

# Introduction to Dune-Fem



The DUNE-FEM developers\*

September 24, 2010

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

<http://dune.mathematik.uni-freiburg.de/>

[dune@mathematik.uni-freiburg.de](mailto:dune@mathematik.uni-freiburg.de)

# Contents

<b>1</b>	<b>What is Dune-Fem?</b>	<b>4</b>
1.1	Available example implementations using DUNE-FEM . . . . .	4
1.2	The DUNE Core Modules . . . . .	5
1.3	Available Grid Implementations . . . . .	6
1.4	Download and installation . . . . .	7
1.5	Create your own project . . . . .	9
<b>2</b>	<b>The Transport-Example with Dune-Fem</b>	<b>10</b>
2.1	A Tiny Introduction to Finite Volume Schemes . . . . .	10
2.2	Implementation overview . . . . .	12
2.2.1	The <code>main</code> function . . . . .	12
2.2.2	The basic algorithm . . . . .	20
2.2.3	The methods <code>initialize</code> and <code>compute</code> . . . . .	24
2.3	Parallelization . . . . .	27
<b>3</b>	<b>Solving the Poisson problem</b>	<b>28</b>
3.1	Implementation . . . . .	29
3.1.1	Algorithm . . . . .	30
3.1.2	Assembling the Laplace operator . . . . .	35
3.1.3	Boundary treatment . . . . .	40
3.1.4	Assembling the right hand side . . . . .	40
3.2	Adaptation . . . . .	41
3.3	Parallelization . . . . .	41
<b>4</b>	<b>An LDG solver for Advection-Diffusion Equations</b>	<b>44</b>
4.1	Advection-Diffusion Equation . . . . .	44
4.2	Implementation overview . . . . .	46
4.3	Main Loop . . . . .	47
4.4	Stepper control class . . . . .	52
4.5	Setting up an LDGPass . . . . .	55
4.6	Implementing your own Pass Operator . . . . .	60

*Contents*

4.7	Visualisation and EOC Output . . . . .	60
<b>5</b>	<b>Solving the Stokes problem</b>	<b>62</b>
5.1	Implementation . . . . .	64
5.1.1	Algorithm . . . . .	65
5.1.2	Assembling the Laplace operator . . . . .	70
5.1.3	Assambling the discrete Divergence operator . . . . .	71
5.1.4	Boundary treatment . . . . .	76
5.1.5	Assembling the right hand side . . . . .	76
5.2	Parallelization . . . . .	77
<b>6</b>	<b>Documentation and reference guide for Dune-Fem</b>	<b>78</b>

# 1 What is Dune-Fem?

DUNE-FEM is a DUNE module providing a framework for implementing discretizations based on grids described by the DUNE-GRID interface. DUNE-FEM is also free software, but in contrast to DUNE, it is licensed under the standard GPL licence (see <http://dune.mathematik.uni-freiburg.de>).

DUNE, the Distributed and Unified Numerics Environment is a modular toolbox for solving partial differential equations with grid-based methods. DUNE is free software licensed under the GPL with a so called “runtime exception” (<http://www.dune-project.org/license.html>). The main intention is to create slim interfaces allowing an efficient use of legacy and/or new libraries. Using C++ techniques DUNE allows to use very different implementations of the same concept (e.g., grids, solvers) using a common interface with a very low overhead. For more information we refer to <http://www.dune-project.org/dune.html>.

## 1.1 Available example implementations using Dune-Fem

This section describes briefly which examples are available in this DUNE-FEM-HOWTO. These examples demonstrate the most commonly used features from DUNE-FEM. The DUNE-FEM-HOWTO contains the following examples, ordered by increasing difficulty:

**Getting started** (`dune-fem-howto/examples/gettingstarted`) This example shows how to calculate a Lagrange interpolation for a given analytical function.

**A Finite Volume scheme** (`dune-fem-howto/examples/finitevolume`) This example demonstrates the implementation of a first order Finite Volume scheme using DUNE-FEM. A detailed code description is given in Chapter 2.

**The Poisson problem** (`dune-fem-howto/examples/poisson`) This is an example for calculating a solution of the Poisson problem  $-\Delta u = f$  with Dirichlet boundary. The code can be executed in a parallel environment. Details can be found in Chapter 3.

## 1 What is DUNE-FEM?

**$L^2$ -projection** (`dune-fem-howto/examples/l2projection`) This is an example for calculating the  $L^2$ -projection of an analytical function to a discrete function space. The code uses basically the same features as for the Poisson example from Chapter 3. Therefore, no detailed description is available.

**LDG for Advection-Diffusion equations** (`dune-fem-howto/examples/localdg`) This is an example for implementing a Local DG solver for advection-diffusion problems. All features are described in Chapter 4. This example code can be used for parallel computations.

**The Stokes problem** (`dune-fem-howto/examples/stokes`) This example shows how to implement a Stokes solver in the DUNE-FEM context. The description is presented in Chapter 5.

**Data I/O and check pointing** (`dune-fem-howto/examples/dataio`) This example shows how to incorporate data I/O and check pointing into your simulation code. At the moment no detailed description, other than the code example, is available.

## 1.2 The Dune Core Modules

The framework consists of a number of modules which are downloadable as separate packages. The current core modules are:

**Dune-Common** contains the basic classes used by all DUNE modules. It provides some infrastructural classes for debugging and exception handling as well as a dense matrix/vector template library.

**Dune-Grid** is the most mature module. It defines nonconforming, hierarchically nested, multi-element-type, parallel grids in arbitrary space dimensions. Graphical output through several packages is available, e.g., file output to VTK (parallel XML format for unstructured grids). The graphics package GraPE has been integrated in interactive mode.

**Dune-Istl** (Iterative Solver Template Library) provides generic sparse matrix/vector classes and a variety of solvers based on them. A special feature is the use of templates to exploit the recursive block structure of finite element matrices at compile time. Available solvers include Krylov methods, (block-) incomplete decompositions and aggregation-based algebraic multigrid.

**Dune-LocalFunctions** is the most recent DUNE core module. It provides a common interface for local basis functions, local interpolations, and the local DoF administration.

### 1.3 Available Grid Implementations

So far seven grid implementations are included in the DUNE-GRID module, each geared towards a different purpose:

**AlbertaGrid** The grid manager of the ALBERTA toolbox providing 1d/2d/3d simplicial grids with recursive, conforming bisection refinement

**ALUGrid** Based on the ALUGrid library, the following DUNE grid implementations are available:

**ALUCubeGrid** A parallel (only 3d), locally adaptive (non-conforming) grid providing hexahedral or quadrilateral elements

**ALUSimplexGrid** A parallel (only 3d), locally adaptive (non-conforming) grid providing tetrahedral or triangular elements

**ALUConformGrid** A locally adaptive (conforming) grid providing triangular elements using an iterative bisection algorithm for refinement

**GeometryGrid** A metagrid (i.e., a grid based on another DUNE grid, called the host grid) replacing the host grid's geometry and adding entities of all codimensions

**OneDGrid** A sequential locally adaptive grid in one space dimension

**SGrid** A structured grid in  $n$  space dimensions

**UGGrid** The grid manager of the UG<sup>1</sup>

**YaspGrid** A structured parallel grid in  $n$  space dimensions (used as default grid)

More information on these grids can be found on the DUNE documentation website<sup>2</sup>.

---

<sup>1</sup>UG is not freely available

<sup>2</sup><http://www.dune-project.org/doc/devel/features.html>

## 1.4 Download and installation

This section describes how members of the Section of Applied Mathematics in Freiburg can create a local working installation of DUNE.

These are only the basic steps, maybe you have to tune up some options. For non-Freiburg users, the steps are similiar. For more information, have a look at the installation notes website<sup>3</sup>.

- Create a directory for your DUNE installation and change to it. For example:

```
mkdir dune
cd dune
```

- Download the latest (stable) core modules from the official DUNE homepage <http://www.dune-project.org/download.html> and unpack them to your DUNE directory. For example:

```
tar -xzf dune-common-2.0.tar.gz
tar -xzf dune-grid-2.0.tar.gz
tar -xzf dune-istl-2.0.tar.gz
```

Alternatively, you can obtain a bleeding edge version from the DUNE subversion repository:

```
svn checkout https://svn.dune-project.org/svn/dune-common/trunk dune-common
svn checkout https://svn.dune-project.org/svn/dune-grid/trunk dune-grid
svn checkout https://svn.dune-project.org/svn/dune-istl/trunk dune-istl
```

- Download the latest (stable) release of DUNE-FEM from <http://dune.mathematik.uni-freiburg.de/download.html> and unpack it to your DUNE directory. For example:

```
tar -xzf dune-fem-1.1.tar.gz
tar -xzf dune-fem-howto-1.1.tar.gz
```

Alternatively, if you have read access to the DUNE-FEM svn archive, you can obtain a bleeding edge version of DUNE-FEM as follows:

```
svn checkout https://dune.mathematik.uni-freiburg.de/svn/dune-fem/trunk dune-fem
svn checkout https://dune.mathematik.uni-freiburg.de/svn/dune-femhowto/trunk dune-fem-howto
```

---

<sup>3</sup><http://www.dune-project.org/doc/installation-notes.html>

## 1 What is DUNE-FEM?

- Create a file named `config.opts` with the following content in your DUNE directory:

### Listing 1 (File `../installation/config.opts`)

```
# Standard flags, used as default
STDFLAGS="-O3 -Wall -DNDEBUG -funroll-loops -finline-functions"

# Optimizing flags
# The last option (-march=opteron) has to be adapted to your processor type
OPTIMFLAGS="$STDFLAGS -fomit-frame-pointer -ffast-math -mfpmath=sse -msse3 \
            -march=opteron"

# Debugging flags
DEBUGFLAGS="-g -Wall"

# Paths to installed modules
# You have to adapt these paths if you are not part of the
# Section of Applied Mathematics in Freiburg!
MODDIR="/hosts/raid5/aragorn/dune/modules/$HOSTTYPE/"
MODULEFLAGS="--with-alberta=$MODDIR/alberta \
            --with-alugrid=$MODDIR/alugrid \
            --with-ug=$MODDIR/ug \
            -x-include=/usr/X11R6/include \
            -x-libraries=/usr/X11R6/lib \
            --with-grape=$MODDIR/grape"

# Choose the compiler
COMPILER_FLAGS="CXX=g++ CC=gcc"

# Choose CXXFLAGS from above ($STDFLAGS or $DEBUGFLAGS or $OPTIMFLAGS)
# Remove the option "--disable-documentation" if you want a local doxygen
# documentation (takes quite some time to build).
CONFIGURE_FLAGS="--disable-documentation --disable-parallel $COMPILER_FLAGS \
                \
                CXXFLAGS=\"$STDFLAGS\" $MODULEFLAGS"

MAKE_FLAGS =
```

Please consider this file as an example. It will work correctly only for members of the Section of Applied Mathematics in Freiburg. Other users have to adapt (or comment out) at least the paths to the external modules.

- Finally, configure and compile DUNE using the `dunecontrol` script. Type the following command in your DUNE directory:

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

The script needs several minutes to finish.

## 1 What is DUNE-FEM?

**Warning:** Make sure your DUNE directory contains only one copy of each module. Otherwise the `dunecontrol` script will not work properly.

- Congratulations, you can start working, now.

You can find the above `config.opts` in the subdirectory `doc/installation` of the DUNE-FEM-HOWTO module.

### 1.5 Create your own project

You can create your own DUNE project by using the `duneproject` script. Type the following command in your DUNE directory and follow the instructions:

```
./dune-common/bin/duneproject
```

After creating your project, you have to rerun the `dunecontrol` script to configure your new module:

```
./dune-common/bin/dunecontrol --opts=config.opts --only=YOUR_MODULE_NAME all
```

Take a look in your new DUNE project directory. The `duneproject` script created a sample source code file, which was compiled by `dunecontrol`.

You may replace the sample code with your own. If you want to add source code files, remember to adapt `Makefile.am` as necessary. You can also create your own subdirectories. In this case you will have to create a `Makefile.am` in that directory and modify your project's `configure.ac` accordingly. Then you have to rerun `dunecontrol` in order to create the new make files in these subdirectories. For more information on the build system, have a look at the DUNE Build System Howto (<http://www.dune-project.org/doc/buildsystem/buildsystem.pdf>).

## 2 The Transport-Example with Dune-Fem

This chapter describes the implementation of a simple finite volume scheme for a hyperbolic problem in DUNE-FEM. As an example, we choose the following linear transport problem:

$$\begin{aligned}\partial_t u + \mathbf{a} \cdot \nabla u &= 0 && \text{in } \Omega \times [0, T], \\ u &= u_{in} && \text{on } \Gamma_{in}, \\ u(\cdot, 0) &= u_0 && \text{in } \Omega,\end{aligned}\tag{2.1}$$

where  $\Gamma_{in} = \{(\mathbf{x}, t) \in \partial\Omega \times [0, T] \mid \mathbf{a} \cdot \mathbf{n}(\mathbf{x}) < 0\}$  denotes the inflow boundary. In this example, we choose the following problem data:

$$\Omega = [0, 1]^n,\tag{2.2}$$

$$\mathbf{a} \equiv (1.25, 0.8, \dots, 0.8) \in \mathbb{R}^n,\tag{2.3}$$

$$u_{in}(\mathbf{x}, t) = \sin 2\pi|\mathbf{x} - \mathbf{a}t|^2,\tag{2.4}$$

$$u_0(\mathbf{x}) = \sin 2\pi|\mathbf{x}|^2.\tag{2.5}$$

The exact solution to this problem is given by

$$u(\mathbf{x}, t) := \sin 2\pi|\mathbf{x} - \mathbf{a}t|^2 \quad \text{for } (\mathbf{x}, t) \in \Omega \times [0, T].\tag{2.6}$$

In this discussion, we will restrict ourselves to the case  $n = 2$ . The code, however, is capable of handling arbitrary space dimension.

Change the file `../examples/common/parameter` to work on different grids or in higher space dimensions.

Note: In the current implementation, the values of  $\mathbf{a}$ ,  $u_0$ ,  $u_{in}$ , and  $u$  are hard-coded into the file `../examples/finitevolume/problem.hh`. You must implement a new version of this file in order to use different problem data.

### 2.1 A Tiny Introduction to Finite Volume Schemes

In this section we will give some minimal mathematical background on finite volume schemes and introduce the necessary notation, which will be used throughout this chap-

## 2 The Transport-Example with DUNE-FEM

ter. A rigorous introduction of finite volume schemes can, for example, be found in [7].

For the time interval  $[0, T]$ , we introduce the decomposition

$$J := \{0 = t_0 < \dots < t_N = T\}. \quad (2.7)$$

Furthermore, we consider an unstructured grid  $\mathcal{T}_h$  of  $\Omega$ . The elements of this grid are numbered by  $(T_i)_{i \in I}$  with an index set  $I \subset \mathbb{Z}$ . We introduce the notation

$$S_{i,j} := \bar{T}_i \cap \bar{T}_j \quad (2.8)$$

for the intersection of  $T_i$  and  $T_j$ . For a finite volume scheme we need exactly the intersections of codimension 1:

$$\mathcal{E} := \{(i, j) \in I^2 \mid 0 < \mathcal{H}^{n-1}(S_{i,j}) < \infty\}, \quad (2.9)$$

where  $\mathcal{H}^d$  denotes the  $d$ -dimensional Hausdorff measure on  $\mathbb{R}^n$ . We also introduce the set of  $N(i)$  of neighbors of  $T_i$ , defined by

$$N(T_i) := \{j \in I \mid (i, j) \in \mathcal{E}\} \quad (2.10)$$

and the unit outer normal  $\mathbf{n}_i : \partial T_i \rightarrow \mathbb{R}^d$  to  $T_i$ . Finally, we will need a numerical flux, denoted by  $g : \mathbb{R}^d \times [0, T] \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}$ , with the following properties:

**Consistency**  $g(\mathbf{x}, t, u, u, \mathbf{n}) = (\mathbf{a}u) \cdot \mathbf{n}$ ,

**Conservation**  $g(\mathbf{x}, t, u, v, \mathbf{n}) = g(\mathbf{x}, t, v, u, -\mathbf{n})$ .

Using an explicit discretization for the timestep  $\Delta t_n = t_{n+1} - t_n$ , the numerical treatment of the problem results in

$$u_i^{n+1} = u_i^n - \frac{\Delta t_n}{|T_i|} \sum_{j \in N(T_i)} \int_{S_{ij}} g(\mathbf{x}, t_n, u_i^n, u_j^n, \mathbf{n}_i(\mathbf{x})) d\mathcal{H}^{d-1}(\mathbf{x}), \quad (2.11)$$

$$u_i^0 = \frac{1}{|T_i|} \int_{T_i} u_0(\mathbf{x}) d\mathbf{x} \quad (2.12)$$

Given these quantities, the numerical approximation  $u_h$  of the exact solution is given by

$$u_h(\mathbf{x}, t) := u_i^n \quad \text{for } \mathbf{x} \in T_i \text{ and } t \in [t_n, t_{n+1}). \quad (2.13)$$

## 2 The Transport-Example with DUNE-FEM

Notice that the scheme (2.11) can be interpreted as a forward Euler discretization of the following semidiscrete scheme:

$$\partial_t u_i(t) = -\frac{1}{|T_i|} \sum_{j \in N(T_i)} \int_{S_{ij}} g(\mathbf{x}, t, u_i(t), u_j(t), \mathbf{n}_i(\mathbf{x})) d\mathcal{H}^{d-1}(\mathbf{x}) =: \mathcal{L}_i(t, u) \quad (2.14)$$

$$u_i(0) = \frac{1}{|T_i|} \int_{T_i} u_0(\mathbf{x}) dx \quad (2.15)$$

Therefore, we will implement  $\mathcal{L}_i(t, u)$  and use an ODE solver to solve (2.14). Of course, we could also use an implicit Euler solver or a higher order Runge-Kutta solver.

## 2.2 Implementation overview

In this section we describe the source code of the finite volume example, located in the directory `dune-femhowto/examples/finitevolume`. It consists of the following 2 source files:

`problem.hh` contains the class `U0` representing the initial data  $u_0$  and the exact solution  $u$  for the transport problem (2.1).

