Introduction to dune-fem

The Freiburg DUNE Group*

January 22, 2009

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

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

Dieses Dokument stellt in der vorliegenden Form keine vollständige Referenz in `dune-fem` dar, sondern es soll als eine Art roter Faden dienen, damit `DUNE`-Neulinge eine Idee bekommen, was alles möglich ist und wo man anfangen kann. Das Problem aus meiner Sicht ist, dass die vorhandene Dokumentation auf zu viele verschiedene Orte verteilt und teilweise nur schwer zu finden ist. Dementsprechend fasst dieses Dokument zu einem grossen Teil einfach Informationen aus verschiedenen Quellen zusammen oder verweist auf andere Quellen.

Der erste Teil von Kap.1 ist fast wörtlich von der `DUNE` Homepage kopiert. Kapitel 2 verwendet die Programmbeispiele aus der Dune Summer School 2007 von R. Klöfkorn, und Kapitel 3 ebenso.

Worauf also Wert gelegt wurde, sind einfache Beispiele und viele Verweise auf weitere Dokumentation und Hilfedateien, insbesondere also auf die `doxygen`-Dokumentation.

Freiburg, Januar 2008

Contents

1 What is DUNE?

DUNE, the Distributed and Unified Numerics Environment is a modular toolbox for solving partial differential equations with grid-based methods. It supports easy discretization using methods like Finite Elements, Finite Volumes, and also Finite Differences.

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 (i.e. grids, solvers, ...) using a common interface with a very low overhead. More: http://www.dune-project.org/dune.html (1).

1.1 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 library to handle dense matrices and vectors.
- dune-grid
  is the most mature module. It defines nonconforming, hierarchically nested, multi-element-type, parallel grids in arbitrary space dimensions. Graphical output with several packages is available, e.g. file output to IBM data explorer and 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 these classes. A special feature is the use of templates to exploit the recursive

block structure of finite element matrices at compile time. Available solvers include Krylov methods, (block-) incomplete decompositions and aggregation-based algebraic multigrid.

1.2 Available Grid Implementations

So far six grid implementations are available through the dune-gridinterface. Each is geared towards a different purpose.

- ONEDGRID: A sequential locally adaptive grid in one space dimension
- SGRID: A structured grid in n space dimensions
- YASPGRID: A structured parallel grid in n space dimensions (used as default grid)
- ALUGRID_CUBE, ALUGRID_SIMPLEX: A parallel locally adaptive grid in 2 and 3 space dimensions
- ALBERTAGRID: The grid manager of the Alberta toolbox
- UGGRID: The grid manager of the UG toolbox (UG is not freely available)

More information about these grids: http://www.dune-project.org/doc/devel/features.html.

1.3 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, see http://www.dune-project.org/doc/installation-notes.html.

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

```
mkdir dune
cd dune
```

- Checkout the necessary DUNE repositories via svn. To do this, type the following commands in your DUNE directory:

Listing 1 (File ./installation/checkout.sh)

```
svn checkout https://svn.dune-project.org/svn/dune-common/releases/1.1 dune-
    common
svn checkout https://svn.dune-project.org/svn/dune-grid/releases/1.1 dune-
    grid
svn checkout https://dune.mathematik.uni-freiburg.de/svn/dune-fem/release
    -0.9.1 dune-fem

svn checkout https://svn.dune-project.org/svn/dune-grid-howto/releases/1.1
    dune-grid-howto
svn checkout https://dune.mathematik.uni-freiburg.de/svn/dune-femhowto dune-
    femhowto
```

Alternatively, you can download the corresponding tar.gz archives from `http://www.dune-project.org/download.html` and `http://dune.mathematik.uni-freiburg.de` and extract them into your `DUNE` directory.

- Create a file named `config.opts` with the following content in your `DUNE` directory. Consider this file as an example, that will work correctly only for members of the Section of Applied Mathematics in Freiburg! Other users have to adapt (or to comment out) at least the paths to the external modules.

Listing 2 (File ./installation/config.opts)

```
#Standard flags. Used as default!
STDFLAGS="-O3␣-Wall␣-DNDEBUG␣-funroll-loops␣-finline-functions"

#More optimizing flags.
#The last option (-march=opteron) has to be adapted to your processor type,
    or just leave it out.
OPTIMFLAGS="-O3␣-Wall␣-DNDEBUG␣-funroll-loops␣-finline-functions␣\
--param␣max-inline-insns-single=4000␣\
--param␣large-function-growth=4500␣\
--param␣inline-unit-growth=4000␣\
-ffast-math␣-fomit-frame-pointer␣-msse3␣-mfpmath=sse␣-march=opteron"

#Debugging flags.
DEBUGFLAGS="-g␣-Wall"

#Paths to modules installed in Freiburg. Works only for internal users.
MODDIR="/hosts/morgoth/raid5/dune/modules_$HOSTTYPE/"
#MODDIR="/hosts/raid5/aragorn/robertk/DuneAdds/modules_x86_64"
MODULEFLAGS="--with-alberta=$MODDIR/alberta␣\
--with-alugrid=$MODDIR/alugrid␣\
--with-ug=$MODDIR/ug␣\
--disable-parallel␣\
--x-includes=/usr/X11R6/include␣\
--x-libraries=/usr/X11R6/lib␣\
--with-grape=$MODDIR/grape"
```

```
#Choose CXXFLAGS to one of the above flag options ($STDFLAGS or $DEBUGFLAGS
    or $OPTIMFLAGS)
#Remove the option "--disable-documentation" if you want a local doxygen-
    documentation.
CONFIGURE_FLAGS="--with-grid-dim=2 --disable-documentation CXX=g++ CXXFLAGS
    =\"$STDFLAGS\" $MODULEFLAGS"

MAKE_FLAGS=
```

- Finally, configure und compile DUNE using the dunecontrol script: Type in your DUNE directory:

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

  The script needs some minutes to finish. After finishing, you can start working!

You can find the file config.opts and shell scripts with the commands described above under http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/freiburg_intern/tools.html or in the subdirectory doc/installation of the dune-femhowto module.

1.4 Create your own project

You can create your own DUNE project by using the duneproject script. Type in your DUNE directory

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

and follow the instructions. After creating your project, you have to rerun the dunecontrol script:

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

If you want to write your own code, replace the sample source code with your own code, and modify the file Makefile.am, if necessary. You can also create your own subdirectories, but then you additionally have to modify the file configure.ac and run again dunecontrol in order to create the makefiles in the subdirectories. If you are unsure how to do these modifications, look how this is done in the core DUNE Modules, or read the DUNE Build System Howto on http://www.dune-project.org/doc/buildsystem/buildsystem.pdf.

2 Short Introduction to the dune-grid Interface

This chapter gives a really short introduction to dune-grid. For a more detailed explanation please read the DUNE GRID HOWTO (http://www.dune-project.org/doc/grid-howto/grid-howto.pdf).

Understanding code can be sometimes difficult. The DUNE Coding Style document can help you to understand DUNE code, see http://www.dune-project.org/doc/devel/codingstyle.html.

The source code to all of the following examples is shipped with this documentation, see directory dune-femhowto/src_grid.

