1 Introduction

DuMu$^x$ aims to be a generic framework for the simulation of multiphase fluid flow and transport processes in porous media using continuum mechanical approaches. At the same time, DuMu$^x$ aims to deliver top-notch computational performance, high flexibility, a sound software architecture and the ability to run on anything from single processor systems to highly parallel supercomputers with specialized hardware architectures.

The means to achieve these somewhat contradictory goals are the thorough use of object-oriented design in conjunction with template programming. These requirements call for C++ as the implementation language.

One of the more complex issues when dealing with parallel continuum models is managing the grids used for the spatial discretization of the physical model. To date, no generic and efficient approach exists for all possible cases, so DuMu$^x$ is build on top of DUNE, the Distributed and Unified Numerics Environment [10]. DUNE provides a generic interface to many existing grid management libraries such as UG [14], ALUGrid [4, 3] and a few more. DUNE also extensively uses template programming in order to achieve minimal overhead when accessing the underlying grid libraries.

![Figure 1.1: A high-level overview of DUNE’s design is available on the project’s web site [10].](image)

DUNE’s grid interface is independent of the spatial dimension of the underlying grid. For this purpose, it uses the concept of co-dimensional entities. Roughly speaking, an entity of co-dimension 0 constitutes a cell, co-dimension 1 entities are faces between cells, co-dimension 2 are edges, and so on until co-dimension $n$ which are the cell’s vertices. The DUNE grid interface generally assumes that all entities are convex polytopes, which means that it must be possible to express each entity as the convex hull of a set of vertices. For the sake of efficiency, all entities are further expressed in terms of so-called reference elements which are transformed to the actual spatial incarnation within the grid by a so-called geometry function. Here, a reference element for an entity can be thought of as a prototype for the actual grid entity. For example, if we used a grid which applied hexahedrons as cells, the

---

[1] In fact, the performance penalty resulting from the use of DUNE’s grid interface is usually negligible [3].
1 Introduction

reference element for each cell would be the unit cube \([0, 1]^3\) and the geometry function would scale and translate the cube so that it matches the grid’s cell. A quick overview of reference elements and the related numbering can be obtained from the DUNE cheat sheet [https://www.dune-project.org/pdf/dune-cheat-sheet.pdf](https://www.dune-project.org/pdf/dune-cheat-sheet.pdf). For a more thorough description of DUNE’s grid definition, see [5].

In addition to the grid interface, DUNE also provides quite a few additional modules, of which the `dune-localfunctions` and `dune-istl` modules are the most relevant in the context of this handbook. `dune-localfunctions` provides a set of generic finite element shape functions, while `dune-istl` is the Iterative Solver Template Library and provides generic, highly optimized linear algebra routines for solving the generated systems.

`DuMu` comes in form of an additional module `dumux`. It depends on the DUNE core modules `dune-common`, `dune-geometry`, `dune-grid`, `dune-istl`, and `dune-localfunctions`. The main intention of `DuMu` is to provide a framework for an easy and efficient implementation of new physical models for porous media flow problems, ranging from problem formulation and the selection of spatial and temporal discretization schemes as well as nonlinear solvers, to general concepts for model coupling. Moreover, `DuMu` includes ready to use numerical models and a few example applications.

This is the handbook to a new minor version update of `DuMu`: version 3.1. The release contains improvements and new features compared to the 3.0 version. The update is backwards compatible with the last release 3.0. To facilitate the transition for our users, we have created a changelog helping to update programs from version 3.0 to version 3.1, and giving an overview over new capabilities. It is available online: [https://git.iws.uni-stuttgart.de/dumux-repositories/dumux/blob/master/CHANGELOG.md](https://git.iws.uni-stuttgart.de/dumux-repositories/dumux/blob/master/CHANGELOG.md). We highly recommend all our users to transition with us to the most recent version of `DuMu` and wish everyone an exciting simulation experience.
2 Quick Start

In this chapter we provide a quick start guide to your first DuMu\textsuperscript{x} experience, including an install script with all necessary instructions on how to very quickly install the latest release version of DuMu\textsuperscript{x}. You should have a recent working Linux environment. If you need more information, please have a look at the detailed installation instructions in chapter 3.

2.1 Prerequisites

For this quick start guide the following software packages are required:

- GitLab client
- A standard compliant C++ compiler supporting C++11 and the C++14 feature set of GCC 4.9. We support GCC 4.9 or newer and Clang 3.8 or newer.
- CMake 2.8.12 or newer
- pkg-config
- ParaView (to visualize the results)

2.2 Obtaining code and configuring all modules with a script

We provide you with a shell-script \texttt{installDumux.sh} that facilitates setting up a DUNE/DuMu\textsuperscript{x} directory tree and configures all modules with CMake. Copy the following lines into a text file named \texttt{installDumux.sh}:

```bash
# One click install script for dumux

# make a new folder containing everything
mkdir $(pwd)/DUMUX
cd DUMUX

echo "*************************************************
(1/2) Cloning repositories. This may take a while.
Make sure to be connected to the internet."

echo "*************************************************

# the core modules
for MOD in common geometry grid localfunctions istl; do
git clone -b releases/2.6 https://gitlab.dune-project.org/core/dune-$MOD.git
done

# dumux

git clone -b releases/3.1 https://git.iws.uni-stuttgart.de/dumux-repositories/dumux.git

if [ $? -ne 0 ]; then
```

Place the `installDumux.sh` script in the directory where you want to install DuMu$^x$ and DUNE (a single root folder DUMUX will be produced, so you do not need to provide one). Make `installDumux.sh` executable and run the script by typing into the terminal: `./installDumux.sh`

Configuring DUNE and DuMu$^x$ is done by the command-line script `dunecontrol` using optimized configure options, see the line entitled `# run build` in the `installDumux.sh` script. More details about the build-system can be found in section 3.2.

### 2.2.1 A first test run of DuMu$^x$

When the `installDumux.sh` script from the subsection above has run successfully, you can execute a second script that will compile and run a simple one-phase ground water flow example and will visualize the result using ParaView. The test script can be obtained by copying the following lines into a text file named `test_dumux.sh` that has to be located in the same directory as the installation script.

```bash
cd DUMUX/dumux/build-cmake/test/porousmediumflow/1p/implicit/isothermal
make -B test_1p_tpfa
./test_1p_tpfa params.input
paraview *pvd
```

After making `test_dumux.sh` executable, it can be executed by typing into the terminal: `./test_dumux.sh`. If everything works fine, a ParaView window with the result should open automatically, showing the initial conditions. Advance ParaView to the next frame (green arrow button) and rescale to data range (green double arrow on top right) to admire the colorful pressure distribution.
3 Detailed Installation, Documentation, and Externals

In this chapter, we provide more detailed information on how to obtain source code, build and test DUNE and DuMu\textsuperscript{x}. It further contains information on how to build the documentation and about external libraries and modules. Installing DuMu\textsuperscript{x} means that you first unpack DUNE and DuMu\textsuperscript{x} in a root directory, (section 3.1). In a second step of the installation, all modules are configured with CMake (section 3.2). After successful installation of DuMu\textsuperscript{x}, we guide you to start a test application, described in section 3.3. In section 3.4 we explain how to build the DuMu\textsuperscript{x} documentation. Lastly, section 3.5 provides details on optional libraries and modules.

In a technical sense DuMu\textsuperscript{x} is a module of DUNE. Thus, the installation procedure of DuMu\textsuperscript{x} is the same as that of DUNE. Details regarding the installation of DUNE are provided on the DUNE website \cite{DUNEwebsite}.

3.1 Obtaining Source Code for DUNE and DuMu\textsuperscript{x}

The DuMu\textsuperscript{x} release and trunk (developer tree) are based on the most recent DUNE release 2.6, comprising the core modules \texttt{dune-common}, \texttt{dune-geometry}, \texttt{dune-grid}, \texttt{dune-istl} and \texttt{dune-localfunctions}. For working with DuMu\textsuperscript{x}, these modules are required. All DUNE modules, including the DuMu\textsuperscript{x} module, get extracted into a common root directory, as it is done in an ordinary DUNE installation. We usually name our root directory \texttt{DUMUX} but an arbitrary name can be chosen. Source code files for each DUNE module are contained in their own sub-directory within the root directory. The sub-directories for the modules are named after the module names (depending on how the modules were obtained, a version number is added to the module name). The name of each DUNE module is defined in the file \texttt{dune.module}, which is in the root directory of the respective module. This should not be changed by the user.

Two possibilities exist to get the source code of DUNE and DuMu\textsuperscript{x}. Firstly, DUNE and DuMu\textsuperscript{x} can be downloaded as tar files from the respective DUNE and DuMu\textsuperscript{x} websites. They have to be extracted as described in the next paragraph. Secondly, the most recent source code (or, more generally, any of its previous revisions) can be obtained by direct access to the software repositories of the revision control system. This is described in the second paragraph.