`finitevolume.cc` is the main source file (it contains the function `main`). It uses an interface for a general algorithm for evolutionary problems, which can be found in `dune-fem-howto/dune/fem-howto/baseevolution.hh`.

For the rest of this section we will discuss the `main` loop and the function `evolve` in detail. The class `U0` will not be of particular interest because it mostly contains functions to evaluate  $u$  and  $u_0$  and does not provide ingredients which are crucial to the actual algorithm.

### 2.2.1 The main function

We start by discussing the function `main`, displayed in Listing 2.

#### Listing 2 (Excerpt of `dune-femhowto/examples/finitevolume/finitevolume.cc`)

```
366 int main ( int argc, char **argv )
367 try
368 {
369     typedef Dune::GridSelector::GridType GridType;
```

## 2 The Transport-Example with DUNE-FEM

```
370
371 // initialization
372 Dune::MPIManager::initialize( argc, argv );
373 Dune::Parameter::append( argc, argv );
374 Dune::Parameter::append( argc >= 2 ? argv[ 1 ] : "parameter" );
375
376 Dune::GridPtr< GridType > gridptr
377     = initialize< GridType >( "FiniteVolumeScheme" );
378 GridType &grid = *gridptr;
379
380
381 // EOC loop
382 Stepper< GridType > stepper( grid );
383 compute(stepper);
384
385 // dump parameters
386 Dune::Parameter::write( "parameter.log" );
387
388 return 0;
389 }
390 catch( const Dune::Exception &e )
391 {
392     std::cerr << e << std::endl;
393     return 1;
394 }
395 catch (...)
396 {
397     std::cerr << "Generic exception caught." << std::endl;
398     return 2;
399 }
```

First, we want to discuss the main program in the file `finitevolume.cc`. As the basic numerical scheme is delegated to the `compute` function, the problem specific code parts can be found in the `FiniteVolumeScheme` and `Stepper` structs as can be seen in listing 3. An explanation of the most important lines in the struct definitions and the main loop follows after the listing.

### Listing 3 (Excerpt of `dune-femhowto/examples/finitevolume/finitevolume.cc`)

```
23 template< class DestinationType, class Problem >
24 struct FiniteVolumeScheme
25 : public Dune::SpaceOperatorInterface< DestinationType >
26 {
27     // type of discrete function space
28     typedef typename DestinationType::DiscreteFunctionSpaceType DiscreteSpaceType;
29
30     // type of grid part
31     typedef typename DiscreteSpaceType::GridPartType GridPartType;
32
33     // suitable iterator type over the codim 0 entities of the grid part
```

## 2 The Transport-Example with DUNE-FEM

```
34 typedef typename DiscreteSpaceType::IteratorType IteratorType;
35
36 // type of codim 0 entity and entity pointer
37 typedef typename IteratorType::Entity Entity;
38 typedef typename Entity::EntityPointer EntityPointer;
39
40 // type of geometry
41 typedef typename Entity::Geometry Geometry;
42
43 // type of intersections and intersection iterators
44 typedef typename GridPartType::IntersectionIteratorType
45 IntersectionIteratorType;
46 typedef typename IntersectionIteratorType::Intersection IntersectionType;
47
48 // type of local function
49 typedef typename DestinationType::LocalFunctionType LocalFunctionType;
50
51
52
53
54
55
56
57 // return reference to space (needed by ode solvers)
58 const DiscreteSpaceType &space () const
59 {
60     return space_;
61 }
62
63
64 // initialize solution to initial data
65 void initialize ( DestinationType &solution )
66 {
67     // iterate over all entities
68     const IteratorType end = space().end();
69     for( IteratorType it = space().begin(); it != end; ++it )
70     {
71         // obtain entity and geometry
72         const Entity &entity = *it;
73         const Geometry &geo = entity.geometry();
74
75         // obtain local function
76         LocalFunctionType lf = solution.localFunction( entity );
77
78         // evaluate initial data in element's center
79         RangeType initialValue;
80         problem_.evaluate( geo.center(), initialValue );
81
82         // directly manipulate the only local DoF (degree of freedom)
83         lf[ 0 ] = initialValue[ 0 ];
84     }
85
86     // communicate the data to other processes (for parallel runs only)
87     solution.communicate();
88 }
89
90
91 // application operator
92 void operator () ( const DestinationType &solution,
93                 DestinationType &update ) const
```

## 2 The Transport-Example with DUNE-FEM

```
112 {
113     // clear update (i.e., initialize to zero)
114     update.clear();
115
116     // initialize time step to infinity
117     dt_ = std::numeric_limits< double >::infinity();
118
119     // obtain grid part and index set references
120     const GridPartType &gridPart = space().gridPart();
121     const typename GridPartType::IndexSetType &indexSet = gridPart.indexSet();
122
123     // iterate over all entities
124     const IteratorType end = space().end();
125     for( IteratorType it = space().begin(); it != end; ++it )
126     {
127         // obtain entity and geometry
128         const Entity &entity = *it;
129         const Geometry &geo = entity.geometry();
130
131         // assert CFL condition for quadrilateral grids
132         const double timeFactor = (geo.type().isCube() ? 0.5 : 1.0);
133
134         // obtain local functions for solution and update
135         LocalFunctionType lfSolEn = solution.localFunction( entity );
136         LocalFunctionType lfUpdEn = update.localFunction( entity );
137
138         // element volume
139         const double enVolume = geo.volume();
140
141         // iterate over all intersections
142         const IntersectionIteratorType iend = gridPart.iend( entity );
143         for( IntersectionIteratorType iit = gridPart.ibegin( entity );
144             iit != iend; ++iit )
145         {
146             // obtain intersection
147             const IntersectionType &intersection = *iit;
148
149             // obtain reference element for intersection
150             const ReferenceElementType &referenceElement
151                 = ReferenceElementContainerType::general(intersection.type());
152
153             // local coordinate of intersection's barycenter
154             const FaceLocalCoordinate &x = referenceElement.position( 0, 0 );
155
156             const double volumeIntersection = referenceElement.volume();
157
158             if( !intersection.boundary() )
159             {
160                 // assert that interior intersections have a neighbor
161                 assert( intersection.neighbor() );
162
163                 // access neighbor
164                 const EntityPointer neighborPointer = intersection.outside();
```

## 2 The Transport-Example with DUNE-FEM

```
165     const Entity &neighbor = *neighborPointer;
166
167
168     // compute flux from one side only (except it's not an interior entity)
169     if( (neighbor.partitionType() != Dune::InteriorEntity)
170         || (indexSet.index( entity ) < indexSet.index( neighbor )) )
171
172         DomainType normal = intersection.integrationOuterNormal( x );
173
174         // obtain velocity
175         const DomainType &velocity = problem_.velocity();
176
177         // calculate wave speed
178         const double waveSpeed = (velocity * normal) * volumeIntersection;
179
180         // compute upwind numerical flux
181         RangeType flux;
182         flux[ 0 ]
183             = (waveSpeed > 0 ? lfSolEn[ 0 ] : lfSolNb[ 0 ]) * waveSpeed;
184
185         // compute update for entity and neighbor
186         lfUpdEn[ 0 ] -= flux[ 0 ] / enVolume;
187         lfUpdNb[ 0 ] += flux[ 0 ] / nbVolume;
188
189         // compute time step restriction
190         const double dtLocal = timeFactor * std::min( enVolume, nbVolume ) /
191             std::abs( waveSpeed );
192         dt_ = std::min( dt_, dtLocal );
193     }
194 }
195 else // if( !intersection.boundary() )
196 {
197     DomainType normal = intersection.integrationOuterNormal( x );
198
199     // obtain velocity
200     const DomainType &velocity = problem_.velocity();
201
202     // calculate wave speed
203     const double waveSpeed = (velocity * normal) * volumeIntersection;
204
205     // compute upwind numerical flux
206     RangeType flux;
207     if( waveSpeed < 0 )
208     {
209         const DomainType xBnd
210             = geo.global( intersection.geometryInInside().global( x ) );
211
212         RangeType uBnd;
213         problem_.evaluate ( time_, xBnd, uBnd );
214         flux[ 0 ] = uBnd[ 0 ] * waveSpeed;
215     }
216 }
217 else
218     flux[ 0 ] = lfSolEn[ 0 ] * waveSpeed;
219 }
```

## 2 The Transport-Example with DUNE-FEM

```
225
226     // compute update for entity
227     lfUpdEn[ 0 ] -= flux[ 0 ] / enVolume;
228
229     // compute time step restriction
230     const double dtLocal = timeFactor * enVolume / std::abs( waveSpeed );
231     dt_ = std::min( dt_, dtLocal );
232 }
233 }
234 }
235
236 // communicate the data to other processes (for parallel runs only)
237 update.communicate();
238 }

260 template <class GridImp>
261 struct FVStepperTraits
262 {
263     // type of grid
264     typedef GridImp GridType;
265
266     // Initial data
267     //typedef U0<GridType, Dune::Double> InitialData;
268     typedef U0<GridType, double> InitialData;
269     // Define the discrete function for the unknown solution
270     // ... first the grid part ...
271     typedef Dune::AdaptiveLeafGridPart<GridType> GridPartType;
272     // ... next the function space ...
273     typedef Dune::FunctionSpace
274         <typename InitialData::DomainFieldType,
275          typename InitialData::RangeFieldType,
276          GridType::dimensionworld, InitialData::dimRange>
277         FunctionSpaceType;
278     // ... followed by the discrete function space type ...
279     typedef Dune::FiniteVolumeSpace
280         < FunctionSpaceType, GridPartType, 0, Dune::CachingStorage >
281         DiscreteSpaceType;
282     // ... and finally the type for the discrete function ...
283     typedef Dune::AdaptiveDiscreteFunction
284         < DiscreteSpaceType >
285         DiscreteFunctionType;

291 template <class GridImp>
292 struct Stepper : public AlgorithmBase< FVStepperTraits< GridImp > >
293 {
294     // my traits class
295     typedef FVStepperTraits< GridImp > Traits;
296     // base class
297     typedef AlgorithmBase< Traits > BaseType ;
298
299     // type of grid
300     typedef typename Traits :: GridType GridType;
301
```

## 2 The Transport-Example with DUNE-FEM

```
302 // Initial data
303 typedef typename Traits :: InitialData InitialData;
304 // Define the discrete function for the unknown solution
305 // ... first the grid part ...
306 typedef typename Traits :: GridPartType GridPartType;
307 // ... followed by the discrete function space type ...
308 typedef typename Traits :: DiscreteSpaceType DiscreteSpaceType;
309 // ... and finally the type for the discrete function ...
310 typedef typename Traits :: DiscreteFunctionType DiscreteFunctionType;
311
312 // type of time provider
313 typedef typename BaseType :: TimeProviderType TimeProviderType;
314
315 Stepper(GridType& grid) :
316     BaseType( grid ),
317     problem_(),
318     spaceSolver_(this->space(), problem_),
319     odeSolver_(0),
320     eocId_( Dune::FemEoc::addEntry(std::string("L1-error")) ),
321     verbose_( Dune::Parameter :: verbose() )
322 {
323     // distribute the grid to other processes (for parallel runs only)
324     grid.loadBalance();
325 }
326
327 void initializeStep(TimeProviderType& tp, DiscreteFunctionType& u)
328 {
329     spaceSolver_.initialize(u);
330     odeSolver_ = new DuneODE::ExplicitRungeKuttaSolver<DiscreteFunctionType>(
331         spaceSolver_, tp, 1);
332     odeSolver_->initialize(u);
333 }
334
335 void step(TimeProviderType& tp, DiscreteFunctionType& u) {
336     assert(odeSolver_);
337     odeSolver_->solve(u);
338 }
339
340 void finalizeStep(TimeProviderType& tp, DiscreteFunctionType& u) {
341     Dune::L1Error<DiscreteFunctionType> L1err;
342     // Compute L1 error of discretized solution ...
343     typename InitialData::RangeType error;
344     error = L1err.norm(problem_, u, tp.time());
345     // ... and print the statistics out to the eocOutputPath file
346     typename InitialData::RangeFieldType err = error.two_norm2();
347     Dune::FemEoc::setErrors(eocId_, sqrt(err));
348     delete odeSolver_;
349     odeSolver_ = 0;
350 }
```

## 2 The Transport-Example with DUNE-FEM

We start with some general words about the structs `FiniteVolumeScheme` and `Stepper`. `FiniteVolumeScheme` contains all necessary data and methods to calculate the spatial discretization step of the numerical scheme, i.e., the update vector  $-\frac{1}{|T_i|} \sum_{j \in N(i)} g_{ij}^n(u_i^n, u_j^n)$  in 2.11. `Stepper` on the other hand realizes the calculation of a single timestep using an ODE solver.

In the first few lines of the struct `FiniteVolumeScheme` several typedefs for the grid type, the function spaces etc. are made. After that, the methods `space`, `initialize` and `operator ()` are defined.

The method `space` simply returns a reference to the instance of a discrete function space that is used. It is invoked by the ODE solver.

The method `initialize` should be used to project the initial data to its discrete representative `DestinationType& u`, which in this case is an `AdaptiveDiscreteFunction`. The given implementation does a for-loop over all grid entities in line 88. For each entity the given function  $u_0$  is evaluated at the barycenter. For we calculate a piecewise constant solution, the local function `lf` contains just one degree of freedom, `lf[0]`. A deeper insight into the DOF handling is contained in chapter 3.

We now have a detailed look at the method `operator ()`. As mentioned above, it calculates the update vector regarding the numerical scheme, cf. (2.11). The work mostly consists in calculating the numerical fluxes between the grid entities, so we start with a for-loop over all grid entities in line 125. For the current entity `entity`, the lines 128-139 initialize some geometric information and the index of `entity` as well as the local functions (with respect to `entity`) belonging to `u` and `update`. After that, another for-loop iterates over all intersections, i.e., all edges, of `entity`. The lines 147-156 initialize the `intersection`, its midpoint `xIntersection` and volume `volumeIntersection`.

Each intersection can be either an interior or a boundary intersection. We start with the interior ones, see line 158. In lines 164-166 an entity object `neighbor` and the corresponding entity index are initialized. Due to the nature of the numerical fluxes we need to calculate the flux between two entities  $T_i$  and  $T_j$  only once. This is ensured by the if-clause starting in line 169. After getting the outer normal of the intersection, the numerical flux between `entity` and `neighbor` is calculated in lines 183-191. Here, and Engquist-Osher flux is used. Due to the special form of the given problem, the evaluation of the flux simplifies to the given code. At last we calculate the entity updates and timestep restriction (according to a CFL condition).

The `else` part of the if-clause starts in line 202 and handles boundary intersections. The code is basically the same as in the interior case. The only difference concerns the handling of inflow boundaries. This is done in lines 215-222 where the exact solution  $u$

## 2 The Transport-Example with DUNE-FEM

of the problem is used to calculate the flux.

We now turn to the struct `Stepper` which provides the necessary data and methods for calculating a single timestep. The first few lines of the definition contain several typedefs for the grid type, function spaces and initial data. After that, the methods `initialize`, `step` and `finalize` are defined.

The method `initialize` uses `spaceSolver_`, an instance of `FiniteVolumeScheme`, to initialize the solution vector `u` with the initial data. After that, an explicit ODE solver `odeSolver_` is defined and initialized.

The method `step` uses the ODE solver to calculate a single timestep for our problem. During this process, the ODE solver will execute the method `operator ()` of the `FiniteVolumeScheme` in order to evaluate the spatial discretization operator  $\mathcal{L}(u)$ .

The method `finalize` calculates the  $L^1$ -error between numerical and exact solution. The error vector is then given to the `Dune::FemEoc` singleton. This DUNE-class is used to generate EOC data and stores these data in a `.tex` file. At last the method does some clean-up work.

Finally we give some words about the `main-loop`. For the most work is delegated to the structs `FiniteVolumeScheme` and `Stepper` and the function `compute`, this code part is kept rather short. In lines 33-37 the `Dune::Parameter` singleton is used to process certain runtime parameters. By default it is initialized with the `parameter` file. Other choices can be made via commandline arguments. Some more information about `Dune::Parameter` will be given in the next subsection.

After that, in line 376 the method `initialize` from the file `dune-fem-howto/dune/fem-howto/base.hh` is used to create the gridpointer. Finally, in lines 382-383 a `Stepper` object is initialized and the method `compute` is invoked to calculate the solution. The last few lines 390-399 do some runtime error catching.

### 2.2.2 The basic algorithm

As we have seen in the previous section, the struct `Stepper` is derived from `AlgorithmBase`, which is an interface for an algorithm solving general evolutionary problems. It is located in `dune-fem-howto/dune/fem-howto/baserevolution.hh`. The most important method of this interface is the `operator ()`. A default implementation solving the evolutionary problem on the time interval `[starttime, endtime]` is given. We will discuss all this in more detail after the following listing.