2.1 The dune hello world programm

Listing 3 (File ../src_grid/dune_hello_world.cc)

```
1  #ifdef HAVE_CONFIG_H
2  # include "config.h"
3  #endif
4  #include <iostream>
5  #include"dune/common/mpihelper.hh" // An initializer of MPI
6  #include"dune/common/exceptions.hh" // We use exceptions
7
8  int main(int argc, char** argv)
9  {
10   try{
11     //Maybe initialize Mpi
12     Dune::MPIHelper& helper = Dune::MPIHelper::instance(argc, argv);
13     std::cout << "Hello␣World!␣This␣is␣DUNE." << std::endl;
14     if(Dune::MPIHelper::isFake)
15       std::cout<< "This␣is␣a␣sequential␣program." << std::endl;
16     else
17       std::cout<<"I␣am␣rank␣"<<helper.rank()<<"␣of␣"<<helper.size()
18         <<"␣processes!"<<std::endl;
19     return 0;
20   }
21   catch (Dune::Exception &e){
22     std::cerr << "Dune␣reported␣error:␣" << e << std::endl;
23   }
```

```
24   catch (...){
25     std::cerr << "Unknown␣exception␣thrown!" << std::endl;
26   }
27 }
```

This little programm has a sequentiell and a parallel part. The parallel part is regarded in section 2.4. In fact, this program just do nothing. But you can check whether your DUNE installation is working. Output:

```
Hello World! This is DUNE.
This is a sequential program.
```

2.2 Getting started with your first grid

Now we want to create our first grid:

Listing 4 (File ../src_grid/gettingstarted.cc)

```
1  // Dune includes
2  #include <config.h>           // file constructed by ./configure script
3  #include <dune/grid/sgrid.hh> // load sgrid definition
4  #include <dune/grid/common/gridinfo.hh> // definition of gridinfo
5  #include <dune/common/mpihelper.hh> // include mpi helper class
6
7  template <int dim>
8  struct Info
9  {
10   static void print()
11   {
12     std::cout << "**************************************************" << std::endl
         ;
13     std::cout << "Print␣grid␣infos␣for␣dim␣=␣" << dim << std::endl;
14     std::cout << "**************************************************" << std::endl
         ;
15
16     // make a grid
17     typedef typename Dune::SGrid<dim,dim> GridType;
18     Dune::FieldVector<int,dim> N(1);
19     Dune::FieldVector<typename GridType::ctype,dim> L(0.0);
20     Dune::FieldVector<typename GridType::ctype,dim> H(1.0);
21     GridType grid(N,L,H);
22
23     // print some information about the grid
24     std::cout << "gridinfo:" << std::endl;
25     Dune::gridinfo(grid);
26     std::cout << std::endl;
27
```

```
28      // print level list
29      std::cout << "gridlevellist:" << std::endl;
30      Dune::gridlevellist(grid,0,"LevelList ");
31      std::cout << std::endl;
32
33      // print leaf list
34      std::cout << "gridleaflist:" << std::endl;
35      Dune::gridleaflist(grid,"LeafList ");
36
37      std::cout << std::endl << std::endl;
38    }
39 };
40
41 // main routine
42 int main(int argc, char **argv)
43 {
44   // start try/catch block to get error messages from dune
45   try{
46     // initialize MPI, finalize is done automatically on exit
47     Dune::MPIHelper::instance(argc,argv);
48
49     // print grid info for dim = 1
50     Info<1>::print();
51
52     // print grid info for dim = 2
53     Info<2>::print();
54
55     // print grid info for dim = 3
56     Info<3>::print();
57   }
58   catch (std::exception & e) {
59     std::cout << "STL ERROR: " << e.what() << std::endl;
60     return 1;
61   }
62   catch (Dune::Exception & e) {
63     std::cout << "DUNE ERROR: " << e.what() << std::endl;
64     return 1;
65   }
66   catch (...) {
67     std::cout << "Unknown ERROR" << std::endl;
68     return 1;
69   }
70
71   // done
72   return 0;
73 }
```

Here we create an SGRID (lines 17-21) for several space dimensions (lines 50,53 and 56) und print some information about these grids. Notice we use generic programming:

```
template <int dim>
struct Info
```

```
{ ... };
```

Interesting is the number of codimensions in the several space dimensions. We just created an unitcube for the dimension 1, 2 and 3.

- 1D: We have one codim 0 element (the unit intervall) and two codim 1 elements (the end points of the intervall)

```
=> SGrid(dim=1,dimworld=1)
level 0 codim[0]=1 codim[1]=2
leaf    codim[0]=1 codim[1]=2
```

- 2D: We have one codim 0 element (the unit square), four codim 1 elements (edges) and four codim 2 elements (vertices).

```
=> SGrid(dim=2,dimworld=2)
level 0 codim[0]=1 codim[1]=4 codim[2]=4
leaf    codim[0]=1 codim[1]=4 codim[2]=4
```

- 3D: We have one codim 0 element (the unit cube), six codim 1 elements (faces), twelve codim 2 elements (edges) and eight codim 3 elements (vertices).

```
=> SGrid(dim=3,dimworld=3)
level 0 codim[0]=1 codim[1]=6 codim[2]=12 codim[3]=8
leaf    codim[0]=1 codim[1]=6 codim[2]=12 codim[3]=8
```

Because we did't refined the grid, there only exists a level 0 grid, and this level grid is the same as the leaf grid. You have access to all these different codim elements via iterators, even after refining you have access to all elements of the different levels.

For a more detailed explanation please read the DUNE GRID HOWTO (http://www.dune-project.org/doc/grid-howto/grid-howto.pdf)

2.3 The DGF-Parser: Using different grids / macrogrids

DUNE is designed to easily handle with different grid implementations. Therefore it is really easy to change the used grid. To show this fact we take a look to the following code:

Listing 5 (File ../src_grid/dgfparser.cc)

```
1 #include "config.h"              // know what grids are present
2 #include <iostream>              // for input/output to shell
3 #include <fstream>               // for input/output to files
```

```
4  #include <vector>                    // STL vector class
5  #include <dune/common/mpihelper.hh> // include mpi helper class
6
7  #include <dune/grid/common/gridinfo.hh>
8  #include <dune/grid/io/file/vtk/vtkwriter.hh> // VTK output routines
9
10 // checks for defined gridtype and includes appropriate dgfparser implementation
11 #include <dune/grid/io/file/dgfparser/dgfgridtype.hh>
12
13 // for manual grid definition
14 #include <dune/grid/yaspgrid.hh>
15 #include <dune/grid/io/file/dgfparser/dgfyasp.hh>
16
17
18 void dgfTest ()
19 {
20   using namespace Dune;
21
22   // use unitcube from grids
23   std::stringstream dgfFileName;
24   dgfFileName << "./unitcube2.dgf";
25
26   std::cout << "Try␣to␣open␣" << dgfFileName.str() << std::endl;
27
28   // create grid
29   typedef YaspGrid<2,2> GridType;
30   GridPtr<GridType> gridPtr ( dgfFileName.str() );
31
32   // print info
33   gridinfo( *gridPtr );
34   std::cout << std::endl;
35
36   // write grid in VTK format
37   VTKWriter<GridType> vtkwriter(*gridPtr);
38   vtkwriter.write("grid1",Dune::VTKOptions::ascii);
39 }
40
41 void dgfGridType ()
42 {
43   using namespace Dune;
44
45   // use unitcube from grids
46   std::stringstream dgfFileName;
47   // GridType is defined by dgfgridtype.hh
48   dgfFileName << "./unitcube" << GridType :: dimension << ".dgf";
49
50   std::cout << "Try␣to␣open␣" << dgfFileName.str() << std::endl;
51
52   // create grid pointer
53   GridPtr<GridType> gridPtr( dgfFileName.str() );
54
55   // grid reference
56   GridType& grid = *gridPtr;
```

```
57
58    // half grid width 4 times
59    int level = 4 * DGFGridInfo<GridType>::refineStepsForHalf();
60
61    // refine grid until upper limit of level
62    grid.globalRefine(level);
63
64    // print info
65    gridinfo( *gridPtr );
66    std::cout << std::endl;
67
68    // write grid in VTK format
69    VTKWriter<GridType> vtkwriter(*gridPtr);
70    vtkwriter.write("grid2",Dune::VTKOptions::ascii);
71  }
72
73  //===============================================================
74  // main rountine
75  //===============================================================
76  int main (int argc , char ** argv)
77  {
78    // initialize MPI, finalize is done automatically on exit
79    Dune::MPIHelper::instance(argc,argv);
80
81    // start try/catch block to get error messages from dune
82    try {
83      using namespace Dune;
84
85      // use manual typedef
86      dgfTest();
87
88      // use grid type definition
89      dgfGridType();
90    }
91    catch (std::exception & e) {
92      std::cout << "STL ERROR: " << e.what() << std::endl;
93      return 1;
94    }
95    catch (Dune::Exception & e) {
96      std::cout << "DUNE ERROR: " << e.what() << std::endl;
97      return 1;
98    }
99    catch (...) {
100     std::cout << "Unknown ERROR" << std::endl;
101     return 1;
102   }
103
104   // done
105   return 0;
106 }
```

By default, programs are compiled using the options GRIDTYPE=YASPGRID GRIDDIM=3. If you want to change the grid type or the number of space dimensions, you just have to recompile the program again using some other options for make, for example: make clean GRIDTYPE=ALUGRID_SIMPLEX GRIDDIM=3. How you have to implement this feature is shown in dgfGridType(), lines 45-56. Attention: For changing your grid in the described way, you must define GRIDTYPE somewhere in an makefile.am or in your configuration options during installation!

