Introduction to Dune-Fem

The DUNE-FEM developers[*]

July 30, 2012

[*]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

Contents

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/tutorial/gettingstarted`) This example shows how to calculate a Lagrange interpolation for a given analytical function.

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

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 availavble.

LDG for Advection-Diffusion equations (`dune-fem-howto/tutorial/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/tutorial/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 then 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[1]

`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[2].

---

[1]UG is not freely available

[2]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[3].

- 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
```

---

[3]http://www.dune-project.org/doc/installation-notes.html

- 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-includes=/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 und 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.

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
```

You will be asked the name of your project (the one you use later for `dunecontrol`) and to give a list of Dune modules your project should depend on - this information can be modified later in the `dune.module` file within your project directory. You should at least provide `dune-common dune-geometry dune-grid dune-fem` and including `dune-fem-howto` might also be a good idea. This allows you to simply include all files found under `dune/fem-howto` directly in your project, e.g.,

```
#include <dune/fem-howto/baseadaptive.hh>
```

as found in `tutorial/poisson/mainadaptive.cc`

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. Espeically if you want to base your code on the examples from this tutorial, note the differences in the `Makefile.am`. Most importantly we specifiy a default grid manager and dimension in our `Makefile.am` and this is not the case for the basic make file found in the new project. Its best simply to add the lines

```
GRIDTYPE = YASPGRID
GRIDDIM = 2
```

into your `Makefile.am` as for example found in the `gettingstarted` example.

You can also create a new subdirectory either under `src` or directly in the main directory of your project. In either case you have to add your new directory to the SUBDIRS variable in the `Makefile.am` of that directory and you will have to create a `Makefile.am` in the new directory. After also having modified your project's `configure.ac`, you have to rerun `dunecontrol`. For more information on the build system, have a look at the Dune Build SystemHowto (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:

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

where $\Gamma_{in} = \{(\mathbf{x}, t) \in \partial\Omega \times [0, T) \,|\, \mathbf{a} \cdot \mathbf{n}(\mathbf{x}) < 0\}$ 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 `../parameters/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 `../tutorial/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-

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

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

$$J := \{0 = t_0 < ... < t_N = T\}. \tag{2.7}$$

Furthermore, we consider an unstructured grid $\mathcal{T}_h$ of Ω. 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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/tutorial/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 discussion the function `main`, displayed in Listing 2.

Listing 2 (Excerpt of dune-femhowto/tutorial/finitevolume/finitevolume.cc)

```
363 int main ( int argc , char **argv )
364 try
365 {
366   typedef Dune :: GridSelector :: GridType GridType ;
```

```
367
368   // initialization
369   Dune::MPIManager::initialize( argc, argv );
370   Dune::Parameter::append( argc, argv );
371   Dune::Parameter::append( argc >= 2 ? argv[ 1 ] : "parameter" );
372
373   Dune::GridPtr< GridType > gridptr
374     = initialize< GridType >( "Finite␣Volume␣Scheme" );
375   GridType &grid = *gridptr;
376
377
378   // EOC loop
379   Stepper< GridType > stepper( grid );
380   compute(stepper);
381
382   // dump parameters
383   Dune::Parameter::write( "parameter.log" );
384
385   return 0;
386 }
387 catch( const Dune::Exception &e )
388 {
389   std::cerr << e << std::endl;
390   return 1;
391 }
392 catch (...)
393 {
394   std::cerr << "Generic␣exception␣caught." << std::endl;
395   return 2;
396 }
```

First, we want to discuss the main programm 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/tutorial/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
```

14

```
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;

77    // return reference to space (needed by ode solvers)
78    const DiscreteSpaceType &space () const
79    {
80      return space_;
81    }

83    // initialize solution to initial data
84    void initialize ( DestinationType &solution )
85    {
86      // iterate over all entities
87      const IteratorType end = space().end();
88      for( IteratorType it = space().begin(); it != end; ++it )
89      {
90        // obtain entity and geometry
91        const Entity &entity = *it;
92        const Geometry &geo = entity.geometry();
93
94        // obtain local function
95        LocalFunctionType lf = solution.localFunction( entity );
96
97        // evaluate initial data in element's center
98        RangeType initialValue;
99        problem_.evaluate( geo.center(), initialValue );
100
101        // directly manipulate the only local DoF (degree of freedom)
102        lf[ 0 ] = initialValue[ 0 ];
103      }
104
105      // communicate the data to other processes (for parallel runs only)
106      solution.communicate();
107    }

109    // application operator
110    void operator () ( const DestinationType &solution,
111                       DestinationType &update ) const
```

```
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.neighbor() )
159          {
160            // access neighbor
161            const EntityPointer neighborPointer = intersection.outside();
162            const Entity &neighbor = *neighborPointer;
163
164
```

```
165            // compute flux from one side only (except it's not an interior entity)
166            if( (neighbor.partitionType() != Dune::InteriorEntity)
167                || (indexSet.index( entity ) < indexSet.index( neighbor )) )

173              DomainType normal = intersection.integrationOuterNormal( x );

179              // obtain velocity
180              const DomainType &velocity = problem_.velocity();
181
182              // calculate wave speed
183              const double waveSpeed = (velocity * normal) * volumeIntersection;
184
185              // compute upwind numerical flux
186              RangeType flux;
187              flux[ 0 ]
188                = (waveSpeed > 0 ? lfSolEn[ 0 ] : lfSolNb[ 0 ]) * waveSpeed;
189
190              // compute update for entity and neighbor
191              lfUpdEn[ 0 ] -= flux[ 0 ] / enVolume;
192              lfUpdNb[ 0 ] += flux[ 0 ] / nbVolume;
193
194              // compute time step restriction
195              const double dtLocal = timeFactor * std::min( enVolume, nbVolume ) /
                     std::abs( waveSpeed );
196              dt_ = std::min( dt_, dtLocal );
197            }
198          }
199          else if( intersection.boundary() )
200          {
201            DomainType normal = intersection.integrationOuterNormal( x );
202
203            // obtain velocity
204            const DomainType &velocity = problem_.velocity();
205
206            // calculate wave speed
207            const double waveSpeed = (velocity * normal) * volumeIntersection;
208
209            // compute upwind numerical flux
210            RangeType flux;
211            if( waveSpeed < 0 )
212            {
213              const DomainType xBnd
214                = geo.global( intersection.geometryInInside().global( x ) );
215
216              RangeType uBnd;
217              problem_.evaluate ( time_, xBnd, uBnd );
218              flux[ 0 ] = uBnd[ 0 ] * waveSpeed;
219            }
220            else
221              flux[ 0 ] = lfSolEn[ 0 ] * waveSpeed;
222
223            // compute update for entity
224            lfUpdEn[ 0 ] -= flux[ 0 ] / enVolume;
```

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


257 template <class GridImp >
258 struct FVStepperTraits
259 {
260   // type of grid
261   typedef GridImp GridType;
262
263   // Initial data
264   //typedef U0<GridType, Dune::Double> InitialData;
265   typedef U0<GridType, double > InitialData;
266   // Define the discrete function for the unkown solution
267   // ... first the grid part ...
268   typedef Dune::AdaptiveLeafGridPart <GridType> GridPartType;
269   // ... next the function space ...
270   typedef Dune::FunctionSpace
271          <typename InitialData::DomainFieldType,
272           typename InitialData::RangeFieldType,
273           GridType::dimensionworld, InitialData::dimRange>
274          FunctionSpaceType;
275   // ... followed by the discrete function space type ...
276   typedef Dune::FiniteVolumeSpace
277          < FunctionSpaceType, GridPartType, 0, Dune::CachingStorage >
278          DiscreteSpaceType;
279   // ... and finally the type for the discrete function ...
280   typedef Dune::AdaptiveDiscreteFunction
281          < DiscreteSpaceType >
282          DiscreteFunctionType;


288 template <class GridImp >
289 struct Stepper : public AlgorithmBase< FVStepperTraits < GridImp > >
290 {
291   // my traits class
292   typedef FVStepperTraits < GridImp > Traits;
293   // base class
294   typedef AlgorithmBase< Traits > BaseType ;
295
296   // type of grid
297   typedef typename Traits :: GridType GridType;
298
299   // Initial data
300   typedef typename Traits :: InitialData InitialData;
301   // Define the discrete function for the unkown solution
```