## 2 The Transport-Example with DUNE-FEM

### Listing 4 (Excerpt of dune-fem-howto/dune/fem-howto/baseevolution.hh)

```
36 template <class TraitsImp>
37 class AlgorithmBase

70 ///! return reference to space
71 DiscreteSpaceType& space()
72 {
73     return space_;
74 }
75
76
77 ///! initialize method for time loop, i.e. L2-project initial data
78 virtual void initializeStep(TimeProviderType& tp,
79                             DiscreteFunctionType & solution) = 0;
80
81 ///! solve one time step
82 virtual void step(TimeProviderType& tp,
83                  DiscreteFunctionType & solution) = 0;
84
85 ///! finalize problem, i.e. calculate EOC ...
86 virtual void finalizeStep(TimeProviderType& tp,
87                            DiscreteFunctionType & solution) = 0;
88
89 ///! restore all data from check point (overload to do something)
90 virtual void restoreFromCheckPoint(TimeProviderType& tp,
91                                    DiscreteFunctionType & solution) {}
92
93 ///! restore all data from check point (overload to do something)
94 virtual void writeCheckPoint(TimeProviderType& tp) const {}
95
96 ///! default time loop implementation, overload for changes
97 virtual void operator()(DiscreteFunctionType & solution)
98 {
99     const bool verbose = Dune::Parameter::verbose();
100     int printCount = Dune::Parameter::getValue<int>("femhowto.printCount", -1);
101
102     const double maxTimeStep =
103         Dune::Parameter::getValue("femhowto.maxTimeStep", std::numeric_limits<double>
104                                     >::max());
105     const double startTime = Dune::Parameter::getValue<double>("femhowto.startTime",
106                                                                 0.0);
107     const double endTime = Dune::Parameter::getValue<double>("femhowto.endTime",
108                                                                0.9);
109
110     /// for statistics
111     double maxdt = 0.;
112     double mindt = 1.e10;
113     double averagedt = 0.;
114
115     /// Initialize TimeProvider
116     TimeProviderType tp(startTime, grid_);
117
118     /// type of adaptation manager
```

## 2 The Transport-Example with DUNE-FEM

```

116     typedef Dune::AdaptationManager< GridType, RestrictionProlongationType >
117         AdaptationManagerType;
118
119     RestrictionProlongationType rp( solution );
120
121     // set refine weight
122     rp.setFatherChildWeight( Dune::DGFGGridInfo<GridType> :: refineWeight() );
123
124     // create adaptation manager
125     AdaptationManagerType adaptManager( grid_ , rp );
126
127     // set initial data (and create ode solver)
128     initializeStep(tp, solution);
129
130     // restoreData if checkpointing is enabled (default is disabled)
131     restoreFromCheckPoint( tp, solution );
132
133     typedef Dune::Tuple< DiscreteFunctionType* >          IOTupleType;
134     typedef Dune::DataWriter< GridType, IOTupleType >    DataWriterType;
135
136     IOTupleType dataTup ( &solution );
137
138     // create data writer after restore is done, otherwise tp is not set correctly
139     DataWriterType eocDataOutput( grid_, dataTup, tp, EocDataOutputParameters(
140         loop_ ) );
141
142     // write initial data
143     eocDataOutput.write( tp );
144
145     //*****
146     /* Time Loop
147     /******
148     tp.provideTimeStepEstimate(maxTimeStep);
149     for( tp.init() ; tp.time() < endTime ; tp.next() )
150     {
151         tp.provideTimeStepEstimate(maxTimeStep);
152
153         //*****
154         /* Compute an ODE timestep
155         /******
156         Dune::FemTimer::start(timeStepTimer_);
157         step(tp, solution );
158         Dune::FemTimer::stop(timeStepTimer_ ,Dune::FemTimer::max);
159
160
161         eocDataOutput.write( tp );
162
163     } /***** END of time loop *****/
164
165     averagedt /= double(tp.timeStep());
166     if(verbose)
167     {
168         std::cout << "Minimum dt: " << mindt

```

## 2 The Transport-Example with DUNE-FEM

```
192         << "\nMaximum_dt:_" << maxdt
193         << "\nAverage_dt:_" << averagedt << std::endl;
194     }
195
196     // finalize time step
197     finalizeStep(tp, solution);
198
199     // increase loop counter
200     ++loop_;
201 }
202
203 //! finalize problem, i.e. calculated EOC ...
204 virtual void finalize(DiscreteFunctionType & solution)
205 {
206
207 protected:
208     GridType&          grid_;
209     GridPartType      gridPart_;
210     DiscreteSpaceType space_;
211     unsigned int timeStepTimer_;
212     unsigned int loop_ ;
213 };
```

After the declaration of the `AlgorithmBase` class a few typedefs are made. Because these are just plain forwardings from the given `TraitsImp` class this part is excluded from the listing. The most relevant code parts are the virtual methods `initializeStep`, `step`, `finalizeStep` and `operator ()`. For the first three of them are implemented by the `Stepper` class as seen in the last section, we just need to give some explanation to the last one.

As mentioned prior to the listing, `operator ()` gets a default implementation computing the solution of the underlying evolutionary problem within the time interval `[starttime, endtime]`. We will discuss this code now.

The first few lines define some important parameters like start- and endtime. In order to calculate the numerical solution, an instance of a `TimeProviderType` is needed (line 113). This object is used by the ODE solver and within time loop. Amongst others it provides time step estimates and ensures a CFL number to be granted. With this `TimeProvider` we can set the initial data using the method `initializeStep`, cf. line 127.

The lines 132-?? define and initialize two datatypes. The first one is `IOTupleType` that is used to store some pointers to discrete functions in. The second one is `DataOutputType`. It gets a grid and an `IOTupleType` to create some output, e.g. for visualizing the calculated solution in Paraview. In this example only the (discrete) solution  $u$  is used as output.

## 2 The Transport-Example with DUNE-FEM

Within the time loop at first a time step estimate is needed. This is followed by starting the `FemTimer`. This is a DUNE-FEM singleton that provides means to measure the calculation time of the algorithm. After that the next time step is calculated using the `step` method (line 158). Remember that this method is implemented within the `Stepper` struct. As mentioned above this invokes an ODE solver to solve  $\partial_t u = \mathcal{L}(u)$  where the operator  $\mathcal{L}$  on the right hand side is given by the `operator ()` method of the `FiniteVolumeScheme` struct. When this is done the timer is stopped, some EOC output and additional statistical data are generated. This finishes the time loop.

After the time loop the `finalize` method of the `Stepper` struct is called in line 96.

### 2.2.3 The methods `initialize` and `compute`

As mentioned earlier we also use two functions which are contained within the header file `dune-fem-howto/dune/fem-howto/base.hh`, namely `initialize` and `compute`. Both functions will be discussed after the listing.

#### Listing 5 (Excerpt of `dune-fem-howto/dune/fem-howto/base.hh`)

```
28 template< class HGridType >
29 Dune::GridPtr< HGridType > initialize ( const std::string &problemDescription )
30 {
31     // ----- read in runtime parameters -----
32     const std::string filekey = Dune::IOInterface::defaultGridKey( HGridType::
        dimension );
33     const std::string filename = Dune::Parameter::getValue< std::string >( filekey )
        ;
34     int startLevel = Dune::Parameter::getValue<int>("femhowto.startLevel", 0);
35
36     // output of error and eoc information
37     std::string eocOutPath = Dune::Parameter::getValue<std::string>("femhowto.
        eocOutputPath",
38                                                                                     std::string("."));
39     std::string eocFile = eocOutPath + std::string("/eoc.tex");
40     Dune::FemEoc::initialize(eocOutPath, "eoc", problemDescription);
41
42     // initialize grid
43     Dune::GridPtr< HGridType > gridptr(filename);
44     // and refine the grid until the startLevel is reached
45     for(int level=0; level < startLevel ; ++level)
46         Dune::GlobalRefine::apply(*gridptr, 1 );
47     return gridptr;
48 }

55 template <class Algorithm>
56 void compute(Algorithm& algorithm)
```

## 2 The Transport-Example with DUNE-FEM

```
57 {
58     typedef typename Algorithm::DiscreteFunctionType DiscreteFunctionType;
59     typename Algorithm::DiscreteSpaceType& space = algorithm.space();
60     typename Algorithm::GridPartType& gridPart = space.gridPart();
61     typedef typename Algorithm::GridPartType::GridType HGridType;
62     HGridType& grid = gridPart.grid();
63
64     // solution function
65     DiscreteFunctionType u("solution",space);
66
67     // get some parameters
68     const int eocSteps = Dune::Parameter::getValue<int>("femhowto.eocSteps", 1);
69
70     // Initialize the DataOutput that writes the solution on the harddisk in a
71     // format readable by e.g. Paraview
72     // in each loop for the eoc computation the results at
73     // the final time is stored
74     typedef Dune::Tuple<DiscreteFunctionType*> IOTupleType;
75     typedef Dune::DataOutput<HGridType, IOTupleType> DataOutputType;
76     IOTupleType dataTup ( &u );
77     DataOutputType dataOutput( grid, dataTup );
78
79     const unsigned int femTimerId = Dune::FemTimer::addTo("timestep");
80     for(int eocloop=0; eocloop < eocSteps; ++eocloop)
81     {
82         Dune::FemTimer :: start(femTimerId);
83
84         // do one step
85         algorithm(u);
86
87         double runTime = Dune::FemTimer::stop(femTimerId);
88
89         Dune::FemTimer::printFile("./timer.out");
90         Dune::FemTimer::reset(femTimerId);
91
92         // Write solution to hd
93         dataOutput.writeData(eocloop);
94
95         // finalize algorithm (compute errors)
96         algorithm.finalize(u);
97
98         // calculate grid width
99         const double h = Dune::GridWidth::calcGridWidth(gridPart);
100
101         if( Dune::Parameter :: verbose() )
102             Dune::FemEoc::write(h,grid.size(0),runTime,0, std::cout);
103         else
104             Dune::FemEoc::write(h,grid.size(0),runTime,0);
105
106         // Refine the grid for the next EOC Step. If the scheme uses adaptation,
107         // the refinement level needs to be set in the algorithms' initialize method.
108         if(eocloop < eocSteps-1)
109             {
```

## 2 The Transport-Example with DUNE-FEM

```
110     Dune::GlobalRefine::apply(grid,Dune::DGFGGridInfo<HGridType>::
        refineStepsForHalf());
111     grid.loadBalance();
112 }
113 } /***** END of EOC Loop *****/
114 Dune::FemTimer::removeAll();
115 }
```

First we discuss the method `initialize`. The `Dune::Parameter` singleton parses the given parameter file line by line. A parameter line must have the format `<key>: <value>`. Whilst `<key>` must be an alphanumerical string, `<value>` can contains strings or numbers. Lines starting with a `#` are understood as commentaries. The lines [33-37](#) show you how to work with `Dune::Parameter`. Have a look at the most important methods of `Dune::Parameter`:

- `get` expects three parameter. The first one is a `std::string` representing the `<key>`. The second one is a reference to the variable where the related `<value>` will be stored. The third one is a default value that will be used if the key cannot be found. This last parameter is optional.
- `getValue` works similar to `get`. The difference is that the value is not stored in a reference but returned by the method. Thats why `getValue` has only two parameters: The key as `std::string` and an optional default value.

In line [40](#) the `Dune::FemEoc` singleton is initialized with the path for a `.tex` file where the EOC data will be stored in. Finally the `apply` method from the `Dune::GlobalRefine` singleton is used in line [46](#) to do a given number of initial refinements on the grid.

Now we have a look at the `compute` method which produces EOC data for the evolutionary problem represented by the given instance of `Algorithm`, cf. the `AlgorithmBase` interface from the last subsection. The first few lines so some typedefs and define some important variables like the grid, the discrete solution  $u$  etc. After that the `Dune::FemTimer` singleton is used to create a new timer for measuring the runtime needed for solving the given problem. The variable `timeStepTimer` is the id of the created timer.

Finally, the EOC loop starts is started doing a given number of EOC steps. For each step the complete (discrete) numerical solution  $u$  is calculated using the given `algorithm` (line [85](#)). After the solution is calculated the runtime is measured, written out to a file `./timer.out` and the timer is reset for the next loop. In line [96](#) the `finalize` method of `algorithm` struct is called. At last, in line [110](#) the grid is globally refined for the next EOC step using the same technique as in the `initialize` method.

## 2.3 Parallelization

The parallelization of this example is very easy. Here, we show the few lines of code that differ from the serial algorithm. The first step in each parallel code is to distribute the grid to all processes available. This is done by calling `grid.loadBalance()` in line 324 of Listing 3.

During the execution of the algorithm we need to adjust three things. After setting the initial data we have to invoke a communication that updates all non-interior entities on each process. This is done by simply calling `solution.communicate()` in line 106 in Listing 3. In the same way the flux update is communicated by calling `update.communicate()` in line 237 of Listing 3. Note that during the flux calculation we have to make sure that the flux is calculated when the neighbor is not an interior entity, see line 169 of Listing 3.

The synchronization of the time step size is done automatically by the `TimeProvider` class.

In order to start the program `finitevolume` on two processors, type

```
mpiexec -n 2 ./finitevolume
```

at the command line. Make sure that all DUNE modules were configured with the option `--enable-parallel`.

Please read also Section 3.3.

### 3 Solving the Poisson problem

In this chapter we want to show how to use DUNE-FEM for solving an elliptic test problem with the Finite Element method. As a test problem, we choose the Poisson problem

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{d=2,3}, \quad (3.1)$$

$$u = g \quad \text{on } \partial\Omega. \quad (3.2)$$

The source code of the implementation described in this chapter is located in the directory `dune-femhowto/examples/poisson`. In this example, we choose the following problem data (with  $n = \text{dimworld}$ ):

$$\Omega := ]0, 1[^n, \quad \mathbf{x} = (x_1, \dots, x_n) \quad (3.3)$$

$$f(\mathbf{x}) := 4n\pi^2 \prod_{i=1}^n \sin(2\pi x_i) \quad \forall \mathbf{x} \in \Omega \quad (3.4)$$

$$u(\mathbf{x}) := \prod_{i=1}^n \sin(2\pi x_i) \quad \forall \mathbf{x} \in \Omega \quad (3.5)$$

$$g(\mathbf{x}) := u(\mathbf{x}) \quad \forall \mathbf{x} \in \partial\Omega \quad (3.6)$$

where  $u$  is the exact solution. The boundary values are just the values of the exact solution on the boundary points.

In the current implementation we implemented several alternative right hand side functions  $f$  und corresponding exact solutions  $u$  for the problem (3.1) (see `problemdata.hh`). You can switch between them by changing the runtime parameter `femhowto.poissonproblem` in the parameter file `parameter`.

The numerical treatment of problem (3.1) is described in chapter 1 of the script [8]. The numerical discussion ends with the equation system

$$\mathbf{A}\mathbf{u} = \mathbf{b} \quad (3.7)$$

### 3 Solving the Poisson problem

that needs to be solved by the numerical scheme for the unknown discrete function  $\mathbf{u} \in V_h$ , where  $V_h \in H_0^1(\Omega)$  is a suitable discrete function space with a basis  $\{\varphi_1, \dots, \varphi_N\}$ . The right hand side  $\mathbf{b}$  is given by

$$\mathbf{b} := (f\varphi_i)_{1 \leq i \leq N}. \quad (3.8)$$

Except boundary corrections the matrix  $A$  equals the stiffness matrix

$$S := \left( \int_{\Omega} \nabla \varphi_j \nabla \varphi_i \right)_{1 \leq i, j \leq N}. \quad (3.9)$$

The degrees of freedom (DOFs), that lie on the boundary  $\partial\Omega$  however, are defined by the values of the boundary data function  $g$ . Therefore, for matrix  $A$  the corresponding rows in the stiffness matrix need to be substituted by a unit vector such that for the  $i$ -th DOF  $u_i$  of the discrete solution  $\mathbf{u}$  we ensure  $u_i = g(p_i)$ , where  $p_i$  is the node coordinate corresponding to the DOF. Schematically, the substitution in the stiffness matrix  $S$  looks like this:

$$i \rightarrow \begin{pmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{pmatrix} \begin{pmatrix} \vdots \\ u_i \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ g(p_i) \\ \vdots \end{pmatrix}. \quad (3.10)$$

$\uparrow$   
 $i$

## 3.1 Implementation

We first describe the basic implementation of the algorithm without adaptive or parallel calculations. Adaptive and parallel calculations will be the topic of the sections 3.2 and 3.3, respectively. The basic implementation consists of the following source files:

- **main.hh**: Contains the `main` function. The algorithm first initializes the problem data (implemented in the file `problemdata.hh`) and then calls the main algorithm (implemented in the file `algorithm.hh`). We utilize here the function `compute` from the header file `dune-femhowto/examples/common/base.hh`, which is an abstract implementation of a numerical scheme for stationary problems.
- **problemdata.hh**: Contains several classes which implement several right hand side functions  $f$  and the corresponding exact solutions  $u$  for the problem 3.1. The actual problem class is chosen at runtime, see the parameter file `parameter`. In

### 3 Solving the Poisson problem

cases where an exact solution is available we use it as Dirichlet boundary data (see class `Algorithm` in the file `algorithm.hh`), so there is no need to explicitly implement the boundary data  $g$ .

- `algorithm.hh`: Contains a class `Algorithm` which provides the main algorithm. This algorithm is described in section 3.1.1 in detail.
- `laplaceoperator.hh`: Contains the classes `LaplaceOperator` and `RightHandSideAssembler` which model the discrete stiffness matrix  $A$  with entries for DOFs on the region's boundary and the discrete right hand side  $\mathbf{b}$  as described in the last section. The implementation of the right hand side will be the topic of section 3.1.2.

#### 3.1.1 Algorithm

We begin with the `main` function in the file `main.hh`, see Listing 6. We delegated the basic numerical scheme to the `compute` function in the file `../common/base.hh`, the problem specific code parts are implemented in the class `Algorithm` (described below), so the `main` function is rather short.

#### Listing 6 (Excerpt of `dune-femhowto/examples/poisson/main.cc`)

```
63 int main( int argc, char **argv )
64 {
65     typedef Dune::GridSelector::GridType HGridType;
66
67     // initialize MPI
68     Dune::MPIManager::initialize ( argc, argv );
69     const int rank = Dune::MPIManager::rank ();
70
71     try
72     {
73         // append parameters from the comand line
74         Dune::Parameter::append( argc, argv );
75
76         // append parameters from the parameter file
77         Dune::Parameter::append( (argc < 2) ? "parameter" : argv[ 1 ] );
78
79         // generate GridPointer holding grid instance
80         Dune::GridPtr< HGridType > gridptr = initialize< HGridType >(std::string("
            Poisson_ problem"));
81
82         // get grid reference
83         HGridType& grid = *gridptr ;
84
85         // create problem
86         ProblemType* problem = Dune::createProblem<HGridType> ();
87         assert( problem );
```

### 3 Solving the Poisson problem

```
88
89 // create stepper class
90 Algorithm<HGridType, polynomialOrder> algorithm(grid, *problem);
91 // compute solution
92 compute(algorithm);
93
94 // write parameter logfile
95 Dune::Parameter :: write("parameter.log");
96
97 // remove problem
98 delete problem;
99
100 return 0;
101 }
102 catch( const Dune::Exception &exception )
103 {
104     if( rank == 0 )
105         std::cerr << exception << std::endl;
106     return 1;
107 }
108 }
```

After initializing the grid in line 80, we get an instance of a class containing the problem data (line 86) and an instance of the class `Algorithm`, containing the algorithm. After that, the function `compute` is executed in line 92. The function `compute`, defined in the file `../common/base.hh`, calls the `operator()` routine from the class `Algorithm` several times at different grid sizes and the experimental order of convergence is computed.