Another possibility is shown in dgfTest(), lines 22-30. Here the used grid is hardcoded in the source code (line 29). This is similar to the version in Listing 4. The only difference is that you use there an SGrid without macrogrids and without the DGF-Parser.

For more information: See DGF-Parser documentation (Link is below). We also refer to the list of the available grid implementations (1.2).

Are you bored about these stupid unit cubes? You can change your macrogrid just by editing the corresponding dgf files. How this can be done is explained in the documentation of the DGF-Parser. This documentation you find in your local dune-grid documentation, under the point I/O - The Dune Grid Format (DGF), or online under http://www.dune-project.org/doc/doxygen/dune-grid-html/group__DuneGridFormatParser.html.

You can find the example dgf-files from that documentation in your local dune-fem installation, in the subdirectory macrogrids/DGFMacrogrids.

The program above also contains an output routine, which writes the grid information into a vtk file (lines 37-38 and 69-70). You can visualize your grid by opening these files with paraview (http://www.paraview.org/) or another VTK-Viewer.

Excercise 1 Play with different dimensions, different grids implementations and different macrogrids, and visualize the results with paraview!

2.4 The parallel dune_hello_world programm

Now we want to run the dune_hello_world (listing 3) programm parallel on several computers.

To do that, you have to recompile the package with a modified config.opts. Replace in listing 2 --disable-parallel with --enable-parallel (line 20) and CXX=g++ with CXX=mpiCC (line

27). After that, recompile the dune-fem HOWTO with your modified config.opts, which ist now called config_parallel.opts:

```
./dune-common/bin/dunecontrol --opts=config_parallel.opts --only=dune-femhowto
    all
```

Notice: By using the --only=XXXX option, dunecontrol will only work on the module XXXX.

1. Running the program on one computer:

   ```
   ./dune_hello_world
   ```

   Output:

   ```
   Hello World! This is DUNE.
   I am rank 0 of 1 processes!
   ```

   Compare that with the output of the serial version!

2. Running on several computers using the "Simple Linux Utility for Resource Managment (SLURM)":

   ```
   srun -A -n 4
   mpirun nice ./dune_hello_world
   exit
   ```

   Here we use 4 machines (-n 4) for running the program, resulting in an output similar to this:

   ```
   Hello World! This is DUNE.
   Hello World! This is DUNE.
   I am rank 0 of 5 processes!
   Hello World! This is DUNE.
   I am rank 3 of 5 processes!
   I am rank 1 of 5 processes!
   Hello World! This is DUNE.
   I am rank 2 of 5 processes!
   Hello World! This is DUNE.
   I am rank 4 of 5 processes!
   ```

   Don't forget the final exit to terminate your job!!!!

3. Running on several computer without using SLURM:

   ```
   nice mpirun -np 4 ./dune_hello_world
   ```

   The output will be the same as above. Using SLURM has some advantages, for example it can avoid conflicts when running several jobs on one machine. So better use it! More information: SLURM manpages (type: man slurm).

You should also use `nice` like in the examples above to adjust scheduling priority.

Some maybe helpful commands:

- `squeue`: Returns the list of jobs managed by SLURM (and their job IDs).
- `scancel <JOB ID>`: Cancels the corresponding job.
- `sinfo`: Print some general information.

Attention: Only YaspGrid and AluGrid are able to perform parallel computations!

For real applications please read section 8 in the DUNE GRID HOWTO (`http://www.dune-project.org/doc/grid-howto/grid-howto.pdf`).

More information: `dune-common` documentation - `Parallel Communication` (`http://www.dune-project.org/doc/doxygen/dune-common-html/group__ParallelCommunication.html`).

2.5 Implementation of a Finite Volume Scheme with dune-grid

You can find different implementations of a Finite Volume Scheme in the DUNE GRID HOWTO (`http://www.dune-project.org/doc/grid-howto/grid-howto.pdf`).

- Basic scheme: Section 6.3
- Adaptive scheme: Section 7.2
- Parallel scheme: Section 8.3

3 The Poisson-Example with `dune-fem`

In this chapter we want to show how to use `dune-fem` for solving a simple test problem. As a test problem, we choose the Poisson problem:

$$
\begin{aligned}
-\triangle u &= f \quad \text{in } \Omega && (3.1)\\
u &= g \quad \text{on } \partial\Omega && (3.2)
\end{aligned}
$$

We want to solve this problem with the finite element method using Lagrangian elements. The implementation of this method is done in the files in the directory `dune-femhowto/src_poisson`.

The numerical treatment of this problem is described in chapter 1 of this skript (in german): `http://www.mathematik.uni-freiburg.de/IAM/Teaching/ubungen/sci_com_SS06/skriptum.pdf`.

In this example, we choose the following problem data:

$$
\begin{aligned}
\Omega &:=]0,1[^{dimworld}, \quad x = (x_1, ..., x_{dimworld}) && (3.3)\\
f(x) &:= 2 \sum_{i=1}^{dimworld} \prod_{j \neq i} (x_j - x_j^2) \quad \forall x \in \Omega && (3.4)\\
u(x) &:= \prod_{i=1}^{dimworld} (x_i - x_i^2) \quad \forall x \in \Omega && (3.5)\\
g(x) &:= u(x) \quad \forall x \in \partial\Omega && (3.6)
\end{aligned}
$$

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

Note: In the current implementation we implemented another right hand side f.

$$
\begin{aligned}
f(x,y) &:= 8\pi^2 \cos(2\pi x)\cos(2\pi y) \quad \forall x \in \Omega && (3.7)\\
u(x,y) &:= \cos(2\pi x)\cos(2\pi y) \quad \forall x \in \Omega && (3.8)\\
g(x,y) &:= u(x,y) \quad \forall x \in \partial\Omega && (3.9)
\end{aligned}
$$

The second problem data should better be used only with `dimworld=2`. For changing between these two versions, please modify `problem.hh`.

3.1 Implementing the algorithm and the problem data

3.1.1 The algorithm in `lagrange.cc`

The function `main()`

We want to skip all the `includes` at the beginning of the file and jump directly to the `main` routine.

- Initialize grid and some more initialization stuff
- Call `Algorithm::calc()` twice for EOC calculation:

```
for( int i=0; i<2; ++i )
{
    // calculate L2-projection
    projectionError[i] =
       Algorithm<GridType, L2ProjectionProblem, MassOperator>
          ::calc(grid,"l2-projection", false);

    // calculate poisson problem
    poissonError[i] =
       Algorithm<GridType, PoissonProblem, LaplaceOperator>
          ::calc(grid,"poisson",true);

    // make refinement for next step
    grid.globalRefine( DGFGridInfo<GridType>::refineStepsForHalf() );
}
```

- Calculate EOC:

```
const double poissonEOC =
               log( poissonError[0] / poissonError[1] ) / M_LN2;
const double projectionEOC =
               log( projectionError[0] / projectionError[1] ) / M_LN2;
```

`L2ProjectionProblem` and `PoissonProblem` are defined in `problem.hh`, `MassOperator` and `LaplaceOperator` are defined in `massmatrix.hh` respective `laplace.hh`. These files will be explained in detail later.