In section 2.1 we list some prerequisites for running DUNE and DuMu\textsuperscript{x}. Please check in said paragraph whether you can fulfill them before continuing.

Obtaining the software by installing tar files The slightly old-fashioned tape-archive-file, shortly named tar file or tarball, is a common file format for distributing collections of files contained within these archives. The extraction from the tar files is done as follows: Download the tarballs from the respective DUNE (version 2.6) and DuMu\textsuperscript{x} websites to a certain folder in your file system. Create the common root directory, named \texttt{DUMUX} in the example below. Then extract the content of the tar
files, e.g. with the command-line program `tar`. This can be achieved by the following shell commands. Replace `path_to_tarball` with the directory name where the downloaded files are actually located. After extraction, the actual name of the dumux subdirectory is `dumux-3.1` (or whatever version you downloaded).

```bash
$ mkdir DUMUX
$ cd DUMUX
$ tar xzvf path_to_tarball_of/dune-common-2.6.0.tar.gz
$ tar xzvf path_to_tarball_of/dune-geometry-2.6.0.tar.gz
$ tar xzvf path_to_tarball_of/dune-grid-2.6.0.tar.gz
$ tar xzvf path_to_tarball_of/dune-istl-2.6.0.tar.gz
$ tar xzvf path_to_tarball_of/dune-localfunctions-2.6.0.tar.gz
$ tar xzvf path_to_tarball_of/dumux-3.1.tar.gz
```

Furthermore, if you wish to install the optional DUNE Grid-Howto which provides a tutorial on the Dune grid interface, act similar.

### Obtaining DUNE and DuMu\(^x\) from software repositories

Direct access to a software revision control system for downloading code can be of advantage later on. It is easier to keep up with code changes and to receive important bug fixes. DUNE and DuMu\(^x\) use Git for their software repositories. To access them, a Git client is needed.

In the technical language of Git, **cloning a certain software version** means nothing more then fetching a local copy from the software repository and laying it out in the file system. In addition to the software, some more files for the use of the software revision control system itself are created. If you have developer access to DuMu\(^x\), it is also possible to do the opposite, i.e. to load up a modified revision of software into the software repository. This is usually termed as **commit** and **push**.

The installation procedure is done as follows: Create a common root directory, named e.g. `DUMUX` in the lines below. Then, enter the previously created directory and check out the desired modules. As you see below, the check-out uses two different servers for getting the sources, one for DUNE and one for DuMu\(^x\).

```bash
$ mkdir DUMUX
$ cd DUMUX
$ git clone -b releases/2.6 https://gitlab.dune-project.org/core/dune-common.git
$ git clone -b releases/2.6 https://gitlab.dune-project.org/core/dune-geometry.git
$ git clone -b releases/2.6 https://gitlab.dune-project.org/core/dune-grid.git
$ git clone -b releases/2.6 https://gitlab.dune-project.org/core/dune-istl.git
$ git clone -b releases/2.6 https://gitlab.dune-project.org/core/dune-localfunctions.git
$ git clone -b releases/3.1 https://git.iws.uni-stuttgart.de/dumux-repositories/dumux.git
```

The newest and maybe unstable developments of DUNE and DuMu\(^x\) are also provided in these repositories and can be found in the **master** branch. Please check the DUNE website [10] for further information on the DUNE development. We always try to keep up with the latest developments of DUNE. However, the current DuMu\(^x\) release is based on the stable 2.6 release and it might not compile without further adaptations using the newest versions of DUNE.

Furthermore, if you wish to install the optional DUNE Grid-Howto which provides a tutorial on the Dune grid interface, act similar.
3.2 Building DUNE and DuMu\textsuperscript{x}

Configuring DUNE and DuMu\textsuperscript{x} is done by the shell-command \texttt{dunecontrol} which is part of the DUNE build system. If you are interested in more details about the build system that is used, they can be found in the DUNE build system documentation\cite{buildsystem} and CMake’s documentation\cite{cmake}. If something fails during the execution of \texttt{dunecontrol}, feel free to report it to the DUNE or DuMu\textsuperscript{x} developer mailing list, but please include error details.

It is possible to compile DuMu\textsuperscript{x} with nearly no explicit options to the build system. However, for the successful compilation of DUNE and DuMu\textsuperscript{x}, it is currently necessary to pass the option \texttt{-fno-strict-aliasing} to the C++ compiler, which is done here via a command-line argument to \texttt{dunecontrol}:

\begin{verbatim}
$ # make sure you are in the common root directory
$ ./dune-common/bin/dunecontrol --configure-opts="CXXFLAGS=-fno-strict-aliasing" --use-cmake all
\end{verbatim}

Too many options within the command line can make life hard. That’s why usually option files are being used together with \texttt{dunecontrol} and its sub-tools. Larger sets of options are kept in them. If you are going to compile with modified options, the following can be a starting point:

\begin{verbatim}
$ # make sure you are in the common root directory
$ cp dumux/cmake.opts my-cmake.opts # create a personal version
$ gedit my-cmake.opts # optional editing the options file
$ ./dune-common/bin/dunecontrol --opts=my-cmake.opts all
\end{verbatim}

Sometimes, it is necessary to have additional options which are specific to a package set of an operating system or sometimes you have your own preferences. Feel free to work with your own set of options, which may evolve over time. The option file that comes with the distribution is to be understood more as a starting point for setting up an own customization than as something which is fixed. The use of external libraries can make it necessary to add quite many options in an option file. It can be helpful to give your customized option file its own name, as done above, to avoid confusing it with the option files which came out of the distribution.

3.3 The First Run of a Test Application

The previous section showed how to install and compile DuMu\textsuperscript{x}. This section shall give a very brief introduction how to run a first test application and how to visualize the first output files.

All executables are compiled in the build sub-directories of DuMu\textsuperscript{x}. If not specified differently in the options file, this is build-cmake as default.

1. Enter the folder test/porousmediumflow/2p/implicit/incompressible within your build directory.
   Type \texttt{make test\_2p\_incompressible\_tpfa} in order to compile the application

\begin{verbatim}
https://www.dune-project.org/buildsystem/
https://cmake.org/documentation/
\end{verbatim}
test_2p.incompressible_tpfa. To run the simulation, type ./test_2p.incompressible_tpfa params.input into the console. The added params.input specifies that all important run-time parameters (like first time step size, end of simulation and location of the grid file) can be found in a text file in the same directory with the name params.input.

2. The simulation starts and produces some VTU output files and also a PVD file. The PVD file can be used to examine time series and summarizes the VTU files. It is possible to stop a running application by pressing <Ctrl><c>.

3. You can display the results using the visualization tool ParaView (or alternatively VisIt). Just type paraview in the console and open the PVD file. On the upper left hand side, you can choose the desired parameter to be displayed.

3.4 Building Documentation

Building the included documentation like this handbook requires \LaTeX and the auxiliary tool bibtex. One usually chooses a \LaTeX distribution like texlive for this purpose. It is possible to switch off building the documentation by setting the switch --disable-documentation in the CONFIGURE_FLAGS of the building options, see section 3.2.

3.4.1 Doxygen

Doxygen documentation is done by specifically formatted comments integrated in the source code, which can get extracted by the program doxygen. Beside extracting these comments, doxygen builds up a web-browsable code-structure documentation like class hierarchy of code displayed as graphs, see http://www.stack.nl/~dimitri/doxygen/.

The Doxygen documentation of a module can be built if doxygen is installed, by running dunecontrol, entering the build-*directory, and executing make doc. Then point your web browser to the file MODULE_BUILD_DIRECTORY/doc/doxygen/html/index.html to read the generated documentation. This should also work for other DUNE modules.

3.4.2 Handbook

To build the DuMu\textsuperscript{x} handbook go into the build-directory and run make doc or make 0_dumux-handbook.pdf. The pdf can then be found in MODULE_BUILD_DIRECTORY/doc/handbook/0_dumux-handbook.pdf.

3.5 External Libraries and Modules

The libraries described below provide additional functionality but are not generally required to run DuMu\textsuperscript{x}. If you are going to use an external library, check the information provided on the DUNE website\textsuperscript{3} If you are going to use an external DUNE module, the website on external modules\textsuperscript{4} can be helpful.

\textsuperscript{3}DUNE: External libraries, https://www.dune-project.org/doc/external-libraries/  
\textsuperscript{4}DUNE: External modules, https://www.dune-project.org/groups/external/
Installing an external library can require additional libraries which are also used by DUNE. For some libraries, such as BLAS or MPI, multiple versions can be installed on the system. Make sure that it uses the same library as DUNE when configuring the external library.