Now, we are going to explain the problem specific algorithm in the class `Algorithm`, see Listing 7. An explanation of the most important lines in the class definition follows after the listing.

#### Listing 7 (Excerpt of `dune-femhowto/examples/poisson/algorithm.hh`)

```
105 class Algorithm
106 {
107 public:
108
109 //! constructor
180 Algorithm(HGridType & grid, const ProblemType& problem)
181 : grid_( grid ),
182   gridPart_( grid_ ),
183   space_( gridPart_ ),
184   problem_( problem )
185 {
186     // add entries to eoc calculation
187     std::vector<std::string> femEocHeaders;
188     // we want to calculate L2 and H1 error (and EOC)
189     femEocHeaders.push_back("$L^2-error");
```

### 3 Solving the Poisson problem

```
190     femEocHeaders.push_back("$H^1$-error");
197 }
198
199 //! setup and solve the linear system
200 void operator()(DiscreteFunctionType & solution)
201 {
220     // initialize solution with zero
221     solution.clear();
222
223     // create laplace assembler (is assembled on call of systemMatrix by solver)
224     LaplaceOperatorType laplace( space_ , problem_ );
225
226     // create right hand side
227     DiscreteFunctionType rhs( "rhs", space_ );
228
229     // initialize as zero
230     rhs.clear();
231
232     // setup right hand side
233     const int quadratureOrder = 2 * space_.order() + 1;
234     Dune::RightHandSideAssembler< DiscreteFunctionType >
235         ::assemble ( problem_ , rhs , quadratureOrder );
236
237     // set Dirichlet Boundary to boundary data
238     bool hasDirichletBoundary = false;
239     const IteratorType endit = space_.end();
240     for( IteratorType it = space_.begin(); it != endit; ++it )
241     {
242         const EntityType &entity = *it;
243         // if an entity has intersections with the boundary adjust dirichlet nodes
244         if( entity.hasBoundaryIntersections() )
245             hasDirichletBoundary |= boundaryTreatment( entity, problem_ , rhs , solution
246                 );
247
248         // if no dirichlet boundary is set then assume Neumann boundary
249         if( !hasDirichletBoundary )
250         {
251             clearMeanValue( rhs );
252             setExactMeanValue( ugrid , solution );
253         }
254
255         // solve the linear system @\ref{
256         solve( laplace , rhs , solution );
257     }
258
259     //! finalize computation by calculating errors and EOCs
260     void finalize(DiscreteFunctionType & solution)
261     {
262         // create exact solution
263         ExactSolutionType uexact( problem_ );
```

### 3 Solving the Poisson problem

```
264
265 // create grid function adapter
266 GridExactSolutionType ugrid( "exact_solution", uexact, gridPart_,
267                               DiscreteSpaceType :: polynomialOrder + 1 );
268
269 // create L2 - Norm
270 Dune::L2Norm< GridPartType > l2norm( gridPart_ );
271 // calculate L2 - Norm
272 const double l2error = l2norm.distance( ugrid, solution );
273
274 // create H1 - Norm
275 Dune::H1Norm< GridPartType > h1norm( gridPart_ );
276 // calculate H1 - Norm
277 const double h1error = h1norm.distance( ugrid, solution );
278
279 // store values
280 std::vector<double> errors;
281 errors.push_back( l2error );
282 errors.push_back( h1error );
283
284 // submit error to the FEM EOC calculator
285 Dune::FemEoc :: setErrors( eocId_, errors);
286 }
287
288 //! return reference to discrete space
289 DiscreteSpaceType & space() { return space_; }
```

```
291 private:
292 //! set the dirichlet points to exact values
293 template< class EntityType, class DiscreteFunctionType >
294 bool boundaryTreatment( const EntityType &entity,
295                          const ProblemType& problem,
296                          DiscreteFunctionType &rhs,
297                          DiscreteFunctionType &solution )
298 {
375 }
376
377 //! solve the resulting linear system
378 void solve ( LaplaceOperatorType &laplace,
379              const DiscreteFunctionType &rhs,
380              DiscreteFunctionType &solution )
381 {
393 // create inverse operator (note reduction is not used here)
394 InverseOperatorType cg( laplace, reduction, solverEps );
395
396 // solve the system
397 cg( rhs, solution );
405 }
406
```

### 3 Solving the Poisson problem

```
407 //! subtract the mean value from the right hand side
408 void clearMeanValue ( DiscreteFunctionType &rhs ) const
409 {
410
423
424 }
425
426 //! set the exact mean value to the solution
427 template< class GridFunctionType >
428 void setExactMeanValue ( const GridFunctionType &ugrid, DiscreteFunctionType &
    solution ) const
429 {
430
463
464 }
465
466 protected :
467   HGridType& grid_ ;           // reference to grid, i.e. the hierarchical grid
468   GridPartType gridPart_ ;    // reference to grid part, i.e. the leaf grid
469   DiscreteSpaceType space_ ;  // the discrete function space
470   const ProblemType& problem_ ; // the problem data
471   int eocId_ ;                // id for FemEOC
472 };
```

The definition of the class `Algorithm` starts with some typedefs (not printed here) defining the grid type, the function spaces and discrete functions, the Laplace operator and a solver for linear systems. Furthermore, the constructor and the methods `operator()`, `finalize`, and `space` need to be defined. The methods `operator()` and `finalize` are evaluated by the `compute` function in this order every time a new numerical solution needs to be computed. The method `operator()` should be used for the implementation of the actual numerical scheme, while the constructor and the method `finalize` can be used for pre- respectively post-processing.

In this example, `operator()` initializes the Laplace operator  $A$  from equation (3.7) in line 224, the right hand side function  $\mathbf{b}$  in lines 227-246, and finally solves the linear equation system for the unknown discrete function  $\mathbf{u}$  in line 256. This line executes the private method `solve` initializing a CG Solver in line 394 and executing it in line 397.

After the solution step is executed, we compute the  $L^2$  and the  $H^1$  error between the exact and the discrete solution in the `finalize` method. The method `space` simply returns a reference to the instance of a discrete function space.

To change certain runtime parameters the user needs to edit the file `parameter` in the directory `dune-femhowto/examples/poisson/` or the file `parameter` in the direc-

### 3 Solving the Poisson problem

tory `dune-femhowto/examples/base/`. These parameter are processed by the `Dune::Parameter` singleton which can be accessed anywhere in the program.

The initial grid size can be manipulated by the `femhowto.startLevel` parameter indicating the number of refinement steps that are executed on the grid before the first numerical solution gets computed. The parameter `femhowto.eocSteps` then controls the number of subsequent EOC computations. The `compute` function makes use of the DUNE-FEM utility classes `FemEOC` and `FemTimer` which write out nicely formatted output information on the evolution of the EOC error and the elapsed time of computations to the files `eoc_main.tex` and `timer.out`.

#### 3.1.2 Assembling the Laplace operator

The assembling of the Laplace operator  $A$  and the right hand side  $\mathbf{b}$  from the equation (3.7) is done in the file `laplaceoperator.hh`. The important parts of the operator implementation are shown in listing 8. After the listing we are going to elaborate on certain concepts of DUNE-FEM like numerical integration that are used in these algorithms for the assembling of  $A$ .

##### Listing 8 (Excerpt of `dune-femhowto/examples/poisson/laplaceoperator.hh`)

```
21  template< class DiscreteFunction , class MatrixTraits >
22  class LaplaceOperator
23  : public Operator< typename DiscreteFunction::RangeFieldType ,
24                  typename DiscreteFunction::RangeFieldType ,
25                  DiscreteFunction ,
26                  DiscreteFunction > ,
27  public OEMSolver::PreconditionInterface
28  {
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114  public:
115    ///! apply the operator
116    virtual void operator() ( const DiscreteFunctionType &u ,
117                             DiscreteFunctionType &w ) const
118    {
119      systemMatrix().apply( u , w );
120    }
121
122
123
124
125
126
127
128
129  LinearOperatorType &systemMatrix () const
130  {
131    /// if stored sequence number it not equal to the one of the
132    /// dofManager (or space) then the grid has been changed
133    /// and matrix has to be assembled new
134    if( sequence_ != dofManager_.sequence() )
135      assemble();
```

### 3 Solving the Poisson problem

```
166
167     return linearOperator_;
168 }
169
170 /** \brief perform a grid walkthrough and assemble the global matrix */
171 void assemble () const
172 {
173     const DiscreteFunctionSpaceType &space = discreteFunctionSpace();
174
175     // reserve memory for matrix
176     linearOperator_.reserve();
177
178     // clear matrix
179     linearOperator_.clear();
180
181     // apply local matrix assembler on each element
182     typedef typename DiscreteFunctionSpaceType :: IteratorType IteratorType;
183     IteratorType end = space.end();
184     for(IteratorType it = space.begin(); it != end; ++it)
185     {
186         assembleLocal( *it );
187     }
188
189     // correct matrix due to boundary conditions
190     boundaryCorrectOnGrid();
191
192     // get grid sequence number from space (for adaptive runs)
193     sequence_ = dofManager_.sequence();
194 }
195
196 protected:
197 /** assemble local matrix for given entity
198 template< class EntityType >
199 void assembleLocal( const EntityType &entity ) const
200 {
201     // extract type of geometry from entity
202     typedef typename EntityType :: Geometry Geometry;
203
204     // assert that matrix is not build on ghost elements
205     assert( entity.partitionType() != GhostEntity );
206
207     // cache geometry of entity
208     const Geometry &geometry = entity.geometry();
209
210     // get local matrix from matrix object
211     LocalMatrixType localMatrix
212         = linearOperator_.localMatrix( entity, entity );
213
214     // get base function set
215     const BaseFunctionSetType &baseSet = localMatrix.domainBaseFunctionSet();
216
217     // get number of local base functions
218     const size_t numBaseFunctions = baseSet.numBaseFunctions();
219 }
```

### 3 Solving the Poisson problem

```
228
229 // create quadrature of appropriate order
230 QuadratureType quadrature( entity, 2 * (polynomialOrder - 1) );
231
232 // loop over all quadrature points
233 const size_t numQuadraturePoints = quadrature.nop();
234 for( size_t pt = 0; pt < numQuadraturePoints; ++pt )
235 {
236     // get local coordinate of quadrature point
237     const typename QuadratureType :: CoordinateType &x
238         = quadrature.point( pt );
239
240     // get jacobian inverse transposed
241     const typename Geometry :: Jacobian& inv
242         = geometry.jacobianInverseTransposed( x );
243
244     // extract type of diffusion coefficient from problem
245     typedef typename ProblemType :: DiffusionMatrixType DiffusionMatrixType;
246     DiffusionMatrixType K;
247
248     // evaluate diffusion matrix
249     problem().K( geometry.global( x ), K );
250
251     // for all base functions evaluate the gradient
252     // on quadrature point pt and apply jacobian inverse
253     baseSet.jacobianAll( quadrature[ pt ], inv, gradCache_ );
254
255     // apply diffusion tensor
256     for( size_t i = 0; i < numBaseFunctions; ++i )
257         K.mv( gradCache_[ i ][ 0 ], gradDiffusion_[ i ][ 0 ] );
258
259     // evaluate integration weight
260     weight_ = quadrature.weight( pt ) * geometry.integrationElement( x );
261
262     // add scalar product of gradients to local matrix
263     updateLocalMatrix( localMatrix );
264 }
265 }
266
267 //! add scalar product of cached gradients to local matrix
268 void updateLocalMatrix ( LocalMatrixType &localMatrix ) const
269 {
270     const size_t rows    = localMatrix.rows();
271     const size_t columns = localMatrix.columns();
272     for( size_t i = 0; i < rows; ++i )
273     {
274         for ( size_t j = 0; j < columns; ++j )
275         {
276             const RangeFieldType value
277                 = weight_ * (gradCache_[ i ][ 0 ] * gradDiffusion_[ j ][ 0 ]);
278             localMatrix.add( j, i, value );
279         }
280     }
```

### 3 Solving the Poisson problem

```
281     }

309     template< class EntityType >
310     void boundaryCorrectOnEntity ( const EntityType &entity ) const
311     {

328         // get slave dof structure (for parallel runs)
329         SlaveDofsType &slaveDofs = this->slaveDofs();
330         const int numSlaveDofs = slaveDofs.size();
331
332         // get local matrix from matrix object
333         LocalMatrixType localMatrix = linearOperator_.localMatrix( entity, entity );
334
335         // loop over all intersections and find dirichlet
336         // boundaries
337         const IntersectionIteratorType endit = gridPart.iend( entity );
338         for( IntersectionIteratorType it = gridPart.ibegin( entity );
339             it != endit ; ++it )
340         {

358             // iterate over face dofs and set unit row
359             for( ; faceIt != faceEndIt; ++faceIt )
360             {
361                 const int localDof = *faceIt;
362                 // clear the whole row
363                 localMatrix.clearRow( localDof );
364
365                 // set diagonal to 1
366                 localMatrix.set( localDof, localDof, 1 );
367
368                 // clear all other columns
369                 const int globalDof = dfSpace.mapToGlobal( entity, localDof );
370
371                 // cancel all slave dofs (for parallel runs)
372                 for( int i = 0; i < numSlaveDofs; ++i )
373                 {
374                     // slave dofs are canceled
375                     if( globalDof == slaveDofs[ i ] )
376                         localMatrix.set( localDof, localDof, 0 );
377                 }
378             }
379         }
380     }
381
382
383     // return slave dofs
384     static SlaveDofsType *getSlaveDofs ( const DiscreteFunctionSpaceType &space )
385     {
386         SlaveDofsKeyType key( space, space.mapper() );
387         return &(SlaveDofsProviderType :: getObject( key ));
388     }
389 }
```

### 3 Solving the Poisson problem

```

390 // return reference to slave dofs
391 SlaveDofsType &slaveDofs () const
392 {
393     slaveDofs_ -> rebuild();
394     return *slaveDofs_;
395 }
416 };

```

The class `LaplaceOperator` is derived from the `Dune::Operator` class. This is why we need to override the `operator()` method. In line 119 this method delegates the work to the `evaluate` method of the underlying system matrix. Note that the system matrix is accessed through a call to the `systemMatrix` method. This method first checks in line 164 whether the grid and therefore the container storing the function's DOFs were changed since the last assembling of the system matrix. This is done with help of the integer variable `dofManager_.sequence` which gets incremented each time the grid changes because of an adaptation or global refinement step. In case it does not equal the private member `sequence_`, the system matrix is invalid and gets updated by the `assemble` method. This method does a grid traversal and calls the method `assembleLocal` on each element in line 189.

#### Local degrees of freedom

We denote by  $I(T) := \{i \in \{1, \dots, N\} \mid \varphi_i|_T \neq 0\}$  the set of all DOF indices with corresponding basefunctions that have positive support on the grid entity  $T \in \mathcal{T}$ . Then the DOFs  $v_k^T := v_{\mu^T(k)}$  are called the *local degrees of freedom* on the grid entity  $T$  for  $k = 1, \dots, |I(T)|$ , where  $\mu^T : \{1, \dots, |I(T)|\} \rightarrow I(T)$  is an enumeration of  $I(T)$ . For the full concept of local degrees of freedom we refer to [5, Definition 20] (see also [6]).

With this notation the `LocalMatrix` that is retrieved in line 220 can now be read as a matrix  $L^T \in \mathbb{R}^{|I(T)| \times |I(T)|}$  with matrix entries

$$(L^T)_{i,j} = (A)_{\mu^T(i), \mu^T(j)}, \quad 1 \leq i, j \leq |I(T)|. \quad (3.11)$$

Note, that changes in the local matrix also affect the corresponding entries in the system matrix  $A$ .

In order to assemble this local matrix on every grid entity  $T$ , we need the base functions corresponding to the local degrees of freedom on the entity. In line 224 we retrieve a so-called `BaseFunctionSet` that consists of all base functions  $\varphi_{\mu^T(1)}, \dots, \varphi_{\mu^T(|I(T)|)}$  living

### 3 Solving the Poisson problem

on the reference element  $\hat{T}$ . Together with the reference mapping  $\Phi^T : \hat{T} \rightarrow T$  we get the wanted base functions  $\varphi_{\mu^T(i)} = \Phi(\varphi_{\mu^{\hat{T}}(i)})$  for  $i = 1, \dots, |I(T)|$ .

#### Quadrature

The DUNE-FEM quadrature object which is initialized in line 230 gives us quadrature points  $\hat{x}_q^T$  and associated weights  $w_q^T$  for  $q = 1, \dots, Q^T$ , such that we can compute the stiffness matrix entries

$$\begin{aligned} (S)_{ij} &= \int_T \nabla \varphi_i(x) \nabla \varphi_j(x) dx = \int_{\hat{T}} \nabla \hat{\varphi}_i(\hat{x}) \nabla \hat{\varphi}_j(\hat{x}) D\Phi^{-1}|_{\hat{x}} d\hat{x} \\ &\approx \sum_{q=1}^{Q^T} w_q^T \nabla \hat{\varphi}_i(\hat{x}_q^T) \nabla \hat{\varphi}_j(\hat{x}_q^T) D\Phi^{-1}|_{\hat{x}_q^T}. \end{aligned} \tag{3.12}$$

A complete definition is given in [5, Definition 14] (see also [6]). Note that the quadrature points are given in local coordinates on the reference element  $\hat{T}$ . In order to convert them into global coordinates, we had to use the `global` method from the `Geometry` class. For more details, see the DUNE-FEM documentation [2] at the module page [3] for quadratures.

#### 3.1.3 Boundary treatment

After the assembling of the stiffness matrix  $S$  (section 3.1.2), the matrix entries for boundary DOFs have to be updated according to (3.10). This is done in the class method `boundaryCorrectOnGrid` which is not printed in this document.

#### 3.1.4 Assembling the right hand side

The class `RightHandSideAssembler` in the file `laplaceoperator.hh` (not printed in this document) assembles the right hand side discrete function and is implemented analogously to the `LaplaceOperator` (see section 3.1.2). Instead of a local matrix it uses a local function.

## 3.2 Adaptation

To perform adaptive calculations, only very few things have to be added to the basic implementation as described in section 3.1. Most of the work is already done! We give a short overview over the involved files and highlight the differences to the basic implementation from section 3.1.