The struct Algorithm

```
template < class GridImp ,
           template <class> class ProblemImp ,
           template <class> class OperatorImp >
struct Algorithm
```

This struct contains some (important) typedefs and the method `calc()`, which holds the algorithm. Explanation of the typedefs:

- `typedef GridImp GridType`
  New name for template parameter.
- `typedef LeafGridPart<GridType> GridPartType`
  A `GridPart` is a subset of entities of the whole grid. Here our `GridPart` is build up by all leaf elements.
- `FunctionSpace<double, double, GridType::dimension, 1> FunctionSpaceType`
  The analytical functionspace we want to use: We use functions $f : \mathbb{R}^{dimension} \to \mathbb{R}$, where Elements from $\mathbb{R}$ are represented by `doubles`.
- `typedef LagrangeDiscreteFunctionSpace<FunctionSpaceType, GridPartType, POLORDER>`
  `DiscreteFunctionSpaceType`
  The discrete function space we want to use, here the lagrange space.
- `typedef AdaptiveDiscreteFunction<DiscreteFunctionSpaceType> DiscreteFunctionType`
  Build up discrete functions from the discrete function space.
- `typedef ProblemImp<FunctionSpaceType> ProblemType`
  `typedef typename ProblemType::RHSFunctionType RHSFunctionType`
  `typedef typename ProblemType::ExactSolutionType ExactSolutionType`
  The problem data and data functions (as defined in problem.hh)
- `typedef OperatorImp<DiscreteFunctionType> OperatorType`
  New name for template parameter (will be the laplace or the massmatrix operator).
- `typedef OEMBICGSTABOp<DiscreteFunctionType,OperatorType> InverseOperatorType`
  Define the type of inverse operator we are using to solve the system. There are different inverse operators available.

Explanation of the algorithm in the method calc()

```
static double calc (GridType& grid,
                    const std::string name,
                    bool verbose = false )
```

- Create instances of the types defined above:
  - `discreteFunctionSpace` of type `DiscreteFunctionSpaceType` (and print also some information)
  - `solution` and `rhs` of type `DiscreteFunctionType` (discrete functions for solution and right hand side)
  - `f` of type `RHSFunctionType` (analytical right hand side)
- Build right hand side (see boundary.hh):

```
assembleRHS<2*DiscreteFunctionSpaceType::polynomialOrder> ( f , rhs );
```

- Set the dirichlet boundary points to the corresponding values of the exact solution u by modifying `rhs` (see boundary.hh, problem.hh):

```
ExactSolutionType u( discreteFunctionSpace );
// ...
for( IteratorType it = discreteFunctionSpace.begin(); it != endit; ++it )
{
  boundaryTreatment( *it, u, rhs );
}
```

- Create instance of our problem operator (here: Laplace or MassMatrix Operator), see `laplace.hh` and `massmatrix.hh`.

```
OperatorType problemOperator( discreteFunctionSpace );
```

- Solve the linear system by using the inverse operator (defined above) of our `problemOperator`:

```
InverseOperatorType cg( problemOperator, 1e-8, 1e-8, 20000, verbose );
cg( rhs, solution );
```

  For more information see the doxygen documentation of `dune-fem` (Operators - Linear solver)
- Error calculation and graphical output.

3.1.2 The problem data in `problem.hh`

First we implement the problem data: The right hand side f and the exact solution u of our Poisson problem (see page 17). The exact solution is needed for the EOC calculation and also for setting the boundary values.

```
template <class FunctionSpaceType>
class myRHSFunction
    : public Function< FunctionSpaceType, myRHSFunction<FunctionSpaceType> >
{
  // Implementation...
};
```

```
template <class FunctionSpaceType>
class myExactSolution
    : public Function< FunctionSpaceType, myExactSolution<FunctionSpaceType> >
{
  // Implementation...
};
```

Note: These classes are derived from the `Function` class by using the Barton-Nackman-Trick in the second template parameter.

Now we put all problem data together in two classes:

```
// right hand side and exact solution for poisson problem
template <class FunctionSpaceType>
struct PoissonProblem
{
    // type of right hand side
    typedef myRHSFunction<FunctionSpaceType> RHSFunctionType;

    // type of exact solution
    typedef myExactSolution<FunctionSpaceType> ExactSolutionType;
};

// for the L2-Projection use the exact solution also for the right hand side
template <class FunctionSpaceType>
struct L2ProjectionProblem
{
    // type of right hand side, use exact solution(!)
    typedef myExactSolution<FunctionSpaceType> RHSFunctionType;

    // type of exact solution
    typedef myExactSolution<FunctionSpaceType> ExactSolutionType;
};
```

3.1.3 Right hand side assembler and boundary treatment in `boundary.hh`

You can find more details about this topic in chapter 1 of this skript (in german): `http://www.mathematik.uni-freiburg.de/IAM/Teaching/ubungen/sci_com_SS06/skriptum.pdf`.

The function `assembleRHS()`

This function assembles the right hand side for a given `function` and returns the result in `discreteFunction`. The (given) right hand side function f itself is implemented in `problem.hh`.

```
template< int polOrd,
          class FunctionType,
          class DiscreteFunctionType >
static void assembleRHS( const FunctionType &function,
                         DiscreteFunctionType &discreteFunction )
{
// Implementation...
}
```

The function `boundaryTreatment()`

Sets the dirichlet points to the corresponding values of `boundaryValue`. `boundaryValue` can be arbitrary, but `langrange.cc` uses the (given) exact solution u for setting the boundary values. The exact solution u itself is implemented in `problem.hh`.

```
template< class EntityType,
          class FunctionType,
          class DiscreteFunctionType >
static void boundaryTreatment( const EntityType &entity,
                               const FunctionType& boundaryValue,
                               DiscreteFunctionType &rhs )
{
// Implementation...
}
```

3.2 Implementing Laplace- and MassOperator

Remember the following lines of the algorithm in `lagrange.cc`:

```
// type of inverse operator
typedef OEMBICGSTABOp<DiscreteFunctionType,OperatorType> InverseOperatorType;

// assemble right hand side and so on ...
// ...

// create problem operator (Laplace or MassMatrix Operator)
OperatorType problemOperator( discreteFunctionSpace );

// apply inverse operator (solve the linear system)
InverseOperatorType cg( problemOperator, 1e-8, 1e-8, 20000, verbose );
cg( rhs, solution );
```

This means, for solving our problem with the inverse operator `cg`, we first have to provide a suitable problem operator. How this can be done for the Poisson Problem is shown in the following subsections.

3.2.1 An abstract matrix operator in matrixoperator.hh

In this file the class MatrixOperator is implemented. This class is the base class for the problem operator classes LaplaceOperator and MassOperator, which are described below. The derived classes have to implement at least the (protected) method assembleLocalMatrix().

Private member variables

- const DiscreteFunctionSpaceType &discreteFunctionSpace_: reference to the used discrete function space.
- mutable MatrixObjectImp matrixObj_: pointer to the system matrix.
- mutable bool matrix_assembled_: flag indicating whether the system matrix has already been assembled.
- const bool boundaryCorrect_: true if boundary correction should be done.

The constructor MatrixOperator::MatrixOperator()

The constructor of this class is protected, so you cannot create any instances of this class.

```
MatrixOperator( const DiscreteFunctionSpaceType &discreteFunctionSpace, bool bnd )
    : discreteFunctionSpace_( discreteFunctionSpace ),
      matrixObj_( discreteFunctionSpace, discreteFunctionSpace , "" ),
      matrix_assembled_( false ),
      boundaryCorrect_(bnd)
{
}
```

Public methods

The public methods are all very easy.

- MatrixOperator::operator() ()
  MatrixOperator::multOEM()
  Apply the operator: Multiplies a vector with the system matrix and returns the result.

- `MatrixOperator::systemMatrix()`
  Returns the system matrix `matrixObj_`. Calls `MatrixOperator::assemble()` before, if needed.
- `MatrixOperator::preconditionMatrix()`
  Returns a reference to the preconditioning matrix.
- `MatrixOperator::print()`
  Prints the system matrix into a stream.
- `MatrixOperator::discreteFunctionSpace()`
  Returns the used discreteFunctionSpace.
- `MatrixOperator::assemble()`
  Allocates memory for the system matrix `matrixObj_` and then calls the method `MatrixOperator::assembleOnGrid()`.