Some of the libraries are then compiled within that directory and are not installed in a different place, but DUNE may need to know their location. Thus, one may have to refer to them as options for **dunecontrol**, for example via the options file `my-cmake.opts`. Make sure you compile the required external libraries before you run **dunecontrol**.

An easy way to install some of the libraries and modules given below is the `installexternal.sh` script located in `bin`. The script has to be called from your common root directory.

### 3.5.1 List of External Libraries and Modules

In the following list, you can find some external modules and external libraries, and some more libraries and tools which are prerequisites for their use.

- **dune-ALUGrid**: Grid library, comes as a DUNE module. The parallel version needs also a graph partitioner, such as ParMETIS. Download: [https://gitlab.dune-project.org/extensions/dune-alugrid](https://gitlab.dune-project.org/extensions/dune-alugrid)

- **dune-foamgrid**: External grid module. One- and two-dimensional grids in a physical space of arbitrary dimension; non-manifold grids, growth, element parameterizations, and movable vertices. This makes FoamGrid the grid data structure of choice for simulating structures such as foams, discrete fracture networks, or network flow problems. Download: [https://gitlab.dune-project.org/extensions/dune-foamgrid](https://gitlab.dune-project.org/extensions/dune-foamgrid)

- **opm-grid**: opm-grid is a DUNE module supporting grids in a corner-point format. Download: [https://github.com/OPM/opm-grid.git](https://github.com/OPM/opm-grid.git)

- **dune-subgrid**: The dune-subgrid module is a meta-grid implementation that allows to mark elements of another hierarchical dune grid and use this sub-grid just like a regular grid. The set of marked elements can then be accessed as a hierarchical dune grid in its own right. Dune-Subgrid provides the full grid interface including adaptive mesh refinement. Download: [https://git.imp.fu-berlin.de/agnumpde/dune-subgrid.git](https://git.imp.fu-berlin.de/agnumpde/dune-subgrid.git)

- **dune-spgrid**: The DUNE module dune-spgrid provides a structured, parallel grid and supports periodic boundary conditions. Download: [https://gitlab.dune-project.org/extensions/dune-spgrid.git](https://gitlab.dune-project.org/extensions/dune-spgrid.git)


- **UMFPack**: External library for solving linear equations. It is part of SuiteSparse.

- **dune-UG**: External library for use as grid. UG is a toolbox for unstructured grids, released under GPL. To build UG the tools `lex/yacc` or the GNU variants of `flex/bison` must be provided. Download: [https://gitlab.dune-project.org/staging/dune-uggrid](https://gitlab.dune-project.org/staging/dune-uggrid)
The following are dependencies of some of the used libraries. You will need them depending on which modules of DUNE and which external libraries you use.

- **MPI**: The parallel version of DUNE and also some of the external dependencies need MPI when they are going to be built for parallel computing. OpenMPI and MPICH in a recent version have been reported to work.

- **BLAS**: SuperLU makes use of BLAS. Thus install GotoBLAS2, ATLAS, non-optimized BLAS or BLAS provided by a chip manufacturer. Take care that the installation scripts select the intended version of BLAS.

- **METIS** and **ParMETIS**: This are dependencies of ALUGrid and can be used with UG, if run in parallel.

- **Compilers**: Beside g++, DUNE can be built with Clang from the LLVM project and Intel C++ compiler. C and Fortran compilers are needed for some external libraries. As code of different compilers is linked together, they have to be be compatible with each other.
4 Learning to use DuMu\textsuperscript{x}

So, you’ve downloaded your very own copy of DuMu\textsuperscript{x} and its dependencies. You’ve run dunecontrol, and your example “test\_dumux” not only compiles, but it even shows a nice simulation in ParaView. Maybe you’ve read through parts of the handbook, and even started looking through the Doxygen documentation. Well done. What now? “How on earth is this going to help me solve my multi-(phase, component, scale, physics) flow and transport problems in porous media systems?”, you begin to wonder. Don’t panic! In order to best ease our prospective users and developers into the wonderful DuMu\textsuperscript{x} simulation environment, we’ve prepared a DuMu\textsuperscript{x} course and extensively-documented examples.

4.1 Hands-on DuMu\textsuperscript{x} experience – the DuMu\textsuperscript{x} course

This course is offered once a year over a period of 3 days at the University of Stuttgart. If you’re looking for information on attending, subscribe to the DuMu\textsuperscript{x} mailing list and stay tuned for updates: https://listserv.uni-stuttgart.de/mailman/listinfo/dumux.

“But the course won’t take place for another 6 months!” and, “I want to start developing a numerical model of my challenging and interesting process now!”, you think. Not a problem. The course materials are all shared online in their own git repository. A series of beginner-level exercises are explained such that you can see how a model is developed in DuMu\textsuperscript{x}. As a teaser, we’ve also included a suite of examples from hot topics we’re working on. Models exploring “Coupling free flow and porous-media flow”, “Flow in fractured porous media” and “Fluid-solid phase change” are all introduced.

“Sounds great, but where is this material? I can’t find it within what I’ve downloaded.”, you question. The DuMu\textsuperscript{x} course material is available online: https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-course.

In order to download this repository, which acts as an additional module to the DuMu\textsuperscript{x} base, you can download an installation script with the following command:

```
$ wget https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-course/raw/releases/3.1/scripts/install.sh
```

This script will install dumux, its Dune dependencies, and the dumux-course repository. Within the directory dumux-course there are a series of exercises and slides describing the previously described examples.

The DuMu\textsuperscript{x} course will be updated with each DuMu\textsuperscript{x} release. The above script will download the correct version (releases/3.1) of both the dumux and dumux-course module.
4.2 Experience DuMu\textsuperscript{x} by reading – the DuMu\textsuperscript{x} examples

As an alternative to going through exercises, you can have a look at our well-documented DuMu\textsuperscript{x} examples. They show how to apply DuMu\textsuperscript{x} to typical physical problems. In the README.md files, the setup is explained, the used code is presented and documented and images resulting from the simulation are included. The README.md files are located directly in the subfolders of examples and can be displayed by web browsers.

We currently have the examples

- 1ptracer: one-phase groundwater flow including a tracer
- 2pinfiltration: two-phase infiltration problem
- shallowwaterfriction: steady subcritical shallow water flow including bottom friction

The number of examples is continuously growing.

4.3 Further Practice

If there is a need for further practice, we refer here to the test problems that are already implemented in DuMu\textsuperscript{x}. Several examples for all models can be found in the test-directory. An overview over the available test cases can be found in the class documentation \url{http://www.dumux.org/documentation.php}.

Another possibility to gain more experience with DuMu\textsuperscript{x} is the dumux-lecture module that contains different application examples that are used in the lectures at the Department of Hydromechanics and Modelling of Hydrosystems in Stuttgart. The dumux-lecture module can be obtained as follows:

```
$ git clone https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-lecture.git
```

The module is structured based on the different lectures:

- mm: Multiphase Modelling,
- efm: Environmental Fluid Mechanics,
- mhs: Modelling of Hydrosystems.

The majority of applications is covered in the course Multiphase Modelling (mm), while there are also some basic examples in the courses Environmental Fluid Mechanics (efm) and Modelling of Hydrosystems (mhs). These applications are primarily designed to enhance the understanding of conceptualizing the governing physical processes and their implementation in a numerical simulator. Different aspects of modeling multi-phase multi-component flow and transport processes are shown. The lectures focus on questions like the assignment of boundary conditions, the choice of the appropriate physics for a given problem (which phases, which components), discretization issues, time stepping. You can find, e. g., a comparison of different two-phase flow problems: The simpler approach considers two immiscible fluids while components in both phases with inter-phase mass transfer are considered in the more complex approach. All scenarios and their physical background are explained in additional .tex-files, which are provided in sub-directories named description. The following test cases are contained in the dumux-lecture module:
4 Learning to use DuMuX

- **buckleyleverett**: The Buckley-Leverett Problem is a classical porous media flow show case
- **co2plume**: Analysis of the influence of the gravitational number on a CO$_2$ plume
- **columnxylene**: An experiment of the Research Facility for Subsurface Remediation, University of Stuttgart
- **convectivemixing**: A test case related to CO$_2$ storage
- **fractures**: Two-phase flow in fractured porous media
- **fuelcell**: Water management in PEM fuel cells
- **heatpipe**: A show case related to salt water intrusion
- **mcwhorter**: The McWhorter Problem is a classical porous media flow show case
- **naplinfiltration**: Infiltration of non-aqueous phase liquid (NAPL) into soil
- **remediationscenarios**: Test case for NAPL contaminated unsaturated soils
- **groundwater**: Simple groundwater flow case for the course Modelling of Hydrosystems (mhs)
- Different single/two-phase, single/two-component problems: Examples from the course Environmental Fluid Mechanics (efm)
5 Overview and Infrastructure

This chapter provides an overview of the general structure in DuMu\textsuperscript{x} \citep{5.1} and gives help for basic work with DuMu\textsuperscript{x} \citep{5.2-5.5}. Further it presents useful external tools \citep{5.6} and basic concepts \citep{5.7}.