- `mainadaptive.hh`: Contains the `main` function. Mainly the same as the `main` function in `main.hh` (see section 3.1), but it calls the *adaptive* algorithm implemented in the file `algorithmadaptive.hh`.
- `algorithmadaptive.hh`: Contains a class `AdaptiveAlgorithm` which provides the main algorithm. The class `AdaptiveAlgorithm` is derived from the class `Algorithm` (see section 3.1.1) and simply provides an additional method `estimateAndMark`. This method marks grid elements for refining/coarsening based on the results of an error estimator (implemented in the file `estimator.hh`).
- `estimator.hh` contains a class `Estimator` which provides an a-posteriori error estimator for the numerical solution. The estimator is utilized by the adaptive scheme.

Furthermore, we want to emphasize that the classes defining the problem data (in file `problemdata.hh`, see section 3.1), the class `Algorithm` (in file `algorithm.hh`, see section 3.1.1) and the classes `LaplaceOperator` and `RightHandSideAssembler` (in file `laplaceoperator.hh`, see section 3.1.2) are used in the adaptive calculations without any changes in code.

**Important:** You have to use an adaptive grid (for example `AlbertaGrid` or `ALUGrid` with conforming refinement) for performing adaptive calculations. Non-adaptive grids (like the default(!) `YaspGrid`) will work, but result in nonadaptive calculations.

The program refines the grid until the error drops beneath the bound given by `femhowto.tolerance` in the `parameter` file.

## 3.3 Parallelization

The Makefile in the directory `examples/poisson` it produces two executables: `poisson` and `adaptive`. The first one is compiled with additional `$(DUNEMPICPPFLAGS)` as can be seen in line 18 of `Makefile.am` and therefore allows parallel execution on many processors, if the used grid implementation supports parallelization. At the moment this

### 3 Solving the Poisson problem

applies to `ALUSimplexGrid` and `ALUCubeGrid` for 3 dimensional problems and `YaspGrid` and `UGGrid` for 2 and 3 dimensional problems.

*Note:* The `YaspGrid` implementation does not include any routines for repartitioning of the grid, thus the initial domain decomposition is fixed during the whole computation. For `ALUSimplexGrid` and `ALUCubeGrid` the partitioning can be adapted by calling the method `loadBalance` of the grid. This only has to be used in adaptive parallel computations.

*Note:* Adaptive and parallel computations of the Poisson problem are currently not possible with the implementation of the Poisson solver, because of the following reasons:

- `YaspGrid` is not capable of local grid adaptation,
- although `UG` would be capable of local adaptation in parallel the `DUNE-GRID` interface implementation `UGGrid` does not support this feature completely yet, and
- `ALUSimplexGrid` and `ALUCubeGrid` produce *hanging nodes* during local refinement and the Poisson solver is not able to handle those. For discretizations with discontinuous basis functions this is not a problem at all, see Chapter 4.

Parallel computations for this problem without local grid adaptivity are very well possible.

In order to start the program `poisson` on two processors, type

```
mpiexec -n 2 ./poisson
```

at the command line.

When the grid gets initialized in line 80, for each process only a partition of the grid is returned. The partitioning is done automatically. At some steps of the algorithm, however, the processes need to interact and exchange their computed results. In order to understand how `DUNE` handles communication at the boundary of grid partitions, the reader should be familiar with concepts like ghost cells and overlap cells. This is, for example, described in [5, Section 3.2]. An other introduction can be found in the chapter on parallelization of the `dune-grid-howto` [4]. Most times the `DUNE-FEM CommunicationManager` handles code parts where communication is needed automatically, for example, the call `discreteFunction.communicate()` on a discrete function object does the appropriate communication according to the discrete function space. But sometimes things get more complicated.

We want to explain one of these complicated communication routines and therefore have a deeper look at the `boundaryCorrectOnEntity` method in listing 8. When the system

### 3 Solving the Poisson problem

matrix  $A$  gets assembled, only the DOFs on the boundary between two partitions, i.e. the DOFs on the border of a partition need to be communicated. After the assembling of the stiffness matrix also the DOFs on the overlap respectively in ghost cells are set on each partition.

After the partitioning of the grid, some vertices lie on more than one grid partition. For the final discrete function living on the entire grid, the contributions of the different processes to the DOFs corresponding to these vertices are added up. For Dirichlet boundaries this leads to problems, because a global evaluation of the discrete function at the vertices would result in twice the expected value. Therefore, in lines 369-377 the lines of the system matrix corresponding to these DOFs are set to the zero vector for all but one so-called “master” process. The selection of the matrix lines that need to be erased is quite easy because of the `SlaveDof` concept. All DOFs that are duplicated on a grid partition with a lesser process number than the current process, are automatically marked as `SlaveDofs`. In lines 383-395 we can see how a `SlaveDof` container can be constructed for discrete function spaces on each grid partition with the help of the class `SlaveDofsProviderType`, i.e. a `SingletonList`, and how such a container is updated by the `rebuild` method.

## 4 An LDG solver for Advection-Diffusion Equations

This chapter introduces the `Pass` concept in DUNE-FEM and explains the steps that are necessary in order to implement the Local Discontinuous Galerkin solver using the `LocalDGPass` class.

### 4.1 Advection-Diffusion Equation

The example problem is a scalar advection-diffusion equation on  $\Omega = [0, 1]^3$

$$\begin{aligned} \partial_t u + \nabla \cdot (\mathbf{a}u) - \varepsilon \Delta u &= 0 && \text{in } \Omega \times \mathbf{T} \\ &= g_D && \text{on } \partial\Omega \times \mathbf{T} \\ u(0, \cdot) &= u_0 && \text{in } \Omega \times \{0\}, \end{aligned} \quad (4.1)$$

where  $u$  belongs to a function space  $V$  and  $\mathbf{a} := (0.8, 0.6, 0)^t$ . An exact solution is specified by

$$u(x, t) = \sum_{i=1}^2 \exp(-\varepsilon t \pi^2 |c^i|) \left( \prod_{j=1}^3 \tilde{c}_j^i \cos(c_j^i \pi (x_j - \mathbf{a}_j t)) + \tilde{c}_j^i \sin(c_j^i \pi (x_j - \mathbf{a}_j t)) \right), \quad (4.2)$$

where

$$c^1 := (2, 1, 1.3)^t \qquad c^2 := (0.7, 0.5, 0.1) \quad (4.3)$$

$$\hat{c}^1 := (0.8, 0.4, -0.4)^t \qquad \hat{c}^2 := (0.2, 0.1, 0.2) \quad (4.4)$$

$$\tilde{c}^1 := (0.6, 1.2, 0.1)^t \qquad \tilde{c}^2 := (0.9, 0.3, -0.3). \quad (4.5)$$

Therefore the initial and boundary data functions are defined by

$$u(x, 0) = u_0(x) \qquad u|_{\partial\Omega}(x, t) = g_D(x, t). \quad (4.6)$$

Note that the problem is also implemented for two dimensions by projecting everything onto the first two coordinates. The discretization of the problem is done as described in [8, Chapter 4].

#### 4 An LDG solver for Advection-Diffusion Equations

The LDG Ansatz uses auxiliary functions  $u_0, u_2 \in V$  and  $u_1 \in V^3$

$$\begin{aligned} u_0 &= u \\ u_1 &= -\sqrt{\varepsilon} \nabla u_0 \\ u_2 &= -\nabla \cdot (\mathbf{a}u_0 + \sqrt{\varepsilon}u_1) \\ \partial_t u &= u_2. \end{aligned} \tag{4.7}$$

Now  $u_0$  and  $u_1$  can be projected onto the Discontinuous Galerkin function spaces  $V_h := \{\varphi_1, \dots, \varphi_n\}$  resp.  $[V_h]^3 = \{\psi_1, \dots, \psi_{n^3}\}$  with discrete domain in space. The equations in (4.7) can then be rewritten as a variational formulation

$$\int_T u_1 \psi = \int_T \underbrace{\sqrt{\varepsilon}u_0}_{\tilde{f}_1(u_0)} \nabla \cdot \psi - \int_{\partial T} \underbrace{\sqrt{\varepsilon}u_0}_{\tilde{f}_1(u_0)} \psi \cdot n \quad \forall \psi \in [V_h]^3 \tag{4.8}$$

$$\int_T u_2 \varphi = \int_T \underbrace{(\mathbf{a}u_0 + \sqrt{\varepsilon}u_1)}_{\tilde{f}_2(u_0, u_1)} \cdot \nabla \varphi - \int_{\partial T} \underbrace{(\mathbf{a}u_0 + \sqrt{\varepsilon}u_1)}_{\tilde{f}_2(u_0, u_1)} \varphi \cdot n \quad \forall \varphi \in V_h \tag{4.9}$$

for every grid cell  $T$ . To complete the discretization in the space domain, the numerical fluxes between the cell interfaces must be defined to approximate  $\tilde{f}_1$  and  $\tilde{f}_2$  defined in (4.8) and (4.9). In this example we choose

$$\tilde{f}_{1,h}(u_0) = \begin{cases} \sqrt{\varepsilon}g_D(x) & \text{on } \partial\Omega \\ \sqrt{\varepsilon}\{u_0\} & \text{else} \end{cases} \tag{4.10}$$

$$\tilde{f}_{2,h}(u_0, u_1) = \begin{cases} \mathbf{a}g_D(x) + \sqrt{\varepsilon}u_1(x^-) & \text{on } \partial\Omega \\ w(\mathbf{a}, u_0) + \sqrt{\varepsilon}\{u_1\} & \text{else} \end{cases} \tag{4.11}$$

where

$$\{u\} := \frac{1}{2}(u(x^+) + u(x^-)) \quad \text{and} \quad w(\mathbf{a}, u_0) := \begin{cases} \mathbf{a}u_0(x^+) & \text{if } \mathbf{a} \cdot n \leq 0 \\ \mathbf{a}u_0(x^-) & \text{if } \mathbf{a} \cdot n > 0 \end{cases} \tag{4.12}$$

define the *mean value* function and the *upwind* function over the cell interfaces. Now an operator  $L : V_h \rightarrow V_h$  can be defined as a concatenation  $L := \Pi \circ L_2 \circ L_1$  of the two operators

$$\begin{aligned} L_1 : V_h &\rightarrow V_h \times V_h^3 & (u_0) &\mapsto (u_0, u_1) \\ L_2 : V_h \times V_h^3 &\rightarrow V_h \times V_h^3 \times V_h & (u_0, u_1) &\mapsto (u_0, u_1, u_2) \end{aligned} \tag{4.13}$$

and the projection operator  $\Pi$  that projects to the last component. The concatenation  $\Pi \circ L_2 \circ L_1$  can later be implemented by the `Dune::Pass` class. This allows to shorten the problem formulation to

$$\partial_t u + L[u] = 0. \tag{4.14}$$

This ODE in the time domain can be solved with standard ODE solvers requiring evaluations of the discrete operator  $L$  in each time step. The so-called “methods of lines” is further examined in [8, Chapter 4.3].

## 4.2 Implementation overview

The source code of the implementation described in this section is located in the directory `dune-femhowto/examples/localdg`. It consists of 6 source files:

- `dgtest.cc` is the source file for the main program and is explained in the next section. It utilizes the `../common/baseevolution.hh` header file for EOC handling and time loop routines which was described in the chapter on the finite volume transport example. (Ch. 2)
- `problem.hh` contains the class `U0` which defines the exact solution  $u$  (4.2), the boundary data function  $g_D$  and the initial data function  $u_0$  both defined in (4.6). Those functions can be accessed through the methods

```
void evaluate(const DomainType & arg, RangeType& res) const
```

for  $u$  resp.  $g_D$ , and

```
void evaluate(const DomainType & arg, double t, RangeType& res) const
```

for  $u_0$ . The class `U0` is derived from the virtual interface class `ProblemInterface` allowing definition of different problems the user can select at runtime by specifying a parameter on the command line or in a parameter file.

- `models.hh` contains the class `AdvectionDiffusionModel` implementing the model data given in equation (4.7) and the class `UpwindFlux` which is just a helper class for the upwind flux given in (4.12).
- `discretemodels.hh` contains two classes `AdvDiffDModel1` and `AdvDiffDModel2` that describe the two passes of the LDG implementation, i.e. the terms and fluxes given by (4.8–4.11). For further details on these classes see section 4.5.
- `advectdiff.hh` provides the LDG Operator `DGAdvectionDiffusionOperator` constructed out of the models given in `discretemodels.hh`. For further details, refer to section 4.5.

### 4.3 Main Loop

The main loop can be found in the file `dgtest.cc` (9). It uses the `evolve` function of the source file `baseevolution.hh`. This function provides an interface for a generic evolution scheme with routines for EOC data computation and data visualization. For details on this function refer to chapter ref TODO.

In order to customize the `evolve` function for our Local Discontinuous Galerkin problem, we only need to declare a `Stepper` class that we give to the `evolve` function as a template parameter. An explanation of the most important lines in the definition of this class follows in section 10.

#### Listing 9 (`./../examples/localdg/dgtest.cc`)

```

2 // include host specific macros set by configure script
3 #include <config.h>
4
5 // include std libs
6 #include <iostream>
7 #include <string>
8
9 // Dune includes
10 #include <dune/fem/misc/l2error.hh>
11 #include <dune/fem/operator/projection/l2projection.hh>
12 #include <dune/fem/gridpart/gridpart.hh>
13 #include <dune/fem/solver/odesolver.hh>
14
15 // include local header files
16 #include <dune/fem-howto/baseevolution.hh>
17 #include "models.hh"
18
19 // approximation order
20 const int order = POLORDER;
21 const int rkSteps = POLORDER + 1;
22
23 template <class GridImp, class InitialDataType>           LST1S
24 struct StepperTraits {
25     // type of Grid
26     typedef GridImp                                     GridType;
27
28     // Choose a suitable GridView
29     typedef Dune::DGAdaptiveLeafGridPart< GridType >
30         GridPartType;
31
32     // An analytical version of our model
33     typedef AdvectionDiffusionModel< GridPartType, InitialDataType >
34         ModelType;
35
36     // The flux for the discretization of advection terms
37     typedef UpwindFlux< ModelType >                   FluxType;

```

## 4 An LDG solver for Advection-Diffusion Equations

```

36
37 // The DG Operator (using 2 Passes)
38 typedef Dune::DGAdvectionDiffusionOperator< ModelType, UpwindFlux,
39                                             order >                               DgType;
40
41 // The discrete function for the unknown solution is defined in the DgOperator
42 typedef typename DgType :: DestinationType
43     DiscreteFunctionType;
44 // ... as well as the Space type
45 typedef typename DgType :: SpaceType
46     DiscreteSpaceType;
47
48 // The ODE Solvers
49
50
51
52
53 };
54
55 template <class GridImp, class InitialDataType>                               LST1S
56 struct Stepper : public AlgorithmBase< StepperTraits< GridImp, InitialDataType > >
57 {
58     // my traits class
59     typedef StepperTraits< GridImp, InitialDataType > Traits ;
60
61     // my base class
62     typedef AlgorithmBase < Traits > BaseType;
63
64     // type of Grid
65     typedef typename Traits :: GridType                                     GridType;
66
67     // Choose a suitable GridView
68     typedef typename Traits :: GridPartType                               GridPartType;
69
70     // An analytical version of our model
71     typedef typename Traits :: ModelType                                  ModelType;
72
73     // The flux for the discretization of advection terms
74     typedef typename Traits :: FluxType                                   FluxType;
75
76     // The DG Operator (using 2 Passes)
77     typedef typename Traits :: DgType                                     DgType;
78
79     // The discrete function for the unknown solution is defined in the DgOperator
80     typedef typename Traits :: DiscreteFunctionType                       DiscreteFunctionType;
81

```

#### 4 An LDG solver for Advection-Diffusion Equations

```

82 // ... as well as the Space type
83 typedef typename Traits :: DiscreteSpaceType      DiscreteSpaceType;
84
85 // The ODE Solvers
86 typedef typename Traits :: OdeSolverInterfaceType OdeSolverInterfaceType;
87 typedef typename Traits :: ImplicitOdeSolverType  ImplicitOdeSolverType;
88 typedef typename Traits :: ExplicitOdeSolverType  ExplicitOdeSolverType;
89
90 typedef typename BaseType :: TimeProviderType     TimeProviderType;
91
92 using BaseType :: grid_;
93
94 Stepper(GridType& grid, const InitialDataType& problem) :
95     BaseType ( grid ),
96     problem_(problem),
97     model_(problem_),
98     convectionFlux_(model_),
99     dgOperator_(grid_, convectionFlux_),
100     eocId_( Dune::FemEoc::addEntry(std::string("$L^2$-error")) ),
101     odeSolver_(0)
102 {
103 }
104
105 ~Stepper()
106 {
107     delete odeSolver_;
108     odeSolver_ = 0;
109 }
110
111 // before first step, do data initialization
112 void initializeStep(TimeProviderType& tp, DiscreteFunctionType& u)
113 {
114     // choice of explicit or implicit ode solver
115     static const std::string odeSolver []
116         = { "explicit", "implicit" };
117     int odeSolve = Dune::Parameter::getEnum( "femhowto.odesolver", odeSolver, 0 );
118
119     // create implicit or explicit ode solver
120     if( odeSolve == 0 )
121         odeSolver_ = new ExplicitOdeSolverType(dgOperator_, tp, rkSteps);
122     else
123         odeSolver_ = new ImplicitOdeSolverType(dgOperator_, tp, rkSteps);
124
125     assert( odeSolver_ );
126
127     Dune::L2Projection< double, double,
128         InitialDataType, DiscreteFunctionType > l2pro;
129     l2pro(problem_, u);
130
131     // ode.initialize applies the DG Operator once to get an estimate for dt.
132     //odeSolver->initialize(u);
133 }
134

