick lines) and Neumann boundary Γ_N.

into your shell. This will output lots of information while it runs, but non of it should be of any concern to you right now (unless actual error messages appear somewhere). Once `dunecontrol` has terminated you can find a new executable `dune-foo` in `dune-foo/build_cmake/src`. Start it with

```
sander@affe:~/dune: ./dune-foo/build-cmake/src/dune-foo
```

and it will greet you with

```
Hello World! This is dune-foo.
This is a sequential program.
```

Congratulations! You have just run your first DUNE program.

3 Example: Solving the Poisson Equation Using Finite Elements

To get started with a real example we will solve the Poisson equation with the finite element method. We will compute the weak solution of the equation

$$-\Delta u = -5 \tag{1}$$

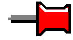
on the L-shaped domain $\Omega = (0,1)^2 \setminus (0.5,1)^2$, with Dirichlet boundary conditions

$$u = \begin{cases} 0 & \text{on } \{0\} \times [0,1] \cup [0,1] \times \{0\}, \\ 0.5 & \text{on } \{0.5\} \times [0.5,1] \cup [0.5,1] \times \{0.5\}, \end{cases}$$

and homogeneous Neumann conditions on the remainder of the boundary. (If you are unsure about how the finite element method works, please consult one of the

numerous textbooks on the subject.) The domain and boundary conditions are shown in Figure 1.

In the following we will walk you through a complete example program that solves the Poisson equation and outputs the result to a file which you can look at with the ParaView visualization software.[7] We will not show you quite the entire source code here, because complete C++ programs take a lot of space. However, the entire code is printed in Section 5. Also, if you are reading this document in electronic form you can access the source code file through the little pin icon on the side.



A Dune example program for solving the Poisson equation can be written at different levels of abstraction. It could use many Dune modules and the high-level features. In that case, the user code would be short, but you would lose the detailed control over the inner working of your program. On the other hand, you may want to depend on only a few low-level modules. In this case your would have to write more code by hand, which is more work, but which also means that you have more control.

The example in this chapter tries to strike a middle ground. It uses the Dune modules for grids, shape functions, discrete functions spaces, and linear algebra. It does not use a Dune module for the assembly of the algebraic system—that part is written by hand. If you are interested in having Dune assembly your systems we recommend that you have a look at the `dune-pdelab`, `dune-fem`, and `dune-fufem` modules.

At the same time, it is quite easy to rewrite the example to not use the Dune function spaces or a different linear algebra implementation. This is Dune—you are in control. The finite volume example in Section 4 uses much less parts from Dune than the Poisson example here does.

3.1 The Main Program

We begin with the main program, i.e., the part that sits in

```
int main (int argc, char** argv)
{
  [...]
}
```

Right away, we recommend you to catch exceptions at the end of the main block, i.e., instead of the previous code snippet use

```
int main (int argc, char** argv) try
{
  [...]
} catch (std::exception& e) {
  std::cout << e.what() << std::endl;
  return 1;
}
```

This will allow you to see more error messages if something goes wrong in the Dune methods you call.

[7] `http://www.paraview.org/`

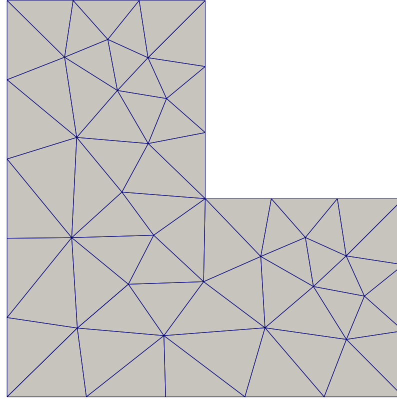
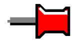


Figure 2: Unstructured grid for the example domain

3.1.1 Creating a Grid

The first thing we need to do is to create a grid. We will use the triangle grid shown in Figure 2. A good grid implementation for unstructured triangle grids is the `UGGrid` grid manager. The grid itself is read from a file in the GMSH format [11].[8] You can obtain the file by clicking on the annotation icon in the margin. The following code sets up a two-dimensional `UGGrid` object with the grid from the GMSH file `l-shape.msh`.



```
273    const int dim = 2;
274    typedef UGGrid<dim> GridType;
275    std::shared_ptr<GridType> grid(GmshReader<GridType>::read("l-shape.msh"));
276
277    grid->globalRefine(2);
278
279    typedef GridType::LeafGridView GridView;
280    GridView gridView = grid->leafGridView();
```

Line 277 refines the grid twice uniformly, to get a smoother-looking result. Note that the grid dimension `dim` is a compile-time parameter. For this code to compile you need to place the line

```
10  #include <dune/grid/uggrid.hh>
```

at the top of your program. Also remember that all DUNE code resides in the `Dune` namespace. Hence you either need to write the `Dune::` prefix before the type names `UGGrid` and `GmshReader`, or write the line

```
30  using namespace Dune;
```

once at the top of the program.

[8] `http://geuz.org/gmsh`

We have introduced the type variable `GridType` to store the type of the grid we use. This hints at one of the strengths of DUNE: It is easy to use the same code with different grid data structures. For the rest of the code, whenever the type of the grid is needed, we will only refer to `GridType`. This allows to change to, say, a structured cube grid changing only the definition of `GridType` and the subsequent constructor call. Indeed, to replace the unstructured grid by a structured cube grid for the unit square, replace Lines 273–280 by

```
const int dim = 2;
typedef YaspGrid<dim> GridType;

GridType grid({1.0,1.0},      // upper right corner, the lower left one is implicitly (0,0) here
              {10, 10});      // number of elements per direction

grid.globalRefine(2);

typedef GridType::LeafGridView GridView;
GridView gridView = grid.leafGridView();
```

The `YaspGrid` class,[9] from the file `dune/grid/yaspgrid.hh`, is the standard implementation of a structured cube grid in DUNE.

3.1.2 Assembling the Stiffness Matrix and Load Vector

Now that we have a grid we can assemble the matrix and the right-hand-side vector. For this we first need a matrix and a vector object to assemble into. We get these with the lines

```
288   typedef BlockVector<FieldVector<double,1> > VectorType;
289   typedef BCRSMatrix<FieldMatrix<double,1,1> > MatrixType;
290
291   VectorType rhs;
292   MatrixType stiffnessMatrix;
```

These are data structures from the `dune-istl` module, and you need to put

```
17 #include <dune/istl/bvector.hh>
18 #include <dune/istl/bcrsmatrix.hh>
```

at the top of your program. If you look closely at our example code you will see that it is pretty easy to use other linear algebra implementations instead of the ones from `dune-istl`. That is another advantage of DUNE.

To give you a quick idea of what's happening in Lines 288 and 289: `BCRSMatrix` and `BlockVector` are *block* matrix and *block* vector types, respectively. Their template parameters are the types of the blocks. Since we deal with a scalar problem the blocks are 1×1 matrices and vectors of size 1.

We are now ready to assemble the stiffness matrix and right-hand side. To make the `main` method more readable we have put the assembly code into a subroutine and discuss it later. So for the time being write

[9] `Yet another structured parallel Grid`, for historical reasons

```
300     Functions::PQkNodalBasis<GridView,1> basis(gridView);
301
302     auto sourceTerm = [](const FieldVector<double,dim>& x){return −5.0;};
303     assemblePoissonProblem(basis, stiffnessMatrix, rhs, sourceTerm);
```

into your code. Line 300 selects the finite element space we are using. This example uses first-order Lagrangian finite elements, and we select the space of these elements by specifying a *basis* of this space. Line 302 then creates a function object that implements your analytical source term; in our case it is the function with the constant value -5. We will show you how to implement the `assemblePoissonProblem` method in Chapter 3.2.

3.1.3 Incorporating the Dirichlet Boundary Conditions

After the call to `assemblePoissonProblem`, the variable `stiffnessMatrix` contains the stiffness matrix A of the Laplace operator, and the variable `rhs` contains the weak right hand side. However, we still need to incorporate the Dirichlet boundary conditions. We do this in the standard way; viz. if the i-th degree of freedom is on the Dirichlet boundary we overwrite the corresponding matrix row with a row from the identity matrix, and the entry in the right hand side vector with the prescribed Dirichlet value.

The implementation needs two steps. First we need to figure out which degrees of freedom are Dirichlet degrees of freedom. Since we are using Lagrangian finite elements, we can use the positions of the Lagrange nodes to determine which degrees of freedom are fixed by the Dirichlet boundary conditions. First, we define a predicate class that returns `true` or `false` depending on whether a given position is on the Dirichlet boundary or not. Then, we interpolate this predicate with respect to the Lagrange basis to obtain a vector of booleans with the desired information.

```
312     auto predicate = [](auto p){return p[0]< 1e−8 or p[1] < 1e−8 or (p[0] > 0.4999 and p[1] > 0.4999);};
313
314     // Interpolating the predicate will mark all desired Dirichlet degrees of freedom
315     std::vector<char> dirichletNodes;
316     Functions::interpolate(basis, dirichletNodes, predicate);
```

In general, there is no single approach to the determination of Dirichlet degrees of freedom that fits all needs. The simple method used here works well for Lagrange spaces and simple geometries. However, Dune also supports other ways to find the Dirichlet boundary.