18

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

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. As 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 161-163 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 166. After getting the outer normal of the intersection, the numerical flux between `entity` and `neighbor` is calculated in lines 180-188. 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 199 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 212-219 where the exact solution u 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 explicite 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. As most work is delegated to the structs `FiniteVolumeScheme` and `Stepper` and the function `compute`, this code part is kept rather short. In lines 36-**??** 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 373 the method `initialize` from the file
`dune-fem-howto/dune/fem-howto/base.hh` is used to create the gridpointer. Finally, in lines 379-380 a `Stepper` object is initialized and the method `compute` is invoked to calculate the solution. The last few lines 387-396 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/baseevolution.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.

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

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

```
79   //! return reference to space
80   DiscreteSpaceType& space()
81   {
82     return space_;
83   }
84
85
86   //! initialize grid, i.e. to start adaptation (overload to do something)
87   //! Return true if grid should be adapted; this method is then called again on
         the
88   //! new grid. Now that on the final grid, the data should be set in the method
89   //! initializeTimeStep().
90   virtual bool initializeGrid(DiscreteFunctionType & solution) {return false;}
91
92   //! initialize method for time loop, i.e. L2-project initial data
93   virtual void initializeStep(TimeProviderType& tp,
94                               DiscreteFunctionType & solution) = 0;
95
96   //! solve one time step
97   virtual void step(TimeProviderType& tp,
98                     DiscreteFunctionType & solution) = 0;
99
100  //! finalize problem, i.e. calculate EOC ...
101  virtual void finalizeStep(TimeProviderType& tp,
102                            DiscreteFunctionType & solution) = 0;
103
104  //! restore all data from check point (overload to do something)
105  virtual void restoreFromCheckPoint(TimeProviderType& tp,
106                                     DiscreteFunctionType & solution) {}
107
108  //! write a check point (overload to do something)
109  virtual void writeCheckPoint(TimeProviderType& tp, DiscreteFunctionType &
         solution ) const {}
110
111  //! default time loop implementation, overload for changes
112  virtual void operator()(DiscreteFunctionType & solution)
113  {
114    const bool verbose = Dune::Parameter :: verbose ();
115    int printCount = Dune::Parameter::getValue<int>("femhowto.printCount", -1);
116
117    const double maxTimeStep =
118      Dune::Parameter::getValue("femhowto.maxTimeStep", std::numeric_limits<double
           >::max());
119    const double startTime = Dune::Parameter::getValue<double>("femhowto.startTime
           ", 0.0);
120    const double endTime   = Dune::Parameter::getValue<double>("femhowto.endTime",
           0.9);
121
122    // for statistics
123    double maxdt    = 0.;
124    double mindt    = 1.e10;
125    double averagedt = 0.;
126
```

22

```
127       // Initialize TimeProvider
128       TimeProviderType tp(startTime, grid_ );
129
130       RestrictionProlongationType rp( solution );
131
132       // set refine weight
133       rp.setFatherChildWeight( Dune::DGFGridInfo<GridType> :: refineWeight() );
134
135       // create adaptation manager
136       AdaptationManagerType adaptManager( grid_ , rp );
137
138       // restoreData if checkpointing is enabled (default is disabled)
139       restoreFromCheckPoint( tp, solution );
140
141       // create data writer after restore is done, otherwise tp is not set correctly
142       IOTupleType dataTup ( &solution );
143       DataWriterType eocDataOutput( grid_, dataTup, tp, EocDataOutputParameters(
             loop_) );
144
145       // adapt the grid to the initial data
146       int startCount = 0;
147       while ( initializeGrid(solution) )
148       {
149         eocDataOutput.write( tp );
150         adaptManager.adapt();
151         if(verbose)
152           std::cout << "start:␣" << startCount << "␣grid␣size:␣" << grid_.size(0)
153                     << std::endl;
154         ++startCount;
155       }
156       // set initial data (and create ode solver)
157       initializeStep(tp, solution);
158
159       // write initial data
160       eocDataOutput.write( tp );
161
162       //***********************************************
163       //* Time Loop                                   *
164       //***********************************************
165       tp.provideTimeStepEstimate(maxTimeStep);
166       for( tp.init() ; tp.time() < endTime ; tp.next() )
167       {
168         tp.provideTimeStepEstimate(maxTimeStep);
169
173         //***************************************************
174         //* Compute an ODE timestep                         *
175         //***************************************************
176         Dune::FemTimer::start(timeStepTimer_);
177         step(tp, solution );
178         Dune::FemTimer::stop(timeStepTimer_,Dune::FemTimer::max);
202         eocDataOutput.write( tp );
```

```
211     } /****** END of time loop *****/
212
213     averagedt /= double(tp.timeStep());
214     if(verbose)
215     {
216       std::cout << "Minimum␣dt:␣" << mindt
217           << "\nMaximum␣dt:␣" << maxdt
218           << "\nAverage␣dt:␣" << averagedt << std::endl;
219     }
220
221     // finalize time step
222     finalizeStep(tp, solution);
223
224     // increase loop counter
225     ++loop_;
226   }
227
228   //! finalize problem, i.e. calculated EOC ...
229   virtual void finalize(DiscreteFunctionType & solution)
230   {}
231
232 protected:
233   GridType&           grid_;
234   GridPartType        gridPart_;
235   DiscreteSpaceType   space_;
236   unsigned int timeStepTimer_;
237   unsigned int loop_ ;
238 };
```

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 ()`. As 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 128). 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 157.

The lines **??-??** 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.

Within the time loop at first an 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 177). 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 222.

