Getting started with DUNE

For Dune version 2.7

Oliver Sander*

December 18, 2020

This document describes how to get started with the Distributed and Unified Numerics Environment (DUNE). DUNE is a set of open-source C++ libraries that help to 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 a first own DUNE module. 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, published by Springer Verlag [14].[1] Springer kindly consented to having this chapter published independently from the rest of the book.

*Fakultät für Mathematik, TU Dresden, Germany, email: oliver.sander@tu-dresden.de

[1]DOI: 10.1007/978-3-030-59702-3

Contents

Preamble

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,[2] 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`. It has 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 [2, 3] on the DUNE grid interface, [1, 5, 6] on linear algebra, [9] on discrete functions, and [12] on the `dune-typetree` library. The DUNE release 2.4 has been presented in [7], and the release 2.7 in [4].

1 Installation of Dune

The first step into the DUNE world is its installation. We have tried to make this as painless as possible. The following instructions assume a Unix-style command shell and toolchain installed. This will be easy to obtain on all flavors of Linux and Apple OS X. On Windows there is the WINDOWS SUBSYSTEM FOR LINUX,[3] or the CYGWIN environment.[4]

Running the examples in this text requires eight DUNE modules: `dune-common`, `dune-geometry`, `dune-grid`, `dune-istl`, `dune-localfunctions`, `dune-uggrid`, `dune-typetree`, and `dune-functions`.

1.1 Installation from Binary Packages

Installation is easiest when using precompiled packages. At the time of writing this is the case for the Debian Linux distribution and many of its derivatives, but there may be more. An up-to-date list is available on the DUNE project web page at `www.dune-project.org`. On a Debian-type system, type

```
sudo apt-get install libdune-functions-dev \
                     libdune-istl-dev \
                     libdune-uggrid-dev
```

to install all those eight DUNE modules. The ones not listed explicitly are added automatically. The DUNE modules are then installed globally on the machine, and the DUNE build system will find them.

1.2 Installation from Source

If there are no precompiled packages available, or when working without `sudo` rights, then DUNE has to be installed from the source code. First, download the code from the

[2] `https://www.dune-project.org/about/license`
[3] `https://docs.microsoft.com/en-us/windows/wsl/about`
[4] `http://cygwin.com`

DUNE website at www.dune-project.org. It is recommended to download the release tarballs, named

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

where <modulename> is one of common, geometry, grid, etc., and X.Y.Z is the release version number. The code examples in this book require at least version 2.7.0.

Those who need very recent features can also get the bleeding edge development versions of the DUNE modules. The DUNE source code is stored and managed using the git version control software.[5] 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 <modulename> is replaced by common, geometry, grid, localfunctions, or istl. The remaining three modules are available from the staging namespace:

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

These eight commands create eight directories, one for each module. Each directory contains the latest development version of the source code of the corresponding module. If desired, this development version can be replaced by a particular release version by calling, e.g.,

```
git checkout releases/2.7
```

in the directory. See the git documentation for details.

Suppose that there is an empty directory called dune, and that the sources of the eight DUNE modules have been downloaded into this directory. When using the tarballs, these have to be unpacked by

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

for each DUNE module. To build them all, enter the dune directory and type[6]

```
./dune-common/bin/dunecontrol cmake : make
```

This configures and builds all DUNE modules, which may take several minutes. For brevity, the two commands cmake and make can be called together as:

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

Once the process has completed, DUNE can be installed by typing

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

[5] https://git-scm.com

[6] Replace dune-common by dune-commmon-X.Y.Z when using the tarballs.

This will install the DUNE core modules to `/usr/local`, and requires root access.

To install DUNE to a non-standard location, a custom installation path can be set. For this, create a text file `dune.opts`, which should contain

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

Then call

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

and

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

The `dunecontrol` program will pick up the variable `CMAKE_FLAGS` from the options file and use it as a command line option for any call to `cmake`, which in turn is used to configure the individual modules. The particular option of this example will tell `cmake` that all modules should be installed to `/the/desired/installation/path`.

Unfortunately, it has to be mentioned that as of DUNE Version 2.7, working with installed modules is still not very mature, and may lead to build failures. As a consequence of this, the installation of DUNE modules is optional, and the DUNE build system will also accept non-installed modules as build dependencies.

When using the `dunecontrol` program to manage further modules, it has to be told where to find the DUNE core modules. This is done by prepending the path `/the/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 the current directory should be searched then the `DUNE_CONTROL_PATH` variable has to contain `:.:` somewhere.[7]

2 A First Dune Application

DUNE is organized in modules, where each module is roughly a directory with a predefined structure.Each such module implements a C++ library that can be used from other code. Although not necessary, however, it can be convenient to write new code into a new DUNE module, because that simplifies dependency tracking, and allows further DUNE modules to easily depend on the new code. The first step for this is to create a new DUNE module template.

2.1 Creating a new Module

In the following we assume again that there is a Unix-type command shell available, and that DUNE has been installed successfully either into the standard location, or that `DUNE_CONTROL_PATH` contains the installation path. To create a new module, DUNE provides a special program called `duneproject`. To invoke it, simply type

[7] The dot denotes the current directory, and the colons are separators.

in the shell. If the program is not installed globally, use the version from dune-common/bin.

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

The next question asks for other DUNE modules that the new module will depend upon. To help, duneproject has already collected a list of all modules it sees on the system. These are the globally installed ones, and the ones in directories listed in the DUNE_CONTROL_PATH environment variable. After the installation described above one should see at least dune-common, dune-geometry, dune-grid, dune-istl, dune-localfunctions, dune-typetree, dune-uggrid, and dune-functions. The required ones should be entered in a white-space-separated list (and it is easy to add further ones later). For the purpose of this introduction please select them all.

Next is the question for a module version number. These should start with X.Y (X and Y being numbers), and can optionally end with a third number .Y or an arbitrary string. This is followed by a question for an email address. This address will appear in the file dune.module of the module (and nowhere else), and will be a point of contact for others with an interest in the module. After this, the duneproject program exits and there is now an blank module dune-foo:

```
~/dune: ls
dune-foo
```

A tool like the tree program can be used to 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

7 directories, 15 files
```

This tree contains:

- The `cmake` configuration files for the DUNE build system,
- A text file `dune.module`, which contains some meta data of the module,
- a small example program in `dune-foo.cc`.

2.2 Testing the new Module

The new module created by `duneproject` contains one C++ source code file `src/dune-foo.cc`. This is a small test program that allows to verify whether the module has been built properly. Configuring and building the module is controlled by the `dunecontrol` program again.

The process is hardly any different from configuring and building the DUNE core modules as described in the previous section. Just move to the `dune` directory again, and type

```
dunecontrol all
```

in the shell. This will output lots of information while it runs, but none of it should be of any concern right now (unless actual error messages appear somewhere). Once `dunecontrol` has terminated there is a new executable `dune-foo` in `dune-foo/build-cmake/src`. Start it with

```
~/dune: ./dune-foo/build-cmake/src/dune-foo
```

and it will print

```
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.[8] More specifically, we will compute the weak solution of the Poisson equation

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

[8] Readers that are unsure about how the finite element method works should consult one of the numerous textbooks on the subject.

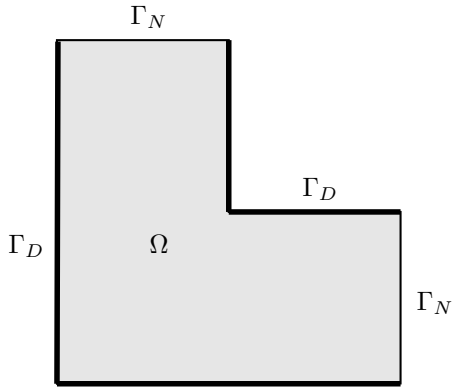

Figure 1: A simple domain Ω with Dirichlet boundary Γ_D (thick lines) and Neumann boundary Γ_N

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

$$u = \begin{cases} 0 & \text{on } \{0\} \times [0,1] \cup [0,1] \times \{0\}, \\ 0.5 & \text{on } \{0.5\} \times [0.5,1] \cup [0.5,1] \times \{0.5\}, \end{cases} \tag{2}$$

and zero Neumann conditions on the remainder of the boundary. The domain and boundary conditions are shown in Figure 1.