5.1 Directory Structure

DuMu\textsuperscript{x} has the following folder structure, which is similar to other DUNE modules.

- **bin**: binaries, e.g. used for the automatic testing, post-processing, installation
- **cmake**: the configuration options for building DuMu\textsuperscript{x}
- **doc**: files necessary for the Doxygen documentation and this handbook, and various logos
- **dumux**: the main folder, containing the source files. See Fig. 5.1 for a visualized structure. For more information on the models have a look at the Doxygen documentation.
- **examples**: well-documented examples of applying DuMu\textsuperscript{x} to typical physical problems. In the README.md files, the setup is explained, the used code is presented and documented and images resulting from the simulation are included. The README.md files are located directly in the subfolders of examples and can be displayed by web browsers.
- **test**: tests for each numerical model and some functionality. The structure is equivalent to the dumux folder, the references folder contains solutions for the automatic testing. Each test program consist of a main file main.cc, the problem definition *problem.hh (specifying initial and boundary conditions), and an input file params.input. If necessary, spatially dependent parameters are defined in *spatialparameters.hh. For more detailed descriptions of the tests, please have a look at the Doxygen documentation.

5.2 Setup of new Folders and new Tests

This section describes how to set up a new folder and how to tell the build system there is a new one.

**Adding new Folders**

1) create new folder with content
2) adapt the CMakeList.txt in the folder above by adding a line with add_subdirectory(NEW_FOLDER)
3) create a CMakeList.txt in the newly created folder
4) go to your build-directory and type make to re-configure the system
5 Overview and Infrastructure

Figure 5.1: Structure of the directory dumux containing the DuMu\(^x\) source files.
Adding new Test Programs  To add a test use the `add_dune_test` macro within the `CMakeList.txt` file. The macro can be used with a variable amount of arguments. A simple call could look like this:

```haskell
1) dune_add_test(NAME my_test
2) SOURCES main.cc
3) CMD_ARGS my_test params.input)
```

Here, we create an executable called `my_test` from a source file `main.cc`. The name of the test will also be `my_test` (has to be unique). The last argument specifies a command - here, we just run the executable `my_test` with an input file `params.input`. For more advanced uses of the `add_dune_test` macro, have a look at the `test` directory. A complete documentation is given under [https://www.dune-project.org/sphinx/core-2.6/](https://www.dune-project.org/sphinx/core-2.6/).

5.3 Parameters in DuMuX

Simulation parameters can be parsed to the program via a parameter file or via the command line.

After having run the example application from section 3.3 you will get the following output at the end of the simulation run:

```
# Runtime-specified parameters used:
[ Grid ]
Cells = "48 32"
UpperRight = "6 4"
[ Newton ]
EnablePartialReassembly = "true"
[ Problem ]
EnableGravity = "true"
Name = "2p"
[ SpatialParams ]
LensLowerLeft = "1.0 2.0"
LensUpperRight = "4.0 3.0"
[ TimeLoop ]
DtInitial = "250"
TEnd = "3000"

# Global default parameters used:
[ Assembly ]
NumericDifferenceMethod = "1"
[ Flux ]
UpwindWeight = "1.0"
[ LinearSolver ]
MaxIterations = "250"
PreconditionerIterations = "1"
PreconditionerRelaxation = "1.0"
ResidualReduction = "1e-13"
Verbosity = "0"
[ Newton ]
EnableAbsoluteResidualCriterion = "false"
EnableChop = "false"
```

1If you did not get the output, add `Parameters::print();` to your main file.
A number of things can be learned:

- **run-time** parameters can be changed without re-compiling
- **default parameters** are set by default
- **unused** parameters are not used by the simulation (maybe typo or wrong group in input file)

### 5.3.1 Parameter Values

To get the value of an input parameter please use:

```c++
1 static const TYPE paramname = getParam<TYPE>("GROUPNAME.PARAMNAME");
```

If you also want to set a default value for a parameter, just add it like this:

```c++
2 static const TYPE paramname = getParam<TYPE>("GROUPNAME.PARAMNAME", default);
```

As this function call is relatively expensive, the respective variables should always be `static` (e.g., if used in a loop). When dealing with multiple group names, e.g., in the context of coupled models, the following methods might be more convenient:

```c++
3 auto modelParamGroup0 = "Model0";
4 static const TYPE paramname0 = getParamFromGroup<TYPE>(modelParamGroup0, "GROUPNAME.PARAMNAME");
5 auto modelParamGroup1 = "Model1";
6 static const TYPE paramname1 = getParamFromGroup<TYPE>(modelParamGroup1, "GROUPNAME.PARAMNAME");
```

The `FVProblem` class provides a convenience function `paramGroup()`. The parameters can then be specified in the input file:

```c++
[ Model0.Grid ]
File = file0.dgf
[ Model1.Grid ]
File = file1.dgf
```
5 Overview and Infrastructure

5.4 Restart DuMu\textsuperscript{x} Simulations

DuMu\textsuperscript{x} has some experimental support for check-pointing (restarting paused/stopped/crashed simulations). You can restart a DuMu\textsuperscript{x} simulation from any time point where a VTK file was written out. This is currently only supported for sequential, non-adaptive simulations. For adaptive simulation the full hierarchical grid has to be stored. This is usually done with the grid’s BackupRestoreFacility. There is currently no special support by DuMu\textsuperscript{x} for that, but it is possible to implement a restart using BackupRestoreFacility with plain Dune.

For VTK files the output can be read with the free function loadSolution. Grids can be read with the Dumux::VTKReader or you can simply recreate the grid as you did in the first simulation run. Unfortunately, writing double-precision floating point numbers to VTK files is only available with Dune master (will be in 2.7). That’s why we currently only support single precision restart, meaning some information will be lost if you are computing in double precision.

The restart capabilities will hopefully be improved in future versions of DuMu\textsuperscript{x}-3. We are happy about any contributions (especially HDF5 / XDMF support, improvement of VTK support).

5.5 Developing DuMu\textsuperscript{x}

5.5.1 Communicate with DuMu\textsuperscript{x} Developers

Issues and Bug Tracking The bug-tracking system GitLab Issues offers the possibility to report bugs or discuss new development requests. Feel free to register (if you don’t have an account at out Git yet) and to contribute at https://git.iws.uni-stuttgart.de/dumux-repositories/dumux/issues

Commits, Merges, etc. To be up-to-date with the latest changes made to any git-repository you can use RSS Feeds. Simply click on Issues or Activity and then select a tab you are interested in and use your favorite RSS-application for receiving the news.

Automatic Testing Dashboard The automatic testing using BuildBot helps to constantly check the DuMu\textsuperscript{x} problems for compiling and running correctly. It is available at https://git.iws.uni-stuttgart.de/buildbot/#/builders

The General Mailing List: If you have questions, specific problems (which you really struggle to solve on your own), or hints for the DuMu\textsuperscript{x}-developers, please contact the mailing list dumux@iws.uni-stuttgart.de You can subscribe to the mailing list via https://listserv.uni-stuttgart.de/mailman/listinfo/dumux then you will be informed about upcoming releases or events.

5.5.2 Coding Guidelines

Writing code in a readable manner is very important, especially for future code developers (e.g. for adding features, debugging, etc.). For the style guide and instructions how to contribute to DuMu\textsuperscript{x} visit https://git.iws.uni-stuttgart.de/dumux-repositories/dumux/blob/master/CONTRIBUTING.md
5 Overview and Infrastructure

5.5.3 Tips and Tricks
DuMu\textsuperscript{x} users and developers at the LH2 are also referred to the internal confluence pages for more information.

**Optimized computation vs debugging** DUNE and DuMu\textsuperscript{x} are built with the help of dunecontrol, as explained on page 9. Per default, DuMu\textsuperscript{x} is compiled using optimization options, which leads to faster runtimes but is unsuitable for debugging. For debug opts you can set DCMAKE\_BUILD\_TYPE to Debug or RelWithDebInfo in your options file. You can also do this in any of the CMakeLists.txt in Dumux by adding:

```
set(DCMAKE\_BUILD\_TYPE Debug)
```

Afterwards rerun cmake (run cmake \textless path-to-build-dir\textgreater).

**Dunecontrol for selected modules** A complete build using dunecontrol takes some time. In many cases not all modules need to be re-built. Pass the flag --only=dumux to dunecontrol for configuring or building only DuMu\textsuperscript{x}. A more complex example would be a case in which you have to configure and build only e.g. DUNE-grid and DuMu\textsuperscript{x}. This is achieved by adding --only=MODULE,dumux.