Protected methods

- `MatrixOperator::assembleOnGrid()`
  Runs over all entities, calls on each entity `MatrixOperator::assembleOnEntity()` and calls, if needed, `MatrixOperator::boundaryCorrectOnEntity()`. Furthermore, prints out the time needed for the hole matrix to assemble.
- `MatrixOperator::assembleOnEntity()`
  Performs matrix assemble for one entity. Creates for that entity a local matrix and calls `MatrixOperator::assembleLocalMatrix()` on the entity.
- `MatrixOperator::assembleLocalMatrix()`
  Uses the Barton-Nackman Trick to avoid using virtual functions: Just calls the corresponding method `assembleLocalMatrix()` in the derived class. The derived class has to implement this method!
- `MatrixOperator::boundaryCorrectOnEntity()`
  Modifies the system matrix in order to implement the dirichlet boundary condition. Compare with the function `boundaryTreatment()` on page 22, where the corresponding thing was done for the right hand side.

Private methods

- `MatrixOperatorType& asImp()`
  `const MatrixOperatorType& asImp() const`

Performs the Barton-Nackman Trick to avoid virtual functions. Looks strange, but works, so don't worry about!

3.2.2 The Laplace operator in laplace.hh

The struct LaplaceOperatorTraits at the beginning of the file contains only some typedefs, so let's start directly with the class LaplaceOperator.

```
template< class DiscreteFunctionImp >
class LaplaceOperator : public MatrixOperator< LaplaceOperatorTraits<
    DiscreteFunctionImp > >
```

This class is derived from the class MatrixOperator (see matrixoperator.hh) to implement the Laplace operator. The typedefs at the beginning of the class are not so important.

Most of the work is already be done in the base class, here we have only to implement the method LaplaceOperator::assembleLocalMatrix(). For details about that method we refer again to this skript: http://www.mathematik.uni-freiburg.de/IAM/Teaching/ubungen/sci_com_SS06/skriptum.pdf, chapter 1.3.2.

3.2.3 The mass matrix operator in massmatrix.hh

This file holds another class derived from MatrixOperator, but this time to implement the mass matrix operator. The hole file is very similar with the file laplace.hh, only the method MassOperator::assembleLocalMatrix() is (of course) different.

4 An LDG solver for Advection-Diffusion Equations

This chapter is to introduce the `Pass` concept in `dune-fem` and to explain the steps that are necessary in order to implement a Local Discontinuous Galerkin solver using the `LocalDGPass` class. In the end it is shown how the solution can be analysed visually with help of the `DataWriter` class.

4.1 Advection-Diffusion Equation

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

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

with u living in a function space V and $\mathbf{a} := (0.8, 0.8, 0)^t$. An exact solution is specified by

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

where

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

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

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

Therefore the initial and boundary functions are defined by

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

Note that the problem is also implemented for 2 dimension by projecting everything onto the first two coordinates. The discretization of the problem is done as described in (2, ch.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 Discontinous Galerkin function spaces $V_h := \{\varphi_1, \dots \varphi_n\}$ resp. $[V_h]^3 = \{\psi_1, \dots \ \psi_{n^3}\}$ with discrete domain in space. The equations in (4.7) can then be rewritten as a variational formulation

$$\int_T u_1\psi = \int_T \underbrace{\sqrt{\varepsilon}u_0}_{f_1(u_0)} \nabla \cdot \psi - \int_{\partial T} \underbrace{\sqrt{\varepsilon}u_0}_{\tilde{f}_1(u_0)} \psi \cdot n \qquad \forall \psi \in [V_h]^3 \tag{4.8}$$

$$\int_T u_2\varphi = \int_T \underbrace{\left(\mathbf{a}u_0 + \sqrt{\varepsilon}u_1\right)}_{f_2(u_0,u_1)} \cdot \nabla\varphi - \int_{\partial T} \underbrace{\left(\mathbf{a}u_0 + \sqrt{\varepsilon}u_1\right)}_{\tilde{f}_2(u_0,u_1)} \varphi \cdot n \qquad \forall \varphi \in V_h \tag{4.9}$$

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

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

define a mean respectively an 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. This allows to shorten the problem formulation to

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

This is an ODE in the time domain and 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 (2, ch.4.3).

4.2 Implementation overview

The source code of the implmentation described in this section is located in the directory `dune-femhowto/src\_localdg`. It consists of 7 source files:

- `problem.hh` contains the class `U0` which defines the exact solution u as defined in (4.2) and g_D, u_0 as defined in (4.6). Those can be called through the following methods:

```
94    }
111   }
```

- `models.hh` contains the class `AdvectionDiffusionModel` that describes 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 to these classes see section 4.4.
- `advectdiff.hh` provides the LDG Operator `DGAdvectionDiffusionOperator` that is constructed out of the models given in `discretemodels.hh`. For further details, refer to section 4.4.
- `dgtest.cc` is the source file for the main program and is explained in the next section.
- `datadisp.cc` and `stuff.cc` are functions with helper class for pretty printing of EOC Data respectively data visualisation. Further details on this topic can be found in section 4.5.

4.3 Main Loop

The major typedefs for problem and model definitions are included via the `models.hh` header file. If the program shall run with a different ODE solver e.g., the definitions

that can be seen in the header files excerpt in listing 6 need to be adapted.

Listing 6 (End of ../src_localdg/models.hh)

```
325   mutable DomainType velocity_;
326   const Model& model_;
327 };
328
329
330 /*********************************************
331  * Definition of model and solver             *
332  *********************************************/
333
334 // approximation order
335 const int order   = POLORDER;
336 const int rkSteps = POLORDER + 1;
337
338 // Choose a suitable GridView
339 typedef Dune::LeafGridPart<GridType> GridPartType;
340
341 // The initial function u_0 and the exact solution
342 typedef U0<GridType> InitialDataType;
343 // An analytical version of our model
344 typedef AdvectionDiffusionModel<GridPartType, InitialDataType> ModelType;
345 // The flux for the discretization of advection terms
346 typedef UpwindFlux<ModelType> FluxType;
347 // The DG Operator (using 2 Passes)
348 typedef DGAdvectionDiffusionOperator<ModelType,UpwindFlux,order> DgType;
```

The main loop can be found in the file `dgtest.cc`. An explanation of the most important lines in this file, follows after listing 7.