The example program solves the Poisson equation and outputs the result to a file which can be opened with the ParaView visualization software.[9] The program code is contained in a single file. We will not show 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, readers of this document in electronic form can access the source code file through the little pin icon in the page margin. The easiest way to build the example is to copy the program file into `dune-foo/src/`, and then either to adjust the file `dune-foo/src/CMakeLists.txt`, or to simply replace the existing file `dune-foo.cc` with the example, leaving `CMakeLists.txt` intact.

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 there would be no detailed control over the inner working of the program. On the other hand, an example implementation could be written to depend on only a few low-level modules. In this case more code would have to be written by hand. This would mean more work, but also more control and understanding of how the programs works exactly.

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

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

written by hand. Chapter 11.3 of [14] contains an alternative implementation that uses `dune-pdelab` to assemble the algebraic problem.

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—the user is 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` Method

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

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

It is located at the end of the file, and is preceded by a few classes that make up the finite element assembler. As a first action, it sets up MPI [10] if available:

```
275   // Set up MPI, if available
276   MPIHelper::instance(argc, argv);
```

Note that this command is needed even if the program is purely sequential, because it does some vital internal initialization work.

Remember that all DUNE code resides in the `Dune` namespace. Hence type names like `MPIHelper` need to be prefixed by `Dune::`. Since that makes the code more difficult to read, the example file contains the line

```
28  using namespace Dune;
```

near the top. This allows to omit the `Dune::` prefixes.

3.1.1 Creating a Grid

The first real action is to create a grid. The example program will use the triangle grid shown in Figure 2. A grid implementation in DUNE that supports unstructured triangle grids is the `UGGrid` grid manager.

The grid itself is read from a file in the GMSH format [10].[11] The file can be obtained 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`:

```
284   constexpr int dim = 2;
285   using Grid = UGGrid<dim>;
286   std::shared_ptr<Grid> grid = GmshReader<Grid>::read("l-shape.msh");
287
288   grid->globalRefine(2);
289
290   using GridView = Grid::LeafGridView;
291   GridView gridView = grid->leafGridView();
```

[10] The Message Passing Interface, used for distributed computing (see `www.mpi-forum.org`).
[11] `http://gmsh.info`

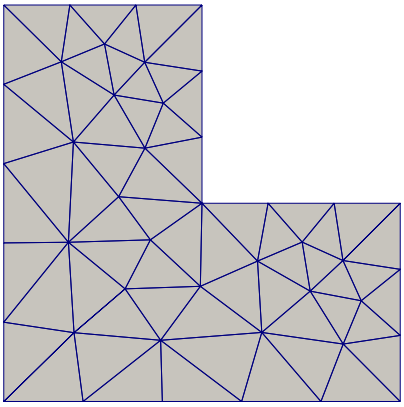

Figure 2: Unstructured coarse grid for the domain used by the example. The actual simulation happens on a grid that results from this one after two steps of uniform refinement.

The first two lines of this code block define the C++ data structure used for the finite element grid, and the third line loads the grid from the file into an object of this data structure. Note that the grid dimension `dim` is a compile-time parameter. Line 288 refines the grid twice uniformly, to get a result with higher resolution. For this part of the code to compile, it needs to have the lines

```
8  #include <dune/grid/uggrid.hh>
9  #include <dune/grid/io/file/gmshreader.hh>
```

at the top. The final two lines extract the result of the second refinement step as a non-hierarchical grid. This so-called *leaf grid view* is where the actual finite element computation will take place on.

Line 285 has introduced the type variable `Grid` to store the type of the grid data structure that is used. 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 `Grid`. This allows to change to, say, a structured cube grid by replacing only the definition of `Grid` and the subsequent constructor call. Indeed, to replace the unstructured grid by a structured cube grid for the unit square, replace Lines 285–286 by

```
using Grid = YaspGrid<dim>;
auto grid = std::make_shared<Grid>({1.0,1.0},    // Upper right corner,
                                                 // the lower left one
                                                 // is implicitly (0,0) here
                                   {10, 10});    // Number of elements
                                                 // per direction
```

The `YaspGrid` class,[12] from the file `dune/grid/yaspgrid.hh`, is the standard imple-

[12]The name means "Yet another structured parallel `Grid`", for historical reasons.

mentation 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 stiffness matrix and the load vector. For this we first need matrix and vector objects to assemble into. We get these with the lines

```
299   using Matrix = BCRSMatrix<double>;
300   using Vector = BlockVector<double>;
301
302   Matrix stiffnessMatrix;
303   Vector b;
```

Both `BCRSMatrix` and `BlockVector` are data structures from the `dune-istl` module, and obtained by placing

```
15  #include <dune/istl/bcrsmatrix.hh>
16  #include <dune/istl/bvector.hh>
```

near the top of the program. It is, however, easy to use other linear algebra implementations instead of the one from `dune-istl`. That is another advantage of DUNE.

The next code block selects the first-order finite element space and the volume source term:

```
311   Functions::LagrangeBasis<GridView,1> basis(gridView);
312
313   auto sourceTerm = [](const FieldVector<double,dim>& x){return -5.0;};
```

The space is specified by providing a *basis* for it. The closed-form volume source term is written as a C++ lambda object.

To make the `main` method more readable, the actual assembly code has been put into a subroutine, which is called next:

```
316   assemblePoissonProblem(basis, stiffnessMatrix, b, sourceTerm);
```

The `assemblePoissonProblem` method will be discussed 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 `b` 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 belongs to 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 proceeds in 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. We define a predicate class

that returns **true** or **false** depending on whether a given position is on the Dirichlet boundary implied by (2) or not. Then, we evaluate this predicate with respect to the Lagrange basis to obtain a vector of booleans with the desired information:

```
322  auto predicate = [](auto x)
323  {
324    return x[0] < 1e-8
325        || x[1] < 1e-8
326        || (x[0] > 0.4999 && x[1] > 0.4999);
327  };
328
329  // Evaluating the predicate will mark all Dirichlet degrees of freedom
330  std::vector<bool> dirichletNodes;
331  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:

```
338  // Loop over the matrix rows
339  for (size_t i=0; i<stiffnessMatrix.N(); i++)
340  {
341    if (dirichletNodes[i])
342    {
343      auto cIt    = stiffnessMatrix[i].begin();
344      auto cEndIt = stiffnessMatrix[i].end();
345      // Loop over nonzero matrix entries in current row
346      for (; cIt!=cEndIt; ++cIt)
347        *cIt = (cIt.index()==i) ? 1.0 : 0.0;
348    }
349  }
```

Line 339 loops over all matrix rows, and Line 341 tests whether the row corresponds to a Dirichlet degree of freedom. If this is the case then we loop over all nonzero matrix entries of the row, using the iterator loop that starts in Line 346. Note how this loop is very similar to iterator loops in the C++ standard library. Finally, Line 347 sets the matrix entries to the corresponding values of the identity matrix, by comparing column and row indices.

Modifying the right hand side vector is even easier. The previous loop could be extended to also overwrite the appropriate entries of the `b` array, but it is equally possible to use the interpolation functionality of the `dune-functions` module a second time:

```
354  auto dirichletValues = [](auto x)
355  {
356    return (x[0]< 1e-8 || x[1] < 1e-8) ? 0 : 0.5;
357  };
```

```
358   Functions::interpolate(basis,b,dirichletValues, dirichletNodes);
```

The code defines a new lambda object that implements the Dirichlet value function, and computes its Lagrange interpolation coefficients in the `b` vector object. The fourth argument of the `Functions::interpolate` method restricts the interpolation to those degrees of freedom where the corresponding entry in `dirichletNodes` is set. All others are untouched.

At this point, we have set up the linear system

$$Ax = b \tag{3}$$

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 `b`.

3.1.4 Solving the Algebraic Problem

To solve the algebraic system (3) 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 require the headers `dune/istl/solvers.hh` and `dune/istl/preconditioners.hh`, respectively. The following code constructs the preconditioned solver, and applies it to the algebraic problem:

```
376   // Choose an initial iterate that fulfills the Dirichlet conditions
377   Vector x(basis.size());
378   x = b;
379
380   // Turn the matrix into a linear operator
381   MatrixAdapter<Matrix,Vector,Vector> linearOperator(stiffnessMatrix);
382
383   // Sequential incomplete LU decomposition as the preconditioner
384   SeqILU<Matrix,Vector,Vector> preconditioner(stiffnessMatrix,
385                                               1.0);  // Relaxation factor
386
387   // Preconditioned conjugate gradient solver
388   CGSolver<Vector> cg(linearOperator,
389                       preconditioner,
390                       1e-5, // Desired residual reduction factor
391                       50,   // Maximum number of iterations
392                       2);   // Verbosity of the solver
393
394   // Object storing some statistics about the solving process
395   InverseOperatorResult statistics;
396
397   // Solve!
398   cg.apply(x, b, statistics);
```

After this code has run, the variable `x` contains the approximate solution of (3), `b` 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 does not want to compete), but the result can be written to a file in a variety of different formats for post-processing. In this example we will use the VTK file format [11], which is the standard format of the PARAVIEW software.[13] This requires the header `dune/grid/io/file/vtk/vtkwriter.hh`, and the following code:

```
403   VTKWriter<GridView> vtkWriter(gridView);
404   vtkWriter.addVertexData(x, "solution");
405   vtkWriter.write("getting-started-poisson-fem-result");
```

The first line creates a `VTKWriter` object and registers the grid view. 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 `getting-started-poisson-fem-result.vtu`.

3.1.6 Running the Program

With the exception of the stiffness matrix assembler (which is covered in the next section), the complete program has now been discussed. It can be built by typing `make` in the directory `build-cmake`. The executable `getting-started-poisson-fem` will then appear in the `build-cmake/src` directory. After program start one can see the `GmshReader` object giving some information about the grid file it is reading, followed by the conjugate gradients iterations:

```
Reading 2d Gmsh grid...
version 2.2 Gmsh file detected
file contains 43 nodes
file contains 90 elements
number of real vertices = 43
number of boundary elements = 22
number of elements = 62
=== Dune::IterativeSolver
 Iter          Defect            Rate
    0          3.26472
    1         0.851622         0.260856
    2         0.510143         0.599025
[...]
   21      6.52302e-05         0.925161
```

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

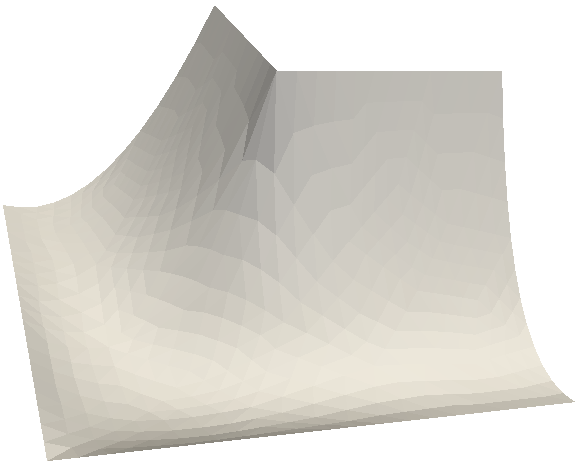

Figure 3: Output of the Poisson example program, visualized as a height field

```
   22      4.68241e-05         0.717829
   23       2.2387e-05         0.478109
=== rate=0.596327, T=0.0213751, TIT=0.000929351, IT=23
```

After program termination there is a file called `getting-started-poisson-fem-result.vtu` in the `build-cmake/src` directory, which can be opened with, e.g., ParaView. It contains the grid and the solution function u_h, and when visualized using a height field, the result should look like Figure 3.

3.2 Assembling the Stiffness Matrix

We now show 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`. This will illustrate how to use the grid and discrete function interfaces, and can be used as a starting point for writing assemblers for other PDEs. Several additional Dune modules provide full frameworks for finite element assemblers. Have a look at the `dune-pdelab`,[14] `dune-fem`,[15] and `dune-fufem`[16] modules.

[14] `www.dune-project.org/modules/dune-pdelab`
[15] `www.dune-project.org/modules/dune-fem`
[16] `www.dune-project.org/modules/dune-fufem`

3.2.1 The Global Assembler

The main assembler loop is contained in the method `assemblePoissonProblem`, which is located above the `main` method in the example file. It has the signature

```
191 template<class Basis>
192 void assemblePoissonProblem(const Basis& basis,
193                             BCRSMatrix<double>& matrix,
194                             BlockVector<double>& b,
195                             const std::function<
196                                 double(FieldVector<double,
197                                                    Basis::GridView::dimension>)
198                                             > volumeTerm)
```

The method implements the standard finite element assembly loop; in particular, it in particular, it assembles the individual element stiffness matrices, and adds them up to obtain the global stiffness matrix. As the first step, it retrieves the grid object from the finite element basis by

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

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

Next, the code initializes the global stiffness matrix. Before a `BCRSMatrix` object can be filled with values, it has to be given its occupation pattern, i.e., the set of all row/column pairs where nonzero matrix entries may appear:

```
208   MatrixIndexSet occupationPattern;
209   getOccupationPattern(basis, occupationPattern);
210   occupationPattern.exportIdx(matrix);
```

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

For all matrix entries that are part of the pattern, the next line then writes an explicit zero into the matrix:

```
215   matrix = 0;
```

Finally, the vector `b` is set to the correct size, and filled with zeros as well:

```
219   // Set b to correct length
220   b.resize(basis.dimension());
221
222   // Set all entries to zero
223   b = 0;
```

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

```
228   auto localView = basis.localView();
229
230   for (const auto& element : elements(gridView))
231   {
```

The variable `localView` implements a restriction of the finite element basis to individual elements. Among other things, it provides the local set of shape functions, and how they relate to global degrees of freedom. The **`for`** loop in Line 230 iterates over the elements of the grid. The free method `elements` from the `dune-grid` module acts like a container of all elements of the grid in `gridView`. At each iteration, the object `element` will be a **`const`** reference to the current grid element.

Within the loop, we first bind the `localView` object to the current element. All subsequent calls to this `localView` will now implicitly refer to that element. Then, we create a small dense matrix and call the element matrix assembler for it:

```
237     localView.bind(element);
238
239     Matrix<double> elementMatrix;
240     assembleElementStiffnessMatrix(localView, elementMatrix);
```

In this implementation, the element assembler sets the correct matrix size for the current element. After the call to `assembleElementStiffnessMatrix`, the variable `elementMatrix` contains the element stiffness matrix for the element referenced by the `element` variable.

Finally, the element matrix is added to the global one:

```
244     for(size_t p=0; p<elementMatrix.N(); p++)
245     {
246       // The global index of the p-th degree of freedom of the element
247       auto row = localView.index(p);
248
249       for (size_t q=0; q<elementMatrix.M(); q++ )
250       {
251         // The global index of the q-th degree of freedom of the element
252         auto col = localView.index(q);
253         matrix[row][col] += elementMatrix[p][q];
254       }
255     }
```

The two **`for`**-loops iterate over all pairs of shape functions. The `localView` object knows the corresponding global degrees of freedom, and provides their numbers via its `index` method.

3.2.2 The Element Assembler

Finally, there is the local problem: given a grid element T, assemble the element stiffness matrix A_T for the Laplace operator and the given finite element basis. Remember that an entry $(A_T)_{pq}$ of the element stiffness matrix for the Poisson problem has the form

$$(A_T)_{pq} = \int_T \langle \nabla \phi_i, \nabla \theta_j \rangle \, dx,$$

where ϕ_i and θ_j are the basis functions from the trial and test spaces corresponding to the p-th and q-th local degree of freedom, respectively. For simplicity, the example

implementation uses the same basis for both spaces, and we therefore only use the symbol ϕ for basis functions.

The matrix entry is computed by transforming the integral over T to an integral over the reference element T_{ref}

$$(A_T)_{pq} = \int_{T_{\text{ref}}} \langle \nabla F^{-T} \nabla \hat{\phi}_p, \nabla F^{-T} \nabla \hat{\phi}_q \rangle |\det \nabla F| \, d\xi, \tag{4}$$

where F is the mapping from T_{ref} to T, and $\hat{\phi}_p$, $\hat{\phi}_q$ are the shape functions on T_{ref} corresponding to the basis functions ϕ_i, ϕ_j on T. We approximate (4) by a quadrature rule with points ξ^k and weights ω_k,

$$(A_T)_{pq} \approx \sum_k \omega_k \Big\langle \nabla F^{-T}(\xi^k) \nabla \hat{\phi}_i(\xi^k), \nabla F^{-T}(\xi^k) \nabla \hat{\phi}_j(\xi^k) \Big\rangle |\det \nabla F(\xi^k)|.$$

This is the formula that the local assembler has to implement.

The corresponding method has the following signature:

```
33 template<class LocalView, class Matrix>
34 void assembleElementStiffnessMatrix(const LocalView& localView,
35                                      Matrix& elementMatrix)
```

The first parameter is the `LocalView` object of the finite element basis. From this view we get information about the current element, in particular its dimension and its shape:

```
39   using Element = typename LocalView::Element;
40   constexpr int dim = Element::dimension;
41   auto element = localView.element();
42   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 $\{\hat{\phi}_p\}_{p=0}^{n_T-1}$ for this element:

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

In DUNE-speak, the object that holds the set of shape functions is called a *local finite element*. The need to invoke the method `tree` is a technicality. It exists to support vector-valued or mixed finite element spaces, and can be ignored for the time being.