**Patching Files or Modules** If you want to send changes to an other developer of DuMu\textsuperscript{x} providing patches can be quite smart. To create a patch simply type:

```
$ git diff > PATCHFILE
```

which creates a text file containing all your changes to the files in the current folder or its subdirectories. To apply a patch in the same directory type:

```
$ patch -p1 < PATCHFILE
```

**File Name and Line Number by Predefined Macro** If you want to create output in order to later know where some output or debug information came from, use the predefined macros \_FILE\_ and \_LINE\_:

```
std::cout << "# This was written from " << \_FILE\_ << ", line " << \_LINE\_ << std::endl;
```

**Using DUNE Debug Streams** DUNE provides a helpful feature, for keeping your debug-output organized. It uses simple streams like std::cout, but they can be switched on and off for the whole project. You can chose five different levels of severity:

5 - grave (dgrave)
4 - warning (dwarn)
3 - info (dinfo)
2 - verbose (dverb)
1 - very verbose (dvverb)

They are used as follows:
5 Overview and Infrastructure

Make headercheck: To check one header file for all necessary includes to compile the contained code, use `make headercheck`. Include the option `-DENABLE_HEADERCHECK=1` in your opts file and run `dunecontrol`. Then go to the top level in your build-directory and type `make headercheck` to check all headers or press ‘tab’ to use the auto-completion to search for a specific header.

5.6 External Tools

5.6.1 Git

Git is a version control tool which we use. The basic Git commands are:

- `git checkout`: receive a specified branch from the repository
- `git clone`: clone a repository; creates a local copy
- `git diff`: to see the actual changes compared to your last commit
- `git pull`: pull changes from the repository; synchronizes the repository with your local copy
- `git push`: push committed changes to the repository; synchronizes your local copy with the repository
- `git status`: to check which files/folders have been changed
- `git gui`: graphical user interface, helps selecting changes for a commit

5.6.2 Gnuplot

A gnuplot interface is available to plot or visualize results during a simulation run. This is achieved with the help of the `Dumux::GnuplotInterface` class provided in `io/gnuplotinterface.hh`.

To use the gnuplot interface you have to make some modifications in your file, e.g., your main file. First, you have to include the corresponding header file for the gnuplot interface:

```cpp
#include <dumux/io/gnuplotinterface.hh
```

Second, you have to define an instance of the class `Dumux::GnuplotInterface` (e.g. called `gnuplot`).

```cpp
Dumux::GnuplotInterface<double> gnuplot;
```

As an example, to plot the mole fraction of nitrogen (\(y\)) over time (\(x\)), extract the variables after each time step in the time loop. The actual plotting is done using the method of the gnuplot interface:

```cpp
gnuplot.resetPlot(); // reset the plot
gnuplot.setXRange(0.0, 72000.0); // specify xmin and xmax
gnuplot.setYRange(0.0, 1.0); // specify ymin and ymax
gnuplot.setXlabel("time [s]"); // set xlabel
gnuplot.setYlabel("mole fraction mol/mol"); // set ylabel
```
It is also possible to add several data sets to one plot by calling `addDataSetToPlot()` more than once. For more information have a look into a test including the gnuplot interface header file, the doxygen documentation of `Dumux::GnuplotInterface`, or the header file itself (`dumux/io/gnuplotinterface.hh`).

5.6.3 Gstat

Gstat is an open source software tool which generates geostatistical random fields (see [www.gstat.org](http://www.gstat.org)). In order to use gstat, execute the `bin/installexternal.sh` from your DuMuX root directory or download, unpack and install the tarball from the gstat-website. Then rerun cmake (in the second case set `GSTAT_ROOT` in your input file to the path where gstat is installed).

5.6.4 ParaView

Reload Button: There are scripts to reload PVD or series of VTU files since ParaView 4.2. The scripts can be found under this link. Just save the specific code portion in a file and load it via Macros → Add new macro.


5.7 Assembling the linear system

The physical system is implemented as the mathematical differential equation in local operators. DuMuX generates the linear system automatically. Read on, to learn what is done internally.

5.7.1 Newton’s method

The differential equations are implemented in the residual form. All terms are on the left hand side and are summed up. The terms contain values for the primary variables which are part of the solution vector \( u \). The sum of the terms is called residual \( r(u) \) which is a function of the solution. For example:

\[
\frac{\partial \varrho_{\alpha} S_{\alpha}}{\partial t} - \text{div} \left( \varrho_{\alpha} \frac{k_{\alpha}}{\mu_{\alpha}} K \left( \text{grad} \ p_{\alpha} - \varrho_{\alpha} g \right) \right) - q_{\alpha} = 0
\]

We don’t know the solution \( u \), so we use the iterative Newton’s method to obtain a good estimate of \( u \). We start with an initial guess \( u^0 \) and calculate it’s residual \( r(u^0) \). To minimize the error, we calculate the derivative of the residual with respect to the solution. This is the Jacobian matrix:

\[
\frac{d}{du} r (u^i) = J_r(u^i) = \left( \frac{d}{du^i_n} r (u^i) \right)_{m,n}
\]
5 Overview and Infrastructure

with $i$ denoting the Newton iteration step. Each column is the residual derived with respect to the $m$th entry of $u^i$.

The Jacobian indicates the direction where the residual increases. By solving the linear system

$$J_{r(u^i)} \cdot x^i = r(u^i)$$

we calculate the direction of maximum growth $x^i$. We subtract it from our current solution to get a new, better solution $u^{i+1} = u^i - x^i$.

We repeat the calculation of of the Jacobian $J_{r(u^i)}$ and the direction of maximum growth $x^i$ until our approximated solution becomes good enough.

5.7.2 Structure of matrix and vectors

To understand the meaning of an entry in the matrix or the vector of the linear system, we have to define their structure. Both have a blocking structure. Each block contains the degrees of freedom (also called variables or unknowns) for a control volume. The equation index is used to order of the degrees of freedom. For each control volume we have one block. The mapper is used to order the blocks.

![Figure 5.2: Structure of matrix and vector, left: block structure, right: within block](image)

Accessing entries follows this structure. You can access the pressure value in the third ($n = 3$) sub-control volume in a solution vector sol with sol[n−1][pressureIdx]=sol[2][pressureIdx].
6 Advanced DuMu\textsuperscript{x} – Detailed Instructions

This chapter contains detailed information for those who are interested in deeper modifications of underlying DuMu\textsuperscript{x} models, classes, functions, etc.

6.1 Physical Basics

Here the basic definitions, the general models concept, and a list of models available in DuMu\textsuperscript{x} are given. The actual differential equations can be found in the local residuals (see Doxygen documentation of the model's \texttt{LocalResidual} class).

6.1.1 Basic Definitions and Assumptions

Basic definitions and assumptions are given. More information can be found e.g. in [2, 6].

**Phases:** A phase is defined as a continuum having distinct properties (e.g. density and viscosity). If phases are miscible, they contain dissolved portions of the substance of the other phase. Fluid and solid phases are distinguished. The fluid phases have different affinities to the solid phases. The phase, which has a higher affinity to the solid phases is referred to as the (more) wetting phase. In the case of two phases, the less wetting one is called the non-wetting phase.

For compositional multi-phase models, fluid phases may be composed of several components, while the solid phases are assumed to consist exclusively of a single component.

**Components:** The term component stands for constituents of the phases which can be associated with a unique chemical species, or, more generally, with a group of species exploiting similar physical behavior. For example, Fig. 6.1 shows a water-gas-NAPL system composed of the phases water (subscript w), gas (g), and NAPL (n). These phases are composed of the components water (superscript w), the pseudo-component air (a), and an organic contaminant (c).

The composition of the components in a phase can influence the phase properties. Furthermore, for mass transfer, the phase behavior is quite different from the component behavior.

**Equilibrium:** For the non-isothermal, multi-phase, multi-component processes in porous media we state the assumption of local thermodynamic equilibrium. Chemical equilibrium means that the mass/mole fractions of a component in different phases are in equilibrium. Thermal equilibrium assumes the same temperature for all considered phases. Mechanical equilibrium is not valid in a porous medium, since discontinuities in pressure can occur across a fluid-fluid interface due to capillary effects.

**Notation:** The subscript index $\alpha$, e.g. w, n and g in the example of Fig. 6.1 refers to the phase, while the superscript $\kappa$, e.g. w, a and c in the example of Fig. 6.1 refers to the component.
Table 6.1: Notation list for most of the variables and indices used in DuMux.