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)

```
31  template< class HGridType >
32  Dune::GridPtr< HGridType > initialize ( const std::string &problemDescription )
33  {
34    // ——— read in runtime parameters ———
35    const std::string filekey  = Dune::IOInterface::defaultGridKey( HGridType::
          dimension );
36    const std::string filename = Dune::Parameter::getValue< std::string >( filekey )
          ;
37
38    // initialize grid
39    Dune::GridPtr< HGridType > gridptr(filename);
40    gridptr ->loadBalance ();
41    Dune::Parameter::appendDGF( filename );
42
43    // initialize FemEoc
44    initializeFemEoc( problemDescription );
45
46    // and refine the grid until the startLevel is reached
47    const int startLevel = Dune::Parameter::getValue<int>("femhowto.startLevel", 0);
48    for(int level=0; level < startLevel ; ++level)
```

```
49       gridptr ->globalRefine( 1 );
50
51    return gridptr ;
52 }

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

```
107     else
108       Dune::FemEoc::write(h,grid.size(0),runTime ,0);
109
110     // Refine the grid for the next EOC Step. If the scheme uses adaptation ,
111     // the refinement level needs to be set in the algorithms ' initialize method.
112     if(eocloop < eocSteps -1)
113     {
114       Dune::GlobalRefine::apply(grid,Dune::DGFGridInfo<HGridType >::
                refineStepsForHalf ());
115       grid.loadBalance ();
116     }
117   } /***** END of EOC Loop *****/
118   Dune::FemTimer::removeAll ();
119 }
```

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 contain strings or numbers. Lines starting with a # are understood as commentaries. The lines 36-?? 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 ?? 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 ?? 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 do 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 89). 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 222 the `finalize` method of `algorithm` struct is called. At last, in line 114 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 321 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 234 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 166 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

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

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

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

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

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

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

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

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

$$A\mathbf{u} = \mathbf{b} \tag{3.7}$$

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 $\{\varphi_1, \cdots, \varphi_N\}$. The right hand side $\mathbf{b}$ is given by

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

Except boundary corrections the matrix A equals the stiffness matrix

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

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:

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

3.1 Implementation

We first describe the basic implementation of the algorithm without adaptive or parallel calulations. Adaptive and parallel calulations 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/fem-howto/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

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 stiff-
  ness 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 `dune/fem-howto/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/tutorial/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 );
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 `dune/fem-howto/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/tutorial/poisson/algorithm.hh)

```
108 class Algorithm
109 {
110 public :
```

```
180   //! constructor
181   Algorithm ( HGridType & grid , const ProblemType & problem )
182   : grid_ ( grid ),
183     gridPart_ ( grid_ ),
184     space_ ( gridPart_ ),
185     problem_ ( problem )
186   {
```

```
187      // add entries to eoc calculation
188      std::vector<std::string> femEocHeaders;
189      // we want to calculate L2 and H1 error (and EOC)
190      femEocHeaders.push_back("$L^2$-error");
191      femEocHeaders.push_back("$H^1$-error");


198    }

199
200    //! setup and solve the linear system
201    void operator()(DiscreteFunctionType & solution)
202    {


212      // initialize solution with zero
213      solution.clear();

214
215      // create laplace assembler (is assembled on call of systemMatrix by solver)
216      LaplaceOperatorType laplace( space_ , problem_ );

217
218      // functional describing the right hand side
219      typedef Dune::IntegralFunctional< ProblemType , DiscreteFunctionType >
              FunctionalType ;
220      FunctionalType rhsFunctional( problem_ );

221
222      Dune::DirichletConstraints< DiscreteSpaceType , ProblemType >
223        constraints( space_ , problem_ );

224
225      Dune::NeumannConstraints< DiscreteSpaceType , ProblemType >
226        neumann( space_ , problem_ );

227
228      // solve the linear system  @\ref{
229      solve( laplace , rhsFunctional , solution , constraints , neumann );
230    }

231
232    //! finalize computation by calculating errors and EOCs
233    void finalize(DiscreteFunctionType & solution)
234    {
235      // create exact solution
236      ExactSolutionType uexact( problem_ );

237
238      // create grid function adapter
239      GridExactSolutionType ugrid( "exact solution", uexact, gridPart_ ,
240                                   DiscreteSpaceType :: polynomialOrder + 1 );

241
242      // create L2 - Norm
243      Dune::L2Norm< GridPartType > l2norm( gridPart_ );
244      // calculate L2 - Norm
245      const double l2error = l2norm.distance( ugrid, solution );

246
247      // create H1 - Norm
248      Dune::H1Norm< GridPartType > h1norm( gridPart_ );
249      // calculate H1 - Norm
250      const double h1error = h1norm.distance( ugrid, solution );

251
```

```
252      // store values
253      std::vector<double> errors;
254      errors.push_back( l2error );
255      errors.push_back( h1error );
256
257      // submit error to the FEM EOC calculator
258      Dune::FemEoc :: setErrors(eocId_, errors);
259    }
260
261    //! return reference to discrete space
262    DiscreteSpaceType & space() { return space_; }
263
264 private:
265    //! solve the resulting linear system
266    template <class FunctionalType,
267             class Constraints,
268             class Neumann >
269    void solve ( LaplaceOperatorType &laplace,
270               const FunctionalType& functional,
271               DiscreteFunctionType &solution,
272               const Constraints& constraints,
273               const Neumann& neumann )
274    {
275
300      // create inverse operator (note reduction is not used here)
301      InverseOperatorType cg( laplace, reduction, solverEps );
302
303      // solve the system
304      cg( rhs, solution );
305
312    }
313
314 protected:
315    HGridType& grid_;              // reference to grid, i.e. the hierarchical grid
316    GridPartType gridPart_;        // reference to grid part, i.e. the leaf grid
317    DiscreteSpaceType space_;      // the discrete function space
318    const ProblemType& problem_;   // the problem data
319    int eocId_;                    // id for FemEOC
320 };
```

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 216, the right hand side function **b** in lines **??**-223, and finally solves the linear equation system for the unknown discrete function **u** in line 229. This line executes the private method `solve` initializing a CG Solver in line 301 and executing it in line 304.

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/tutorial/poisson/` or the file `parameter` in the directory `dune-femhowto/tutorial/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 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 **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/tutorial/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    {

104   public:
```

```
105      //! apply the operator
106      virtual void operator() ( const DiscreteFunctionType &u,
107                                 DiscreteFunctionType &w ) const
108      {
109        systemMatrix().apply( u, w );
110      }


149      LinearOperatorType &systemMatrix () const
150      {
151        // if stored sequence number it not equal to the one of the
152        // dofManager (or space) then the grid has been changed
153        // and matrix has to be assembled new
154        if( sequence_ != dofManager_.sequence() )
155          assemble();
156
157        return linearOperator_;
158      }
159
160      /** \brief perform a grid walkthrough and assemble the global matrix */
161      void assemble () const
162      {
163        const DiscreteFunctionSpaceType &space = discreteFunctionSpace();
164
165        // reserve memory for matrix
166        linearOperator_.reserve();
167
171        // clear matrix
172        linearOperator_.clear();
173
174        // apply local matrix assembler on each element
175        typedef typename DiscreteFunctionSpaceType :: IteratorType IteratorType;
176        IteratorType end = space.end();
177        for(IteratorType it = space.begin(); it != end; ++it)
178        {
179          assembleLocal( *it );
180        }
181
182        // get elapsed time
183        const double assemblyTime = timer.elapsed();
184        // in verbose mode print times
185        if ( Parameter :: verbose () )
186          std :: cout << "Time␣to␣assemble␣matrix:␣" << assemblyTime << "s" << std
                 :: endl;
187
188        // get grid sequence number from space (for adaptive runs)
189        sequence_ = dofManager_.sequence();
190      }
191
192  protected :
193      //! assemble local matrix for given entity
194      template< class EntityType >
195      void assembleLocal( const EntityType &entity ) const
196      {
```

```
197        // extract type of geometry from entity
198        typedef typename EntityType :: Geometry Geometry;
199
200        // assert that matrix is not build on ghost elements
201        assert( entity.partitionType() != GhostEntity );
202
203        // cache geometry of entity
204        const Geometry &geometry = entity.geometry();
205
206        // get local matrix from matrix object
207        LocalMatrixType localMatrix
208          = linearOperator_.localMatrix( entity, entity );
209
210        // get base function set
211        const BaseFunctionSetType &baseSet = localMatrix.domainBaseFunctionSet();
212
213        // get number of local base functions
214        const std::size_t numBaseFunctions = baseSet.size();
215
216        // create quadrature of appropriate order
217        QuadratureType quadrature( entity, 2 * (polynomialOrder - 1) );
218
219        // loop over all quadrature points
220        const size_t numQuadraturePoints = quadrature.nop();
221        for( size_t pt = 0; pt < numQuadraturePoints; ++pt )
222        {
223          // get local coordinate of quadrature point
224          const typename QuadratureType :: CoordinateType &x
225            = quadrature.point( pt );
226
227          // get jacobian inverse transposed
228          const typename Geometry :: Jacobian& inv
229            = geometry.jacobianInverseTransposed( x );
230
231          // extract type of diffusion coefficient from problem
232          typedef typename ProblemType :: DiffusionMatrixType DiffusionMatrixType;
233          DiffusionMatrixType K;
234
235          // evaluate diffusion matrix
236          problem().K( geometry.global( x ), K );
237
238          // for all base functions evaluate the gradient
239          // on quadrature point pt and apply jacobian inverse
240          baseSet.jacobianAll( quadrature[ pt ], inv, gradCache_ );
241
242          // apply diffusion tensor
243          for( size_t i = 0; i < numBaseFunctions; ++i )
244            K.mv( gradCache_[ i ][ 0 ], gradDiffusion_[ i ][ 0 ] );
245
246          // evaluate integration weight
247          weight_ = quadrature.weight( pt ) * geometry.integrationElement( x );
248
249          // add scalar product of gradients to local matrix
```

```
250          updateLocalMatrix( localMatrix );
251        }
252     }
253
254     //! add scalar product of cached gradients to local matrix
255     void updateLocalMatrix ( LocalMatrixType &localMatrix ) const
256     {
257       const size_t rows    = localMatrix.rows();
258       const size_t columns = localMatrix.columns();
259       for( size_t row = 0; row < rows; ++row )
260       {
261         for ( size_t col = 0; col < columns; ++col )
262         {
263           const RangeFieldType value
264             = weight_ * (gradCache_[ row ][ 0 ] * gradDiffusion_[ col ][ 0 ]);
265           localMatrix.add( row, col, value );
266         }
267       }
268     }

286   };
```