We can now ask the `localView` object for the number of shape functions for this element, and initialize the element matrix accordingly:

```
52   elementMatrix.setSize(localView.size(),localView.size());
53   elementMatrix = 0;      // Fill the entire matrix with zeros
```

Then we need a quadrature rule. Such rules are provided by the `dune-geometry` module in the file `dune/geometry/quadraturerules.hh`:

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

Line 58 estimates an appropriate quadrature order for simplex grids, and Line 59 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 is straightforward:

```
65   for (const auto& quadPoint : quadRule)
66   {
```

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:

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

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

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

Note how the gradients of the $\{\hat{\phi}_p\}$ are stored in an array of *matrices* with one row in Line 83. This is because dune-localfunctions regards all shape functions as vector-valued functions, with a vector size of 1 for scalar-valued spaces. In the scalar case, getting the gradient $\nabla\hat{\phi}_p$ as a vector requires the suffix [0] in Line 90, which returns an object of type FieldVector<double,dim>.

Finally we compute the actual matrix entries:

```
95     for (size_t p=0; p<elementMatrix.N(); p++)
96     {
97       auto localRow = localView.tree().localIndex(p);
98       for (size_t q=0; q<elementMatrix.M(); q++)
99       {
100        auto localCol = localView.tree().localIndex(q);
101        elementMatrix[localRow][localCol] += (gradients[p] * gradients[q])
```

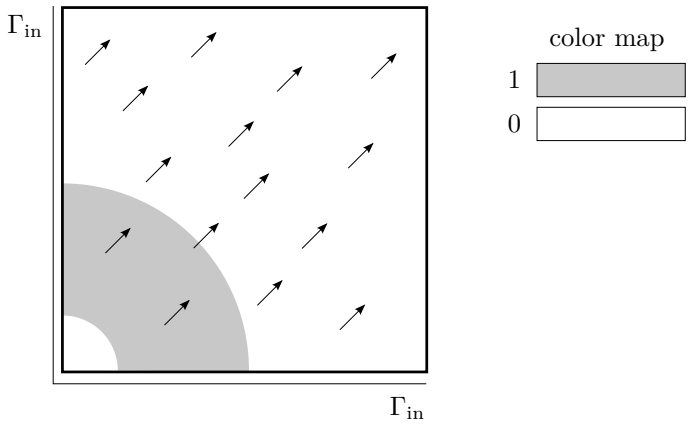

Figure 4: Domain, velocity field (not to scale), and initial condition of the finite volume example

```
102                                   * quadPoint.weight() * integrationElement;
103         }
104       }
```

By operator overloading, `gradients[p]*gradients[q]` implements the scalar product between two vectors. The expressions `localView.tree().localIndex(p)` and `localView.tree().localIndex(q)` compute the element matrix indices from the shape function numbers. In this simple case they simply map `p` to `p` and `q` to `q`, respectively. See the `dune-functions` documentation [8] for more details.

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

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

Compared to the Poisson solver of the previous section, the presented finite volume implementation uses much less features of the DUNE libraries. Instead of using a dedicated linear algebra library, C++ standard library types are used for coefficient vectors. Similarly, 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. The example therefore does not depend on `dune-functions` at all.

4.1 Discrete Linear Transport Equation

As the example problem we will use a linear scalar transport equation. Let $\mathbf{v} : \Omega \times (0, t_{\text{end}}) \to \mathbb{R}^d$ 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

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

For this example, we choose the domain $\Omega = (0, 1)^2$, and the final time $t_{\text{end}} = 0.6$. As velocity field we pick

$$\mathbf{v}(x, t) = (1, 1),$$

which is stationary and divergence-free (Figure 4). By the choice of this field, a part of the boundary becomes the inflow boundary

$$\Gamma_{\text{in}}(t) := \{x \in \partial\Omega \ : \ \langle \mathbf{v}(x, t), \mathbf{n}(x)\rangle < 0\},$$

where $\mathbf{n}$ is the domain unit outer normal. 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 $n = 80\times80$ quadrilateral elements. The time interval $[0, t_{\text{end}}]$ is split into uniform substeps

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

with step size $\Delta t_k := t_{k+1} - t_k = 0.006$. 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 that facet, and $\mathbf{n}_{ij}$ its unit normal pointing from T_i to T_j. The velocity field $\mathbf{v}$ evaluated at the center of γ_{ij} will be called $\mathbf{v}_{ij}$. 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 the value $\bar{c}_i$ of this function on element T_i, $i = 0, \ldots, n-1$ by the mean value of c over that element. We obtain the following equation for the unknown element averages $\bar{c}_i^{k+1}$ at time t_{k+1}:

$$\bar{c}_i^{k+1}|T_i| - \bar{c}_i^{k}|T_i| + \Delta t_k \sum_{\gamma_{ij}} |\gamma_{ij}|\, \phi(\bar{c}_i^k, \bar{c}_j^k, \langle \mathbf{v}_{ij}, \mathbf{n}_{ij}\rangle) = 0 \qquad \forall i = 0, \ldots, n-1. \tag{5}$$

The flux function ϕ is an approximation of the flux $\langle c\mathbf{v}, \mathbf{n}_{ij}\rangle$ across the element boundary γ_{ij}. One common choice is

$$\phi(\bar{c}_i^k, \bar{c}_j^k, \langle \mathbf{v}_{ij}, \mathbf{n}_{ij}\rangle) := \bar{c}_i^k \max(0, \langle \mathbf{v}_{ij}, \mathbf{n}_{ij}\rangle) - \bar{c}_j^k \max(0, -\langle \mathbf{v}_{ij}, \mathbf{n}_{ij}\rangle). \tag{6}$$

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

Inserting the flux function (6) into (5) and rearranging terms, we can solve (5) for the unknown coefficients $\bar{c}_i^{k+1}$ at time t_{k+1}. The resulting formula is a simple vector update

$$\bar{c}^{k+1} = \bar{c}^k + \Delta t_k \delta^k \tag{7}$$

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

$$\delta_i^k := -\sum_{\gamma_{ij}} \frac{|\gamma_{ij}|}{|T_i|} \left(\bar{c}_i^k \max(0, \langle \mathbf{v}_{ij}, \mathbf{n}_{ij}\rangle) + \bar{c}_j^k \max(0, -\langle \mathbf{v}_{ij}, \mathbf{n}_{ij}\rangle)\right). \tag{8}$$

4.2 The `main` Method

The implementation of the finite volume example is again 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 again has

```
13  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 (8). We first discuss the `main` method. As in the finite element case, it begins by setting up the `MPIHelper` variable:

```
112 int main(int argc, char *argv[])
113 {
114   // Set up MPI, if available
115   MPIHelper::instance(argc, argv);
```

The `MPIHelper` instance sets up the MPI message passing system if it is installed. Even though this example does not use MPI, some parts of DUNE call it internally, and not initializing it would lead to run-time errors.

The first real code block sets up the grid:

```
119   constexpr int dim = 2;
120   using Grid = YaspGrid<dim>;
121   Grid grid({1.0,1.0},    // Upper right corner, the lower left one is (0,0)
122             { 80, 80});   // Number of elements per direction
123
124   using GridView = Grid::LeafGridView;
125   GridView gridView = grid.leafGridView();
```

Unlike the previous example, the finite volume implementation uses a structured grid. 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 for the element concentration averages $\bar{c}_i$:

```
130   MultipleCodimMultipleGeomTypeMapper<GridView>
131   mapper(gridView, mcmgElementLayout());
132
133   // Allocate a vector for the concentration
134   std::vector<double> c(mapper.size());
```

The `MultipleCodimMultipleGeomTypeMapper` object constructed in Line 130 is a device that assigns 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. It is provided by the `dune-grid` module.

Line 134 creates the array that is used to store the concentration values $\bar{c}_i$. Observe that an array type from the C++ standard library is used. There is no dependence on the DUNE linear algebra module `dune-istl`, or any other dedicated linear algebra library.

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

```
139   auto c0 = [](const FieldVector<double,dim>& x)
140   {
141     return (x.two_norm()>0.125 && x.two_norm()<0.5) ? 1.0 : 0.0;
142   };
```

Then, the code loops over the elements and samples `c0` at the element centers. These one-point evaluations are used as approximations of the element averages that the algebraic variables $\bar{c}_i^k$ represent:

```
146   // Iterate over grid elements and evaluate c0 at element centers
147   for (const auto& element : elements(gridView))
148   {
149     // Get element geometry
150     auto geometry = element.geometry();
151
152     // Get global coordinate of element center
153     auto global = geometry.center();
154
155     // Sample initial concentration c_0 at the element center
156     c[mapper.index(element)] = c0(global);
157   }
```

Loops over the elements have already appeared in the previous example. Note the special method `center` used in Line 153 to obtain the coordinates of the center of an 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 156 shows how the mapper object is used: Its `index` method returns a nonnegative integer for the given element, which is used 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:

```
162   auto vtkWriter = std::make_shared<Dune::VTKWriter<GridView> >(gridView);
163   VTKSequenceWriter<GridView>
164       vtkSequenceWriter(vtkWriter,
165                         "getting-started-transport-fv-result");   // File name
166
167   // Write the initial values
168   vtkWriter->addCellData(c,"concentration");
169   vtkSequenceWriter.write(0.0);  // 0.0 is the current time
```

The `VTKWriter` constructed in Line 162 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:

```
174   double t=0;                       // Initial time
175   const double tend=0.6;            // Final time
176   const double dt=0.006;            // Time step size
177   int k=0;                          // Time step counter
178
179   // Inflow boundary values
180   auto inflow = [](const FieldVector<double,dim>& x)
181   {
182     return 0.0;
183   };
184
185   // Velocity field
186   auto v = [](const FieldVector<double,dim>& x)
187   {
188     return FieldVector<double,dim> (1.0);
189   };
190
191   while (t<tend)
192   {
193     // Apply finite volume scheme
194     evolve(gridView,mapper,dt,c,v,inflow);
195
196     // Augment time and time step counter
197     t += dt;
198     ++k;
199
```

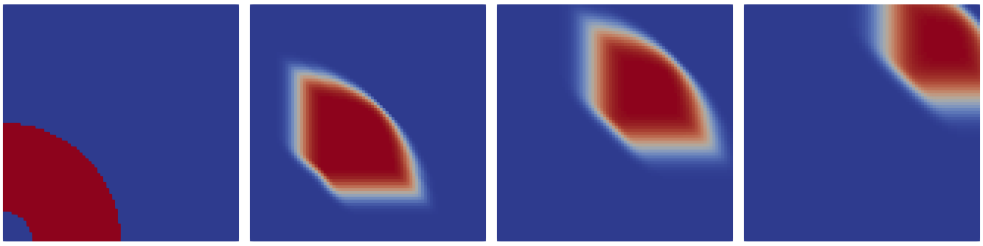

Figure 5: Evolution of the concentration c at times $t = 0$, $t = 0.204$, $t = 0.402$, $t = 0.6$

```
200     // Write data. We do not have to call addCellData again!
201     vtkSequenceWriter.write(t);
202
203     // Print iteration number, time, and time step size
204     std::cout << "k=" << k << " t=" << t << std::endl;
205   }
```

Lines 174–177 initialize several variables, and two further lambda objects `inflow` and `v` for the inflow boundary condition and the velocity field, respectively. The loop starting in Line 191 iterates 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. The main loop then writes the concentration vector to a file, and proceeds to the next time step.

When run, the example program produces 101 output files, and the sequence file `getting-started-transport-fv-result.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 `evolve` Method

The `evolve` method does the main part of the work: After each call to `evolve`, the current iterate has advanced to the next time step.

The method signature is:

```
17  template<class GridView, class Mapper>
18  void evolve(const GridView& gridView,
19              const Mapper& mapper,
20              double dt,       // Time step size
21              std::vector<double>& c,
22              const std::function<FieldVector<double,GridView::dimension>
23                                  (FieldVector<double,GridView::dimension>)> v,
24              const std::function<double
25                                  (FieldVector<double,GridView::dimension>)> inflow)
```

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 `inflow` are the velocity field and the inflow boundary condition functions, respectively.

The method starts by a bit of initialization code:

```
29   // Grid dimension
30   constexpr int dim = GridView::dimension;
31
32   // Allocate a temporary vector for the update
33   std::vector<double> update(c.size());
34   std::fill(update.begin(), update.end(), 0.0);
```

The array set up in Line 33 is the correction δ^k defined in (8).

The code then loops over all grid elements T_i:

```
39   for (const auto& element : elements(gridView))
40   {
41     // Element geometry
42     auto geometry = element.geometry();
43
44     // Element volume
45     double elementVolume = geometry.volume();
46
47     // Unique element number
48     typename Mapper::Index i = mapper.index(element);
```

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^k defined in (8). At the top of the loop, the element volume $|T_i|$ and its index i are precomputed.

The formula (8) for the correction δ_i^k for element T_i consists of a sum over all elements T_j whose boundaries intersect with the boundary of T_i in a $d-1$-dimensional set. Such neighborhood relations are represented in the DUNE grid interface by objects of type `Intersection`. These provide 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 (8) is therefore coded as a loop over all intersections of the current element:

```
53     for (const auto& intersection : intersections(gridView,element))
54     {
55       // Geometry of the intersection
56       auto intersectionGeometry = intersection.geometry();
57
58       // Center of intersection in global coordinates
59       FieldVector<double,dim>
60           intersectionCenter = intersectionGeometry.center();
61
62       // Velocity at intersection center v_ij
63       FieldVector<double,dim> velocity = v(intersectionCenter);
64
65       // Center of the intersection in local coordinates
```

```
66        const auto& intersectionReferenceElement
67            = ReferenceElements<double,dim-1>::general(intersection.type());
68        FieldVector<double,dim-1> intersectionLocalCenter
69            = intersectionReferenceElement.position(0,0);
70
71        // Normal vector scaled with intersection area: n_ij |γ_ij|
72        FieldVector<double,dim> integrationOuterNormal
73            = intersection.integrationOuterNormal(intersectionLocalCenter);
74
75        // Compute factor occuring in flux formula: ⟨v_ij, n_ij⟩|γ_ij|
76        double intersectionFlow = velocity*integrationOuterNormal;
```

The loop itself uses the convenient range-based **for** syntax already seen when looping over the grid elements. It then computes the velocity $\mathbf{v}_{ij}$ and the product $\mathbf{n}_{ij}|\gamma_{ij}|$. To compute $\mathbf{v}_{ij}$, the value of the velocity field $\mathbf{v}$ at the center of the common intersection between T_i and its current neighbor T_j, Line 56 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 60. Line 63 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 knows its center. That reference element is acquired in Line 67, and its center is evaluated in Line 69.

Rather than computing $|\gamma_{ij}|$ and $\mathbf{n}_{ij}$ separately, the code then calls a dedicated DUNE grid interface method called `integrationOuterNormal` that directly yields the product of the two. This product is used 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^k for the element T_i:

```
80        // Outflow contributions
81        update[i] -= c[i]*std::max(0.0,intersectionFlow)/elementVolume;
82
83        // Inflow contributions
84        if (intersectionFlow<=0)
85        {
86          // Handle interior intersection
87          if (intersection.neighbor())
88          {
89            // Access neighbor
90            auto j = mapper.index(intersection.outside());
91            update[i] -= c[j]*intersectionFlow/elementVolume;
92          }
93
94          // Handle boundary intersection
95          if (intersection.boundary())
```