Listing 7 (../src_localdg/dgtest.cc)

```
1  /**
2   *  \file dgtest.cc
3   *  \brief dgtest.cc
4   **/
5
6  // Dune includes
7  #include <config.h>
8
9  #include <dune/common/fvector.hh>
10 #include <dune/fem/misc/utility.hh>
11 #include <dune/fem/gridpart/gridpart.hh>
12
13 #include <dune/common/misc.hh>
14 #include <dune/grid/common/quadraturerules.hh>
15
16 #include <dune/fem/misc/l2error.hh>
17
```

```
18 #include <iostream>
19 #include <string>
20
21 #include <dune/fem/io/parameter.hh>
22 #include <dune/fem/io/file/datawriter.hh>
23 #include <dune/common/timer.hh>
24 #include <dune/common/mpihelper.hh>
25
26 #include "models.hh"
27 // Include helper function for EOC LaTeX output and L2Projection
28 #include "stuff.cc"
29
30 using namespace Dune;
31
32 /**
33  * @brief main function for the LocalDG Advection-Diffusion application
34  *
35  * \c main starts the Simulation of an advection-diffusion pde with
36  * the localdg method with EOC analysis and visual output to grape, paraview or
37  * gnuplot.
38  * \attention The localdg implementation uses the \c Dune::Pass
39  * concept.
40  *
41  * @param argc number of arguments from command line
42  * @param argv array of arguments from command line
43  * @param envp array of environmental variables
44  * @return 0 we don't program bugs. :)
45  */
46 int main(int argc, char ** argv, char ** envp) {
47
48   /* Parallelization is not implemented */
49   MPIHelper::instance(argc,argv);
50
51   try {
52
53   // *** Initialization
54   Parameter::append(argc,argv);
55   if (argc == 2) {
56     Parameter::append(argv[1]);
57   } else {
58     Parameter::append("parameter");
59   }
60
61   // initialize grid
62   std::string filename;
63   Parameter::get("femhowto.localdg.gridFile", filename);
64   GridPtr<GridType> grid(filename); // ,MPI_COMM_WORLD);
65
66   // ------ read in runtime parameters -------
67   int eocSteps   = Parameter::getValue<int>("femhowto.localdg.eocSteps", 1);
68   int startLevel = Parameter::getValue<int>("femhowto.localdg.startLevel", 0);
69   int printCount = Parameter::getValue<int>("femhowto.localdg.printCount", -1);
70   int verbose    = Parameter::getValue<int>("femhowto.localdg.verbose", 0);
```

```
71  const double maxTimeStep =
72      Parameter::getValue("femhowto.localdg.maxTimeStep", std::numeric_limits<
            double>::max());
73  std::string eocOutPath = Parameter::getValue<std::string>("femhowto.localdg.
        eocOutputPath",
74                                                        std::string("."));
75  EocOutput eocOutput(eocOutPath + std::string("/eoc.tex"));
76
77  // ------ read in model parameters -------
78  double cfl; // CFL coefficient for SSP ODE Solver
79  switch (order)
80  {
81    case 0: cfl=0.9;  break;
82    case 1: cfl=0.2;  break;
83    case 2: cfl=0.15; break;
84    case 3: cfl=0.05; break;
85    case 4: cfl=0.09; break;
86  }
87  Parameter::get("femhowto.localdg.cfl", cfl, cfl);
88  double startTime   = Parameter::getValue<double>("femhowto.localdg.startTime",
        0.0);
89  double endTime     = Parameter::getValue<double>("femhowto.localdg.endTime",
        0.9);
90
91  std::cout << "␣CFL␣:␣" << cfl << std::endl;
92
93
94  // InitialDataType is a Dune::Operator that evaluates to $u_0$ and also has a
95  // method that gives you the exact solution.
96  InitialDataType problem;
97  // Initialize the model
98  ModelType model(problem);
99  // Initialize flux for advection discretization (UpwindFlux)
100 FluxType convectionFlux(model);
101
102 // Initialize Timer for CPU time measurements
103 Timer timer;
104 // Initialize variables needed for EOC computation
105 double prevTime  = 0.;
106 double error     = 0.;
107 double prevError = 0.;
108 int    level     = 0;
109 double maxdt     = 0.;
110 double mindt     = 1.e10;
111 double averagedt = 0.;
112
113 // Initialize L2Error for computation of error between discretized and exact
114 // solution
115 L2Error<DgType::DestinationType> L2err;
116 FieldVector<double,ModelType::dimRange> err;
117
118
119 // Refine the grid until the startLevel is reached
```

```
120   for(int level=0; level < startLevel ; ++level)
121     grid->globalRefine(DGFGridInfo<GridType>::refineStepsForHalf());
122
123   /*****************************************
124    * EOC Loop                               *
125    *****************************************/
126   for(int eocloop=0; eocloop < eocSteps; ++eocloop)
127   {
128     // Initialize DG Operator
129     DgType dg(*grid, convectionFlux);
130     // Initialize TimeProvider
131     GridTimeProvider< GridType > tp(startTime, cfl, *grid);
132     // Initialize ODE Solver needed for the time discretization (Runge-Kutta)
133     ODEType ode(dg, tp, rkSteps, verbose);
134
135     // Initialize the discrete Function $u$
136     DgType::DestinationType U("U", dg.space());
137     initialize(problem,U);
138
139     // Print problem info to the eocOutput file
140     if (eocloop==0) {
141       eocOutput.printInput(problem, *grid, ode, filename);
142     }
143
144     // Initialize the DataWriter that writes the solution on the harddisk in a
145     // format readable by Paraview e.g.
146     // IOTupleType is a Tuple of discrete functions
147     IOTupleType dataTup ( &U );

148     typedef DataWriter<GridType, IOTupleType> DataWriterType;
149     DataWriterType dataWriter( *grid, filename, dataTup, startTime, endTime );

150
151     dataWriter.write(0.0, 0);
152
153     FieldVector<double,ModelType::dimRange> projectionError =
154       L2err.norm(problem, U, startTime);
155     std::cout << "Projection␣error␣" << problem.myName << ":␣" << projectionError
          <<
156       std::endl;
157
158     /*****************************************
159      * Time Loop                              *
160      *****************************************/
161     tp.provideTimeStepEstimate(maxTimeStep);
162     // ode.initialize applies the DG Operator once in order to get an estimate for
          dt.
163     ode.initialize(U);
164     for( tp.init() ; tp.time() < endTime ; tp.next() )
165     {
166       tp.provideTimeStepEstimate(maxTimeStep);
167       const double tnow  = tp.time();
168       const double ldt   = tp.deltaT();
```

```
169       const int counter  = tp.timeStep();
170
171       /*********************************************
172        * Compute an ODE timestep                   *
173        *********************************************/
174       ode.solve(U);
175
176       if (!U.dofsValid()) {
177         std::cout << "Invalid␣DOFs" << std::endl;
178         if(eocloop == eocSteps-1) {
179           dataWriter.write(1e10, counter+1);
180         }
181         abort();
182       }
183
184       if(verbose > 1 && printCount > 0 && counter % printCount == 0) {
185         std::cout << "step:␣" << counter << "␣␣time␣=␣" << tnow << ",␣dt␣=␣" <<
              ldt << "\n";
186       }
187
188       if( eocloop == eocSteps -1 ) {
189         dataWriter.write(tnow, counter);
190       }
191
192       // some statistics
193       mindt = (ldt<mindt)?ldt:mindt;
194       maxdt = (ldt>maxdt)?ldt:maxdt;
195       averagedt += ldt;
196
197       // Abort if the ODE solver does not converge
198       if(counter%100 == 0)
199       {
200         err = L2err.norm(problem, U, tp.time());
201         if(err.one_norm() > 1e5 || ldt < 1e-10)
202         {
203           averagedt /= double(counter);
204           std::cout << "Solution␣doing␣nasty␣things!" << std::endl;
205           std::cout << tnow << std::endl;
206           eocOutput.printTexAddError(err[0], prevError, -1, grid->size(0), counter
                , averagedt);
207           eocOutput.printTexEnd(timer.elapsed());
208           exit(EXIT_FAILURE);
209         }
210       }
211
212     } /****** END of time loop *****/
213
214     averagedt /= double(tp.timeStep());
215     if(verbose > 3)
216     {
217       std::cout << "Minimum␣dt:␣" << mindt
218          << "\nMaximum␣dt:␣" << maxdt
219          << "\nAverage␣dt:␣" << averagedt << endl;
```

```
220     }
221
222     // Write solution to hd
223     if( eocloop == eocSteps-1 )
224     {
225       dataWriter.write(tp.time(), tp.timeStep());
226     }
227
228     // Compute L2 error of discretized solution ...
229     err = L2err.norm(problem, U, tp.time());
230     std::cout << "Error␣" << problem.myName << ":␣" << err << endl;
231     error       = err.two_norm();
232     double time = timer.elapsed() - prevTime;
233
234     // ... and print the statistics out to the eocOutputPath file
235     eocOutput.printTexAddError(error, prevError, time, grid->size(0), tp.timeStep
            (), averagedt);
236     prevTime  = time;
237     prevError = error;
238
239     // Stop if too much time passed by (energy prices are soo high!)
240     if(time > 3000.)
241       break;
242
243     // Refine the grid for the next EOC Step.
244     if(eocloop < eocSteps-1) {
245       grid->globalRefine(DGFGridInfo<GridType>::refineStepsForHalf());
246       ++level;
247     }
248   } /***** END of EOC Loop *****/
249
250   // Close the eocOutputPath file
251   eocOutput.printTexEnd(timer.elapsed());
252
253   Parameter::write("parameter.log");
254
255   }
256   catch (Dune::Exception &e) {
257     std::cerr << e << std::endl;
258     return 1;
259   } catch (...) {
260     std::cerr << "Generic␣exception!" << std::endl;
261     return 2;
262   }
263
264   return 0;
265 }
```

In lines 54-73 runtime options specified in a parameter file are read in with help of the `Dune::Parameter` class. The default name for the parameter file is “parameter” (as defined in line 58) and can be changed through the command line.