The class `LaplaceOperator` is derived from the `Dune::Operator` class. This is why we need to override the `operator()` method. In line 109 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 154 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 179.

Local degrees of freedom

We denote by $I(T) := \{i \in \{1, \ldots, N\} \mid \varphi_i|_T \not\equiv 0\}$ 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 $k = 1, \ldots, |I(T)|$, where $\mu^T : \{1, \ldots, |I(T)|\} \to I(T)$ 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 207 can now be read as a matrix $L^T \in \mathbb{R}^{I(T) \times |I(T)|}$ with matrix entries

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

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 211 we retrieve a so-called `BaseFunctionSet` that consists of all base functions $\varphi_{\mu^T(1)}, \ldots, \varphi_{\mu^T(|I(T)|)}$ living on the reference element $\hat{T}$. Together with the reference mapping $\Phi^T : \hat{T} \to T$ we get the wanted base functions $\varphi_{\mu^T(i)} = \Phi(\varphi_{\mu^{\hat{T}}(i)})$ for $i = 1, \ldots, |I(T)|$.

Quadrature

The DUNE-FEM quadrature object which is initialized in line 217 gives us quadrature points $\hat{x}_q^T$ and associated weights w_q^T for $q = 1, \ldots, Q^T$, such that we can compute the stiffness matrix entries

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

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 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 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 hole 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 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 **??-??** 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 `SlaveDof`s. In lines **??–??** 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 $\Omega = [0,1]^3$

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

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

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

where

$$c^1 := (2, 1, 1.3)^t \qquad\qquad c^2 := (0.7, 0.5, 0.1) \tag{4.3}$$
$$\hat{c}^1 := (0.8, 0.4, -0.4)^t \qquad\qquad \hat{c}^2 := (0.2, 0.1, 0.2) \tag{4.4}$$
$$\tilde{c}^1 := (0.6, 1.2, 0.1)^t \qquad\qquad \tilde{c}^2 := (0.9, 0.3, -0.3). \tag{4.5}$$

Therefore the initial and boundary data functions are defined by

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

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].

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 $V_h := \{\varphi_1, \ldots \varphi_n\}$ resp. $[V_h]^3 = \{\psi_1, \ldots \psi_{n^3}\}$ with discrete domain in space. The equations in (4.7) can then be rewritten as a variational formulation

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

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

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

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

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

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

and the projection operator Π 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/tutorial/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 `dune/fem-howto/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 (../../tutorial/localdg/dgtest.cc)

```
1  // include host specific macros set by configure script
2  #include <config.h>
3
4  // include std libs
5  #include <iostream>
6  #include <string>
7
8  // Dune includes
9  #include <dune/fem/misc/l2error.hh>
10 #include <dune/fem/operator/projection/l2projection.hh>
11 #include <dune/fem/gridpart/common/gridpart.hh>
12 #include <dune/fem/solver/odesolver.hh>
13
14 // include local header files
15 #include <dune/fem-howto/baseevolution.hh>
16 #include "models.hh"
17
18 // approximation order
19 const int order   = POLORDER;
20 const int rkSteps = POLORDER + 1;

182 int main(int argc, char ** argv, char ** envp) {
183    typedef Dune::GridSelector :: GridType GridType;
184
185    /* Initialize MPI (always do this even if you are not using MPI) */
186    Dune::MPIManager :: initialize( argc, argv );
187
188    try {
189
190    // *** Initialization
191    Dune::Parameter::append(argc,argv);
192    if (argc == 2) {
193      Dune::Parameter::append(argv[1]);
194    }
195    else
```

```
196    {
197       Dune::Parameter::append("parameter");
198    }
199
200    // ProblemType is a Dune::Function that evaluates to $u_0$ and also has a
201    // method that gives you the exact solution.
202    typedef Dune::U0< GridType > ProblemType;
203    ProblemType problem;
204
205    // Note to me: problem description is for FemEOC
206    Dune::GridPtr<GridType> gridptr = initialize< GridType >( std::string("Local␣DG␣
          scheme") +
207                                              problem.description() );
208    // get grid reference
209    GridType & grid = *gridptr;
210
211    /***********************************************
212     * EOC Loop                                    *
213     **********************************************/
214    Stepper<GridType, ProblemType :: BaseType> stepper(grid, problem);
215
216    compute(stepper);
217
218    Dune::Parameter::write("parameter.log");
219
220    }
221    catch (Dune::Exception &e) {
222      std::cerr << e << std::endl;
223      return 1;
224    } catch (...) {
225      std::cerr << "Generic␣exception!" << std::endl;
226      return 2;
227    }
228
229    return 0;
230 }
```

Note that the entire main function is embedded into a `try` block such that exceptions can be caught in lines 221-227. This is recommended in order to get useful error messages in case something goes wrong the runtime.