Now, with the bit field `dirichletNodes`, the following code snippet does the corresponding modifications of the stiffness matrix.

```
323   // loop over the matrix rows
324   for (size_t i=0; i<stiffnessMatrix.N(); i++)
325   {
326     if (dirichletNodes[i])
327     {
328       auto cIt    = stiffnessMatrix[i].begin();
```

```
329        auto cEndIt = stiffnessMatrix[i].end();
330        // loop over nonzero matrix entries in current row
331        for (; cIt!=cEndIt; ++cIt)
332          *cIt = (i==cIt.index()) ? 1.0 : 0.0;
333      }
334    }
```

In Line 324 we loop over all matrix rows, and in Line 326 we test whether the row corresponds to a Dirichlet degree of freedom. If this is the case then we loop over all matrix entries of the row. The matrix is sparse and we only want to loop over the nonzero entries of the row. This is done by the iterator loop you see in Line 331. Note that it is very similar to iterator loops in the C++ standard library. Finally we set the matrix entries in Line 332.

Modifying the right hand side vector is even easier.

```
339    auto dirichletValues = [](auto p){return (p[0]< 1e-8 or p[1] < 1e-8) ? 0 : 0.5;};
340    Functions::interpolate(basis,rhs, dirichletValues, dirichletNodes);
```

We define another function object implementing the Dirichlet value function in Line 339, and interpolate it into the `rhs` vector object. The fourth argument restricts the interpolation to those entries where the corresponding entry in `dirichletNodes` is set. All others are untouched.

At this point, we have set up the linear system

$$Ax = b \tag{2}$$

corresponding to the Poisson problem (1), and this system contains the Dirichlet boundary conditions. The matrix A is stored in the variable `stiffnessMatrix`, and the load vector b is stored in the variable `rhs`.

3.1.4 Solving the Algebraic Problem

To solve the algebraic system (2) we will use the conjugate gradient (CG) method with an ILU preconditioner (see [13] for some background on how these algorithms work). Both methods are implemented in the `dune-istl` module, and you need to include the headers `<dune/istl/solvers.hh>` and `<dune/istl/preconditioners.hh>`. The following code constructs the preconditioned solver, and applies it to the algebraic problem

```
356    // Choose an initial iterate
357    VectorType x(basis.size());
358    x = 0;
359
360    // Technicality: turn the matrix into a linear operator
361    MatrixAdapter<MatrixType,VectorType,VectorType> linearOperator(stiffnessMatrix);
362
363    // Sequential incomplete LU decomposition as the preconditioner
364    SeqILU0<MatrixType,VectorType,VectorType> preconditioner(stiffnessMatrix,1.0);
365
```

```
366   // Preconditioned conjugate−gradient solver
367   CGSolver<VectorType> cg(linearOperator,
368                           preconditioner,
369                           1e−4, // desired residual reduction factor
370                           50,   // maximum number of iterations
371                           2);   // verbosity of the solver
372
373   // Object storing some statistics about the solving process
374   InverseOperatorResult statistics;
375
376   // Solve!
377   cg.apply(x, rhs, statistics);
```

After this code has run, the variable `x` contains the approximate solution of (2), `rhs` contains the corresponding residual, and `statistics` contains some information about the solution process, like the number of iterations that have been performed.

3.1.5 Outputting the Result

Finally we want to access the result and, in particular, view it on screen. Dune itself does not provide any visualization features (because dedicated visualization tools do a great job, and the Dune team did not want to compete), but you can write the result to a file in a variety of different file formats. In this example we will use the VTK format [1], which is the standard format of the ParaView software. Include the header `<dune/grid/io/file/vtk/vtkwriter.hh>` and add the lines

```
382   VTKWriter<GridView> vtkWriter(gridView);
383   vtkWriter.addVertexData(x, "solution");
384   vtkWriter.write("poissonequation−result");
```

to your program. The first line creates a `VTKWriter` object and initializes it with your grid. The second line adds the solution vector `x` as vertex data to the writer object. The string "`solution`" is a name given to the data field. It appears within ParaView and prevents confusion when there is more then one field. The third line actually writes the file, giving it the name `poissonequation-result.vtu`.

3.1.6 Running the Program

With the exception of the stiffness matrix assembler (which we cover in the next section), the program is now complete. You can build it by typing `make` in the directory `build-cmake`. The executable `poissonequation` will then appear in the `build-cmake/src` directory. When you start it you will first see the `GmshReader` class give some information about the grid file it is reading. Then, you see the CG method iterating

```
Reading 2d Gmsh grid...
version 2.2 Gmsh file detected
file contains 43 nodes
```

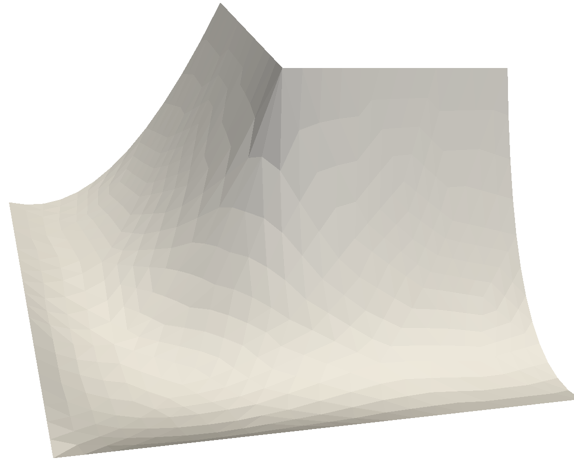


Figure 3: Output of the Poisson example program

```
file contains 90 elements
number of real vertices = 43
number of boundary elements = 22
number of elements = 62
=== CGSolver
 Iter          Defect            Rate
    0          2.50571
    1         0.944781         0.377051
    2         0.660777         0.699397
[...]
   19      0.000414162         0.453941
   20      0.000263526         0.636287
   21      0.000221982         0.842355
=== rate=0.641237, T=0.015707, TIT=0.000747953, IT=21
```

After program termination there is a file called `poissonequation-result.vtu` in the `build_cmake/src` directory, which you can open in, e.g., ParaView. It contains the grid and solution function, which should look like what you see in Figure 3.

3.2 Assembling the Stiffness Matrix

We now show you how the stiffness matrix and load vector of the Poisson problem are assembled. The example program here does it "by hand"—it contains a complete assembler loop that uses only the Dune core modules and `dune-functions` for discrete

function spaces. This will teach you a few things about the grid and discrete function interfaces, and will allow you to write your own special-purpose assemblers. Several additional DUNE modules do provide such assemblers for you. Have a look at the `dune-pdelab`, `dune-fem`, and `dune-fufem` modules.

3.2.1 The Global Assembler

We now discuss the method `assemblePoissonProblem`, which was called by the `main` method for the problem assembly. It has the signature

```
181 template <class Basis>
182 void assemblePoissonProblem(const Basis& basis,
183                             BCRSMatrix<FieldMatrix<double,1,1> >& matrix,
184                             BlockVector<FieldVector<double,1> >& rhs,
185                             const std::function<double(FieldVector<double,Basis::GridView::dimension>)> volume
```

This method will assemble the individual element stiffness matrices, and add up the local results to obtain the global stiffness matrix. As the first step, we get the grid object from the finite element basis by writing

```
189   auto gridView = basis.gridView();
```

The object `gridView` is the finite element grid that the basis is defined on.

Next, we set up the global stiffness matrix occupation pattern, and initialize the matrix values to zero.

```
195   MatrixIndexSet occupationPattern;
196   getOccupationPattern(basis, occupationPattern);
197
198   // ... and give it the occupation pattern we want.
199   occupationPattern.exportIdx(matrix);
200
201   // Set all entries to zero
202   matrix = 0;
```

Then, we set the vector `rhs` to the correct size, and fill it with zeros as well:

```
206   // set rhs to correct length
207   rhs. resize (basis . size ());
208
209   // Set all entries to zero
210   rhs = 0;
```

For brevity we do not show the code of the `getOccupationPattern` method here, because it is very similar to the actual assembler loop. See the complete source code in Section 5 to find out how it works in detail.

After these preliminaries starts the main loop over the elements in the grid.

```
215   auto localView = basis.localView();
216   auto localIndexSet = basis.localIndexSet();
217
218   for (const auto& element : elements(gridView))
219   {
```

The variables `localView` and `localIndexSet` are means to access the finite element basis on individual grid elements. While their use may seem overly complicated in this example program, they show their real power when using more advanced finite element spaces.