```

LST1E

## 4 An LDG solver for Advection-Diffusion Equations

```

135 void initializeOdeSolver( DiscreteFunctionType& u )
136 {
137     odeSolver_ -> initialize(u);
138 }
139
140 // solve ODE for one time step
141 void step(TimeProviderType& tp, DiscreteFunctionType& u)
142 {
143     assert(odeSolver_);
144     odeSolver_ -> solve(u);
145 }
146
147 // after last step, do EOC calculation
148 void finalizeStep(TimeProviderType& tp, DiscreteFunctionType& u)
149 {
150     Dune::L2Error<DiscreteFunctionType> L2err;
151     // Compute L2 error of discretized solution ...
152     Dune::FieldVector<double, ModelType::dimRange> error;
153     error = L2err.norm(problem_, u, tp.time());
154     // ... and print the statistics out to the eocOutputPath file
155     Dune::FemEoc::setErrors(eocId_, error.two_norm());
156
157     // delete ode solver
158     delete odeSolver_;
159     odeSolver_ = 0;
160 }
161
162 private:
163     // InitialDataType is a Dune::Operator that evaluates to $u_0$ and also has a
164     // method that gives you the exact solution.
165     const InitialDataType &problem_;
166     ModelType model_;
167     // Initial flux for advection discretization (UpwindFlux)
168     FluxType convectionFlux_;
169     DgType dgOperator_;
170     const unsigned int eocId_;
171     OdeSolverInterfaceType *odeSolver_;
172 };
173
174 /**
175  * @brief main function for the LocalDG Advection-Diffusion application
176  *
177  * \c main starts the Simulation of an advection-diffusion pde with
178  * the localdg method with EOC analysis and visual output to grape, paraview or
179  * gnuplot.
180  * \attention The localdg implementation uses the \c Dune::Pass
181  * concept.
182  *
183  * @param argc number of arguments from command line
184  * @param argv array of arguments from command line
185  * @param envp array of environmental variables
186  * @return 0 we don't program bugs. :)
187  */
188 int main(int argc, char ** argv, char ** envp) {

```

#### 4 An LDG solver for Advection-Diffusion Equations

```

188 typedef Dune::GridSelector :: GridType GridType;
189
190 /* Initialize MPI (always do this even if you are not using MPI) */
191 Dune::MPIManager :: initialize( argc, argv );
192
193 try {
194
195 // *** Initialization
196 Dune::Parameter::append(argc,argv);
197 if (argc == 2) {
198     Dune::Parameter::append(argv[1]);
199 }
200 else
201 {
202     Dune::Parameter::append("parameter");
203 }
204
205 // ProblemType is a Dune::Function that evaluates to $u_0$ and also has a
206 // method that gives you the exact solution.
207 typedef Dune::U0< GridType > ProblemType;
208 ProblemType problem;
209
210 // Note to me: problem description is for FemEOC
211 Dune::GridPtr<GridType> gridptr = initialize< GridType >( std::string("Local_DG_
        scheme") +
212                                     problem.description() );
213 // get grid reference
214 GridType & grid = *gridptr;
215
216 /*****
217  * EOC Loop
218  *****/
219 Stepper<GridType, ProblemType :: BaseType> stepper(grid, problem);
220
221 compute(stepper);
222
223 Dune::Parameter::write("parameter.log");
224
225 }
226 catch (Dune::Exception &e) {
227     std::cerr << e << std::endl;
228     return 1;
229 } catch (...) {
230     std::cerr << "Generic_exception!" << std::endl;
231     return 2;
232 }
233
234 return 0;
235 }

```

Note, that the entire main function is embedded into a `try` block such that exceptions can be caught in lines [226-232](#). This is recommended in order to get useful error messages

in case something goes wrong the runtime.

In lines 196-203 the singleton `Dune::Parameter` is initialized for later use. It manages all parameters the user can set after the compilation of the program either via the command line or a parameter file. The default name for the parameter file is “parameter” (as defined in line 202) and can also be changed through the command line.

Among other parameters the user needs to specify the name of a DGF file from the function `initialize` in line 211 constructs a `GridPtr`. This function is also included from the header file `base.hh`. After we get the grid, we can define the `Stepper` in line 219 and pass it to the `evolve` function in line 221 which computes the entire evolution scheme with additional EOC handling.

## 4.4 Stepper control class

The `Stepper` needs to provide four methods `space`, `initialize`, `step` and `finalize`. The first one simply returns a reference of the discrete function space which needs to be defined in the `Stepper` class, because the more important methods `initialize` and `step` need to be aware of the discrete solution. They are called inside the `evolve` function at the beginning of the evolution scheme respectively at each time step.

### Listing 10 (Stepper class from `../examples/localdg/dgtest.cc`)

```

23 template <class GridImp, class InitialDataType>
24 struct StepperTraits {
25     // type of Grid
26     typedef GridImp                                     GridType;
27
28     // Choose a suitable GridView
29     typedef Dune::DGAdaptiveLeafGridPart< GridType >
        GridPartType;
30
31     // An analytical version of our model
32     typedef AdvectionDiffusionModel< GridPartType, InitialDataType > ModelType;
33
34     // The flux for the discretization of advection terms
35     typedef UpwindFlux< ModelType > FluxType;
36
37     // The DG Operator (using 2 Passes)
38     typedef Dune::DGAdvectionDiffusionOperator< ModelType, UpwindFlux,
        order > DgType;
39
40
41
42
43
44
45
46 // The ODE Solvers
47 typedef DuneODE :: OdeSolverInterface< DiscreteFunctionType >
        OdeSolverInterfaceType;

```

## 4 An LDG solver for Advection-Diffusion Equations

```

48 typedef DuneODE :: ImplicitOdeSolver < DiscreteFunctionType >
    ImplicitOdeSolverType;
49 typedef DuneODE :: ExplicitOdeSolver < DiscreteFunctionType >
    ExplicitOdeSolverType;
50
51 // type of restriction/prolongation projection for adaptive simulations
52 typedef Dune :: RestrictProlongDefault < DiscreteFunctionType >
    RestrictionProlongationType;
53 };
54
55 template <class GridImp, class InitialDataType >
56 struct Stepper : public AlgorithmBase < StepperTraits < GridImp, InitialDataType > >
57 {
58     // my traits class
59     typedef StepperTraits < GridImp, InitialDataType > Traits ;
60
61     // my base class
62     typedef AlgorithmBase < Traits > BaseType;
63
64     // type of Grid
65     typedef typename Traits :: GridType                GridType;
66
67     // Choose a suitable GridView
68     typedef typename Traits :: GridPartType            GridPartType;
69
70     // An analytical version of our model
71     typedef typename Traits :: ModelType              ModelType;
72
73     // The flux for the discretization of advection terms
74     typedef typename Traits :: FluxType                FluxType;
75
76     // The DG Operator (using 2 Passes)
77     typedef typename Traits :: DgType                  DgType;
78
79     // The discrete function for the unknown solution is defined in the DgOperator
80     typedef typename Traits :: DiscreteFunctionType    DiscreteFunctionType;
81
82     // ... as well as the Space type
83     typedef typename Traits :: DiscreteSpaceType      DiscreteSpaceType;
84
85     // The ODE Solvers
86     typedef typename Traits :: OdeSolverInterfaceType  OdeSolverInterfaceType;
87     typedef typename Traits :: ImplicitOdeSolverType   ImplicitOdeSolverType;
88     typedef typename Traits :: ExplicitOdeSolverType   ExplicitOdeSolverType;
89
90     typedef typename BaseType :: TimeProviderType      TimeProviderType;
91
92     using BaseType :: grid_;
93
94     Stepper(GridType& grid, const InitialDataType& problem) :
95         BaseType ( grid ),
96         problem_(problem),
97         model_(problem_),

```

## 4 An LDG solver for Advection-Diffusion Equations

```
98     convectionFlux_(model_),
99     dgOperator_(grid_, convectionFlux_),
100     eocId_( Dune::FemEoc::addEntry(std::string("$L^2$error")) ),
101     odeSolver_(0)
102   {
103   }
```

At the time step 0 the discrete function  $U$  needs to get initialised with data from the initial data function  $u_0$ . This is done in lines 127-129 by a projection of the analytical function  $u_0$  onto the discrete function space with help of the utility class `L2Projection`. Furthermore, in lines 121 and 123 two ODE solver are defined, an implicit and an explicit Runge-Kutta solver, between which the user can switch at runtime. The arguments for the constructor of the solver include

- the `Dune::Operator` `dg` which is the discrete implementation of operator  $L$  in equation (4.14),
- the time provider `tp` that manages the current time and time step length that is to be used for the next computation of the ODE solver and
- the order of the Runge-Kutta solver `rkSteps`.

Each time the ODE solver calls one of its methods `initialize` or `solve` (see lines 137, 144), the operator `dg` is evaluated and afterwards the solver asks the operator for a new time step estimate by calling the operator's method `timeStepEstimate` which returns an estimate for the next time step length. The estimate must be chosen such that the operator is stable under the forward Euler integration. The ODE solver afterwards weights the estimate with a factor which is specific to the chosen ODE integration method. In addition, the user has the possibility to multiply a factor to the time step estimate via the `fem.timeprovider.factor` parameter. For details on the time step length estimation refer to the `TimeProvider` section in the DUNE-FEM documentation.

The method `finalize` gets called at the end of the evolution scheme and can be used for formatting output and computation of EOC data. In this example the DUNE-FEM class `FemTimer` is used for this purpose.

The next section is supposed to shed light on the construction of the discrete space operator `dg`, the main object of the Local DG solver.

## 4.5 Setting up an LDGPass

In order to set up the operator  $L$  as it is defined in (4.13) the DUNE-FEM Pass concept is used.

The passes  $L_1$  and  $L_2$  are implemented as `LocalDGPass` instances. Each `LocalDGPass` solves an equation of the form

$$v + \operatorname{div}(f(x, u)) + A(x, u)\nabla u = S(x, u) \quad \text{in } \Omega. \quad (4.15)$$

with the argument  $u$  and the computed solution  $v$ . Both required passes (see (4.7)) are in this form. In the weak formulation of equation (4.15) we identify methods that a model needs to provide to a `LocalDGPass` instance. These are `analyticalFlux`, `source`, `numericalFlux` and `boundaryFlux` shown in equation `refeq:fourmethods`.

$$\int_T v\varphi = - \int_{\partial T} \underbrace{\left( f(x, u) + \tilde{A}(x, u)[u] \cdot n \right)}_{\substack{\text{numericalFlux} \\ \text{boundaryFlux}}} \varphi + \int_T \overbrace{f(x, u)}^{\text{analyticalFlux}} \cdot \nabla \varphi + \int_T \underbrace{(S - A(x, u)\nabla u)}_{\text{source}} \varphi. \quad (4.16)$$

The header file `advectdiff.hh` (see listing 12) defines the two models that implement these four methods. Each of these methods describe the discrete version of its corresponding term in the variational formulation (4.16). For the second pass, for example, in equation (4.9) the `analyticalFlux` method represents  $f_2(u_0, u_1)$ , whereas in equation (4.11) `numericalFlux` and `boundaryFlux` represent  $\tilde{f}_{2,h}(u_0, u_1)$ . `numericalFlux` implements the flux on interfaces between inner cells and the `boundaryFlux` on the boundary domain. The second pass has no source term, which is why the `source` needs not to be implemented. The following listing 11 shows the actual implementation of the second pass, with some explanations on the code given in the next paragraph.

### Listing 11 (`AdvDiffDModel2` in `../examples/localdg/discretemodels.hh`)

```

267 // Discrete Model for Pass2
268 template <class Model, class NumFlux, int polOrd, int passUid, int passGradId>
269 class AdvDiffDModel2 :
270     public DGDDiscreteModelDefaultWithInsideOutside
271     <AdvDiffTraits2<Model, NumFlux, polOrd, passUid, passGradId >,
272         passUid, passGradId>
273 {
274     typedef DGDDiscreteModelDefaultWithInsideOutside
275         < AdvDiffTraits2< Model, NumFlux, polOrd, passUid, passGradId >,
276         passUid, passGradId >
277         BaseType;

```

#### 4 An LDG solver for Advection-Diffusion Equations

```

278     using BaseType :: numericalFlux;
279     using BaseType :: boundaryFlux;
280
281     // These type definitions allow a convenient access to arguments of pass.
282     Int2Type<passUIId>    uVar;
283     Int2Type<passGradId> sigmaVar;
284 public:
285     typedef AdvDiffTraits2< Model, NumFlux, polOrd, passUIId,
286                           passGradId >                               Traits;

305     AdvDiffDModel2(const Model& mod, const NumFlux& numf) :
306         model_(mod),
307         numflux_(numf),
308         penalty_(Parameter::getValue<double>("femhowto.penalty")),
309         // Set CFL number for penalty term (compare diffusion in first pass)
310         cflDifflinv_(2.*(polOrd+1.))
311     {}
312
313     bool hasSource() const { return false; }
314     bool hasFlux() const { return true; }

329     template <class ArgumentTuple>
330     double numericalFlux(const Intersection& it,
331                         double time, const FaceLocalCoordinate& x,
332                         const ArgumentTuple& uLeft,
333                         const ArgumentTuple& uRight,
334                         RangeType& gLeft,
335                         RangeType& gRight)
336     {
337         const DomainType normal = it.integrationOuterNormal(x);
338
339
340         /*****
341          * Advection
342          *****/
343         // delegated to numflux_
344         double wave = numflux_.
345             numericalFlux(it, time, x, uLeft[uVar], uRight[uVar], gLeft, gRight);
346
347         /*****
348          * Diffusion
349          *****/
350         JacobianRangeType diffmatrix;
351         RangeType diffflux(0.);
352         /* Central differences */
353         model_.
354             diffusion2(this->inside(), time, it.geometryInInside().global(x),
355                       uLeft[uVar], uLeft[sigmaVar], diffmatrix);
356         diffmatrix.umv(normal, diffflux);
357         model_.
358             diffusion2(this->outside(), time, it.geometryInOutside().global(x),
359                       uRight[uVar], uRight[sigmaVar], diffmatrix);
360         diffmatrix.umv(normal, diffflux);

```

## 4 An LDG solver for Advection-Diffusion Equations

```

361     diffflux*=0.5;
362
363     // add penalty term ( enVolume() is available since we derive from
364     //     DGDiscreteModelDefaultWithInsideOutside )
365     double factor = penalty_ * model_.diffusionCoefficient() *
366                 normal.two_norm2() /
367                 ( 0.5 * ( this->enVolume()+this->nbVolume() ) );
368
369     RangeType jump( uLeft[uVar] );
370     jump -= uRight[uVar];
371     diffflux.axy(factor, jump);
372
373     gLeft  += diffflux;
374     gRight += diffflux;
375     return std::max( wave, factor*cflDiffinv_ );
376 }
377
378 /**
379  * @brief same as numericalFlux() but for fluxes over boundary interfaces
380  */
381 template <class ArgumentTuple>
382 double boundaryFlux(const Intersection& it,
383                   const double time,
384                   const FaceLocalCoordinate& x,
385                   const ArgumentTuple& uLeft,
386                   RangeType& gLeft)
387
429
430 /**
431  * @brief analytical flux function $f_2(u_0, u_1)$
432  */
433 template <class ArgumentTuple>
434 void analyticalFlux( const EntityType& en,
435                   const double time,
436                   const LocalCoordinate& x,
437                   const ArgumentTuple& u, JacobianRangeType& f )
438 {
439     /*****
440     * Advection
441     *****/
442     model_.advection(en,time, x, u[uVar],f);
443     /*****
444     * Diffusion
445     *****/
446     JacobianRangeType diffmatrix;
447     model_.diffusion2(en, time, x, u[uVar], u[sigmaVar], diffmatrix);
448     f += diffmatrix;
449 }
450 private:
451     const Model    &model_;
452     const NumFlux  &numflux_;
453     const double   penalty_;
454     const double   cflDiffinv_;

```

455 };

Note that in line 313 the model notifies the `LocalDGPass` that there is no source term, and therefore the `source` method is never called and does not need to be implemented. All the implemented fluxes have a parameter of type `ArgumentTuple`. This is the argument type for the passes' flux functions. In the case of the second pass  $L_2$  this type refers to the product function space  $V_h \times V_h^3$  (see (4.13)). In order to access the correct component of the argument, one can use the `Int2Type` helper class: Every `Pass` get a unique id after it is defined (see line 37) and the `Int2Type` class can convert this unique id to a unique type as done in lines 282 and 283. These types then allow `ArgumentTuples` to be used like arrays, as in line 353 e.g., where `uLeft[u0Var]` returns  $u_0(x^-)$  and `uLeft[u1Var]` returns  $u_1(x^-)$ .

With models defined for both passes the implementation of the LDG operator becomes quite simple. The implementation of the operator can be found in the file `advectdiff.hh` and excerpts of this file are shown in listing 12.

#### Listing 12 (./../examples/localdg/advectdiff.hh)

```

30 template <class Model,template<class M> class NumFlux,int polOrd >
31 class DGAdvectionDiffusionOperator :
32   public SpaceOperatorInterface<
33     typename PassTraits<Model, Model::Traits::dimRange, polOrd>::DestinationType >
34   {
35 public:
36   // Id's for the three Passes (including StartPass)
37   enum PassIdType { u, gradPass, advectPass };
38
39   static const int dimRange = Model::dimRange;
40   static const int dimDomain = Model::Traits::dimDomain;
41
42   typedef NumFlux< Model > NumFluxType;
43
44   /*****
45    * Declare Models for Passes 1&2
46    *****/
47   // Pass 1 Model (gradient)
48   typedef AdvDiffDModel1< Model, NumFluxType, polOrd, u >
49     DiscreteModel1Type;
49   // Pass 2 Model (advection)
50   typedef AdvDiffDModel2< Model, NumFluxType, polOrd, u, gradPass >
51     DiscreteModel2Type;
52
53   typedef typename DiscreteModel1Type :: Traits Traits1;
54   typedef typename DiscreteModel2Type :: Traits Traits2;
55
56   typedef typename Model :: Traits :: GridType GridType;

```

#### 4 An LDG solver for Advection-Diffusion Equations

```

56
57 typedef typename Traits2 :: DomainType           DomainType;
58 typedef typename Traits2 :: DiscreteFunctionType DiscreteFunctionType;
59
60 typedef typename Traits1 :: DiscreteFunctionSpaceType Space1Type;
61 typedef typename Traits2 :: DiscreteFunctionSpaceType Space2Type;
62 typedef typename Traits1 :: DestinationType
63     Destination1Type;
64 typedef typename Traits2 :: DestinationType
65     Destination2Type;
66
67 typedef Destination2Type
68     DestinationType;
69
70 typedef Space2Type           SpaceType;
71
72 typedef typename Traits1 :: GridPartType           GridPartType;
73
74
75 /*****
76  * Join the Passes 0-2
77  *
78  *****/
79 typedef StartPass< DiscreteFunction2Type , u >           Pass0Type;
80 typedef LocalDGPass< DiscreteModel1Type , Pass0Type , gradPass > Pass1Type;
81 typedef LocalDGPass< DiscreteModel2Type , Pass1Type , advectPass > Pass2Type;
82
83
84 DGAvectionDiffusionOperator(GridType& grid,
85                             const NumFluxType& numf) :
86     grid_(grid),
87     verbose_(Parameter :: getValue<int>("femhowto.verbose")),
88     model_(numf.model()),
89     numflux_(numf),
90     gridPart_(grid_),
91     space1_(gridPart_),
92     space2_(gridPart_),
93     problem1_(model_ , numflux_),
94     problem2_(model_ , numflux_),
95     pass1_(problem1_ , pass0_ , space1_),
96     pass2_(problem2_ , pass1_ , space2_)
97 {
98 }
99
100
101 /**
102  * @brief evaluates the operator (pass)
103  *
104  * @param arg the operator argument
105  * @param dest the result of the operator evaluation
106  */
107 void operator()(const DestinationType& arg, DestinationType& dest) const
108 {
109     pass2_(arg,dest);
110 }

```

Lines 73 and 74 contain the typedefs for both passes specifying the underlying model

and the pass id of the `LocalDG` pass. Note that we need to define a `StartPass` in order to define the first pass because every regular pass needs a predecessor to be specified. The `StartPass` is just a dummy class that does nothing.