```
96            update[i] -= inflow(intersectionCenter)
97                                 * intersectionFlow/elementVolume;
98        }
```

Line 81 adds $-\bar{c}_i \frac{|\gamma_{ij}|}{|T_i|} \max(0, \langle \mathbf{v}_{ij}, \mathbf{n}_{ij} \rangle)$, which covers the case that the current intersection is an outflow boundary of the element T_i. If $\langle \mathbf{v}_{ij}, \mathbf{n}_{ij} \rangle < 0$, i.e., if there is flow into element T_i, we need to distinguish between 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 on the inflow boundary Γ_{in}). Lines 84–98 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.

This ends the loop over the intersections, and the loop over the elements ends as well:

```
100     }  // End loop over all intersections
101   }  // End loop over the grid elements
```

Finally, the concentration vector is updated:

```
105   // Update the concentration vector
106   for (std::size_t i=0; i<c.size(); ++i)
107     c[i] += dt*update[i];
108 }
```

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/geometry/quadraturerules.hh>
6
7   // {  include_uggrid_begin  }
8   #include <dune/grid/uggrid.hh>
9   #include <dune/grid/io/file/gmshreader.hh>
10  // {  include_uggrid_end  }
11  #include <dune/grid/io/file/vtk/vtkwriter.hh>
12
13  #include <dune/istl/matrix.hh>
14  // {  include_matrix_vector_begin  }
15  #include <dune/istl/bcrsmatrix.hh>
16  #include <dune/istl/bvector.hh>
17  // {  include_matrix_vector_end  }
18  #include <dune/istl/matrixindexset.hh>
19  #include <dune/istl/preconditioners.hh>
20  #include <dune/istl/solvers.hh>
21  #include <dune/istl/matrixmarket.hh>
22
23  #include <dune/functions/functionspacebases/lagrangebasis.hh>
24  #include <dune/functions/functionspacebases/interpolate.hh>
25
26
27  // { using_namespace_dune_begin }
28  using namespace Dune;
29  // { using_namespace_dune_end }
30
31  // Compute the  stiffness  matrix  for  a  single  element
32  // {  local_assembler_signature_begin  }
33  template<class LocalView, class Matrix>
34  void assembleElementStiffnessMatrix(const LocalView& localView,
```

```
35                                          Matrix& elementMatrix)
36  // {  local_assembler_signature_end  }
37  {
38  // {  local_assembler_get_geometry_begin  }
39    using Element = typename LocalView::Element;
40    constexpr int dim = Element::dimension;
41    auto element = localView.element();
42    auto geometry = element.geometry();
43  // {  local_assembler_get_geometry_end  }
44
45    // Get set of shape functions for this element
46  // {  get_shapefunctions_begin  }
47    const auto& localFiniteElement = localView.tree().finiteElement();
48  // {  get_shapefunctions_end  }
49
50    // Set all matrix entries to zero
51  // {  init_element_matrix_begin  }
52    elementMatrix.setSize(localView.size(), localView.size());
53    elementMatrix = 0;      // Fill the entire matrix with zeros
54  // {  init_element_matrix_end  }
55
56    // Get a quadrature rule
57  // {  get_quadrature_rule_begin  }
58    int order = 2 * (localFiniteElement.localBasis().order()-1);
59    const auto& quadRule = QuadratureRules<double, dim>::rule(element.type(),
60                                                              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    {
67  // {  loop_over_quad_points_end  }
68
69  // {  get_quad_point_info_begin  }
70      // Position of the current quadrature point in the reference element
71      const auto quadPos = quadPoint.position();
72
73      // The transposed inverse Jacobian of the map from the reference element
74      // to the grid element
75      const auto jacobian = geometry.jacobianInverseTransposed(quadPos);
76
77      // The determinant term in the integral transformation formula
78      const auto integrationElement = geometry.integrationElement(quadPos);
79  // {  get_quad_point_info_end  }
80
81  // {  compute_gradients_begin  }
82      // The gradients of the shape functions on the reference element
83      std::vector<FieldMatrix<double,1,dim> > referenceGradients;
84      localFiniteElement.localBasis().evaluateJacobian(quadPos,
85                                                       referenceGradients);
86
87      // Compute the shape function gradients on the grid element
88      std::vector<FieldVector<double,dim> > gradients(referenceGradients.size());
89      for (size_t i=0; i<gradients.size(); i++)
90        jacobian.mv(referenceGradients[i][0], gradients[i]);
91  // {  compute_gradients_end  }
92
93      // Compute the actual matrix entries
94  // {  compute_matrix_entries_begin  }
95      for (size_t p=0; p<elementMatrix.N(); p++)
96      {
97        auto localRow = localView.tree().localIndex(p);
98        for (size_t q=0; q<elementMatrix.M(); q++)
99        {
100         auto localCol = localView.tree().localIndex(q);
101         elementMatrix[localRow][localCol] += (gradients[p] * gradients[q])
102                                           * quadPoint.weight() * integrationElement;
103       }
104     }
105 // {  compute_matrix_entries_end  }
106   }
107 }
108
109
110 // Compute the source term for a single element
111 template<class LocalView>
112 void assembleElementVolumeTerm(
113         const LocalView& localView,
114         BlockVector<double>& localB,
115         const std::function<double(FieldVector<double,
116                                    LocalView::Element::dimension>)> volumeTerm)
117 {
118   using Element = typename LocalView::Element;
119   auto element = localView.element();
120   constexpr int dim = Element::dimension;
121
122   // Set of shape functions for a single element
123   const auto& localFiniteElement = localView.tree().finiteElement();
```

```
124
125     // Set all entries to zero
126     localB.resize(localFiniteElement.size());
127     localB = 0;
128
129     // A quadrature rule
130     int order = dim;
131     const auto& quadRule = QuadratureRules<double, dim>::rule(element.type(), order);
132
133     // Loop over all quadrature points
134     for (const auto& quadPoint : quadRule)
135     {
136       // Position of the current quadrature point in the reference element
137       const FieldVector<double,dim>& quadPos = quadPoint.position();
138
139       // The multiplicative factor in the integral transformation formula
140       const double integrationElement = element.geometry().integrationElement(quadPos);
141
142       double functionValue = volumeTerm(element.geometry().global(quadPos));
143
144       // Evaluate all shape function values at this point
145       std::vector<FieldVector<double,1> > shapeFunctionValues;
146       localFiniteElement.localBasis().evaluateFunction(quadPos, shapeFunctionValues);
147
148       // Actually compute the vector entries
149       for (size_t p=0; p<localB.size(); p++)
150       {
151         auto localIndex = localView.tree().localIndex(p);
152         localB[localIndex] += shapeFunctionValues[p] * functionValue
153                               * quadPoint.weight() * integrationElement;
154       }
155     }
156   }
157
158   // Get the occupation pattern of the stiffness matrix
159   template<class Basis>
160   void getOccupationPattern(const Basis& basis, MatrixIndexSet& nb)
161   {
162     nb.resize(basis.size(), basis.size());
163
164     auto gridView = basis.gridView();
165
166     // A loop over all elements of the grid
167     auto localView = basis.localView();
168
169     for (const auto& element : elements(gridView))
170     {
171       localView.bind(element);
172
173       for (size_t i=0; i<localView.size(); i++)
174       {
175         // The global index of the i-th vertex of the element
176         auto row = localView.index(i);
177
178         for (size_t j=0; j<localView.size(); j++ )
179         {
180           // The global index of the j-th vertex of the element
181           auto col = localView.index(j);
182           nb.add(row,col);
183         }
184       }
185     }
186   }
187
188
189   /** \brief Assemble the Laplace stiffness matrix on the given grid view */
190   // {  global_assembler_signature_begin  }
191   template<class Basis>
192   void assemblePoissonProblem(const Basis& basis,
193                               BCRSMatrix<double>& matrix,
194                               BlockVector<double>& b,
195                               const std::function<
196                                   double(FieldVector<double,
197                                                      Basis::GridView::dimension>)
198                                                  > volumeTerm)
199   // {  global_assembler_signature_end  }
200   {
201   // {  assembler_get_grid_info_begin  }
202     auto gridView = basis.gridView();
203   // {  assembler_get_grid_info_end  }
204
205     // MatrixIndexSets store the occupation pattern of a sparse matrix.
206     // They are not particularly efficient, but simple to use.
207   // {  assembler_matrix_pattern_begin  }
208     MatrixIndexSet occupationPattern;
209     getOccupationPattern(basis, occupationPattern);
210     occupationPattern.exportIdx(matrix);
211   // {  assembler_matrix_pattern_end  }
212
```

```
213     // Set all entries to zero
214   // { assembler_zero_matrix_begin }
215     matrix = 0;
216   // { assembler_zero_matrix_end }
217
218   // { assembler_zero_vector_begin }
219     // Set b to correct length
220     b.resize(basis.dimension());
221
222     // Set all entries to zero
223     b = 0;
224   // { assembler_zero_vector_end }
225
226     // A loop over all elements of the grid
227   // { assembler_element_loop_begin }
228     auto localView = basis.localView();
229
230     for (const auto& element : elements(gridView))
231     {
232   // { assembler_element_loop_end }
233
234       // Now let's get the element stiffness matrix
235       // A dense matrix is used for the element stiffness matrix
236   // { assembler_assemble_element_matrix_begin }
237       localView.bind(element);
238
239       Matrix<double> elementMatrix;
240       assembleElementStiffnessMatrix(localView, elementMatrix);
241   // { assembler_assemble_element_matrix_end }
242
243   // { assembler_add_element_matrix_begin }
244       for(size_t p=0; p<elementMatrix.N(); p++)
245       {
246         // The global index of the p-th degree of freedom of the element
247         auto row = localView.index(p);
248
249         for (size_t q=0; q<elementMatrix.M(); q++ )
250         {
251           // The global index of the q-th degree of freedom of the element
252           auto col = localView.index(q);
253           matrix[row][col] += elementMatrix[p][q];
254         }
255       }
256   // { assembler_add_element_matrix_end }
257
258       // Now get the local contribution to the right-hand side vector
259       BlockVector<double> localB;
260       assembleElementVolumeTerm(localView, localB, volumeTerm);
261
262       for (size_t p=0; p<localB.size(); p++)
263       {
264         // The global index of the p-th vertex of the element
265         auto row = localView.index(p);
266         b[row] += localB[p];
267       }
268     }
269   }
270
271
272   int main(int argc, char *argv[])
273   {
274   // { mpi_setup_begin }
275     // Set up MPI, if available
276     MPIHelper::instance(argc, argv);
277   // { mpi_setup_end }
278
279     ////////////////////////////////
280     //   Generate the grid
281     ////////////////////////////////
282
283   // { create_grid_begin }
284     constexpr int dim = 2;
285     using Grid = UGGrid<dim>;
286     std::shared_ptr<Grid> grid = GmshReader<Grid>::read("l-shape.msh");
287
288     grid->globalRefine(2);
289
290     using GridView = Grid::LeafGridView;
291     GridView gridView = grid->leafGridView();
292   // { create_grid_end }
293
294     ////////////////////////////////////////////////////////
295     //   Stiffness matrix and right hand side vector
296     ////////////////////////////////////////////////////////
297
298   // { create_matrix_vector_begin }
299     using Matrix = BCRSMatrix<double>;
300     using Vector = BlockVector<double>;
301
```

```
302     Matrix stiffnessMatrix;
303     Vector b;
304   // {  create_matrix_vector_end  }
305
306     //////////////////////////////////////////////////////
307     //   Assemble the system
308     //////////////////////////////////////////////////////
309
310   // {  setup_basis_begin  }
311     Functions::LagrangeBasis<GridView,1> basis(gridView);
312
313     auto sourceTerm = [](const FieldVector<double,dim>& x){return -5.0;};
314   // {  setup_basis_end  }
315   // {  call_assembler_begin  }
316     assemblePoissonProblem(basis, stiffnessMatrix, b, sourceTerm);
317   // {  call_assembler_end  }
318
319     // Determine Dirichlet dofs by marking all degrees of freedom whose Lagrange nodes
320     // comply with a given predicate .
321   // {  dirichlet_marking_begin  }
322     auto predicate = [](auto x)
323     {
324       return x[0] < 1e-8
325           || x[1] < 1e-8
326           || (x[0] > 0.4999 && x[1] > 0.4999);
327     };
328
329     // Evaluating the predicate will mark all Dirichlet degrees of freedom
330     std::vector<bool> dirichletNodes;
331     Functions::interpolate(basis, dirichletNodes, predicate);
332   // {  dirichlet_marking_end  }
333
334     //////////////////////////////////////////
335     //   Modify Dirichlet rows
336     //////////////////////////////////////////
337   // {  dirichlet_matrix_modification_begin  }
338     // Loop over the matrix rows
339     for (size_t i=0; i<stiffnessMatrix.N(); i++)
340     {
341       if (dirichletNodes[i])
342       {
343         auto cIt    = stiffnessMatrix[i].begin();
344         auto cEndIt = stiffnessMatrix[i].end();
345         // Loop over nonzero matrix entries in current row
346         for (; cIt!=cEndIt; ++cIt)
347           *cIt = (cIt.index()==i) ? 1.0 : 0.0;
348       }
349     }
350   // {  dirichlet_matrix_modification_end  }
351
352     // Set Dirichlet values
353   // {  dirichlet_rhs_modification_begin  }
354     auto dirichletValues = [](auto x)
355     {
356       return (x[0]< 1e-8 || x[1] < 1e-8) ? 0 : 0.5;
357     };
358     Functions::interpolate(basis,b,dirichletValues, dirichletNodes);
359   // {  dirichlet_rhs_modification_end  }
360
361     //////////////////////////////////////////////////////////////////////////
362     // Write matrix and load vector to files , to be used in later examples
363     //////////////////////////////////////////////////////////////////////////
364   // {  matrix_rhs_writing_begin  }
365     std::string baseName = "getting-started-poisson-fem-"
366                            + std::to_string(grid->maxLevel()) + "-refinements";
367     storeMatrixMarket(stiffnessMatrix, baseName + "-matrix.mtx");
368     storeMatrixMarket(b, baseName + "-rhs.mtx");
369   // {  matrix_rhs_writing_end  }
370
371     ////////////////////////////
372     //   Compute solution
373     ////////////////////////////
374
375   // {  algebraic_solving_begin  }
376     // Choose an initial iterate that fulfills the Dirichlet conditions
377     Vector x(basis.size());
378     x = b;
379
380     // Turn the matrix into a linear operator
381     MatrixAdapter<Matrix,Vector,Vector> linearOperator(stiffnessMatrix);
382
383     // Sequential incomplete LU decomposition as the preconditioner
384     SeqILU<Matrix,Vector,Vector> preconditioner(stiffnessMatrix,
385                                                 1.0);  // Relaxation factor
386
387     // Preconditioned conjugate gradient solver
388     CGSolver<Vector> cg(linearOperator,
389                         preconditioner,
390                         1e-5, // Desired residual reduction factor
```

```
391                         50,    // Maximum number of iterations
392                         2);    // Verbosity of the solver
393
394     // Object storing some statistics about the solving process
395     InverseOperatorResult statistics;
396
397     // Solve!
398     cg.apply(x, b, statistics);
399   // { algebraic_solving_end }
400
401     // Output result
402   // { vtk_output_begin }
403     VTKWriter<GridView> vtkWriter(gridView);
404     vtkWriter.addVertexData(x, "solution");
405     vtkWriter.write("getting-started-poisson-fem-result");
406   // { vtk_output_end }
407   }
```