The loop in Line 218 is a so-called *range-based* `for` loop, new in C++11. The free method `elements` from the `dune-grid` module acts like a container of all elements of the grid in `gridView`. The `for` loop will iterate over this container. At each iteration, the object `element` will be a `const` reference to the current grid element.

Now, within the loop, we first bind the `localView` and `localIndexSet` objects to the current element. All subsequent calls to these two objects will now implicitly refer to that element. Then, we create a small dense matrix and call the element matrix assembler for it

```
225         localView.bind(element);
226         localIndexSet.bind(localView);
227
228         Matrix<FieldMatrix<double,1,1> > elementMatrix;
229         assembleElementStiffnessMatrix(localView, elementMatrix);
```

Note that we rely on the element assembler to set the correct matrix size for the current element. After this call, the variable `elementMatrix` contains the element stiffness matrix for the element referenced by the `element` variable.

Finally, we need to add the element matrix to the global one. This is done by

```
234         for( size_t i=0; i<elementMatrix.N(); i++)
235         {
236             // The global index of the i-th degree of freedom of the element
237             auto row = localIndexSet.index(i);
238
239             for ( size_t j=0; j<elementMatrix.M(); j++ )
240             {
241                 // The global index of the j-th degree of freedom of the element
242                 auto col = localIndexSet.index(j);
243                 matrix[row][col] += elementMatrix[i][j];
244             }
245         }
```

The two `for`-loops loop over all entries of the element stiffness matrix. Since the matrix is a dense matrix type we do not need to use an iterator loop (but of course we can if we want to). Within the loops a pair `i,j` signifies an entry in the element matrix. The function space basis knows the corresponding global degrees of freedom, and provides this information via the `localIndexSet` object.

3.2.2 The Element Assembler

Finally, there is the local problem: given a grid element T, assemble the element stiffness matrix for the Laplace operator and the given finite element basis. We write a method with the following signature:

```
35 template <class LocalView, class MatrixType>
36 void assembleElementStiffnessMatrix( const LocalView& localView, MatrixType& elementMatrix)
```

Remember that the entry a_{ij} of the element matrix has the form

$$a_{ij} = \int_T \langle \nabla\varphi_i, \nabla\varphi_j \rangle \, dx,$$

φ_i, φ_j the nodal basis functions, and is computed by transforming the integral over T to an integral over the reference element T_{ref}

$$a_{ij} = \int_{T_{\text{ref}}} (\nabla F^{-1})^T \nabla\hat{\varphi}_i \cdot (\nabla F^{-1})^T \nabla\hat{\varphi}_j \det|\nabla F| \, d\xi, \tag{3}$$

where F is the mapping from T_{ref} to T, and $\hat{\varphi}_i$, $\hat{\varphi}_j$ are the shape functions on T_{ref} corresponding to the basis functions φ_i, φ_j on T. We approximate (3) by a quadrature rule with points ξ_k and weights w_k,

$$a_{ij} = \sum_{k=1}^{m} w_k (\nabla F^{-1}(\xi_k))^T \nabla\hat{\varphi}_i(\xi_k) \cdot (\nabla F^{-1}(\xi_k))^T \nabla\hat{\varphi}_j(\xi_k) \det|\nabla F(\xi_k)|. \tag{4}$$

Formula (4) is what we need to implement.

The first argument of the method `assembleElementStiffnessMatrix` is the `LocalView` object of the finite element basis. From the view we get information about the current element, namely the element itself, its dimension, and its shape:

```
40     using Element = typename LocalView::Element;
41     auto element = localView.element();
42     const int dim = Element::dimension;
43     auto geometry = element.geometry();
```

The `geometry` object contains the transformation F from the reference element T_{ref} to the actual element T. Then, we get the set of shape functions $\{\hat{\varphi}_i\}_i$ for this element

```
48     const auto& localFiniteElement = localView.tree().finiteElement();
```

In Dune-speak, the object that holds the set of shape functions is called a *local finite element*, because it is actually a little more than just the shape functions. The method `tree()` is there to accommodate the possibility to have vector-valued or mixed finite element spaces, and we advise you to ignore it for the time being.

We can now ask the shape function set for the number of shape functions for this element, and initialize the element matrix accordingly

```
53     elementMatrix.setSize(localFiniteElement.size(), localFiniteElement.size());
54     elementMatrix = 0;       // fills the entire matrix with zeros
```

The last thing we need to compute the integrals (3) is a quadrature rule. We get it from the `dune-geometry` module by including `<dune/geometry/quadraturerules.hh>`, and calling

```
59      int order = 2*(dim*localFiniteElement.localBasis().order()−1);
60      const auto& quadRule = QuadratureRules<double, dim>::rule(element.type(), order);
```

Line 59 estimates an appropriate quadrature order, and Line 60 gets the actual rule, as a reference to a singleton held by the `dune-geometry` module. A quadrature rule in DUNE is little more than a `std::vector` of quadrature points, and hence looping over all points looks familiar:

```
65      for (const auto& quadPoint : quadRule) {
```

Now, with `quadPoint` the current quadrature point, we need its position ξ_k, the inverse transposed Jacobian $\nabla F^{-T}(\xi_k)$, and the factor $\det \nabla F(\xi_k)$ there. This information is provided directly by the DUNE grid interface via the `geometry` object:

```
69          // Position of the current quadrature point in the reference element
70          const auto quadPos = quadPoint.position();
71
72          // The transposed inverse Jacobian of the map from the reference element to the element
73          const auto jacobian = geometry.jacobianInverseTransposed(quadPos);
74
75          // The multiplicative factor in the integral transformation formula
76          const auto integrationElement = geometry.integrationElement(quadPos);
```

Then we compute the derivatives of all shape functions on the reference element, and multiply them from the left by ∇F^{-T} to obtain the gradients of the basis functions $\{\nabla \varphi_i\}_i$ on the element T

```
80          // The gradients of the shape functions on the reference element
81          std::vector<FieldMatrix<double,1,dim> > referenceGradients;
82          localFiniteElement.localBasis().evaluateJacobian(quadPos, referenceGradients);
83
84          // Compute the shape function gradients on the real element
85          std::vector<FieldVector<double,dim> > gradients(referenceGradients.size());
86          for (size_t i=0; i<gradients.size(); i++)
87              jacobian.mv(referenceGradients[i][0], gradients[i]);
```

Note how the gradients of the $\{\hat{\varphi}_i\}_i$ are stored in a vector of *matrices* with one row in Line 81. The reason for this is to be able to handle vector-valued shape functions. In the scalar case, getting the *vector* $\nabla \varphi_i$ requires the suffix `[0]` in Line 87.

Finally we compute the actual matrix entries

```
92          for (size_t i=0; i<elementMatrix.N(); i++)
93              for (size_t j=0; j<elementMatrix.M(); j++ )
94                  elementMatrix[localView.tree().localIndex(i)][localView.tree().localIndex(j)]
95                      += ( gradients[i] * gradients[j] ) * quadPoint.weight() * integrationElement;
```

By operator overloading, `gradients[i]*gradients[j]` is actually the scalar product between two vectors. The expressions `localView.tree().localIndex(i)` and `localView.tree().localIndex(j)` are there to accommodate more complicated finite element bases. In this particular case, `elementMatrix[i][j] += ...` would work just as well. See the `dune-functions` documentation for details.

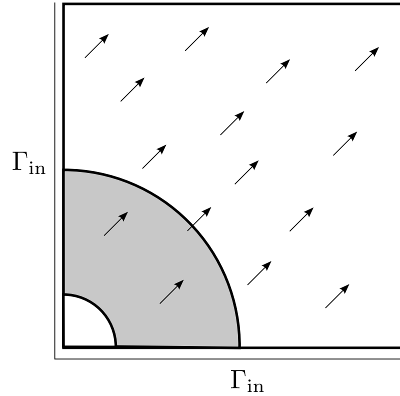


Figure 4: Domain, (scaled) velocity field, and initial condition of the finite volume example

4 Example: Solving the Transport Equation with a Finite Volume Method

The second example program will show you how to implement a simple first-order finite volume method. This will demonstrate a few more things about the DUNE grid interface, e.g., how to obtain face normals and volumes.

Compared to the Poisson solver of the previous section, the finite volume implementation uses much less features of the DUNE libraries. C++ standard library types are used for coefficient vectors, instead of using a dedicated linear algebra library. While cell-centered finite volume methods may be implemented using the function space basis objects from the `dune-functions` module, this would not make the code much simpler. We therefore implement the example without `dune-functions`.