The initialisation of the passes in lines 96 and 97 then is straightforward. Line 109 shows how a pass can be evaluated like an `Operator` calling all previous passes and projecting on the last parameter afterwards.

## 4.6 Implementing your own Pass Operator

If you want to implement your own Pass operator, you need to derive it either from the `LocalPass` or from the more general `Pass` class. The `LocalDGPass` class, e.g., is a specialisation of the `LocalPass` class which in turn is a `Pass`. Figure 4.1 shows a schema of a general pass  $\Pi_s \circ \mathcal{L}_s \circ \dots \circ \mathcal{L}_2 \circ \mathcal{L}_1$  with all public type names. If you want to implement your own pass, you need to know, that each instance of a `Pass` needs to specify typedefs for `DestinationType` and `GlobalArgumentType` and needs to override the virtual methods `compute` and `allocateLocalMemory` if a class instance needs to allocate memory. For further details on the pass concept and the `LocalPass` class, see the DUNE-FEM documentation.

## 4.7 Visualisation and EOC Output

The program writes data snapshots into the directory given by `fem.prefix`. These snapshots can be visualized with GraPE or Paraview.

Furthermore, `dgtest` produces on every run, a `TeX`file called `eoc.tex`, which comprises statistics about the numerical convergence of the implemented LDG scheme. The computation of the EOC and the formatted output to the `TeX`file is done in the DUNE-FEM utility class `FemEoc`. For further information on the usage of this class, see the source code files in the `../examples/localdg` directory or browse the documentation websites.

4 An LDG solver for Advection-Diffusion Equations

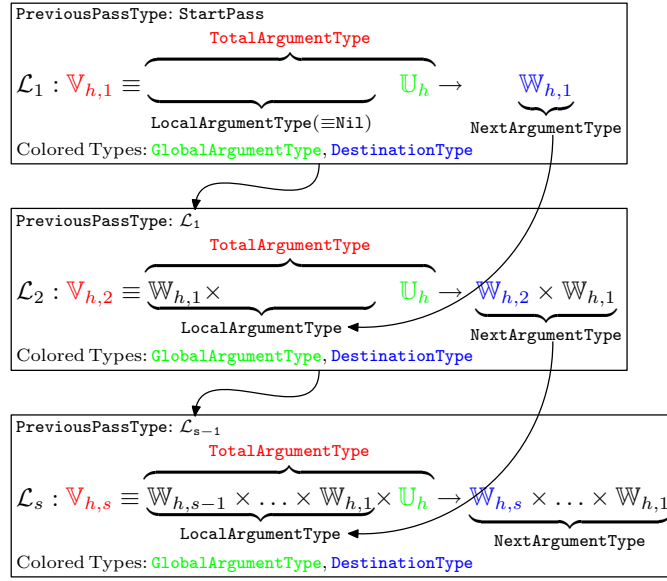


Figure 4.1: Schema of a pass  $\Pi_s \circ \mathcal{L}_s \circ \dots \circ \mathcal{L}_2 \circ \mathcal{L}_1$  with highlighted type names.

## 5 Solving the Stokes problem

In this chapter we want to show how to use DUNE-FEM for solving the Stokes problem with the Finite Element method. The Stokes problem reads:

$$-\Delta \mathbf{u} + \nabla p = \mathbf{f} \quad \text{in } \Omega \quad (5.1)$$

$$\operatorname{div} \mathbf{u} = 0 \quad \text{in } \Omega \quad (5.2)$$

$$u = g \quad \text{on } \partial\Omega. \quad (5.3)$$

The source code of the implementation described in this chapter is located in the directory `dune-femhowto/examples/stokes`. In this example, we choose the following problem data (with  $n = \text{dimworld}$ ):

$$\Omega := ]-1, 1[^2, \quad x = (x_1, x_2) \quad (5.4)$$

$$\mathbf{f}(x) := 0 \quad \forall x \in \Omega \quad (5.5)$$

$$\mathbf{u}(x) := \begin{pmatrix} -\exp^{x_1} \cdot (x_2 \cos(x_2) + \sin(x_2)) \\ \exp^{x_1} \cdot x_2 \sin(x_2) \end{pmatrix} \quad \forall x \in \Omega \quad (5.6)$$

$$p(x) := 2 \exp^{x_1} \sin(x_2) \quad \forall x \in \Omega \quad (5.7)$$

$$\mathbf{g}(x) := \mathbf{u}(x) \quad \forall x \in \partial\Omega \quad (5.8)$$

where  $\mathbf{u}$  is the exact solution for the velocity field and  $p$  the exact solution for the pressure. The boundary values are just the values of the exact solution on the boundary points. The boundary values for the pressure will be achieved by assuming that

$$\int_{\Omega} p = 0. \quad (5.9)$$

In the current implementation we also implemented alternative boundary functions and right hand side functions  $\mathbf{f}$  to simulate the driven cavity problem and the carman vortex street problem. Only for the data described above is equipped with an exact solution  $\mathbf{u}$ . You can switch between them by changing the runtime parameter `femhowto.stokesproblem` in the parameter file `parameter`. For the carman vortex street simulation the macro grid file has to be adapted. In the folder `dune-femhowto/examples/stokes`

## 5 Solving the Stokes problem

is an adapted macro grid file for this type of problem, called `dune-femhowto/examples/stokes/carman.dgf`. Infact the actual problem interface does not support additional pressure functions. So the pressure is not exported or treated in the error calculation process.

The numerical treatment of problem (5.1) will be described in the following. In order to solve the problem with the finite element method a weak formulation of the Stokes problem is needed. A weak formulation of the Stokes problem is given as

$$a(\mathbf{u}, \mathbf{v}) - b^T(\mathbf{v}, p) = F(\mathbf{v}) \quad \forall \mathbf{v} \in V \quad (5.10)$$

$$b(\mathbf{u}, q) = 0 \quad \forall q \in P \quad (5.11)$$

with the linear forms

$$a(\mathbf{u}, \mathbf{v}) := \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \quad \text{for } \mathbf{u}, \mathbf{v} \in V \quad (5.12)$$

$$b(\mathbf{v}, p) := \int_{\Omega} \nabla \mathbf{v} \cdot \mathbf{p} \quad \text{for } \mathbf{v} \in V \text{ and } p \in P. \quad (5.13)$$

The space  $V, P$  are usually chosen as

$$V := H^{1,2}(\Omega)^2$$

and

$$P := L_0^2(\Omega) = \{p \in L^2(\Omega) \mid \int_{\Omega} p = 0\}.$$

As we are interested in the numerics for the Stokes problem, we need discrete dimensional subspaces  $V_h$  and  $P_h$  of  $V$  and  $P$ . This two subspaces  $V_h$  and  $P_h$  have to fulfill the condition

$$\inf_{q \in P_h \setminus \{0\}} \sup_{v \in V_h \setminus \{0\}} \frac{b(v, q)}{\|v\| \|q\|} \geq \beta. \quad (5.14)$$

It is needed to guarantee solutions for the weak formulation of the Stokes problem and also for the discrete formulation based on the weak one. Hence the spaces  $V_h$  and  $P_h$  have to be chosen conforming this condition. As stated in the literature (e.g. Breass) the Taylor-Hood- elements fulfill this condition. A Taylor-Hood-element is given as the space

$$\begin{aligned} V_h \times P_h := & \{v \in C^0(\bar{\Omega}) : v|_T \in \mathbb{P}^k(T) \quad \forall T \in G_h\} \\ & \times \{v \in C^0(\bar{\Omega}) : v|_T \in \mathbb{P}^{k-1}(T) \quad \forall T \in G_h\} \end{aligned} \quad (5.15)$$

## 5 Solving the Stokes problem

with polynomial order  $k$  and a grid  $G_h$ . This choice of the subspaces  $V_h$  and  $P_h$  leads to a system of equations. Similar to the Poisson problem we get

$$A\mathbf{u} - B^T p = \mathbf{F} \quad (5.16)$$

$$B\mathbf{u} = 0 \quad (5.17)$$

where  $A, B$  are given as matrices of the form

$$A_{i,j} := \int_{\Omega} \nabla \varphi_i : \nabla \varphi_j \quad (5.18)$$

$$B_{i,j} := \int_{\Omega} \operatorname{div} \varphi_i \psi_j . \quad (5.19)$$

The functions  $\varphi_i$  and  $\psi_j$  are the base functions of the spaces  $V_h$  and  $P_h$ . By solving the linear system 5.16 an approximating solution to the stokes problem in  $V_h \times P_h$  can be found.

### 5.1 Implementation

We first describe the basic implementation of the algorithm. The basic implementation consists of the following source files:

- **main.hh**: Contains the `main` function. The algorithm first initializes the problem data (implemented in the file `problemdata.hh`) and then calls the main algorithm (implemented in the file `algorithm.hh`). We utilize here the function `compute` from the header file `dune-femhowto/examples/common/base.hh`, which is an abstract implementation of a numerical scheme for stationary problems.
- **problemdata.hh**: Contains several classes which implement several boundary data functions `g` and right hand side functions `f` for the problem 5.1. The actual problem class is chosen at runtime, see the parameter file `parameter`. We use the value of the exact solution as Dirichlet boundary data (see class `Algorithm` in the file `algorithm.hh`), so there is no need to explicitly implement the boundary data `g`.
- **algorithm.hh**: Contains a class `Algorithm` which provides the main algorithm. This algorithm is described in section 5.1.1 in detail.

## 5 Solving the Stokes problem

- `laplaceoperator.hh`: Contains the classes `LaplaceFEOp` and `RightHandSideAssembler` which implements the discrete stiffness matrix  $A$  with entries for DOFs on the region's boundary and the discrete right hand side  $b$  as described in section for the Laplace problem 3.1. How this is implemented is the topic of section 3.1.2.
- `stokesoperator.hh`: Contains the classes `DivergenceFEOp` which models the discrete divergence matrix  $B$  with boundary treatment. How this discrete divergence matrix is implemented is topic of section 5.1.3.

### 5.1.1 Algorithm

We begin with the `main` function in the file `main.hh`, see Listing 13. We delegated the basic numerical scheme to the `compute` function in the file `../common/base.hh`, the problem specific code parts are implemented in the class `Algorithm` (described below), so the `main` function is rather short.

#### Listing 13 (Excerpt of `dune-fem-howto/examples/stokes/main.cc`)

```
66 int main( int argc, char **argv )
67 {
68     typedef GridSelector::GridType GridType;
69
70     typedef ProblemInterface<FunctionSpace< double, double, GridType::dimension,
71         GridType::dimension >> ProblemType;
72
73     // initialize MPI
74     MPIManager :: initialize ( argc, argv );
75     const int rank = MPIManager :: rank ();
76
77     try
78     {
79         // append parameters from the comand line
80         Parameter::append( argc, argv );
81
82         // append parameters from the parameter file
83         Parameter::append( (argc < 2) ? "parameter" : argv[ 1 ] );
84
85         // generate GridPointer holding grid instance
86         GridPtr< GridType > gridptr = initialize< GridType >(std::string("Stokes_
87             problem"));
88
89         // get grid reference
90         GridType& grid = *gridptr ;
91
92         // create problem
93         ProblemType* problem = createProblem<GridType> ();
94         assert( problem );
```

## 5 Solving the Stokes problem

```
93
94 // create stepper class
95 Algorithm<GridType, velocityPolynomialOrder, pressurePolynomialOrder >
    algorithm(grid, *problem);
96 // compute solution
97 compute(algorithm);
98
99 // write parameter logfile
100 Parameter :: write("parameter.log");
101
102 // remove problem
103 delete problem;
104
105 return 0;
106 }
107 catch( Exception &exception )
108 {
109     if( rank == 0 )
110         std :: cerr << exception << std :: endl;
111     return 1;
112 }
113 }
```

After initializing the grid in line 85, we get an instance of a class containing the problem data (line 91) and an instance of the class `Algorithm`, containing the algorithm. After that, the function `compute` is executed in line 97. The function `compute`, defined in the file `../common/base.hh`, calls the `operator()` routine from the class `Algorithm` several times at different grid sizes and the experimental order of convergence is computed.

Now, we are going to explain the problem specific algorithm in the class `Algorithm`, see Listing 14. An explanation of the most important lines in the class definition follows after the listing.

### Listing 14 (Excerpt of `dune-fem-howto/examples/stokes/algorithm.cc`)

```
114 class Algorithm
115 {
116 public:
117
118     //! constructor
119     Algorithm(GridType & grid, const ProblemType& problem)
120     : grid_( grid ),
121       gridPart_( grid_ ),
122       velocitySpace_( gridPart_ ),
123       pressureSpace_( gridPart_ ),
124       problem_( problem )
125     {
126         // add entries to eoc calculation
127         std::vector<std::string> femEocHeaders;
```

## 5 Solving the Stokes problem

```
221 // we want to calculate L2 and H1 error (and EOC)
222 femEocHeaders.push_back("$L^2$-error");
223 femEocHeaders.push_back("$H^1$-error");

230 }
231
232 //! setup and solve the linear system
233 void operator()(DiscreteVelocityFunctionType & solution)
234 {

256 // initialize solution with zero
257
258 solution.clear();
259
260 // create laplace assembler (is assembled on call of systemMatrix by solver)
261 LaplaceOperatorType laplace( velocitySpace_ , problem_ );
262
263 // create divergence assambler
264 DivergenceOperatorType div( velocitySpace_ , pressureSpace_ );
265
266 // create right hand side
267 DiscreteVelocityFunctionType rhs( "rhs", velocitySpace_ );

268
269 // create pressure right hand side
270 DiscretePressureFunctionType g("rhs□for□pressure", pressureSpace_);
271
272 // initialize as zero
273 g.clear();
274
275 // setup right hand side
276 const int quadratureOrder = 2 * velocitySpace_.order() + 1;
277 RightHandSideAssembler < DiscreteVelocityFunctionType >
278   ::assemble ( problem_ , rhs , quadratureOrder );
279
280 // set Dirichlet Boundary to exact solution
281 const IteratorType endit = velocitySpace_.end();
282 for( IteratorType it = velocitySpace_.begin(); it != endit; ++it )
283 {
284   const EntityType &entity = *it;
285   // in entity has intersections with the boundary adjust dirichlet nodes
286   if( entity.hasBoundaryIntersections() )
287     boundaryTreatment( entity, problem_ , rhs , solution );
288 }
289
290 // adjust discrete pressure right hand side to fix the divergence free
291 // condition on
292 // boundary, first apply divergence to bnd data in solution
293 div(solution, pressure);
294
295 // apply the changes in the divergence matrix to the right hande side
296 g -= pressure;
```

## 5 Solving the Stokes problem

```
297     pressure.clear();
298
299     // set correct bnd vals to divergence operator
300     div.boundaryTreatment();
301
302     // solve the linear system
303     solve( laplace, div, rhs, g, solution, pressure );
304 }
305
306 //! finalize computation by calculating errors and EOCs
307 void finalize(DiscreteVelocityFunctionType & solution)
308 {
309     // create exact solution
310     ExactVelocitySolutionType uexact( problem_ );
311
312     // create grid function adapter
313     GridExactVelocitySolutionType ugrid( "exact_velocity", uexact, gridPart_,
314                                         DiscreteVelocitySpaceType ::
315                                             polynomialOrder + 1 );
316
317     // create L2 - Norm
318     L2Norm< GridPartType > l2norm( gridPart_ );
319     // calculate L2 - Norm
320     const double l2errorVelo = l2norm.distance( ugrid, solution );
321
322 }
323
324 private:
325 //! set the dirichlet points to exact values
326 template< class EntityType, class DiscreteFunctionType >
327 void boundaryTreatment( const EntityType &entity,
328                         const ProblemType& problem,
329                         DiscreteFunctionType &rhs,
330                         DiscreteFunctionType &solution)
331 {
332
333     // evaluate boundary data
334     typedef typename VelocityFunctionSpaceType :: RangeType RangeType;
335     RangeType phi, phiLocal;
336     problem.g( global, phi );
337
338     solutionLocal.baseFunctionSet().evaluate( localDof, local, phiLocal );
339     assert( phiLocal.two_norm() != 0 );
340
341     const double bndValue = ( phiLocal * phi ) / phiLocal.two_norm();
342
343     // adjust right hand side and solution data
344     rhsLocal[ localDof ] = bndValue;
345     solutionLocal[ localDof ] = bndValue;
346 }
347 }
```

## 5 Solving the Stokes problem

```
435 }
436
437 //! solve the resulting linear system
438 void solve ( LaplaceOperatorType &laplace,
439             DivergenceOperatorType &div,
440             const DiscreteVelocityFunctionType &rhs,
441             const DiscretePressureFunctionType &g,
442             DiscreteVelocityFunctionType &velocity,
443             DiscretePressureFunctionType &pressure)
444 {
445
446     // create inverse operator
447     InverseOperatorType uzawa( laplace, div, dummy, solverEps, maxIterations,
448                               verbose);
449
450     // solve the system
451     uzawa( rhs, g, velocity, pressure );
452
453 }
454
455 protected:
456     GridType& grid_;
457     GridPartType gridPart_;
458     DiscreteVelocitySpaceType velocitySpace_;
459     DiscretePressureSpaceType pressureSpace_;
460     const ProblemType& problem_;
461     int eocId_;
462 };
```

The definition of the class `Algorithm` starts with some typedefs (not printed here) defining the grid type, the function spaces and discrete functions, the Laplace operator and a solver for the linear systems. Furthermore, the constructor and the methods `operator()`, `finalize`, and `space` need to be defined. The methods `operator()` and `finalize` are evaluated by the `compute` function in this order every time a new numerical solution needs to be computed. The method `operator()` should be used for the implementation of the actual numerical scheme, while the constructor and the method `finalize` can be used for post-processing.