6 Complete source code of the finite volume example

```
1   #include "config.h"
2
3   #include <iostream>
4   #include <vector>
5
6   #include <dune/common/parallel/mpihelper.hh>
7
8   #include <dune/grid/common/mcmgmapper.hh>
9   #include <dune/grid/yaspgrid.hh>
10  #include <dune/grid/io/file/vtk.hh>
11
12  // { using_namespace_dune_begin }
13  using namespace Dune;
14  // { using_namespace_dune_end }
15
16  // { evolve_signature_begin }
17  template<class GridView, class Mapper>
18  void evolve(const GridView& gridView,
19              const Mapper& mapper,
20              double dt,       // Time step size
21              std::vector<double>& c,
22              const std::function<FieldVector<double,GridView::dimension>
23                                  (FieldVector<double,GridView::dimension>)> v,
24              const std::function<double
25                                  (FieldVector<double,GridView::dimension>)> inflow)
26  // { evolve_signature_end }
27  {
28  // { evolve_init_begin }
29    // Grid dimension
30    constexpr int dim = GridView::dimension;
31
32    // Allocate a temporary vector for the update
33    std::vector<double> update(c.size());
34    std::fill(update.begin(), update.end(), 0.0);
35  // { evolve_init_end }
36
37    // Compute update vector
38  // { element_loop_begin }
39    for (const auto& element : elements(gridView))
40    {
41      // Element geometry
42      auto geometry = element.geometry();
43
44      // Element volume
45      double elementVolume = geometry.volume();
46
47      // Unique element number
48      typename Mapper::Index i = mapper.index(element);
49  // { element_loop_end }
50
51      // Loop over all intersections γ_ij with neighbors and boundary
52  // { intersection_loop_begin }
53      for (const auto& intersection : intersections(gridView,element))
54      {
55        // Geometry of the intersection
56        auto intersectionGeometry = intersection.geometry();
57
58        // Center of intersection in global coordinates
59        FieldVector<double,dim>
60            intersectionCenter = intersectionGeometry.center();
61
62        // Velocity at intersection center v_ij
63        FieldVector<double,dim> velocity = v(intersectionCenter);
64
65        // Center of the intersection in local coordinates
```

```
66          const auto& intersectionReferenceElement
67              = ReferenceElements<double,dim-1>::general(intersection.type());
68          FieldVector<double,dim-1> intersectionLocalCenter
69              = intersectionReferenceElement.position(0,0);
70
71          // Normal vector scaled with intersection area: n_ij|γ_ij|
72          FieldVector<double,dim> integrationOuterNormal
73              = intersection.integrationOuterNormal(intersectionLocalCenter);
74
75          // Compute factor occuring in flux formula: ⟨v_ij, n_ij⟩|γ_ij|
76          double intersectionFlow = velocity*integrationOuterNormal;
77      // {  intersection_loop_initend  }
78
79      // {  intersection_loop_mainbegin  }
80          // Outflow contributions
81          update[i] -= c[i]*std::max(0.0,intersectionFlow)/elementVolume;
82
83          // Inflow contributions
84          if (intersectionFlow<=0)
85          {
86            // Handle interior intersection
87            if (intersection.neighbor())
88            {
89              // Access neighbor
90              auto j = mapper.index(intersection.outside());
91              update[i] -= c[j]*intersectionFlow/elementVolume;
92            }
93
94            // Handle boundary intersection
95            if (intersection.boundary())
96              update[i] -= inflow(intersectionCenter)
97                                  * intersectionFlow/elementVolume;
98          }
99      // {  intersection_loopend  }
100       } // End loop over all intersections
101     } // End loop over the grid elements
102     // { element_loop_end }
103
104     // {  evolve_laststeps  }
105       // Update the concentration vector
106       for (std::size_t i=0; i<c.size(); ++i)
107         c[i] += dt*update[i];
108     }
109     // { evolve_end }
110
111     // { main_begin }
112     int main(int argc, char *argv[])
113     {
114       // Set up MPI, if available
115       MPIHelper::instance(argc, argv);
116     // { main_signature_end }
117
118     // {  create_grid_begin  }
119       constexpr int dim = 2;
120       using Grid = YaspGrid<dim>;
121       Grid grid({1.0,1.0},     // Upper right corner, the lower left one is (0,0)
122                 { 80, 80});    // Number of elements per direction
123
124       using GridView = Grid::LeafGridView;
125       GridView gridView = grid.leafGridView();
126     // {  create_grid_end  }
127
128       // Assigns a unique number to each element
129     // {  create_concentration_begin  }
130       MultipleCodimMultipleGeomTypeMapper<GridView>
131       mapper(gridView, mcmgElementLayout());
132
133       // Allocate a vector for the concentration
134       std::vector<double> c(mapper.size());
135     // {  create_concentration_end  }
136
137       // Initial concentration
138     // {  lambda_initial_concentration_begin  }
139       auto c0 = [](const FieldVector<double,dim>& x)
140       {
141         return (x.two_norm()>0.125 && x.two_norm()<0.5) ? 1.0 : 0.0;
142       };
143     // {  lambda_initial_concentration_end  }
144
145     // {  sample_initial_concentration_begin  }
146       // Iterate over grid elements and evaluate c0 at element centers
147       for (const auto& element : elements(gridView))
148       {
149         // Get element geometry
150         auto geometry = element.geometry();
151
152         // Get global coordinate of element center
153         auto global = geometry.center();
```

```
154
155       // Sample initial concentration c_0 at the element center
156       c[mapper.index(element)] = c0(global);
157     }
158   // {  sample_initial_concentration_end  }
159
160     // Construct VTK writer
161   // {  construct_vtk_writer_begin  }
162     auto vtkWriter = std::make_shared<Dune::VTKWriter<GridView> >(gridView);
163     VTKSequenceWriter<GridView>
164         vtkSequenceWriter(vtkWriter,
165                           "getting-started-transport-fv-result");  // File name
166
167     // Write the initial values
168     vtkWriter->addCellData(c,"concentration");
169     vtkSequenceWriter.write(0.0);  // 0.0 is the current time
170   // {  construct_vtk_writer_end  }
171
172     // Now do the time steps
173   // {  time_loop_begin  }
174     double t=0;                          // Initial time
175     const double tend=0.6;               // Final time
176     const double dt=0.006;               // Time step size
177     int k=0;                             // Time step counter
178
179     // Inflow boundary values
180     auto inflow = [](const FieldVector<double,dim>& x)
181     {
182       return 0.0;
183     };
184
185     // Velocity field
186     auto v = [](const FieldVector<double,dim>& x)
187     {
188       return FieldVector<double,dim> (1.0);
189     };
190
191     while (t<tend)
192     {
193       // Apply finite volume scheme
194       evolve(gridView,mapper,dt,c,v,inflow);
195
196       // Augment time and time step counter
197       t += dt;
198       ++k;
199
200       // Write data. We do not have to call addCellData again!
201       vtkSequenceWriter.write(t);
202
203       // Print iteration number, time, and time step size
204       std::cout << "k=" << k << " t=" << t << std::endl;
205     }
206   // { time_loop_end }
207   }
208   // { main_end }
```