4.1 Discrete linear transport equation

As the example problem we will use a linear scalar transport equation. Let $v : \Omega \times (0, t_{\text{end}}) \to \mathbb{R}^d$ be a given velocity field, and $c : \Omega \times [0, t_{\text{end}}] \to \mathbb{R}$ an unknown concentration. Transport of the concentration along the velocity flow lines is described by the equation

$$\frac{\partial c}{\partial t} + \operatorname{div}(vc) = 0 \qquad \text{in } \Omega \times (0, t_{\text{end}}). \tag{5}$$

For this example, we choose the unit-square domain $\Omega = (0, 1)^2$ (Figure 4), and the upper time bound $t_{\text{end}} = 0.6$. As velocity field we pick

$$v(x, t) = (1, 1),$$

which is stationary and divergence-free. By the choice of this flow field, a part of the boundary becomes the inflow boundary

$$\Gamma_{\text{in}}(t) := \{y \in \partial\Omega \mid v(y,t) \cdot \mathbf{n}(y) < 0\}.$$

In the current example, the inflow boundary consists of the lower and left sides of the square, and remains fixed over time. On the inflow boundary we prescribe the concentration

$$c(x,t) = 0 \qquad x \in \Gamma_{\text{in}}, \quad t \in (0, t_{\text{end}}).$$

Finally, we provide initial conditions

$$c(x,0) = c_0(x) \qquad \text{for all } x \in \Omega,$$

which we set to

$$c_0(x) = \begin{cases} 1 & \text{if } |x| > 0.125 \text{ and } |x| < 0.5, \\ 0 & \text{otherwise.} \end{cases}$$

For the discretization we cover the domain with a uniform grid consisting of 80×80 quadrilateral elements. The time interval $[0, t_{\text{end}}]$ is split into substeps

$$0 = t_0 < t_1 < t_2 < \cdots < t_m = t_{\text{end}},$$

and we define $\Delta t^k := t_{k+1} - t_k$. These steps are not uniform, but will be determined during the simulation by a simple adaptive time-stepping strategy. We write T_i for the i-th grid element and $|T_i|$ for its volume. Likewise, γ_{ij} will denote the element facet common to elements T_i and T_j, $|\gamma_{ij}|$ the area of the facet, and $\mathbf{n}_{ij}$ its unit normal pointing from T_i to T_j. We use a cell-centered finite volume discretization in space, full upwind evaluation of the fluxes and an explicit Euler scheme in time. In particular, we approximate the unknown concentration c by a piecewise constant function, and identify each such function with its values C_i, $i = 1, \dots, N$ at the element centers.

We obtain the following equation for the unknown cell values C_i^{k+1} at time t_{k+1}:

$$C_i^{k+1}|T_i| - C_i^k|T_i| + \sum_{\gamma_{ij}} \phi \cdot |\gamma_{ij}| \Delta t^k = 0 \qquad \forall i = 1, \dots, N, \tag{6}$$

where v_{ij}^k is the velocity at the center of the facet γ_{ij} at time t_k. The flux function ϕ is an approximation of the flux $cv\mathbf{n}$ across the element boundary γ_{ij}. One common choice is

$$\phi(C_i^k, C_j^k, v_{ij}^n \cdot \mathbf{n}_{ij}) := C_i^k \max(0, v_{ij}^n \cdot \mathbf{n}_{ij}) - C_j^k \max(0, -v_{ij}^n \cdot \mathbf{n}_{ij}). \tag{7}$$

Observe that this expression effectively switches between two cases, depending on whether there is flux from T_i to T_j or vice versa.

Inserting the flux function (7) into (6) and rearranging terms, we can solve the expression for the unknown coefficients C_i^{k+1} at time t_{k+1}. The resulting formula is a simple vector update

$$C^{k+1} = C^k + \Delta t^k \delta \tag{8}$$

with the update vector $\delta \in \mathbb{R}^N$ given by

$$\delta_i := -\sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|T_i|} \left(C_i^k \max(0, v_{ij}^k \cdot \mathbf{n}_{ij}) + C_j^k \max(0, -v_{ij}^k \cdot \mathbf{n}_{ij}) \right). \tag{9}$$

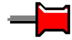
To adaptively select the time steps Δt^k we consider a stability bound. The time stepping scheme is stable provided that

$$\Delta t^k \leq \min_{i=1,\dots,N} \left(\sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|T_i|} \max(0, v_{ij}^k \cdot \mathbf{n}_{ij}) \right)^{-1}. \tag{10}$$

This allows to construct a simple control of the time step size Δt^k. After each iterate we compute the right-hand side of (10), and set Δt^k to just a little below that value.

4.2 The Main Program

The example implementation is contained in a single file. The complete file is printed in Section 6, and users of an electronic version of this text can get it by clicking on the icon in the margin. Do not forget that this program has



```
15  using namespace Dune;
```

at the top, to avoid having to write the `Dune::` prefix over and over again.

The implementation is split between two methods, the `main` method and a method `evolve` that computes and applies the update vector δ defined in (9). We first discuss the `main` method. It has the form

```
131 int main (int argc , char ** argv) try
132 {
133   // Set up MPI, if available
134   Dune::MPIHelper::instance(argc, argv);

      // code

232 }
233 catch (std::exception & e) {
234   std::cout << e.what() << std::endl;
235   return 1;
236 }
```

The `try...catch` block intercepts errors from within DUNE and prints the corresponding messages. The `MPIHelper` instance sets up your MPI system if you have it installed. Even though this example does not use MPI, some parts of DUNE link to it internally, and not initializing it would lead to run-time errors.

The first real code block sets up the grid:

```
138   const int dim = 2;
139   typedef YaspGrid<dim> GridType;
140   GridType grid({1.0,1.0},      // upper right corner, the lower left one is (0,0)
```

```
141                    {80,80});        // number of elements per direction
142
143   typedef GridType::LeafGridView GridView;
144   GridView gridView = grid.leafGridView();
```

Unlike in the previous example we are using a structured grid now. Therefore there is no need to read the grid from a file; giving the bounding box and the number of elements per direction suffices.

We then set up the vector of the concentration values at the element centers:

```
149   MultipleCodimMultipleGeomTypeMapper<GridView,MCMGElementLayout>
150   mapper(gridView);
151
152   // Allocate a vector for the concentration
153   std::vector<double> c(mapper.size());
```

The `MultipleCodimMultipleGeomTypeMapper` object constructed in Line 149 is a device that assigns natural numbers to grid elements. These numbers are then used to address arrays that hold the actual simulation data. The mapper plays a similar role as the function space basis in the previous example, but it is a more low-level construct with less functionality. In particular, it is contained in the `dune-grid` module rather than in `dune-functions`. You do not need the latter module to run this example.

Line 153 creates the array that is to store the concentration values. Observe that this is a type from the C++ standard library. There is no dependence on the DUNE linear algebra module `dune-istl`, or any other linear algebra library.

The array `c` is then filled with the values of the initial-value function c_0 at the cell center. First, the function c_0 is implemented as a lambda object.

```
158   auto c0 = [](auto x) { return (x.two_norm()>0.125 && x.two_norm()<0.5) ? 1.0 : 0.0; };
```

Then, the code loops over the elements and samples `c0` at the element centers.

```
162   // Iterate over grid elements and evaluate c0 at element centers
163   for (const auto& element : elements(gridView))
164   {
165     // Get element geometry
166     auto geometry = element.geometry();
167
168     // Get global coordinate of element center
169     auto global = geometry.center();
170
171     // Sample initial concentration c0 at the element center
172     c[mapper.index(element)] = c0(global);
173   }
```

Looping over the elements has already been used in the previous example. Note the special method `center` used in Line 169 to obtain the coordinates of the center of the current element. While there is a more general mechanism to obtain coordinates for any point in an element (the `global` method of the `geometry` object), the element

center is so frequently used in finite volume schemes that a dedicated method for it exists. The center coordinate is then used as the argument for the function object `c0`, which returns the initial concentration c_0 at that point. Line 172 shows how the mapper object is used: Its `index` method returns a nonnegative integer for the given element, which is suitable to access the data array `c`.

The next code block constructs a writer for the VTK format, and writes the discrete initial concentration to a file

```
178   auto vtkWriter = std::make_shared<Dune::VTKWriter<GridView> >(gridView);
179   VTKSequenceWriter<GridView> vtkSequenceWriter(vtkWriter,
180                                                 "concentration");   // file name
181
182   // Write the initial values
183   vtkWriter->addCellData(c,"concentration");
184   vtkSequenceWriter.write( 0.0 );   // 0.0 is the current time
```

The `VTKWriter` constructed in Line 178 writes individual concentration fields to individual files. The `VTKSequenceWriter` in the following line ties these together to a time series of data. In addition to the individual data files, it writes a *sequence file* (with a `.pvd` suffix) that lists all data files together with their time points t_k. This information allows to properly visualize time-dependent data even if the time steps are not uniform.