In this example, `operator()` initializes the Laplace operator  $A$  and the Divergence operator in line 261 and 264, the right hand side function  $b$  in lines 267, a second right hand side function  $g$  is needed for the boundary treatment and the solving process. It is initialized and cleared in the line 270. In line 287 we get a boundary function stored in the solution holding the boundary values. The lines 292 to 300 are explained in section

## 5 Solving the Stokes problem

5.1.4. Finally the linear system of equations for the unknown discrete function  $u$  is solved in line 303. This line executes the private method `solve` initializing an UZAWA Solver for the saddlepoint problem 5.16 in line 461 and executing it in line 464.

After the solution step is executed, we compute the  $L^2$  and the  $H^1$  error between the exact and the discrete solution in the `finalize` method. The method `space` simply returns a reference to the instance of a discrete function space.

To change certain runtime parameters the user needs to edit the file `parameter` in the directory `dune-femhowto/examples/stokes/` or the file `parameter` in the directory `dune-femhowto/examples/base/`. These parameter are processed by the `Dune::Parameter` singleton which can be accessed anywhere in the program.

The initial grid size can be manipulated by the `femhowto.startLevel` parameter indicating the number of refinement steps that are executed on the grid before the first numerical solution gets computed. The parameter `femhowto.eocSteps` then controls the number of subsequent EOC computations. The `compute` function makes use of the DUNE-FEM utility classes `FemEOC` and `FemTimer` which write out nicely formatted output information on the evolution of the EOC error and the elapsed time of computations to the files `eoc_main.tex` and `timer.out`.

### 5.1.2 Assembling the Laplace operator

The assembling of the Laplace operator is basically the same as for the Poisson problem, see 3.1.2 for further details. The test problems for the Poisson problem are all scalar. Since we are concerning a velocity field the multi dimensional Laplace operator has to be calculated over all range dimensions. The changes made to the version of the Poisson problem are detailed in listing 15.

#### Listing 15 (Excerpt of `dune-femhowto/examples/stokes/laplaceoperator.hh`)

```
221     //! assemble local matrix for given entity
222     template< class EntityType >
223     void assembleLocal( const EntityType &entity ) const
224     {
225
226         // for all base functions evaluate the gradient
227         // on quadrature point pt and apply jacobian inverse
228         for( size_t i = 0; i < numBaseFunctions; ++i )
229         {
230             // JacobianRangeType is a FieldMatrix
231             JacobianRangeType localGradient;
232             // evaluate gradient of basis functions (on reference element)
233         }
234     }
235 }
```

## 5 Solving the Stokes problem

```
270     baseSet.jacobian( i, quadrature[ pt ], localGradient );
271
272     // apply jacobianInverseTransposed to get gradient on current element
273     // see docu of FieldMatrix for multiply functions
274     for(int k=0;k<dimRange;++k)
275         inv.mv( localGradient[k], gradCache_[ i ][ k ] );
276
277     // apply diffusion tensor
278     for(int k=0;k<dimRange;++k)
279         K.mv( gradCache_[ i ][ k ], gradDiffusion_[ i ][ k ] );
280
281 }
282
283
284
285
286
287
288 }
289
290
291 void updateLocalMatrix ( LocalMatrixType &localMatrix ) const
292 {
293     const size_t rows    = localMatrix.rows();
294     const size_t columns = localMatrix.columns();
295
296     for( size_t i = 0; i < rows; ++i )
297     {
298         for ( size_t j = 0; j < columns; ++j )
299         {
300             RangeFieldType value =0.;
301             for(int k=0;k<dimRange;++k)
302                 value += (gradCache_[ i ][ k ] * gradDiffusion_[ j ][ k ]);
303
304             value *= weight_;
305             localMatrix.add( i, j, value );
306         }
307     }
308 }
```

In line 275 and 301 we add up the multidimensional entries of the discrete Laplace operator.

### 5.1.3 Assambling the discrete Divergence operator

This sections shows how the discrete Divergence operator is assambled. The implementation for the discrete Divergence operator can be found in file `dune-femhowto/examples/stokes/stokesoperator.hh`. The important parts of the operator implementation are shown in listing 16.

**Listing 16 (Excerpt of `dune-fem-howto/examples/stokes/stokesoperator.hh`)**

## 5 Solving the Stokes problem

```
18  template< class DiscreteVelocityFunction, class DiscretePressureFunction, class
      MatrixTraits >
19  class DivergenceFEOp
20  : public Operator< typename DiscreteVelocityFunction :: RangeFieldType,
21                    typename DiscretePressureFunction :: RangeFieldType,
22                    DiscreteVelocityFunction,
23                    DiscretePressureFunction >,
24  public OEMSolver::PreconditionInterface
25  {
26
118 public:
119     //! apply the operator
120     virtual void operator() ( const DiscreteVelocityFunctionType &u,
121                               DiscretePressureFunctionType &w ) const
122     {
123         systemMatrix().apply( u, w );
124     }
125
131
132     virtual void applyTransposed ( const DiscretePressureFunctionType &u,
133                                    DiscreteVelocityFunctionType &w ) const
134     {
135         systemMatrix().apply_t(u,w);
136     }
137
163     LinearOperatorType &systemMatrix () const
164     {
165         // if stored sequence number it not equal to the one of the
166         // dofManager (or space) then the grid has been changed
167         // and matrix has to be assembled new
168         if( sequence_ != dofManager_.sequence() )
169             assemble();
170
171         return linearOperator_;
172     }
173
174     /** \brief perform a grid walkthrough and assemble the global matrix */
175     void assemble () const
176     {
177         const DiscreteVelocityFunctionSpaceType &velocitySpace =
178             discreteFunctionSpace();
179
180         // reserve memory for matrix
181         linearOperator_.reserve();
182
185         // clear matrix
186         linearOperator_.clear();
187
188         // apply local matrix assembler on each element
189         typedef typename DiscreteVelocityFunctionSpaceType :: IteratorType
190             IteratorType;
191         IteratorType end = velocitySpace.end();
```

## 5 Solving the Stokes problem

```

191     for(IteratorType it = velocitySpace.begin(); it != end; ++it)
192     {
193         assembleLocal( *it );
194     }
195
196     // get elapsed time
197     const double assemblyTime = timer.elapsed();
198     // in verbose mode print times
199     if ( Parameter :: verbose () )
200         std :: cout << "Time to assemble div matrix: " << assemblyTime << "s" <<
                std :: endl;
201
202     // get grid sequence number from space (for adaptive runs)
203     sequence_ = dofManager_.sequence();
204 }
205
206 // make boundary treatment
207 void boundaryTreatment () const
208 {
209     typedef typename DiscreteVelocityFunctionSpaceType :: IteratorType
                IteratorType;
210     typedef typename IteratorType :: Entity EntityType;
211
212     const DiscreteVelocityFunctionSpaceType &dfSpace = discreteFunctionSpace();
213
214     const IteratorType end = dfSpace.end();
215     for( IteratorType it = dfSpace.begin(); it != end; ++it )
216     {
217         const EntityType &entity = *it;
218         // if entity has boundary intersections
219         if( entity.hasBoundaryIntersections() )
220             boundaryCorrectOnEntity( entity );
221     }
222 }
223
224 protected:
225     //! assemble local matrix for given entity
226     template< class EntityType >
227     void assembleLocal( const EntityType &entity ) const
228     {
229         typedef typename EntityType :: Geometry GeometryType;
230
231         // assert that matrix is not build on ghost elements
232         assert( entity.partitionType() != GhostEntity );
233
234         // cache geometry of entity
235         const GeometryType &geometry = entity.geometry();
236
237         // get local matrix from matrix object
238         LocalMatrixType localMatrix
239             = linearOperator_.localMatrix( entity, entity );
240
241         // get base function set

```

## 5 Solving the Stokes problem

```

242     const VelocityBaseFunctionSetType &velocityBaseSet = localMatrix.
           domainBaseFunctionSet();
243     const PressureBaseFunctionSetType &pressureBaseSet = localMatrix.
           rangeBaseFunctionSet();
244
245     // get number of local base functions
246     const size_t numVelocityBaseFunctions = velocityBaseSet.numBaseFunctions();
247     const size_t numPressureBaseFunctions = pressureBaseSet.numBaseFunctions();
248
249     // create quadrature of appropriate order
250     QuadratureType quadrature( entity, 2 * (velocityPolynomialOrder - 1) );
251
252     // loop over all quadrature points
253     const size_t numQuadraturePoints = quadrature.nop();
254     for( size_t pt = 0; pt < numQuadraturePoints; ++pt )
255     {
256         // get local coordinate of quadrature point
257         const typename QuadratureType :: CoordinateType &x
           = quadrature.point( pt );
258
259
260         // get jacobian inverse transposed
261         const FieldMatrix< double, dimension, dimension > &inv
           = geometry.jacobianInverseTransposed( x );
262
263
264         // for all base functions evaluate the gradient
265         // on quadrature point pt and apply jacobian inverse
266         for( size_t i = 0; i < numVelocityBaseFunctions; ++i )
267         {
268             // JacobianRangeType is a FieldMatrix
269             JacobianRangeType localGradient;
270             // evaluate gradient of basis functions (on reference element)
271             velocityBaseSet.jacobian( i, quadrature[ pt ], localGradient );
272             // apply jacobianInverseTransposed to get gradient on current element
273             // see docu of FieldMatrix for multiply functions
274             for(size_t k=0;k< dimRange;++k)
275                 inv.mv( localGradient[k], gradCache_[ i ][ k ] );
276         }
277
278         for( size_t i=0; i< numPressureBaseFunctions; ++i)
279         {
280             pressureBaseSet.evaluate(i, quadrature[ pt ], values_[i]);
281         }
282
283         // evaluate integration weight
284         weight_ = quadrature.weight( pt ) * geometry.integrationElement( x );
285
286         // add scalar product of gradients to local matrix
287         updateLocalMatrix( localMatrix );
288     }
289 }
290
291 //! add scalar product of cached gradients to local matrix
292 void updateLocalMatrix ( LocalMatrixType &localMatrix ) const

```

## 5 Solving the Stokes problem

```

293     {
294         const size_t rows    = localMatrix.rows();
295         const size_t columns = localMatrix.columns();
296         for( size_t i = 0; i < rows; ++i )
297         {
298             for ( size_t j = 0; j < columns; ++j )
299             {
300                 RangeFieldType value =0.;
301                 for(size_t k = 0;k<dimRange;++k)
302                     value += gradCache_[ j ][ k ][ k ];
303                 value *= -1.* weight_ *values_[ i ];
304                 localMatrix.add( i, j, value );
305             }
306         }
307     }
381 };

```

The class `DivergenceFEOp` is derived from the `Dune::Operator` class. This is why we need to override the `operator()` method. In line 123 this method delegates the work to the evaluate function of the underlying system matrix. Note that the system matrix is accessed through a call to the `systemMatrix()` method. This method first checks in line 168 whether the grid and therefore the container storing the function's DOFs were changed since the last assembling of the system matrix. This is done with help of the integer variable `dofManager_.sequence` which gets incremented each time the grid changes because of an adaptation or global refinement step. In case it does not equal the private member `sequence_`, the system matrix is invalid and gets updated by the `assemble` method. This method does a grid traversal and calls the method `assembleLocal` on each element in line 193. To invert the system of equations 5.16 we also need a method which applies the transposed matrix to a pressure function. In line 135 the corresponding function on the system matrix is called. Also the method `applyTransposed()` delegates the work to the `applyTransposed` function of the underlying system matrix.

### Local degrees of freedom

We denote by  $I_V(T) := \{i \in \{1, \dots, N\} \mid \varphi_i|_T \neq 0\}$  the set of all DOF indices for the velocity space with corresponding basefunctions that have positive support on the grid entity  $T \in \mathcal{T}$ . Analogue,  $I_P(T) := \{i \in \{1, \dots, N\} \mid \psi_i|_T \neq 0\}$  denotes the indices for the pressure space. Then the DOFs  $v_k^T := v_{\mu_V^T}(k)$  are called the *local degrees of freedom* for the velocity space on the grid entity  $T$  for  $k = 1, \dots, |I_V(T)|$ , where  $\mu_V^T : \{1, \dots, |I_V(T)|\} \rightarrow I_V(T)$  is an enumeration of  $I_V(T)$ . And the DOFs

## 5 Solving the Stokes problem

$p_l^T := v_{\mu_P^T}(k)$  are called the *local degrees of freedom* for the pressure space. With this notation the `LocalMatrix` that is retrieved in line 238 can now be read as a matrix  $L^T \in \mathbb{R}^{|I_V(T)| \times |I_P(T)|}$  with matrix entries

$$(L^T)_{i,j} = (A)_{\mu_V^T(i), \mu_P^T(j)}, \quad 1 \leq i \leq |I_V(T)| \quad 1 \leq j \leq |I_P(T)|. \quad (5.20)$$

Note, that changes in the local matrix also affect the corresponding entries in the system matrix  $B$ .

In order to assemble this local matrix on every grid entity  $T$ , we need the base functions corresponding to the local degrees of freedom on the entity. In line 242 and 243 we retrieve the `BaseFunctionSet` that consists of all base functions  $\varphi_{\mu_V^T(1)}, \dots, \varphi_{\mu_V^T(|I_V(T)|)}$  living on the reference element  $\hat{T}$  for the velocity space and the pressure space  $\psi_{\mu_P^T(1)}, \dots, \psi_{\mu_P^T(|I_P(T)|)}$ . Together with the reference mapping  $\Phi^T : \hat{T} \rightarrow T$  we get the base functions  $\varphi_{\mu_V^T(i)} = \Phi(\varphi_{\hat{\mu}_V^T(i)})$  for  $i = 1, \dots, |I_V(T)|$  and  $\psi_{\mu_P^T(i)} = \Phi(\psi_{\hat{\mu}_P^T(i)})$  for  $i = 1, \dots, |I_P(T)|$ . The gradients of the velocity basefunctions and the functionvalues of the pressure basefunctions are cached in line 275, 280 and written into the local matrix in line 304.

### 5.1.4 Boundary treatment

After the assembling of the stiffness matrix  $A$  (section 5.1.2), and the divergence matrix  $B$  (section 5.1.3) we have to build a function with exact boundary values. This is done in the `algorithm` structure. To get the influence of the boundary values to the system of equations, we apply the divergence matrix to this boundary data 14 line 292. This influence is subtracted from the right hand side function. Then the columns of the divergence matrix corresponding to the boundary DOFs are cleared in line 300.

The boundary treatment for the Laplace operator is the same as in the Poisson problem. (see 3.1.3).

### 5.1.5 Assembling the right hand side

As for the Poisson problem a discrete right hand side function is assembled. The structures used for the Poisson problem can be also used for the Stokes problem. How to implement the right hand side function is described in 3.1.4. Although the right hand side function  $\mathbf{f}$  is a vectorial function, the assembler for the right hand side can be used without further modification, due to the implementation of the basefunctions.

## **5.2 Parallelization**

Currently not implemented.

## 6 Documentation and reference guide for Dune-Fem

The complete documentation of DUNE-FEM is done using the tool `doxygen`. You can find this documentation online<sup>1</sup>. You can also build your own local `doxygen` documentation by removing the flag `--disable-documentation` in your `config.opts` file (see listing on page 8)

Some hints about using the `doxygen` documentation:

- The best way to start is from the page [modules.html](#) which gives you access to the documentation by category.
- A list of the central interface classes can be found on page [interfaceclass.html](#).
- A summary of the main features and concepts of DUNE-FEM can be found on the page [group\\_FEM.html](#).
- Newly added implementations are linked on the page [newimplementation.html](#).

---

<sup>1</sup><http://dune.mathematik.uni-freiburg.de/doc/html-current>

## Bibliography

- [1] DUNE – Distributed and Unified Numerics Environment. URL <http://www.dune-project.org/>.
- [2] DUNE-FEM Doxygen Documentation. URL <http://dune.mathematik.uni-freiburg.de/doc/html-current/index.html>.
- [3] DUNE-FEM The FEM Module. URL <http://dune.mathematik.uni-freiburg.de>.
- [4] P. Bastian, M. Blatt, A. Dedner, Ch. Engwer, R. Klöforn, M. Ohlberger, and O. Sander. The DUNE-GRID howto. URL <http://www.dune-project.org/doc/grid-howto/grid-howto.pdf>.
- [5] A. Dedner, R. Klöforn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive scientific computing: Abstraction principles and the DUNE-FEM module. Preprint No. 3, Mathematisches Institut, Universität Freiburg, 2009. URL <http://dune.mathematik.uni-freiburg.de>.
- [6] A. Dedner, R. Klöforn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module. *Computing*, 89(1), 2010. URL <http://www.springerlink.com/content/vj103u6079861001/>.
- [7] D. Kröner. *Numerical Schemes for Conservation Laws*. Wiley-Teubner, 1997.
- [8] M. Ohlberger and A. Dedner. *Wissenschaftliches Rechnen und Anwendungen in der Strömungsmechanik* (in german), 2006. URL [http://www.mathematik.uni-freiburg.de/IAM/Teaching/ubungen/sci\\_com\\_SS06/skriptum.pdf](http://www.mathematik.uni-freiburg.de/IAM/Teaching/ubungen/sci_com_SS06/skriptum.pdf).

For a more complete list of publications see also: <http://www.dune-project.org/publications.html>