References

[1] P. Bastian and M. Blatt. "On the Generic Parallelisation of Iterative Solvers for the Finite Element Method". In: *Int. J. Computational Science and Engineering* 4.1 (2008), pp. 56–69. DOI: 10.1504/IJCSE.2008.021112.

[2] 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". In: *Computing* 82.2-3 (2008), pp. 121–138.

[3] 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". In: *Computing* 82.2-3 (2008), pp. 103–119.

[4] P. Bastian, M. Blatt, A. Dedner, N.-A. Dreier, C. Engwer, R. Fritze, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, M. Ohlberger, and O. Sander. "The DUNE

Framework: Basic Concepts and Recent Developments". In: *Computers and Mathematics with Applications* (2020). DOI: 10.1016/j.camwa.2020.06.007.

[5] M. Blatt and P. Bastian. "The Iterative Solver Template Library". In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Ed. by B. Kagström, E. Elmroth, J. Dongarra, and J. Wasniewski. Lecture Notes in Scientific Computing 4699. 2007, pp. 666–675. URL: 10.1007/978-3-540-75755-9%5C_82.

[6] M. Blatt. "A Parallel Algebraic Multigrid Method for Elliptic Problems with Highly Discontinuous Coefficients". PhD thesis. Universität Heidelberg, 2010.

[7] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. "The Distributed and Unified Numerics Environment, Version 2.4". In: *Archive of Numerical Software* 4.100 (2016), pp. 13–29. DOI: 10.11588/ans.2016.100.26526.

[8] C. Engwer, C. Gräser, S. Müthing, and O. Sander. "Function space bases in the dune-functions module". In: *ArXiv e-prints* (2018). eprint: 1806.09545 (cs.MS).

[9] C. Engwer, C. Gräser, S. Müthing, and O. Sander. "The interface for functions in the dune-functions module". In: *Archive of Numerical Software* 5.1 (2017), pp. 95–105. DOI: 10.11588/ans.2017.1.27683.

[10] C. Geuzaine and F. Remacle. *Gmsh Reference Manual*. 2015. URL: http://www.geuz.org/gmsh/doc/texinfo/gmsh.pdf.

[11] Kitware. *VTK File Formats (for VTK Version 4.2)*. 2003. URL: www.vtk.org/img/file-formats.pdf.

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

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

[14] O. Sander. *DUNE — The Distributed and Unified Numerics Environment*. Springer, 2020.