In lines 191-198 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 197) 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 206 which 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 214 and pass it to the `evolve` function in line 216 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 ../../tutorial/localdg/dgtest.cc)

```
22  template <class GridImp , class InitialDataType >
23  struct StepperTraits {
24    // type of Grid
25    typedef GridImp                                         GridType ;
26
27    // Choose a suitable GridView
28    typedef Dune :: DGAdaptiveLeafGridPart < GridType >
          GridPartType ;
29
30    // An analytical version of our model
31    typedef AdvectionDiffusionModel < GridPartType , InitialDataType >   ModelType ;
32
33    // The flux for the discretization of advection terms
34    typedef UpwindFlux < ModelType >                        FluxType ;
35
36    // The DG Operator (using 2 Passes)
37    typedef Dune :: DGAdvectionDiffusionOperator < ModelType , UpwindFlux ,
38                                          order >                DgType ;
39
40    // The discrete function for the unknown solution is defined in the DgOperator
41    typedef typename DgType :: DestinationType
          DiscreteFunctionType ;
42    // ... as well as the Space type
43    typedef typename DgType :: SpaceType
          DiscreteSpaceType ;
44
45    // The ODE Solvers
46    typedef DuneODE :: OdeSolverInterface < DiscreteFunctionType >
          OdeSolverInterfaceType ;
47    typedef DuneODE :: ImplicitOdeSolver < DiscreteFunctionType >
          ImplicitOdeSolverType ;
48    typedef DuneODE :: ExplicitOdeSolver < DiscreteFunctionType >
          ExplicitOdeSolverType ;
49
50    // type of restriction/prolongation projection for adaptive simulations
```

```
51    typedef Dune :: RestrictProlongDefault < DiscreteFunctionType >
          RestrictionProlongationType;
52  };
53
54  template < class GridImp , class InitialDataType >
55  struct Stepper : public AlgorithmBase < StepperTraits < GridImp , InitialDataType > >
56  {
57    // my traits class
58    typedef StepperTraits < GridImp , InitialDataType > Traits ;
59
60    // my base class
61    typedef AlgorithmBase < Traits > BaseType ;
62
63    // type of Grid
64    typedef typename Traits :: GridType               GridType ;
65
66    // Choose a suitable GridView
67    typedef typename Traits :: GridPartType           GridPartType ;
68
69    // An analytical version of our model
70    typedef typename Traits :: ModelType              ModelType ;
71
72    // The flux for the discretization of advection terms
73    typedef typename Traits :: FluxType               FluxType ;
74
75    // The DG Operator ( using 2 Passes )
76    typedef typename Traits :: DgType                 DgType ;
77
78    // The discrete function for the unknown solution is defined in the DgOperator
79    typedef typename Traits :: DiscreteFunctionType       DiscreteFunctionType;
80
81    // ... as well as the Space type
82    typedef typename Traits :: DiscreteSpaceType          DiscreteSpaceType;
83
84    // The ODE Solvers
85    typedef typename Traits :: OdeSolverInterfaceType     OdeSolverInterfaceType;
86    typedef typename Traits :: ImplicitOdeSolverType      ImplicitOdeSolverType;
87    typedef typename Traits :: ExplicitOdeSolverType      ExplicitOdeSolverType;
88
89    typedef typename BaseType :: TimeProviderType    TimeProviderType;
90
91    using BaseType :: grid_;
92
93    Stepper (GridType & grid , const InitialDataType & problem ) :
94      BaseType ( grid ),
95      problem_ (problem),
96      model_ (problem_),
97      convectionFlux_ (model_),
98      dgOperator_ (grid_ , convectionFlux_),
99      eocId_ ( Dune :: FemEoc :: addEntry (std :: string ("$L^2$-error")) ),
100     odeSolver_ (0)
101   {
102   }
```

49

```
103
104    ~Stepper()
105    {
106       delete odeSolver_;
107       odeSolver_ = 0;
108    }
109
110    // before first step, do data initialization
111    void initializeStep(TimeProviderType& tp, DiscreteFunctionType& u)
112    {
113       // choice of explicit or implicit ode solver
114       static const std::string odeSolver []
115         = { "explicit", "implicit" };
116       int odeSolve = Dune::Parameter::getEnum( "femhowto.odesolver", odeSolver, 0 );
117
118       // create implicit or explicit ode solver
119       if( odeSolve == 0 )
120         odeSolver_ = new ExplicitOdeSolverType(dgOperator_, tp, rkSteps);
121       else
122         odeSolver_ = new ImplicitOdeSolverType(dgOperator_, tp, rkSteps);
123
124       assert( odeSolver_ );
125
126
127       Dune::Fem::L2Projection< InitialDataType, DiscreteFunctionType > l2pro;
128       l2pro(problem_, u);
129
130       // odeSolver_->initialize applies the DG Operator once to get an initial
131       // estimate on the time step. This does not change the initial data u.
132       odeSolver_->initialize(u);
133    }
134
135    // solve ODE for one time step
136    void step(TimeProviderType& tp, DiscreteFunctionType& u)
137    {
138       assert(odeSolver_);
139       odeSolver_->solve(u);
140    }
141
142    // after last step, do EOC calculation
143    void finalizeStep(TimeProviderType& tp, DiscreteFunctionType& u)
144    {
145       Dune::L2Error<DiscreteFunctionType> L2err;
146       // Compute L2 error of discretized solution ...
147       Dune::FieldVector<double,ModelType::dimRange> error;
148       error = L2err.norm(problem_, u, tp.time());
149       // ... and print the statistics out to the eocOutputPath file
150       Dune::FemEoc::setErrors(eocId_,error.two_norm());
151
152       // delete ode solver
153       delete odeSolver_;
154       odeSolver_ = 0;
155    }
```

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 127-128 by a projection of the analytical function u_0 onto the discrete function space with help of the utility class L2Projection. Furthermore, in lines 120 and 122 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 132, 139), 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 formating 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

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

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.