- $p_\alpha$: phase pressure
- $T$: temperature
- $S_\alpha$: phase saturation
- $x_{\kappa}^\alpha$: mole fraction of component $\kappa$ in phase $\alpha$
- $X_\alpha$: mass fraction of component $\kappa$ in phase $\alpha$
- $\varrho_{\text{mol},\alpha}$: molar density of phase $\alpha$
- $\varrho_\alpha$: mass density of phase $\alpha$
- $M$: molar mass of a phase or component
- $k_{\alpha}$: relative permeability
- $\mu_\alpha$: phase viscosity
- $D_\kappa^\alpha$: diffusivity of component $\kappa$ in phase $\alpha$
- $v_\alpha$: velocity (Darcy or free flow)
- $\phi$: porosity
- $K$: absolute permeability tensor
- $\tau$: tortuosity
- $g$: gravitational acceleration
- $q_{\kappa}^\alpha$: volume source term of $\kappa$ in $\alpha$
- $u_\alpha$: specific internal energy
- $h_\alpha$: specific enthalpy
- $c_s$: specific heat enthalpy
- $\lambda_{\text{pm}}$: heat conductivity
- $q_h$: heat source term
- $v_{\kappa,\alpha}$: advective velocity

Figure 6.1: Mass and energy transfer between the phases in a water-NAPL-gas system [9]
6.1.2 Scale

Depending on the scale of interest, physical and chemical processes and properties can be described using different approaches. On the molecular scale, the properties and interactions of individual molecules are described, which is only feasible for a restricted number of molecules. For larger systems, a continuum approach is used, where properties are averaged over groups of similar molecules, assuming continuous matter. This upscaling by averaging from the molecular scale results in the micro-scale, on which the system is described by the pore geometry and the distribution of distinct fluid phases within the pores. However, for larger laboratory or field-scale applications, the micro-scale is still computationally prohibitively expensive and system descriptions on the macro-scale are used for calculations. The macro-scale description is obtained by averaging over the micro-scale properties within a representative elementary volume (REV), which needs to be large enough to ensure that the averaged properties are independent of the REV size or position. However, it should in turn be much smaller than the entire domain size [12]. The detailed pore-geometry and phase-distribution information of the micro-scale is lost on the macro-scale and replaced by volume average quantities, such as porosity, permeability and phase saturation, and relations like the Darcy’s law. The macro-scale is also called the REV (or Darcy) scale and is the scale of the models available in DuMu\textsuperscript{x}.

6.1.3 Porous medium properties

Porosity

The porosity $\phi$ is defined as the fraction of the volume occupied by fluids in an REV $V_{\text{fluid}}$ divided by the total volume of the REV $V_{\text{total}}$.

$$\phi = \frac{V_{\text{fluid}}}{V_{\text{total}}} = 1 - \frac{V_{\text{solid}}}{V_{\text{total}}}.$$  \hfill (6.1)

Intrinsic permeability

The intrinsic permeability is a measure on the REV scale of the ease of fluid flow through porous media. It relates the potential gradient and the resulting flow velocity in the Darcy equation. As the porous medium may have a structure leading to preferential flow in certain directions, intrinsic permeability is in general a tensorial quantity $K$. For isotropic porous media, it can be reduced to a scalar quantity $K$.

\footnote{This subsection is taken from [13] in a slightly adapted form.}
6.1.4 Mass fraction, mole fraction

The composition of a phase is described by mass or mole fractions of the components. The mole fraction \( x_\kappa^\alpha \) of component \( \kappa \) in phase \( \alpha \) is defined as:

\[
x_\kappa^\alpha = \frac{n_\kappa^\alpha}{\sum_\kappa n_\kappa^\alpha},\tag{6.2}
\]

where \( n_\kappa^\alpha \) is the number of moles of component \( \kappa \) in phase \( \alpha \). The mass fraction \( X_\kappa^\alpha \) is defined similarly using the mass of component \( \kappa \) in phase \( \alpha \) instead of \( n_\kappa^\alpha \), \( X_\kappa^\alpha = \frac{\text{mass}_\kappa^\alpha}{\text{mass}_{\text{total}}^\alpha} \). The molar mass \( M_\kappa \) of the component \( \kappa \) relates the mass fraction to the mole fraction and vice versa.

6.1.5 Fluid properties

The most important fluid properties to describe fluid flow on the REV scale are density and viscosity.

**Density**

The density \( \rho_\alpha \) of a fluid phase \( \alpha \) is defined as the ratio of its mass to its volume \( (\rho_\alpha = \frac{\text{mass}_{\alpha}}{\text{volume}_{\alpha}}) \) while the molar density \( \rho_{\text{mol},\alpha} \) is defined as the ratio of the number of moles per volume \( (\rho_{\text{mol},\alpha} = \frac{\text{moles}_{\alpha}}{\text{volume}_{\alpha}}) \).

**Viscosity**

The dynamic viscosity \( \mu_\alpha \) characterizes the resistance of a fluid to flow. As density, it is a fluid phase property. For Newtonian fluids, it relates the shear stress \( \tau_s \) to the velocity gradient \( \frac{dv_{\alpha,x}}{dy} \):

\[
\tau_s = \mu_\alpha \frac{dv_{\alpha,x}}{dy} \tag{6.3}
\]

Density and viscosity are both dependent on pressure, temperature and phase composition.

6.1.6 Fluid phase interactions in porous media

If more than a single fluid is present in the porous medium, the fluids interact with each other and the solids, which leads to additional properties for multi-phase systems.

**Saturation**

The saturation \( S_\alpha \) of a phase \( \alpha \) is defined as the ratio of the volume occupied by that phase to the total pore volume within an REV. As all pores are filled with some fluid, the sum of the saturations of all present phases is equal to one.

**Capillary pressure**

Immiscible fluids form a sharp interface as a result of differences in their intermolecular forces translating into different adhesive and cohesive forces at the fluid-fluid and fluid-fluid-solid interfaces creating interfacial tension on the microscale. From the mechanical equilibrium which has also to be satisfied
at the interface, a difference between the pressures of the fluid phases results defined as the capillary pressure $p_c$:

$$p_c = p_n - p_w.$$  \hfill (6.4)

On the microscale, $p_c$ can be calculated from the surface tension according to the Laplace equation [see 12].

On the REV scale, however, capillary pressure needs to be defined by quantities of that scale. Several empirical relations provide expressions to link $p_c$ to the wetting-phase saturation $S_w$. An example is the relation given by Brooks and Corey [7] to determine $p_c$ based on $S_e$, which is the effective wetting-phase saturation, the entry pressure $p_d$, and the parameter $\lambda$ describing the pore-size distribution:

$$p_c = p_d S_e^{-1/\lambda},$$  \hfill (6.5)

with

$$S_e = \frac{S_w - S_{w,r}}{1 - S_{w,r}},$$  \hfill (6.6)

where $S_{w,r}$ is the residual wetting phase saturation which cannot be displaced by another fluid phase and remains in the porous medium.

Relative permeability

The presence of two fluid phases in the porous medium reduces the space available for flow for each of the fluid phases. This increases the resistance to flow of the phases, which is accounted for by the means of the relative permeability $k_{r,\alpha}$, which scales the intrinsic permeability. It is a value between zero and one, depending on the saturation. The relations describing the relative permeabilities of the wetting and non-wetting phase are different as the wetting phase predominantly occupies small pores and the edges of larger pores while the non-wetting phases occupies large pores. The relative permeabilities for the wetting phase $k_{r,w}$ and the non-wetting phase are e.g. calculated as (also by Brooks and Corey [7]):

$$k_{r,w} = S_e^{2+3/\lambda},$$  \hfill (6.7)

and

$$k_{r,n} = (1 - S_e)^2 \left(1 - S_e^{2+3/\lambda}\right).$$  \hfill (6.8)

6.1.7 Transport processes in porous media

On the macro-scale, the transport of mass can be grouped according to the driving force of the transport process. Pressure gradients result in the advective transport of a fluid phase and all the components constituting the phase, while concentration gradients result in the diffusion of a component within a phase.
Advection

Advection transport is determined by the flow field. On the macro-scale, the velocity $v$ is calculated using the Darcy equation depending on the potential gradient $(\nabla p_\alpha - \rho_\alpha g)$, accounting for both pressure difference and gravitation, the intrinsic permeability of the porous medium, and the viscosity $\mu$ of the fluid phase:

$$v = -\frac{K}{\mu}(\nabla p - \rho g)$$  \hspace{1cm} (6.9)

$v$ is proportional to $(\nabla p - \rho g)$ with the proportional factor $K/\mu$. This equation can be extended to calculate the velocity $v_\alpha$ of phase $\alpha$ in the case of two-phase flow by considering the relative permeability $k_{r,\alpha}$ (Section 6.1.6):

$$v_\alpha = -\frac{k_{r,\alpha} K}{\mu_\alpha} (\nabla p_\alpha - \rho_\alpha g)$$  \hspace{1cm} (6.10)

Diffusion

Molecular diffusion is a process determined by the concentration gradient. It is commonly modeled as Fickian diffusion following Fick’s first law:

$$j_\kappa d = -\rho_\alpha D_\kappa^\alpha \nabla X_\kappa^\alpha$$  \hspace{1cm} (6.11)

where $D_\kappa^\alpha$ is the molecular diffusion coefficient of component $\kappa$ in phase $\alpha$. In a porous medium, the actual path lines are tortuous due to the impact of the solid matrix. This tortuosity and the impact of the presence of multiple fluid phases is accounted for by using an effective diffusion coefficient $D_{p_{\text{m}},\alpha}^\kappa$:

$$D_{p_{\text{m}},\alpha}^\kappa = \phi_\alpha S_\alpha D_\kappa^\alpha$$  \hspace{1cm} (6.12)

where $\tau_\alpha$ is the tortuosity of phase $\alpha$.