Lines 78-87 set the CFL constant that is needed to preserve strong stability of the ODE solver. This constant can also be specified explicitly in the parameterfile. It is known by the TimeProvider `tp` that in turn is given together with the discrete operator for the space domain `dg` to the ODE solver in lines 129 and 131. Each time the ODE solver calls one of its methods `initialize` or `solve` (see lines 163, 174), it evaluates the operator `dg` and afterwards asks the operator for a new time step estimate by calling its method `provideTimeStepEstimate`. This estimate multiplied by the CFL constant then turns out to be the next timestep length. Note that in lines 161 and 166 the maximum for this timestep length is bounded by the value in `maxTimeStep`, which is controlled via the parameterfile.

For visualisation of the found solution the program writes the time snapshots to the harddisk. This snapshot data can then be postprocessed by programs like GRAPE or Paraview. The user hast the ability to change the output directory for the snapshots, the interval at which the snapshots should be generated and the output format in the parameterfile. The options are all handled by an instance of the `dune-fem` class `DataWriter` that is created in lines 147-149 and requested to write snapshots on the harddisk in lines 189 and 225.

4.4 Setting up an LDGPass

This section is to shed light on the construction of the operator `dg` that obviously is the heart of this LDG solver.

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. A `LocalDGPass` solves an equation of the form

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

with the argument u and the computed solution v. Both required passes (see (4.7)) are in this form. A look at the weak formulation of equation (4.15) reveals what methods a model which is passed to the constructor of a `LocalDGPass` instance needs to specify:

$$\int_T v\varphi = -\int_{\partial T} \underbrace{(f(x,u) + A(x,u)\cdot n)}_{\substack{\texttt{numericalFlux}\\ \texttt{boundaryFlux}}}\varphi + \int_T \overbrace{f(x,u)}^{\texttt{analyticalFlux}}\cdot\nabla\varphi + \int_T \underbrace{(S - A(x,u))}_{\texttt{source}}\nabla u\varphi. \tag{4.16}$$

The header file `advectdiff.hh` defines the two models that implement these four methods `analyticalFlux`, `source`, `numericalFlux` and `boundaryFlux`. Each of these methods describes the discrete version of its corresponding term in the variational formulation (4.16). The second pass of this implementation e.g. has no `source` term, the `analyticalFlux` represents $f_2(u_0, u_1)$ (see (4.9)) and the `numericalFlux` resp. the `boundaryFlux` together represent the flux function $f_{2,h}(u_0, u_1)$ (see (4.11)), where the first one implements the flux on interfaces between inner cells and the second one on the boundary domain. Listing 8 shows the actual implementation of this model.

Listing 8 (AdvDiffDModel2 in ../src_localdg/discretemodels.hh)

```
254     const Model& model_;
255     const NumFlux& numflux_;
256     const double cflDiffinv_;
257   };
258
259   // Discrete Model for Pass2
260   template <class Model, class NumFlux, int polOrd, int passId1, int passId2>
261   class AdvDiffDModel2 :
262     public DiscreteModelDefault
263       <AdvDiffTraits2<Model, NumFlux, polOrd, passId1, passId2>, passId1, passId2
              >
264   {
265     // These type definitions allow a convenient access to arguments of pass.
266     Int2Type<passId1> u0Var;
267     Int2Type<passId2> u1Var;
268   public:
269     typedef AdvDiffTraits2<Model, NumFlux, polOrd, passId1, passId2> Traits;
270
271     typedef FieldVector<double, Traits::dimDomain> DomainType;
272     typedef FieldVector<double, Traits::dimDomain-1> FaceDomainType;
273
274     typedef typename Traits::RangeType RangeType;
275     typedef typename Traits::GridType GridType;
276     typedef typename Traits::JacobianRangeType JacobianRangeType;
277     typedef typename Traits::GridPartType::IntersectionIteratorType
278       IntersectionIterator;
279     typedef typename GridType::template Codim<0>::Entity EntityType;
280
281   public:
282     /**
283      * @brief constructor
284      */
285     AdvDiffDModel2(const Model& mod,const NumFlux& numf) :
286       model_(mod),
287       numflux_(numf)
288     {}
289
290     bool hasSource() const { return false; }
291     bool hasFlux() const { return true; }
292
```

```
293     /**
294      * @brief method required by LocalDGPass
295      *
296      * @param it intersection
297      * @param time current time given by TimeProvider
298      * @param x coordinate of required evaluation local to \c it
299      * @param uLeft DOF evaluation on this side of \c it
300      * @param uRight DOF evaluation on the other side of \c it
301      * @param gLeft result for this side of \c it
302      * @param gRight result for the other side of \c it
303      * @return time step estimate
304      */
305     template <class ArgumentTuple>
306     double numericalFlux(IntersectionIterator& it,
307                          double time, const FaceDomainType& x,
308                          const ArgumentTuple& uLeft,
309                          const ArgumentTuple& uRight,
310                          RangeType& gLeft,
311                          RangeType& gRight)
312     {
313       const DomainType normal = it.integrationOuterNormal(x);
314
315
316       /****************************
317        * Advection                *
318        ****************************/
319       // delegated to numflux_
320       double ldt;
321       ldt=numflux_.
322         numericalFlux(it, time, x, uLeft[u0Var], uRight[u0Var], gLeft, gRight);
323
324       /****************************
325        * Diffusion                *
326        ****************************/
327       JacobianRangeType diffmatrix;
328       RangeType diffflux(0.);
329       /* Central differences */
330       model_.
331         diffusion2(*it.inside(), time, it.intersectionSelfLocal().global(x),
332                    uLeft[u0Var], uLeft [u1Var], diffmatrix);
333       diffmatrix.umv(normal, diffflux);
334       model_.
335         diffusion2(*it.outside(), time, it.intersectionNeighborLocal().global(x),
336                    uRight[u0Var], uRight[u1Var], diffmatrix);
337       diffmatrix.umv(normal, diffflux);
338       diffflux*=0.5;
339
340       gLeft += diffflux;
341       gRight += diffflux;
342       return ldt;
343     }
344
345     /**
```

```
346       * @brief same as numericalFlux() but for the boundary
347       */
348      template <class ArgumentTuple>
349      double boundaryFlux(IntersectionIterator& it,
350                          double time, const FaceDomainType& x,
351                          const ArgumentTuple& uLeft,
352                          RangeType& gLeft)
353      {
354        const DomainType normal = it.integrationOuterNormal(x);
355
356        typedef typename ArgumentTuple::template Get<passId1>::Type UType;
357
358        double ldt=0.0;
359        if (model_.hasBoundaryValue(it,time,x))
360        {
361          /****************************
362           * Advection                *
363           ****************************/
364          UType uRight;
365      RangeType gRight;
366
367          // get the boundary value for upwind discretization
368      model_.boundaryValue(it, time, x, uLeft[u0Var], uRight);
369      ldt = numflux_.numericalFlux(it, time, x, uLeft[u0Var], uRight, gLeft, gRight)
             ;
370
371          /****************************
372           * Diffusion                *
373           ****************************/
374      JacobianRangeType diffmatrix;
375      model_.diffusion2(*it.inside(), time,
376                            it.intersectionSelfLocal().global(x),
377                            uLeft[u0Var], uLeft[u1Var], diffmatrix);
378      diffmatrix.umv(normal, gLeft);
379        } else {
380          // not implemented
381          assert(false);
382        }
383        return ldt;
384      }
385
386      /**
387       * @brief method required by LocalDGPass
388       */
389      template <class ArgumentTuple>
390      void analyticalFlux(EntityType& en,
391                          double time, const DomainType& x,
392                          const ArgumentTuple& u, JacobianRangeType& f)
393      {
394        /****************************
395         * Advection                *
396         ****************************/
397        model_.advection(en,time, x, u[u0Var],f);
```

```
398         /****************************
399          * Diffusion                 *
400          ****************************/
401         JacobianRangeType diffmatrix;
402         model_.diffusion2(en, time, x, u[u0Var], u[u1Var], diffmatrix);
403         f += diffmatrix;
```

Note that in line 290 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 to the flux function which for the second pass lives in the product of two function vector spaces. In order to access the correct component of the argument, there is an elegant solution using the `Int2Type` helper class: Every Pass gets a unique id when it is defined (see line 41) and the `Int2Type` class can convert this unique id to a unique type as done in lines 266 and 267. These types then allow `ArgumentTuple`s to be used like arrays, as in line 330 e.g., where `uLeft[u0Var]` returns $u_0(x^-)$ and `uLeft[u1Var]` returns $u_1(x^-)$. Alternatively there is a `Selector` class in `dune-fem` which also provides accessors to any component of a tuple but which is more cumbersome.