The final block in the `main` method is the actual time loop.

```
189   double t=0;
190   double tend=0.6;
191   int k=0;
192   const double saveInterval = 0.1;
193   double saveStep = 0.1;
194
195   auto b = [](const Dune::FieldVector<GridView::ctype,GridView::dimensionworld>& x, double t)
196   {
197     return 0.0;
198   };
199
200   auto v = [](const Dune::FieldVector<GridView::ctype,GridView::dimensionworld>& x, double t)
201   {
202     return Dune::FieldVector<double,GridView::dimensionworld> (1.0);
203   };
204
205   while (t<tend)
206   {
207     // apply finite volume scheme
208     double dt = evolve(gridView,mapper,c,v,b,t);
209
210     // augment time and time step counter
211     t += dt;
212     ++k;
213
```

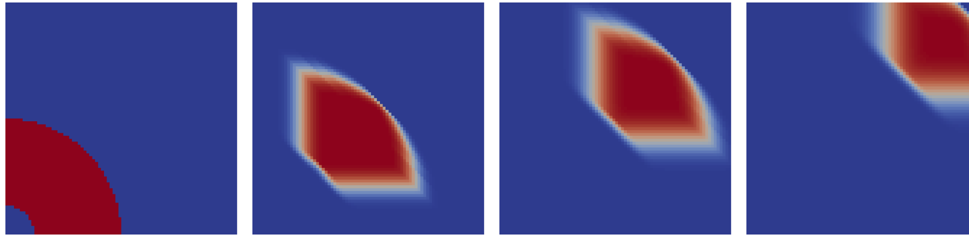


Figure 5: Evolution of the concentration c at times $t = 0$, $t = 0.204187$, $t = 0.402187$, $t = 0.600187$

```
214      // check if data should be written
215      if (t >= saveStep)
216      {
217        // Write data. We do not have to call addCellData again!
218        vtkSequenceWriter.write( t );
219
220        // increase saveStep for next interval
221        saveStep += saveInterval;
222      }
223
224      // print iteration number, time, and time step size
225      std::cout << "k=" << k << " t=" << t << " dt=" << dt << std::endl;
226    }
```

Lines 189–193 initialize several variables, and two further lambda objects `v` and `b` for the velocity field and the inflow boundary condition, respectively. The loop starting in Line 205 loops until the current time `t` has exceeded the specified end time `tend`. Most of the actual work is done in a separate method `evolve`, which we discuss below. That method determines a suitable time step size following the stability criterion (10), and advances the global concentration vector `c` to the next time step. The main loop then increases the global time, and writes the iterate to a file if a predefined amount of time has passed.

When run, the example program produces 97 time steps, seven output files, and the sequence file `concentration.pvd`. A visualization of the simulation result is given in Figure 5. One can clearly see how the initial condition is transported along the velocity field v. The noticeable diffusion is caused by the crude numerical method.

4.3 The method `evolve`

The method `evolve` does the main part of the work. In particular, it implements evolution (8) with optimal time step control via (10). After each call to `evolve`, the current iterate has advanced to the next time step, and a proper time step size Δt^k for this has been determined using the stability bound (10).

The method signature is

```
19 template<class GridView, class Mapper>
20 double evolve (const GridView& gridView,
21                const Mapper& mapper,
22                std::vector<double>& c,
23                const std::function<FieldVector<typename GridView::ctype,GridView::dimensionworld>
24                                   (FieldVector<typename GridView::ctype,GridView::dimension>,double)> v,
25                const std::function<double
26                                   (FieldVector<typename GridView::ctype,GridView::dimension>,double)> b,
27                double t)
```

The first two arguments are the grid and the mapper. The third argument is the array of element concentration values `c`. This argument is a non-const reference, because the array is modified in-place. The arguments `v` and `b` are the velocity field and the inflow boundary condition functions, respectively. The last argument is the current global time t. Even though the velocity field and boundary conditions are independent of time for this specific example, we have kept the infrastructure in the code to also allow time-dependent data.

The method starts by a bit of initialization code:

```
31   // Grid dimension
32   const int dim = GridView::dimension;
33
34   // allocate a temporary vector for the update
35   std::vector<double> update(c.size());
36   std::fill(update.begin(), update.end(), 0.0);
37
38   // initialize dt very large
39   double dt = std::numeric_limits<double>::max();
```

The array set up in Line 35 is the correction δ defined in (9). The variable `dt` in Line 39 will store the time step size Δt^k. As it is defined as the minimum of a sequence of numbers in (10), it is initially set to something that is certainly larger than anything that can occur in (10).

Then starts the loop over all grid elements T_i:

```
44   for (const auto& element : elements(gridView))
45   {
46     // element geometry
47     auto geometry = element.geometry();
48
49     // element volume
50     double elementVolume = geometry.volume();
51
52     // Element index
53     auto i = mapper.index(element);
54
55     // Will store the total element outflow
56     double sumfactor = 0.0;
```

This is the same kind of loop already seen in the global finite element assembler in Section 3. For each element T_i, the loop computes the update δ_i defined in (9). At the top of the loop, a bit of information about T_i like its volume $|T_i|$ and index i are precomputed.

The formula (9) for the correction δ_i for element T_i consists of a sum over all elements T_j whose boundaries intersect with the boundary of T_i in more than just a point. Such neighborhood relations are represented by the DUNE grid interface by objects of type `Intersection`. These return all relevant information about the relationship of an element with a particular neighbor or the domain boundary. The concept is deliberately general enough to allow for nonconforming grids, i.e., grids where the intersection of two elements is not necessarily a common facet.

The sum in (9) is therefore coded with a loop over all intersections of the current element.

```
61      for (const auto& intersection : intersections(gridView,element))
62      {
63        // get geometry type of face
64        auto igeo = intersection.geometry();
65
66        // center of face in global coordinates
67        auto faceCenter = igeo.center();
68
69        // evaluate velocity at face center
70        auto velocity = v(faceCenter,t);
71
72        // get the center of the intersection in local coordinates
73        const auto& intersectionReferenceElement = ReferenceElements<double,dim-1>::general(intersection.type());
74        auto intersectionCenter = intersectionReferenceElement.position(0,0);
75
76        // get normal vector scaled with volume
77        auto integrationOuterNormal = intersection.integrationOuterNormal(intersectionCenter);
78
79        // Compute factor occuring in flux formula v · n_ij · |γ_ij|
80        double factor = velocity*integrationOuterNormal;
```

The loop itself uses the convenient range-based `for` syntax already used for the loops over the grid elements. It then computes the factor $|\gamma_{ij}| v_{ij} \mathbf{n}_{ij}$. To compute v_{ij}, the value of the velocity field v at the center of the common facet between T_i and its current neighbor T_j, Line 64 first acquires the geometry (i.e., the shape) of the intersection between T_i and T_j. Just like an element geometry, this intersection geometry has a `center` method which is called in Line 67. Line 70 then evaluates the velocity field at that position.

To evaluate $\mathbf{n}_{ij}$, we need the center of the intersection in local coordinates of the intersection. A direct method for this does not exist. On the other hand, there is a reference element corresponding to the intersection, which is acquired in Line 73. That reference element knows its center.

Rather than computing $|\gamma_{ij}|$ and $\mathbf{n}_{ij}$ separately, the DUNE grid interface offers a method `integrationOuterNormal` that directly yields the product of the two. This is

again due to the fact that this product is used very frequently in finite volume methods, so that a dedicated method makes sense. Even more importantly, it can be much more efficient to evaluate the scaled normal $|\gamma|_{ij}\mathbf{n}_{ij}$ directly, rather than computing the two separate factors first.

The second half of the intersection loop computes the actual update δ_i for the element T_i.

```
84        // Outflow contributions
85        update[i] -= c[i]*std::max(0.0,factor)/elementVolume;
86
87        // Inflow contributions
88        if (factor<=0) // inflow!
89        {
90          // handle interior face
91          if (intersection.neighbor())
92          {
93            // access neighbor
94            auto j = mapper.index(intersection.outside());
95            update[i] -= c[j]*factor/elementVolume;
96          }
97
98          // handle boundary face
99          if (intersection.boundary())
100           update[i] -= b(faceCenter,t)*factor/elementVolume;
101       }
102
103       // for time step calculation
104       sumfactor += std::max(0.0,factor);
105     }              // end loop over all intersections
```

Line 85 adds $-C_i \frac{|\gamma_{ij}|}{|T_i|} \max(0, v\mathbf{n})$, which covers the case that the current intersection is an outflow boundary of the element T_i. If $v|\gamma_{ij}|\mathbf{n}_{ij} < 0$,i.e., if there is flow into element T_i, we need to distinguish whether γ_{ij} really is the intersection with a second element T_j, or whether γ_{ij} is part of the domain boundary $\partial\Omega$ (in which case it is automatically on the inflow boundary Γ_{in}). Lines 88–101 cover these two cases. Observe how the intersection knows whether there is an adjacent element T_j (through the `neighbor` method), and whether we are on the domain boundary (through the `boundary` method). If the grid is distributed across several processors, both methods may return `false` at the same time. However, in this simple sequential example this cannot happen.

Finally, Line 104 sums up the outflow contributions for the stability bound (10). This ends the loop over the intersections.