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

(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 ../../tutorial/localdg/discretemodels.hh)

```
270    // Discrete Model for Pass2
271    template <class Model, class NumFlux, int polOrd, int passUId, int passGradId>
272    class AdvDiffDModel2 :
273      public DGDiscreteModelDefaultWithInsideOutside
274        <AdvDiffTraits2<Model, NumFlux, polOrd, passUId, passGradId >,
275              passUId , passGradId >
276    {
277      typedef DGDiscreteModelDefaultWithInsideOutside
278              < AdvDiffTraits2< Model, NumFlux, polOrd, passUId, passGradId >,
279                passUId, passGradId >                          BaseType;
280
281      using BaseType :: numericalFlux;
282      using BaseType :: boundaryFlux;
283
284      // These type definitions allow a convenient access to arguments of pass.
285  #if DUNE_VERSION_NEWER_REV(DUNE_COMMON ,2,1,0)
286      integral_constant <int,passUId>     uVar;
287      integral_constant <int,passGradId> sigmaVar;
288  #else
289      Int2Type<passUId>     uVar;
290      Int2Type<passGradId> sigmaVar;
291  #endif
292    public:
293      typedef AdvDiffTraits2< Model, NumFlux, polOrd, passUId,
```

```
294                                passGradId >                                Traits;

313       AdvDiffDModel2(const Model& mod,const NumFlux& numf) :
314         model_(mod),
315         numflux_(numf),
316         penalty_(Parameter::getValue<double>("femhowto.penalty")),
317         // Set CFL number for penalty term (compare diffusion in first pass)
318         cflDiffinv_(2.*(polOrd+1.))
319       {}
320
321       bool hasSource() const { return false; }
322       bool hasFlux() const { return true; }


337       template <class ArgumentTuple>
338       double numericalFlux(const Intersection& it,
339                            double time, const FaceLocalCoordinate& x,
340                            const ArgumentTuple& uLeft,
341                            const ArgumentTuple& uRight,
342                            RangeType& gLeft,
343                            RangeType& gRight)
344       {
345         const DomainType normal = it.integrationOuterNormal(x);


348         /*****************************
349          * Advection                *
350          *****************************/
351         // delegated to numflux_
352         double wave = numflux_.
353           numericalFlux(it, time, x, uLeft[uVar], uRight[uVar], gLeft, gRight);
354
355         /*****************************
356          * Diffusion                *
357          *****************************/
358         JacobianRangeType diffmatrix;
359         RangeType diffflux(0.);
360         /* Central differences */
361         model_.
362           diffusion2(this->inside(), time, it.geometryInInside().global(x),
363                      uLeft[uVar], uLeft[sigmaVar], diffmatrix);
364         diffmatrix.umv(normal, diffflux);
365         model_.
366           diffusion2(this->outside(), time, it.geometryInOutside().global(x),
367                      uRight[uVar], uRight[sigmaVar], diffmatrix);
368         diffmatrix.umv(normal, diffflux);
369         diffflux*=0.5;
370
371         // add penalty term ( enVolume() is available since we derive from
372         //     DGDiscreteModelDefaultWithInsideOutside )
373         double factor = penalty_ * model_.diffusionCoefficient() *
374                         normal.two_norm2() /
375                         ( 0.5 * ( this->enVolume()+this->nbVolume() ) );
376
```

```
377          RangeType jump( uLeft[uVar] );
378          jump -= uRight[uVar];
379          diffflux.axpy(factor,jump);
380
381          gLeft  += diffflux;
382          gRight += diffflux;
383          return std::max( wave, factor*cflDiffinv_ );
384      }
385
386      /**
387       * @brief same as numericalFlux() but for fluxes over boundary interfaces
388       */
389      template <class ArgumentTuple >
390      double boundaryFlux(const Intersection& it,
391                          const double time,
392                          const FaceLocalCoordinate& x,
393                          const ArgumentTuple& uLeft,
394                          RangeType& gLeft)
437
438      /**
439       * @brief analytical flux function $f_2(u_0,u_1)$
440       */
441      template <class ArgumentTuple >
442      void analyticalFlux( const EntityType& en,
443                          const double time,
444                          const LocalCoordinate& x,
445                          const ArgumentTuple& u, JacobianRangeType& f )
446      {
447        /****************************
448         * Advection               *
449         ****************************/
450        model_.advection(en,time, x, u[uVar],f);
451        /****************************
452         * Diffusion               *
453         ****************************/
454        JacobianRangeType diffmatrix;
455        model_.diffusion2(en, time, x, u[uVar], u[sigmaVar], diffmatrix);
456        f += diffmatrix;
457      }
458    private:
459      const Model   &model_;
460      const NumFlux &numflux_;
461      const double   penalty_;
462      const double   cflDiffinv_;
463    };
```

Note that in line 321 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 289 and 290. These types then allow `ArgumentTuple`s to be used like arrays, as in line 361 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 (../../tutorial/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 >
        DiscreteModel2Type;
51
52   typedef typename DiscreteModel1Type :: Traits            Traits1;
53   typedef typename DiscreteModel2Type :: Traits            Traits2;
54
55   typedef typename Model :: Traits :: GridType             GridType;
56
57   typedef typename Traits2 :: DomainType                   DomainType;
58   typedef typename Traits2 :: DiscreteFunctionType
        DiscreteFunction2Type;
59
60   typedef typename Traits1 :: DiscreteFunctionSpaceType    Space1Type;
61   typedef typename Traits2 :: DiscreteFunctionSpaceType    Space2Type;
62   typedef typename Traits1 :: DestinationType
        Destination1Type;
```

55

```
63    typedef typename Traits2 :: DestinationType
          Destination2Type;
64    typedef Destination2Type
          DestinationType;
65    typedef Space2Type                                          SpaceType;
66
67    typedef typename Traits1 :: GridPartType                     GridPartType;
68
69    /***************************************
70     * Join the Passes 0-2               *
71     ***************************************/
72    typedef StartPass< DiscreteFunction2Type , u >              Pass0Type;
73    typedef LocalDGPass< DiscreteModel1Type , Pass0Type, gradPass >   Pass1Type;
74    typedef LocalDGPass< DiscreteModel2Type , Pass1Type, advectPass >  Pass2Type;

85    DGAdvectionDiffusionOperator( GridType& grid,
86                                  const NumFluxType& numf) :
87      grid_(grid),
88      verbose_(Parameter :: getValue<int>("femhowto.verbose")),
89      model_(numf.model()),
90      numflux_(numf),
91      gridPart_(grid_),
92      space1_(gridPart_),
93      space2_(gridPart_),
94      problem1_(model_,numflux_),
95      problem2_(model_,numflux_),
96      pass1_(problem1_, pass0_, space1_),
97      pass2_(problem2_, pass1_, space2_)
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 evaluationt
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 $\Pi_s \circ \mathcal{L}_s \circ \cdots \circ \mathcal{L}_2 \circ \mathcal{L}_1$ 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.
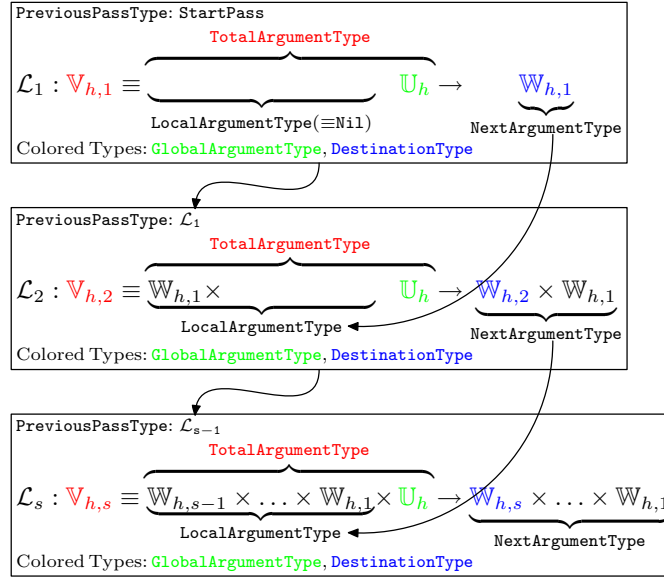

Figure 4.1: Schema of a pass $\Pi_s \circ \mathcal{L}_s \circ \cdots \circ \mathcal{L}_2 \circ \mathcal{L}_1$ with highlighted type names.

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 TEXfile 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 TEXfile 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 `../tutorial/localdg` directory or browse the documentation websites.