With models defined for both passes the implementation of the LDG operator becomes quite simple.

Listing 9 (../src_localdg/advectdiff.hh)

```
1  /**
2   *  \file   advectdiff.hh
3   *  \brief  advectdiff.hh
4   **/
5
6  #ifndef DUNE_DGOPERATORS_HH
7  #define DUNE_DGOPERATORS_HH
8
9  //- system includes
10 #include <string>
11
12 // Dune includes
13 #include <dune/fem/gridpart/gridpart.hh>
14
15 #include <dune/fem/pass/dgpass.hh>
16 #include <dune/fem/pass/discretemodel.hh>
17 #include <dune/fem/pass/selection.hh>
18 #include <dune/fem/space/dgspace.hh>
19 #include <dune/fem/operator/common/spaceoperatorif.hh>
20 #include "discretemodels.hh"
21
22 /************************************
23  * DG Operator                      *
```

```
24  ************************************/
25
26 namespace Dune {
27
28 /**
29  * @brief LDG Operator that is evaluated at every time step by the ODE Solver.
30  *
31  * LDG Operator \f$\mathcal{L}\f$ that implements the two Passes for the
32  * advection-diffusion problem
33  */
34 template <class Model,template<class M> class NumFlux,int polOrd >
35 class DGAdvectionDiffusionOperator :
36   public SpaceOperatorInterface<
37   typename PassTraits<Model, Model::Traits::dimRange, polOrd>::DestinationType>
38   {
39 public:
40   // Id's for the three Passes (including StartPass)
41   enum PassIdType{ u, pass1, pass2 };
42
43   enum { dimRange = Model::dimRange };
44   enum { dimDomain = Model::Traits::dimDomain };
45
46   typedef NumFlux<Model> NumFluxType;
47   typedef typename Model::Traits::GridType GridType;
48
49   // Pass 1 Model
50   typedef AdvDiffDModel1<Model, NumFluxType, polOrd, u>
51     DiscreteModel1Type;
52   // Pass 2 Model
53   typedef AdvDiffDModel2<Model, NumFluxType, polOrd, u, pass1>
54     DiscreteModel2Type;
55
56   typedef typename DiscreteModel1Type::Traits Traits1;
57   typedef typename DiscreteModel2Type::Traits Traits2;
58
59   typedef typename Traits2::DomainType DomainType;
60   typedef typename Traits2::DiscreteFunctionType DiscreteFunction2Type;
61
62   /************************************
63    * Join the Passes 0-2               *
64    ************************************/
65   typedef StartPass<DiscreteFunction2Type, u> Pass0Type;
66   typedef LocalDGPass<DiscreteModel1Type, Pass0Type, pass1> Pass1Type;
67   typedef LocalDGPass<DiscreteModel2Type, Pass1Type, pass2> Pass2Type;
68
69   typedef typename Traits1::DiscreteFunctionSpaceType
70     Space1Type;
71   typedef typename Traits2::DiscreteFunctionSpaceType
72     Space2Type;
73   typedef typename Traits1::DestinationType Destination1Type;
74   typedef typename Traits2::DestinationType Destination2Type;
75   typedef Destination2Type DestinationType;
76   typedef Space2Type SpaceType;
```

```
77
78    typedef typename Traits1::GridPartType GridPartType;
79
80  public:
81    /**
82     * @brief Constructor
83     *
84     * initializes the LDGPasses
85     *
86     * @param grid underlying grid instance
87     * @param numf instance of a numerical flux for the advection term
88     */
89    DGAdvectionDiffusionOperator(GridType& grid,
90                                 const NumFluxType& numf) :
91      grid_(grid),
92      model_(numf.model()),
93      numflux_(numf),
94      gridPart_(grid_),
95      space1_(gridPart_),
96      space2_(gridPart_),
97      problem1_(model_,numflux_),
98      problem2_(model_,numflux_),
99      pass1_(problem1_, pass0_, space1_),
100     pass2_(problem2_, pass1_, space2_)
101   {}
102
103   /**
104    * @brief This methods gets called by the TimeProvider in order to compute
105    * the next time step.
106    */
107   double timeStepEstimate() const
108   {
109     return pass2_.timeStepEstimate();
110   }
111
112   /**
113    * @brief This method is called by the TimeProvider in order to inform the
114    * Operator about the current time.
115    */
116   void setTime(double time) {
117     pass2_.setTime(time);
118   }
119
120   void operator()(const DestinationType& arg, DestinationType& dest) const {
121     pass2_(arg,dest);
122   }
123
124   const SpaceType& space() const {
125     return space2_;
126   }
127
128   /**
129    * @brief LaTeX Information printed by EocOutput
```

```
130     */
131   void printmyInfo(std::string filename) const {
132     std::ostringstream filestream;
133     filestream << filename;
134     std::ofstream ofs(filestream.str().c_str(), std::ios::app);
135     ofs << "Advection-Diffusion␣Op.,␣polynomial␣order:␣" << polOrd << "\\\\\n\n";
136     ofs.close();
137   }
138 private:
139   GridType& grid_;
140   const Model& model_;
141   const NumFluxType& numflux_;
142   GridPartType gridPart_;
143   Space1Type space1_;
144   Space2Type space2_;
145   DiscreteModel1Type problem1_;
146   DiscreteModel2Type problem2_;
147   Pass0Type pass0_;
148   Pass1Type pass1_;
149   Pass2Type pass2_;
150 };
151 }
152 #endif
```

Lines 66 and 67 contain the typedefs for both passes specifying the underlying model and the pass id of the `LocalDG` pass. The initialisation of the passes in lines 99 and 100 is equally straightforward. Line 121 shows how a pass can be evaluated like an `Operator` calling all previous passes.

4.5 Visualisation and EOC Output

The source code shipped with this document contains a file called `datadisp.cc` which implements a GRAPE interface for the data produced by the `dgtest` program. It also provides a file `manager.replay` which preconfigures GRAPE for watching the snapshots as an animation. Just click on the play button.

```
./datadisp paramfile:parameter 0 25
```

starts the visualisation, where the last number must be less than or equal to the number of snapshots produced by `dgtest`.

Furthermore, `dgtest` produces on every run, a TEXfile called `eoc.tex`, which comprises statistics about the numerical convergence of the implemented LDG scheme.

5 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 under http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/index.html. You can also build your own local doxygen documentation, by removing the --disable-documentation command in your config.opts file (see listing on page 6)

Some hints about using the doxygen documentation:

- The best way to start is from the page Modules (http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/modules.html) which gives you access to the documentation by category.
- A list of the central interface classes can be found here: http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/interfaceclass.html.
- A summary of the main features and concepts can be found here: http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/group__FEM.html.
- Newly added implementations are linked on this page: http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/newimplementation.html.
- Some remarks about writting dune-fem documentation is found here: http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/DocRules.html.
- And finally some notes on using subversion (svn): http://www.mathematik.uni-freiburg.de/IAM/Research/projectskr/dune/doc/html/svnhelp.html

Bibliography

[1] DUNE – Distributed and Unified Numerics Environment. http://www.dune-project.org/.

[2] Mario Ohlberger and Andreas Dedner. Wissenschaftliches Rechnen und Anwendungen in der Strömungsmechanik. http://http://www.mathematik.uni-freiburg.de/IAM/Teaching/ubungen/sci_com_SS06/skriptum.pdf, 2006.

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