6.1.8 Gas mixing laws

Prediction of the $p - \rho - T$ behavior of gas mixtures is typically based on two (contradicting) concepts: Dalton’s law or Amagat’s law. In the following the two concepts will be explained in more detail.

Dalton’s law

Dalton’s law assumes that the gases in the mixture are non-interacting (with each other) and each gas independently applies its own pressure (partial pressure), the sum of which is the total pressure:

$$p = \sum_i p_i.$$  \hspace{1cm} (6.13)

Here $p_i$ refers to the partial pressure of component $i$. As an example, if two equal volumes of gas A and gas B are mixed, the volume of the mixture stays the same but the pressures add up (see Figure 6.2). The density of the mixture, $\rho$, can be calculated as follows:
or for an arbitrary number of gases:

$$\rho = \sum_i \rho_i; \quad \rho_m = \sum_i \rho_{m,i}.$$  (6.15)

**Amagat’s law**

Amagat’s law assumes that the volumes of the component gases are additive; the interactions of the different gases are the same as the average interactions of the components. This is known as Amagat’s law:

$$V = \sum_i V_i.$$  (6.16)

As an example, if two volumes of gas A and B at equal pressure are mixed, the pressure of the mixture stays the same, but the volumes add up (see Figure 6.3). The density of the mixture, $\rho$, can be calculated as follows:

$$\rho = \frac{m}{V} = \frac{m_A + m_B}{V} = \frac{\rho_A V + \rho_B V}{V} = \rho_A + \rho_B,$$  \hspace{1cm} (6.14)

or for an arbitrary number of gases:

$$\rho = \sum_i \frac{x_i}{\rho_i}; \quad \rho_m = \sum_i \frac{x_i}{\rho_{m,i}}.$$  (6.18)
Ideal gases

An ideal gas is defined as a gas whose molecules are spaced so far apart that the behavior of a molecule is not influenced by the presence of other molecules. This assumption is usually valid at low pressures and high temperatures. The ideal gas law states that, for one gas:

\[ p = \frac{\varrho RT}{M}; \quad p = \varrho_m RT. \]  

(6.19)

Using the assumption of ideal gases and either Dalton’s law or Amagat’s law lead to the density of the mixture, \( \varrho \), as:

\[ \varrho = \frac{p}{RT} \sum_i M_i x_i; \quad \varrho_m = \frac{p}{RT}. \]  

(6.20)

6.1.9 Available Models


6.2 Temporal Discretization and Solution Strategies

In this section, the temporal discretization as well as solution strategies (monolithic/sequential) are presented.

6.2.1 Temporal discretization

Our systems of partial differential equations are discretized in space and in time.

Let us consider the general case of a balance equation of the following form

\[ \frac{\partial m(u)}{\partial t} + \nabla \cdot f(u, \nabla u) + q(u) = 0, \]  

(6.21)

seeking an unknown quantity \( u \) in terms of storage \( m \), flux \( f \) and source \( q \). All available Dumux models can be written mathematically in form of (6.21) with possibly vector-valued quantities \( u, m, q \) and a tensor-valued flux \( f \). For the sake of simplicity, we assume scalar quantities \( u, m, q \) and a vector-valued flux \( f \) in the notation below.

For discretizing (6.21) we need to choose an approximation for the temporal derivative \( \frac{\partial m(u)}{\partial t} \). While many elaborate methods for this approximation exist, we focus on the simplest one of a first order difference quotient

\[ \frac{\partial m(u_{k+1})}{\partial t} \approx \frac{m(u_{k+1}) - m(u_k)}{\Delta t_{k+1}} \]  

(6.22)

for approximating the solution \( u \) at time \( t_{k+1} \) (forward) or \( t_k \) (backward). The question of whether to choose the forward or the backward quotient leads to the explicit and implicit Euler method, respectively. In case of the former, inserting (6.22) in (6.21) at time \( t_k \) leads to

\[ \frac{m(u_{k+1}) - m(u_k)}{\Delta t_{k+1}} + \nabla \cdot f(u_k, \nabla u_k) + q(u_k) = 0, \]  

(6.23)
whereas the implicit Euler method is described as
\[
m(u_{k+1}) - m(u_k) \frac{\Delta t_{k+1}}{\Delta t_k} + \nabla \cdot f(u_{k+1}, \nabla u_{k+1}) + q(u_{k+1}) = 0.
\] (6.24)

Once the solution \( u_k \) at time \( t_k \) is known, it is straightforward to determine \( m(u_{k+1}) \) from (6.23), while attempting to do the same based on (6.24) involves the solution of a system of equations. On the other hand, the explicit method (6.23) is stable only if the time step size \( \Delta t_{k+1} \) is below a certain limit that depends on the specific balance equation, whereas the implicit method (6.24) is unconditionally stable.

### 6.2.2 Solution strategies to solve equations

The governing equations of each model can be solved monolithically or sequentially. The basic idea of the sequential algorithm is to reformulate the equations of multi-phase flow into one equation for pressure and equations for phase/component/... transport. The pressure equation is the sum of the mass balance equations and thus considers the total flow of the fluid system. The new set of equations is considered as decoupled (or weakly coupled) and can thus be solved sequentially. The most popular sequential model is the fractional flow formulation for two-phase flow which is usually implemented applying an IMplicit Pressure Explicit Saturation algorithm (IMPES). In comparison to solving the equations monolithically, the sequential structure allows the use of different discretization methods for the different equations. The standard method used in the sequential algorithm is a cell-centered finite volume method. Further schemes, so far only available for the two-phase pressure equation, are cell-centered finite volumes with multi-point flux approximation (Mpfa-O method) and mimetic finite differences. An \( h \)-adaptive implementation of both sequential algorithms is provided for two dimensions.

### 6.3 Spatial Discretization

We discretize space with cell-centered finite volume methods (6.3.1), the box method (6.3.2) or a staggered grid scheme (6.3.3). Grid adaption is available for both box and cell-centered finite volume method. In general, the spatial parameters, especially the porosity, have to be assigned on the coarsest level of discretization.

#### 6.3.1 Cell Centered Finite Volume Methods – A Short Introduction

Cell-centered finite volume methods use the elements of the grid as control volumes. For each control volume the discrete values are determined at the element/control volume center (not required to be the barycenters).

We consider a domain \( \Omega \subset \mathbb{R}^d \), \( d \in \{2,3\} \) with boundary \( \Gamma = \partial \Omega \). Within this section, we consider the following elliptic problem
\[
\nabla \cdot (-\Lambda \nabla u) = q \quad \text{in } \Omega \\
(-\Lambda \nabla u) \cdot n = v_N \quad \text{on } \Gamma_N \\
u = u_D \quad \text{on } \Gamma_D.
\] (6.25)

Here, \( \Lambda = \Lambda(x,u) \) is a symmetric and positive definite tensor of second rank (e.g. permeability, diffusivity, etc.), \( u = u(x) \) is unknown and \( q = q(x,u) \) is a source/sink. We denote by \( \mathcal{M} \) the mesh.
that results from the division of the domain $\Omega$ into $n_c$ control volumes $K \subset \Omega$. Each $K$ is a polygonal open set such that $K \cap L = \emptyset$, $\forall K \neq L$ and $\Omega = \cup_{K \in M} K$.