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:

$$-\triangle \mathbf{u} + \nabla p = \mathbf{f} \quad \text{in } \Omega \tag{5.1}$$

$$\text{div } \mathbf{u} = 0 \quad \text{in } \Omega \tag{5.2}$$

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

The source code of the implementation described in this chapter is located in the di-
rectory `dune-femhowto/tutorial/stokes`. In this example, we choose the following
problem data (with $n = $ `dimworld`):

$$\Omega :=] -1, 1[^2, \qquad\qquad\qquad x = (x_1, x_2) \tag{5.4}$$

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

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

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

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

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

$$\int_\Omega p = 0 \ . \tag{5.9}$$

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 solu-
tion $\mathbf{u}$. You can switch between them by changing the runtime parameter `femhowto.`
`stokesproblem` in the parameter file `parameter`. For the carman vortex street simula-
tion the macro grid file has to be adapted. In the folder `dune-femhowto/tutorial/stokes`

is an adapted macro grid file for this type of problem, called
`dune-femhowto/tutorial/stokes/carman.dgf`.
In fact the actual problem interface does not support additional pressure functions. So
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

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

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

with the linear forms

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

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

The space V, P are usualy 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. These two subspaces V_h and P_h have to fullfill the
condition

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

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 fullfill this condition. A Taylor-Hood-element is given as the
space

$$V_h \times P_h := \{v \in C^0(\bar{\Omega}) : v|_T \in \mathbb{P}^k(T) \ \forall T \in G_h\}$$

$$\times \{v \in C^0(\bar{\Omega}) : v|_T \in \mathbb{P}^{k-1}(T) \ \forall T \in G_h\} \tag{5.15}$$

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

$$A\mathbf{u} - B^T p = \mathbf{F} \tag{5.16}$$
$$B\mathbf{u} = 0 \tag{5.17}$$

where A, B are given as matrices of the form

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

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

The functions φ_i and ψ_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/fem-howto/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 choosen 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.

- `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 divergnece 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 `dune/fem-howto/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/tutorial/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 ,
         GridType ::dimension > > ProblemType ;
71
72   // initialize MPI
73   MPIManager :: initialize ( argc, argv );
74   const int rank = MPIManager :: rank ();
75
76   try
77   {
78     // append parameters from the comand line
79     Parameter ::append( argc, argv );
80
81     // append parameters from the parameter file
82     Parameter ::append( (argc < 2) ? "parameter" : argv[ 1 ] );
83
84     // generate GridPointer holding grid instance
85     GridPtr< GridType > gridptr = initialize < GridType >(std::string("Stokes␣
         problem"));
86
87     // get grid reference
88     GridType & grid = *gridptr ;
89
90     // create problem
91     ProblemType * problem = createProblem <GridType> ();
```

```
92      assert ( problem );
93
94      // create stepper class
95      Algorithm<GridType, velocityPolynomialOrder, preasurePolynomialOrder >
              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 `dune/fem-howto/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/tutorial/stokes/algorithm.hh)

```
117 class Algorithm
118 {
119 public :

263   //! constructor
264   Algorithm (GridType & grid, const ProblemType& problem)
265   : grid_( grid ),
266     gridPart_( grid_ ),
267     velocitySpace_( gridPart_ ),
268     pressureSpace_( gridPart_ ),
269     problem_( problem )
```

```
270    {
271      // add entries to eoc calculation
272      std::vector<std::string> femEocHeaders;
273      // we want to calculate L2 and H1 error (and EOC)
274      femEocHeaders.push_back("$L^2$-error");
275      femEocHeaders.push_back("$H^1$-error");

282    }

283
284    //! setup and solve the linear system
285    void operator()(DiscreteVelocityFunctionType & solution)
286    {

308      // initialize solution with zero
309      solution.clear();

310
311      // create laplace assembler (is assembled on call of systemMatrix by solver)
312      LaplaceOperatorType laplace( velocitySpace_ , problem_ );

313
314      // create divergence assambler
315      DivergenceOperatorType div( velocitySpace_ , pressureSpace_);

316
317      // create right hand side
318      DiscreteVelocityFunctionType rhs( "rhs", velocitySpace_ );

319
320      // create pressure right hand side
321      DiscretePressureFunctionType g("rhs␣for␣pressure", pressureSpace_);

322
323      // initialize as zero
324      g.clear();

325
326      // setup right hand side
327      const int quadratureOrder = 2 * velocitySpace_.order() + 1;
328      RightHandSideAssembler< DiscreteVelocityFunctionType >
329        ::assemble ( problem_ , rhs , quadratureOrder );

330
331      // set Dirichlet Boundary to exact solution
332      const IteratorType endit = velocitySpace_.end();
333      for( IteratorType it = velocitySpace_.begin(); it != endit; ++it )
334      {
335        const EntityType &entity = *it;
336        // in entity has intersections with the boundary adjust dirichlet nodes
337        if( entity.hasBoundaryIntersections() )
338          boundaryTreatment( entity, problem_, rhs, solution );
339      }

340
341      // adjust discrete pressure right hand side to fix the divergence free
               condition on
342      // boundary, first apply divergence to bnd data in solution
343      div(solution, pressure);

344
345      // apply the changes in the divergence matrix to the right hande side
```

```
346      g -= pressure ;
347
348      pressure . clear ();
349
350      // set  correct  bnd  vals  to  divergence  operator
351      div . boundaryTreatment ();
352
353
354      // solve  the  linear  system
355      solve ( laplace , div , rhs , g , solution , pressure );
356    }
357
358    //! finalize  computation  by  calculating  errors  and  EOCs
359    void finalize ( DiscreteVelocityFunctionType & solution )
360    {
361      // create  exact  solution
362      ExactVelocitySolutionType uexact ( problem_ );
363
364      // create  grid  function  adapter
365      GridExactVelocitySolutionType ugrid ( "exact␣velocity", uexact , gridPart_ ,
366                                            DiscreteVelocitySpaceType ::
367                                                polynomialOrder + 1 );
368      // create  L2 − Norm
369      L2Norm < GridPartType > l2norm ( gridPart_ );
370      // calculate  L2 − Norm
371      const double l2errorVelo = l2norm . distance ( ugrid , solution );
386    }

394 private :
395    //! set  the  dirichlet  points  to  exact  values
396    template < class EntityType , class DiscreteFunctionType >
397    void boundaryTreatment ( const EntityType & entity ,
398                             const ProblemType & problem ,
399                             DiscreteFunctionType & rhs ,
400                             DiscreteFunctionType & solution )
401    {

469           // evaluate  boundary  data
470           typedef typename VelocityFunctionSpaceType :: RangeType RangeType ;
471           RangeType phi , phiLocal ;
472           problem .g ( global , phi );
473
474           const double bndValue = phi [ i ];
475
476           // adjust  right  hand  side  and  solution  data
477           rhsLocal [ localDof ] = bndValue ;
478           solutionLocal [ localDof ] = bndValue ;
479
```

```
483   }
484
485   //! solve the resulting linear system
486   void solve ( LaplaceOperatorType &laplace,
487                DivergenceOperatorType &div,
488                const DiscreteVelocityFunctionType &rhs,
489                const DiscretePressureFunctionType &g,
490                DiscreteVelocityFunctionType &velocity,
491                DiscretePressureFunctionType &pressure)
492   {
493

511     // create inverse operator
512     InverseOperatorType uzawa( inverseLaplace, div, dummy, solverEps,
            maxIterations, verbose);
513
514     // solve the system
515     uzawa( rhs, g, velocity, pressure );

523   }
524
525
526 protected:
527   GridType& grid_;
528   GridPartType gridPart_;
529   DiscreteVelocitySpaceType velocitySpace_;
530   DiscretePressureSpaceType pressureSpace_;
531   const ProblemType& problem_;
532   int eocId_;
533 };
```

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 312 and 315, the right hand side function b in lines 318, a second right hand side function g is needed for the boundary treatment and the solving process. It is intialized and cleared in the line 321. In line 338 we get a boundary function stored in the solution holding the boundary values. The lines 343 to 351 are explained in section

5.1.4. Finally the linear system of equations for the unknown discrete function u is solved in line 355. This line executes the private method `solve` initializing an UZAWA Solver for the saddlepoint problem 5.16 in line 512 and executing it in line 515.