The loop over the elements ends shortly after

```
109     // compute dt restriction
110     dt = std::min(dt,elementVolume/sumfactor);
111   }        // end loop over the grid elements
```

All that is left to do is to compute the current minimum over the outflows in Line 110.

Finally, the new time step size is multiplied with a safety factor of 0.99, and used to update the concentration vector.

```
115   // scale dt with safety factor
116   dt *= 0.99;
117
118   // update the concentration vector
119   for (unsigned int i=0; i<c.size(); ++i)
120     c[i] += dt*update[i];
121
122   return dt;
123 }
```

This ends the discussion of the `evolve` method.

5 Complete source code of the finite element example

```
 1 #include <config.h>
 2
 3 #include <vector>
 4
 5 #include <dune/common/function.hh>
 6
 7 #include <dune/geometry/quadraturerules.hh>
 8
 9 // { include_uggrid_begin }
10 #include <dune/grid/uggrid.hh>
11 // { include_uggrid_end }
12 #include <dune/grid/io/file/gmshreader.hh>
13 #include <dune/grid/io/file/vtk/vtkwriter.hh>
14
15 #include <dune/istl/matrix.hh>
16 // { include_matrix_vector_begin }
17 #include <dune/istl/bvector.hh>
18 #include <dune/istl/bcrsmatrix.hh>
19 // { include_matrix_vector_end }
20 #include <dune/istl/matrixindexset.hh>
21 #include <dune/istl/preconditioners.hh>
22 #include <dune/istl/solvers.hh>
23 #include <dune/istl/matrixmarket.hh>
24
25 #include <dune/functions/functionspacebases/pqknodalbasis.hh>
26 #include <dune/functions/functionspacebases/interpolate.hh>
27
28
29 // { using_namespace_dune_begin }
30 using namespace Dune;
31 // { using_namespace_dune_end }
32
33 // Compute the stiffness matrix for a single element
34 // { local_assembler_signature_begin }
35 template <class LocalView, class MatrixType>
36 void assembleElementStiffnessMatrix( const LocalView& localView, MatrixType& elementMatrix)
37 // { local_assembler_signature_end }
38 {
39     // { local_assembler_get_geometry_begin }
40     using Element = typename LocalView::Element;
41     auto element = localView.element();
42     const int dim = Element::dimension;
43     auto geometry = element.geometry();
44     // { local_assembler_get_geometry_end }
45
46     // Get set of shape functions for this element
47     // { get_shapefunctions_begin }
48     const auto& localFiniteElement = localView.tree().finiteElement();
49     // { get_shapefunctions_end }
50
51     // Set all matrix entries to zero
52     // { init_element_matrix_begin }
53     elementMatrix.setSize(localFiniteElement.size(), localFiniteElement.size());
54     elementMatrix = 0;     // fills the entire matrix with zeros
55     // { init_element_matrix_end }
56
57     // Get a quadrature rule
58     // { get_quadrature_rule_begin }
```

```
59       int order = 2*(dim*localFiniteElement.localBasis().order()-1);
60       const auto& quadRule = QuadratureRules<double, dim>::rule(element.type(), order);
61       // {    get_quadrature_rule_end    }
62
63       // Loop over all quadrature points
64       // {    loop_over_quad_points_begin    }
65       for (const auto& quadPoint : quadRule) {
66       // {    loop_over_quad_points_end    }
67
68           // {    get_quad_point_info_begin    }
69           // Position of the current quadrature point in the reference element
70           const auto quadPos = quadPoint.position();
71
72           // The transposed inverse Jacobian of the map from the reference element to the element
73           const auto jacobian = geometry.jacobianInverseTransposed(quadPos);
74
75           // The multiplicative factor in the integral transformation formula
76           const auto integrationElement = geometry.integrationElement(quadPos);
77           // {    get_quad_point_info_end    }
78
79           // {    compute_gradients_begin    }
80           // The gradients of the shape functions on the reference element
81           std::vector<FieldMatrix<double,1,dim> > referenceGradients;
82           localFiniteElement.localBasis().evaluateJacobian(quadPos, referenceGradients);
83
84           // Compute the shape function gradients on the real element
85           std::vector<FieldVector<double,dim> > gradients(referenceGradients.size());
86           for (size_t i=0; i<gradients.size(); i++)
87               jacobian.mv(referenceGradients[i][0], gradients[i]);
88           // {    compute_gradients_end    }
89
90           // Compute the actual matrix entries
91           // {    compute_matrix_entries_begin    }
92           for (size_t i=0; i<elementMatrix.N(); i++)
93               for (size_t j=0; j<elementMatrix.M(); j++ )
94                   elementMatrix[localView.tree().localIndex(i)][localView.tree().localIndex(j)]
95                       += ( gradients[i] * gradients[j] ) * quadPoint.weight() * integrationElement;
96           // {    compute_matrix_entries_end    }
97
98       }
99
100  }
101
102
103  // Compute the source term for a single element
104  template <class LocalView>
105  void getVolumeTerm( const LocalView& localView,
106                      BlockVector<FieldVector<double,1> >& localRhs,
107                      const std::function<double(FieldVector<double,LocalView::Element::dimension>)> volumeTerm)
108  {
109      using Element = typename LocalView::Element;
110      auto element = localView.element();
111      const int dim = Element::dimension;
112
113      // Set of shape functions for a single element
114      const auto& localFiniteElement = localView.tree().finiteElement();
115
116      // Set all entries to zero
117      localRhs.resize(localFiniteElement.size());
118      localRhs = 0;
119
120      // A quadrature rule
121      int order = dim;
122      const auto& quadRule = QuadratureRules<double, dim>::rule(element.type(), order);
123
124      // Loop over all quadrature points
125      for (const auto& quadPoint : quadRule) {
126
127          // Position of the current quadrature point in the reference element
128          const FieldVector<double,dim>& quadPos = quadPoint.position();
129
130          // The multiplicative factor in the integral transformation formula
131          const double integrationElement = element.geometry().integrationElement(quadPos);
132
133          double functionValue = volumeTerm(element.geometry().global(quadPos));
134
135          // Evaluate all shape function values at this point
136          std::vector<FieldVector<double,1> > shapeFunctionValues;
137          localFiniteElement.localBasis().evaluateFunction(quadPos, shapeFunctionValues);
138
139          // Actually compute the vector entries
140          for (size_t i=0; i<localRhs.size(); i++)
141              localRhs[i] += shapeFunctionValues[i] * functionValue * quadPoint.weight() * integrationElement;
142      }
143
144  }
145
146  // Get the occupation pattern of the stiffness matrix
147  template <class Basis>
```

```
148 void getOccupationPattern(const Basis& basis, MatrixIndexSet& nb)
149 {
150   nb.resize(basis.size(), basis.size());
151
152   auto gridView = basis.gridView();
153
154   // A loop over all elements of the grid
155   auto localView = basis.localView();
156   auto localIndexSet = basis.localIndexSet();
157
158   for (const auto& element : elements(gridView))
159   {
160     localView.bind(element);
161     localIndexSet.bind(localView);
162
163     for(size_t i=0; i<localIndexSet.size(); i++)
164     {
165       // The global index of the i-th vertex of the element
166       auto row = localIndexSet.index(i);
167
168       for (size_t j=0; j<localIndexSet.size(); j++ )
169       {
170         // The global index of the j-th vertex of the element
171         auto col = localIndexSet.index(j);
172         nb.add(row,col);
173       }
174     }
175   }
176 }
177
178
179 /** \brief Assemble the Laplace stiffness matrix on the given grid view */
180 // { global_assembler_signature_begin }
181 template <class Basis>
182 void assemblePoissonProblem(const Basis& basis,
183                             BCRSMatrix<FieldMatrix<double,1,1> >& matrix,
184                             BlockVector<FieldVector<double,1> >& rhs,
185                             const std::function<double(FieldVector<double,Basis::GridView::dimension>)> volumeTerm)
186 // { global_assembler_signature_end }
187 {
188     // { assembler_get_grid_info_begin }
189     auto gridView = basis.gridView();
190     // { assembler_get_grid_info_end }
191
192     // MatrixIndexSets store the occupation pattern of a sparse matrix.
193     // They are not particularly efficient, but simple to use.
194     // { assembler_zero_matrix_begin }
195     MatrixIndexSet occupationPattern;
196     getOccupationPattern(basis, occupationPattern);
197
198     // ... and give it the occupation pattern we want.
199     occupationPattern.exportIdx(matrix);
200
201     // Set all entries to zero
202     matrix = 0;
203     // { assembler_zero_matrix_end }
204
205     // { assembler_zero_vector_begin }
206     // set rhs to correct length
207     rhs.resize(basis.size());
208
209     // Set all entries to zero
210     rhs = 0;
211     // { assembler_zero_vector_end }
212
213     // A loop over all elements of the grid
214     // { assembler_element_loop_begin }
215     auto localView = basis.localView();
216     auto localIndexSet = basis.localIndexSet();
217
218     for (const auto& element : elements(gridView))
219     {
220     // { assembler_element_loop_end }
221
222         // Now let's get the element stiffness matrix
223         // A dense matrix is used for the element stiffness matrix
224         // { assembler_assemble_element_matrix_begin }
225         localView.bind(element);
226         localIndexSet.bind(localView);
227
228         Matrix<FieldMatrix<double,1,1> > elementMatrix;
229         assembleElementStiffnessMatrix(localView, elementMatrix);
230         // { assembler_assemble_element_matrix_end }
231
232         //
233         // { assembler_add_element_matrix_begin }
234         for(size_t i=0; i<elementMatrix.N(); i++)
235         {
236             // The global index of the i-th degree of freedom of the element
```

```
237                auto row = localIndexSet.index(i);
238
239                for ( size_t  j=0; j<elementMatrix.M(); j++ )
240                {
241                    // The global index of the j−th degree of freedom of the element
242                    auto col = localIndexSet.index(j);
243                    matrix[row][col] += elementMatrix[i][j];
244                }
245            }
246            // {   assembler_add_element_matrix_end   }
247
248            // Now get the local contribution to the right −hand side vector
249            BlockVector<FieldVector<double,1> > localRhs;
250            getVolumeTerm(localView, localRhs, volumeTerm);
251
252            for ( size_t  i=0; i<localRhs.size();  i++)
253            {
254                // The global index of the i−th vertex of the element
255                auto row = localIndexSet.index(i);
256                rhs[row] += localRhs[i];
257            }
258        }
259    }
260
261
262    int main (int argc, char *argv[]) try
263    {
264      // Set up MPI, if available
265      MPIHelper::instance(argc, argv);
266
267
268        // ////////////////////////////////
269        //   Generate the grid
270        // ////////////////////////////////
271
272        // {   create_grid_begin   }
273        const int dim = 2;
274        typedef UGGrid<dim> GridType;
275        std::shared_ptr<GridType> grid(GmshReader<GridType>::read("l−shape.msh"));
276
277        grid−>globalRefine(2);
278
279        typedef GridType::LeafGridView GridView;
280        GridView gridView = grid−>leafGridView();
281        // {   create_grid_end   }
282
283        // /////////////////////////////////////////////////////////
284        //   Stiffness matrix and right hand side vector
285        // /////////////////////////////////////////////////////////
286
287        // {   create_matrix_vector_begin   }
288        typedef BlockVector<FieldVector<double,1> > VectorType;
289        typedef BCRSMatrix<FieldMatrix<double,1,1> > MatrixType;
290
291        VectorType rhs;
292        MatrixType stiffnessMatrix;
293        // {   create_matrix_vector_end   }
294
295        // /////////////////////////////////////////////////////////
296        //   Assemble the system
297        // /////////////////////////////////////////////////////////
298
299        // {   call_assembler_begin   }
300        Functions::PQkNodalBasis<GridView,1> basis(gridView);
301
302        auto sourceTerm = [](const FieldVector<double,dim>& x){return −5.0;};
303        assemblePoissonProblem(basis, stiffnessMatrix, rhs, sourceTerm);
304        // {   call_assembler_end   }
305
306        // Determine Dirichlet dofs by marking all degrees of freedom whose Lagrange nodes comply with a
307        // given predicate .
308        // Since interpolating into a vector <bool> is currently not supported ,
309        // we use a vector <char> which, in contrast to vector <bool>
310        // is a real container .
311        // {   dirichlet_marking_begin   }
312        auto predicate = [](auto p){return p[0]< 1e−8 or p[1] < 1e−8 or (p[0] > 0.4999 and p[1] > 0.4999);};
313
314        // Interpolating the predicate will mark all desired Dirichlet degrees of freedom
315        std::vector<char> dirichletNodes;
316        Functions::interpolate(basis, dirichletNodes, predicate);
317        // {   dirichlet_marking_end   }
318
319      // ////////////////////////////////////////
320      //   Modify Dirichlet rows
321      // ////////////////////////////////////////
322      // {   dirichlet_matrix_modification_begin   }
323      // loop over the matrix rows
324      for ( size_t  i=0; i<stiffnessMatrix.N();  i++)
325      {
```

```
326        if (dirichletNodes[i])
327        {
328          auto cIt    = stiffnessMatrix[i].begin();
329          auto cEndIt = stiffnessMatrix[i].end();
330          // loop over nonzero matrix entries in current row
331          for (; cIt!=cEndIt; ++cIt)
332            *cIt = (i==cIt.index()) ? 1.0 : 0.0;
333        }
334      }
335      // {   dirichlet_matrix_modification_end   }
336
337      // Set Dirichlet values
338      // {   dirichlet_rhs_modification_begin   }
339      auto dirichletValues = [](auto p){return (p[0]< 1e-8 or p[1] < 1e-8) ? 0 : 0.5;};
340      Functions::interpolate(basis,rhs,dirichletValues, dirichletNodes);
341      // {   dirichlet_rhs_modification_end   }
342
343      ////////////////////////////////////////////////////////////////////////////
344      // Write matrix and load vector to files, to be used in later examples
345      ////////////////////////////////////////////////////////////////////////////
346      // {   matrix_rhs_writing_begin   }
347      storeMatrixMarket(stiffnessMatrix, "poisson-matrix.mm");
348      storeMatrixMarket(rhs, "poisson-rhs.mm");
349      // {   matrix_rhs_writing_end   }
350
351      // ///////////////////////
352      //   Compute solution
353      // ///////////////////////
354
355      // {   algebraic_solving_begin   }
356      // Choose an initial iterate
357      VectorType x(basis.size());
358      x = 0;
359
360      // Technicality: turn the matrix into a linear operator
361      MatrixAdapter<MatrixType,VectorType,VectorType> linearOperator(stiffnessMatrix);
362
363      // Sequential incomplete LU decomposition as the preconditioner
364      SeqILU0<MatrixType,VectorType,VectorType> preconditioner(stiffnessMatrix,1.0);
365
366      // Preconditioned conjugate-gradient solver
367      CGSolver<VectorType> cg(linearOperator,
368                              preconditioner,
369                              1e-4, // desired residual reduction factor
370                              50,   // maximum number of iterations
371                              2);   // verbosity of the solver
372
373      // Object storing some statistics about the solving process
374      InverseOperatorResult statistics;
375
376      // Solve!
377      cg.apply(x, rhs, statistics);
378      // {   algebraic_solving_end   }
379
380      // Output result
381      // {   vtk_output_begin   }
382      VTKWriter<GridView> vtkWriter(gridView);
383      vtkWriter.addVertexData(x, "solution");
384      vtkWriter.write("poissonequation-result");
385      // {   vtk_output_end   }
386
387      return 0;
388     }
389    // Error handling
390     catch (std::exception& e) {
391       std::cout << e.what() << std::endl;
392       return 1;
393     }
```