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/tutorial/stokes/` or the file `parameter` in the directory `dune-femhowto/tutorial/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 basicly 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 concering 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 detiled in listing 15.

Listing 15 (Excerpt of dune-femhowto/tutorial/stokes/laplaceoperator.hh)

```
221    //! assemble local matrix for given entity
222    template< class EntityType >
223    void assembleLocal( const EntityType &entity ) const
224    {

263        // for all base functions evaluate the gradient
264        // on quadrature point pt and apply jacobian inverse
265        baseSet.jacobianAll( quadrature[ pt ], inv, gradCache_ );
266
267        // apply diffusion tensor
268        for( size_t i = 0; i < numBaseFunctions; ++i )
269        {
```

```
270          for(int k=0;k<dimRange;++k)
271            K.mv( gradCache_[ i ][ k ], gradDiffusion_[ i ][ k ] );
272        }


280      }


283      void updateLocalMatrix ( LocalMatrixType &localMatrix ) const
284      {
285        const size_t rows    = localMatrix.rows();
286        const size_t columns = localMatrix.columns();

287
288        for( size_t i = 0; i < rows; ++i )
289        {
290          for ( size_t j = 0; j < columns; ++j )
291          {
292            RangeFieldType value =0.;
293            for(int k=0;k<dimRange;++k)
294              value += (gradCache_[ i ][ k ] * gradDiffusion_[ j ][ k ]);

295
296            value *= weight_;
297            localMatrix.add( i, j, value );
298          }
299        }
300      }
```

In line **??** and 293 we add up the multidimensional entries of the discrete Laplace operator.

5.1.3 Assembling the discrete divergence operator

This section shows how the discrete divergence operator is assembled. The implementation for the discrete divergence operator can be found in file
dune-femhowto/tutorial/stokes/stokesoperator.hh. The important parts of the operator implementation are shown in listing 16.

Listing 16 (Excerpt of dune-fem-howto/tutorial/stokes/stokesoperator.hh)

```
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    {
```

```
118    public:
119      //! apply the operator
120      virtual void operator() ( const DiscreteVelocityFunctionType &u,
121                                 DiscretePressureFunctionType &w ) const
122      {
123        systemMatrix().apply( u, w );
124      }


131
132      virtual void applyTransposed ( const DiscretePressureFunctionType &u,
133                                     DiscreteVelocityFunctionType &w) const
134      {
135        systemMatrix().apply_t(u,w);
136      }


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        // reserve memory for matrix
180        linearOperator_.reserve();


185        // clear matrix
186        linearOperator_.clear();
187
188        // apply local matrix assembler on each element
189        typedef typename DiscreteVelocityFunctionSpaceType :: IteratorType
190            IteratorType;
190        IteratorType end = velocitySpace.end();
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 boundry treament
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
242       const VelocityBaseFunctionSetType &velocityBaseSet = localMatrix.
              domainBaseFunctionSet();
243       const PressureBaseFunctionSetType &pressureBaseSet = localMatrix.
              rangeBaseFunctionSet();
244
245       // create quadrature of appropriate order
246       QuadratureType quadrature( entity, 2 * (velocityPolynomialOrder - 1) );
247
248       // loop over all quadrature points
249       const size_t numQuadraturePoints = quadrature.nop();
```

70

```
250          for( size_t pt = 0; pt < numQuadraturePoints; ++pt )
251          {
252            // get local coordinate of quadrature point
253            const typename QuadratureType :: CoordinateType &x
254              = quadrature.point( pt );
255
256            // get jacobian inverse transposed
257            const FieldMatrix< double, dimension, dimension > &inv
258              = geometry.jacobianInverseTransposed( x );
259
260
261            // for all base functions evaluate the gradient
262            // on quadrature point pt and apply jacobian inverse
263            velocityBaseSet.jacobianAll( quadrature[ pt ], inv, gradCache_ );
264
265            // evaluate each base function
266            pressureBaseSet.evaluateAll( quadrature[ pt ], values_ );
267
268            // evaluate integration weight
269            weight_ = quadrature.weight( pt ) * geometry.integrationElement( x );
270
271            // add scalar product of gradients to local matrix
272            updateLocalMatrix( localMatrix );
273          }
274        }
275
276        //! add scalar product of cached gradients to local matrix
277        void updateLocalMatrix ( LocalMatrixType &localMatrix ) const
278        {
279          const size_t rows    = localMatrix.rows();
280          const size_t columns = localMatrix.columns();
281          for( size_t i = 0; i < rows; ++i )
282          {
283            for ( size_t j = 0; j < columns; ++j )
284            {
285              RangeFieldType value =0.;
286              for(size_t k = 0;k<dimRange;++k)
287                value += gradCache_[ j ][ k ][ k ];
288              value *= -1.* weight_ *values_[ i ];
289              localMatrix.add( i, j, value );
290            }
291          }
292        }

366    };
```

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 $I_V(T) := \{i \in \{1, \ldots, N\} \mid \varphi_i|_T \not\equiv 0\}$ the set of all DOF indices for the velocity space with corresponding basefunctions that have positive support on the grid entity $T \in \mathcal{T}$. Analogously, $I_P(T) := \{i \in \{1, \ldots, N\} \mid \psi_i|_T \not\equiv 0\}$ 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 $k = 1, \ldots, |I_V(T)|$, where $\mu_V^T : \{1, \ldots, |I_V(T)|\} \to I_V(T)$ is an enumeration of $I_V(T)$. And the DOFs $p_l^T := v_{\mu_P^T}(k)$ 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

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

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 $\varphi_{\mu_V^T(1)}, \ldots, \varphi_{\mu_V^T(|I_V(T)|)}$ living on the reference element $\hat{T}$ for the velocity space and the pressure space $\psi_{\mu_P^T(1)}, \ldots, \psi_{\mu_P^T(|I_P(T)|)}$. Together with the reference mapping $\Phi^T : \hat{T} \to T$ we get the base functions $\varphi_{\mu_V^T(i)} = \Phi(\varphi_{\mu_V^{\hat{T}}(i)})$ for $i = 1, \ldots, |I_V(T)|$ and $\psi_{\mu_P^T(i)} = \Phi(\psi_{\mu_P^{\hat{T}}(i)})$ for $i = 1, \ldots, |I_P(T)|$. The gradients of the velocity basefunctions and the function values of the pressure basefunctions are cached in line **??**, 266 and written into the local matrix in line 289.

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. Tho get the influence of the boundary values to the system of equations, we apply the divergence matrix to this boundary data 14 line 343. This influence is substracted from the right hand side function. Then the columns of the divergence matrix corresponding to the boundary DOFs are cleared in line 351.
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[1]. 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.

---

[1] 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. Klöfkorn, 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. Klöfkorn, 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. Klöfkorn, 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`.

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