6 Complete source code of the finite volume example

```
1    // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2    // vi: set et ts=4 sw=2 sts=2:
3    #include "config.h"              // know what grids are present
4
5    #include <iostream>              // for input/output to shell
6    #include <vector>                // STL vector class
7
8    #include <dune/common/parallel/mpihelper.hh>
9
10   #include <dune/grid/common/mcmgmapper.hh>
11   #include <dune/grid/yaspgrid.hh>
12   #include <dune/grid/io/file/vtk.hh>
13
14   // {   using_namespace_dune_begin   }
15   using namespace Dune;
```

```
16  // {  using_namespace_dune_end  }
17
18  // {  evolve_signature_begin  }
19  template<class GridView, class Mapper>
20  double evolve (const GridView& gridView,
21                 const Mapper& mapper,
22                 std::vector<double>& c,
23                 const std::function<FieldVector<typename GridView::ctype,GridView::dimensionworld>
24                                     (FieldVector<typename GridView::ctype,GridView::dimension>,double)> v,
25                 const std::function<double
26                                     (FieldVector<typename GridView::ctype,GridView::dimension>,double)> b,
27                 double t)
28  // {  evolve_signature_end  }
29  {
30    // {  evolve_init_begin  }
31    // Grid dimension
32    const int dim = GridView::dimension;
33
34    // allocate a temporary vector for the update
35    std::vector<double> update(c.size());
36    std::fill(update.begin(), update.end(), 0.0);
37
38    // initialize dt very large
39    double dt = std::numeric_limits<double>::max();
40    // {  evolve_init_end  }
41
42    // compute update vector and optimum dt in one grid traversal
43    // {  element_loop_begin  }
44    for (const auto& element : elements(gridView))
45    {
46      // element geometry
47      auto geometry = element.geometry();
48
49      // element volume
50      double elementVolume = geometry.volume();
51
52      // Element index
53      auto i = mapper.index(element);
54
55      // Will store the total element outflow
56      double sumfactor = 0.0;
57      // {  element_loop_end  }
58
59      // run through all intersections γij with neighbors and boundary
60      // {  intersection_loop_begin  }
61      for (const auto& intersection : intersections(gridView,element))
62      {
63        // get geometry type of face
64        auto igeo = intersection.geometry();
65
66        // center of face in global coordinates
67        auto faceCenter = igeo.center();
68
69        // evaluate velocity at face center
70        auto velocity = v(faceCenter,t);
71
72        // get the center of the intersection in local coordinates
73        const auto& intersectionReferenceElement = ReferenceElements<double,dim-1>::general(intersection.type());
74        auto intersectionCenter = intersectionReferenceElement.position(0,0);
75
76        // get normal vector scaled with volume
77        auto integrationOuterNormal = intersection.integrationOuterNormal(intersectionCenter);
78
79        // Compute factor occuring in flux formula v · nij · |γij|
80        double factor = velocity*integrationOuterNormal;
81        // {  intersection_loop_initend  }
82
83        // {  intersection_loop_mainbegin  }
84        // Outflow contributions
85        update[i] -= c[i]*std::max(0.0,factor)/elementVolume;
86
87        // Inflow contributions
88        if (factor<=0) // inflow!
89        {
90          // handle interior face
91          if (intersection.neighbor())
92          {
93            // access neighbor
94            auto j = mapper.index(intersection.outside());
95            update[i] -= c[j]*factor/elementVolume;
96          }
97
98          // handle boundary face
99          if (intersection.boundary())
100           update[i] -= b(faceCenter,t)*factor/elementVolume;
101       }
102
103       // for time step calculation
```

```
104         sumfactor += std::max(0.0,factor);
105       }             // end loop over all intersections
106       // {  intersection_loopend  }
107
108       // {  element_loop_beginend  }
109       // compute dt restriction
110       dt = std::min(dt,elementVolume/sumfactor);
111     }        // end loop over the grid elements
112     // {  element_loop_end  }
113
114     // {  evolve_laststeps  }
115     // scale dt with safety factor
116     dt *= 0.99;
117
118     // update the concentration vector
119     for (unsigned int i=0; i<c.size(); ++i)
120       c[i] += dt*update[i];
121
122     return dt;
123   }
124   // {  evolve_end  }
125
126   //===============================================================
127   // The main function creates objects and does the time loop
128   //===============================================================
129
130   // {  main_begin  }
131   int main (int argc , char ** argv) try
132   {
133     // Set up MPI, if available
134     Dune::MPIHelper::instance(argc, argv);
135     // {  main_signature_end  }
136
137     // {  create_grid_begin  }
138     const int dim = 2;
139     typedef YaspGrid<dim> GridType;
140     GridType grid({1.0,1.0},    // upper right corner, the lower left one is (0,0)
141                   {80,80});     // number of elements per direction
142
143     typedef GridType::LeafGridView GridView;
144     GridView gridView = grid.leafGridView();
145     // {  create_grid_end  }
146
147     // Assigns a unique number to each element
148     // {  create_concentration_begin  }
149     MultipleCodimMultipleGeomTypeMapper<GridView,MCMGElementLayout>
150     mapper(gridView);
151
152     // Allocate a vector for the concentration
153     std::vector<double> c(mapper.size());
154     // {  create_concentration_end  }
155
156     // initial concentration
157     // {  lambda_initial_concentration_begin  }
158     auto c0 = [](auto x) { return (x.two_norm()>0.125 && x.two_norm()<0.5) ? 1.0 : 0.0; };
159     // {  lambda_initial_concentration_end  }
160
161     // {  sample_initial_concentration_begin  }
162     // Iterate over grid elements and evaluate c0 at element centers
163     for (const auto& element : elements(gridView))
164     {
165       // Get element geometry
166       auto geometry = element.geometry();
167
168       // Get global coordinate of element center
169       auto global = geometry.center();
170
171       // Sample initial concentration c_0 at the element center
172       c[mapper.index(element)] = c0(global);
173     }
174     // {  sample_initial_concentration_end  }
175
176     // Construct VTK writer
177     // {  construct_vtk_writer_begin  }
178     auto vtkWriter = std::make_shared<Dune::VTKWriter<GridView> >(gridView);
179     VTKSequenceWriter<GridView> vtkSequenceWriter(vtkWriter,
180                                                   "concentration");  // file name
181
182     // Write the initial values
183     vtkWriter->addCellData(c,"concentration");
184     vtkSequenceWriter.write( 0.0 );  // 0.0 is the current time
185     // {  construct_vtk_writer_end  }
186
187     // now do the time steps
188     // {  time_loop_begin  }
189     double t=0;
190     double tend=0.6;
191     int k=0;
192     const double saveInterval = 0.1;
```

```
193     double saveStep = 0.1;
194
195     auto b = [](const Dune::FieldVector<GridView::ctype,GridView::dimensionworld>& x, double t)
196     {
197       return 0.0;
198     };
199
200     auto v = [](const Dune::FieldVector<GridView::ctype,GridView::dimensionworld>& x, double t)
201     {
202       return Dune::FieldVector<double,GridView::dimensionworld> (1.0);
203     };
204
205     while (t<tend)
206     {
207       // apply finite volume scheme
208       double dt = evolve(gridView,mapper,c,v,b,t);
209
210       // augment time and time step counter
211       t += dt;
212       ++k;
213
214       // check if data should be written
215       if (t >= saveStep)
216       {
217         // Write data. We do not have to call addCellData again!
218         vtkSequenceWriter.write( t );
219
220         // increase saveStep for next interval
221         saveStep += saveInterval;
222       }
223
224       // print iteration number, time, and time step size
225       std::cout << "k=" << k << " t=" << t << " dt=" << dt << std::endl;
226     }
227     // { time_loop_end }
228
229     // done
230     return 0;
231     // { catch_block_begin }
232   }
233   catch (std::exception & e) {
234     std::cout << e.what() << std::endl;
235     return 1;
236   }
237   // { main_end }
```

GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

`<http://fsf.org/>`

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed

to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However,

you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version.

Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts

may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies

that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

References

[1] *VTK File Formats (for VTK Version 4.2).* URL `www.vtk.org/img/file-formats.pdf`.

[2] P. Bastian and M. Blatt. On the Generic Parallelisation of Iterative Solvers for the Finite Element Method. *Int. J. Computational Science and Engineering*, 4 (1):56–69, 2008. URL `http://dx.doi.org/10.1504/IJCSE.2008.021112`.

[3] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz–Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Comp. Vis. Sci*, 1:27–40, 1997.

[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82 (2-3):121–138, 2008.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2-3):103–119, 2008.

[6] M. Blatt and P. Bastian. The Iterative Solver Template Library. In B. Kagström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, number 4699 in Lecture Notes in Scientific Computing, pages 666–675, 2007. URL `http://dx.doi.org/10.1007/978-3-540-75755-9_82`.

[7] Markus Blatt. *A Parallel Algebraic Multigrid Method for Elliptic Problems with Highly Discontinuous Coefficients.* PhD thesis, Universität Heidelberg, 2010.

[8] Markus Blatt, Ansgar Burchardt, Andreas Dedner, Christian Engwer, Jorrit Fahlke, Bernd Flemisch, Christoph Gersbacher, Carsten Gräser, Felix Gruber, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Tobias Malkmus, Steffen Müthing, Martin Nolte, Marian Piatkowski, and Oliver Sander. The distributed and unified numerics environment, version 2.4. *Archive of Numerical Software*, submitted.

[9] Adrian Burri, Andreas Dedner, Robert Klöfkorn, and Mario Ohlberger. An efficient implementation of an adaptive and parallel grid in DUNE. In *Proc. of the 2nd Russian–German Advanced Research Workshop on Computational Science and High Performance Computing*, 2005.

[10] Christian Engwer, Carsten Gräser, Steffen Müthing, and Oliver Sander. The interface for functions in the dune-functions module. arXiv 1512.06136, 2015.

[11] Christophe Geuzaine and François Remacle. *Gmsh Reference Manual*, 2015. URL `http://www.geuz.org/gmsh/doc/texinfo/gmsh.pdf`.

[12] Steffen Müthing. *A Flexible Framework for Multi Physics and Multi Domain PDE Simulations*. PhD thesis, Universität Stuttgart, 2015.

[13] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2nd edition, 2003.

[14] Alfred Schmidt and Kunibert G. Siebert. *Design of Adaptive Finite Element Software – The Finite Element Toolbox ALBERTA*, volume 42 of *Lecture Notes in Computer Science and Engineering*. Springer, 2005.

[15] Bernhard Schupp. *Entwicklung eines effizienten Verfahrens zur Simulation kompressibler Strömungen in 3D auf Parallelrechnern*. PhD thesis, Albert-Ludwigs-Universität Freiburg, Mathematische Fakultät, 1999.