For the derivation of the finite-volume formulation we integrate the first equation of (6.25) over a control volume $K$ and apply the Gauss divergence theorem:

$$\int_{\partial K} (-\Lambda \nabla u) \cdot n \, d\Gamma = \int_K q \, dx. \quad (6.26)$$

Splitting the control volume boundary $\partial K$ into a finite number of faces $\sigma \subset \partial K$ (such that $\sigma = K \cap L$ for some neighboring control volume $L$) and replacing the exact fluxes by an approximation, i.e. $F_{K,\sigma} \approx \int_{\sigma} (-\Lambda_K \nabla u) \cdot n \, d\Gamma$ (here $\Lambda_K$ is the value of $\Lambda$ associated with control volume $K$), yield

$$\sum_{\sigma \subset \partial K} F_{K,\sigma} = Q_K, \quad \forall K \in M, \quad (6.27)$$

where $F_{K,\sigma}$ is the discrete flux through face $\sigma$ flowing out of cell $K$ and $Q_K := \int_K q \, dx$ is the integrated source/sink term. Equation (6.27) is the typical cell-centered finite-volume formulation. Finite-volume schemes differ in the way how the term $(\Lambda_K \nabla u) \cdot n$ is approximated (i.e. the choice of the fluxes $F_{K,\sigma}$). Using the symmetry of the tensor $\Lambda_K$, this term can be rewritten as $\nabla u \cdot \Lambda_K n$, which corresponds to the directional derivative of $u$ in co-normal direction $\Lambda_K n$. In the following, the main ideas of the two-point flux approximation and the multi-point flux approximation methods are briefly described. Hereby, we restrict the discussion to the two-dimensional case.

Please also note that other types of equations, e.g. instationary parabolic problems, can be discretized by applying some time discretization scheme to the time derivatives and by using the finite-volume scheme for the flux discretization. For simplicity the discussion is restricted to the elliptic problem (6.25).

**Tpfa Method**

The linear two-point flux approximation is a simple but robust cell-centered finite-volume scheme, which is commonly used in commercial software. This scheme can be derived by using the co-normal decomposition, which reads

$$\Lambda_K n_{K,\sigma} = t_{K,\sigma} d_{K,\sigma} + d^\perp_{K,\sigma}, \quad t_{K,\sigma} = \frac{n_{K,\sigma}^T A_K d_{K,\sigma}}{d_{K,\sigma}^T d_{K,\sigma}}, \quad d^\perp_{K,\sigma} = \Lambda_K n_{K,\sigma} - t_{K,\sigma} d_{K,\sigma}, \quad (6.28)$$

with the tensor $\Lambda_K$ associated with control volume $K$, the distance vector $d_{K,\sigma} := x_\sigma - x_K$ and $d_{K,\sigma}^T d_{K,\sigma} = 0$, see Figure 6.4 for the used notations. The same can be done for the conormal $\Lambda_L n_{L,\sigma}$. The $t_{K,\sigma}$ and $t_{L,\sigma}$ are the transmissibilities associated with the face $\sigma$. These transmissibilities are calculated in DuMu$^x$ by using the function `computeTpfaTransmissibility`.

With these notations, it follows that for each cell $K$ and face $\sigma$

$$\nabla u \cdot \Lambda_K n_{K,\sigma} = t_{K,\sigma} \nabla u \cdot d_{K,\sigma} + \nabla u \cdot d^\perp_{K,\sigma}. \quad (6.29)$$

For the Tpfa scheme, the second part in the above equation is neglected. By using the fact that $\nabla u \cdot d_{K,\sigma} \approx u_\sigma - u_K$, the discrete fluxes for face $\sigma$ are given by

$$F_{K,\sigma} = -|\sigma| t_{K,\sigma} (u_\sigma - u_K), \quad F_{L,\sigma} = -|\sigma| t_{L,\sigma} (u_\sigma - u_L). \quad (6.30)$$
Enforcing local flux conservation, i.e. \( F_{K,\sigma} + F_{L,\sigma} = 0 \), results in
\[
\begin{align*}
\mathbf{u}_{\sigma} &= \frac{t_{K,\sigma} \mathbf{u}_K + t_{L,\sigma} \mathbf{u}_L}{t_{K,\sigma} + t_{L,\sigma}}.
\end{align*}
\] (6.31)

With this, the fluxes (6.30) are rewritten as
\[
\begin{align*}
F_{K,\sigma} &= |\sigma| \frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}} (\mathbf{u}_K - \mathbf{u}_L), \quad F_{L,\sigma} &= |\sigma| \frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}} (\mathbf{u}_L - \mathbf{u}_K).
\end{align*}
\] (6.32)

By neglecting the orthogonal term, the consistency of the scheme is lost for general grids, where \( \nabla \mathbf{u} \cdot \mathbf{d}_{K,\sigma} \neq 0 \). The consistency is achieved only for so-called K-orthogonal grids for which \( \mathbf{d}_{K,\sigma} = 0 \).

For such grids we deduce that
\[
\begin{align*}
t_{K,\sigma} t_{L,\sigma} = \tau_{K,\sigma} \tau_{L,\sigma},
\end{align*}
\] (6.33)

with \( \tau_{K,\sigma} := \mathbf{n}_{K,\sigma} \Lambda_{K} \mathbf{n}_{K,\sigma} \), \( \tau_{L,\sigma} := \mathbf{n}_{L,\sigma} \Lambda_{L} \mathbf{n}_{L,\sigma} \), \( d_{K,\sigma} := \mathbf{n}_{K,\sigma} \cdot \mathbf{d}_{K,\sigma} \), and \( d_{L,\sigma} := \mathbf{n}_{L,\sigma} \cdot \mathbf{d}_{L,\sigma} \). This reduces, for the case of scalar permeability, to a distance weighted harmonic averaging of permeabilities.

Mpfa Method

Expressions for the face fluxes \( F_{K,\sigma} \) are obtained by introducing intermediate face unknowns \( \mathbf{u}_{\sigma} \) in addition to the cell unknowns \( \mathbf{u}_K \) and enforcing the physically motivated continuity of fluxes and continuity of the solution across the faces. For a face \( \sigma \) between the two polygons \( K \) and \( L \) these conditions read:
\[
\begin{align*}
F_{K,\sigma} + F_{L,\sigma} &= 0, \\
\mathbf{u}_{K,\sigma} &= \mathbf{u}_{L,\sigma} = \mathbf{u}_{\sigma}.
\end{align*}
\] (6.34)

Using these conditions, the intermediate face unknowns \( \mathbf{u}_{\sigma} \) can be eliminated and the fluxes are expressed as a function of the cell unknowns \( \mathbf{u}_N \) and associated transmissibilities \( t_{K,\sigma}^N \):
\[
F_{K,\sigma} = \sum_{N \in S_{K,\sigma}} t_{K,\sigma}^N \mathbf{u}_N.
\] (6.35)

The main difference between the various finite-volume schemes available is the assembly of the face fluxes, i.e. the computation of the \( t_{K,\sigma}^N \) and the size of \( S_{K,\sigma} \). For the Tpfa, that has been presented in
Figure 6.5: Interaction region for the Mpfa-O method. The graphic on the right illustrates how the sub-control volume \( L \) and face \( \sigma \) are embedded in cell \( L \). Note that the face stencils for all sub-control volume faces in the depicted interaction region are \( S_{\sigma} = \{ K, L, M \} \), meaning that the fluxes over the sub-control volume faces depend on the three cell unknowns \( u_K, u_L, u_M \).

In the following, a multi-point flux approximation method (Mpfa-O method), which was introduced in Aavatsmark [1], is presented. The main difference to the Tpfa scheme is the fact that a consistent discrete gradient is constructed, i.e. the term \( \nabla u \cdot d_{K,\sigma} \) is not neglected.

For this scheme, a dual grid is created by connecting the barycenters of the cells with the barycenters of the faces \((d = 2)\) or the barycenters of the faces and edges \((d = 3)\). This divides each cell into sub-control volumes \( K \). Analogously, each face is sub-divided into sub-control volume faces \( \sigma \), see Figure 6.5. We allow for piecewise constant \( \Lambda \) (denoted as \( \Lambda_K \) for each cell \( K \)) and construct discrete gradients \( \nabla_{K,\sigma} u \) (per sub-control volume \( K \)). In the following, we restrict our discussion to the two-dimensional setup that is shown in Figure 6.5. Here, the discrete gradients are constructed to be consistent such that the following conditions hold:

$$
\nabla_{K,\sigma} u \cdot (x_{\sigma_1} - x_K) = u_{\sigma_1} - u_K, \quad \nabla_{K,\sigma} u \cdot (x_{\sigma_3} - x_K) = u_{\sigma_3} - u_K.
$$

Thus, a discrete gradient (for sub-control volume \( K \)) that fulfills these conditions is given as

$$
\nabla_{K,\sigma} u = \mathbb{D}_{K,\sigma} \begin{bmatrix} u_{\sigma_1} - u_K \\ u_{\sigma_3} - u_K \end{bmatrix}, \quad \text{with } \mathbb{D}_{K,\sigma} := \begin{bmatrix} x_{\sigma_1} - x_K \\ x_{\sigma_3} - x_K \end{bmatrix}.
$$
This enables us to write the discrete flux across $\sigma_1^v$ from cell $K$ as follows:

$$F_{K,\sigma_1^v} := -|\sigma_1^v| n_T^\sigma_1^v A_K \nabla_D K u.$$ (6.38)

Inserting the discrete gradient, yields

$$F_{K,\sigma_1^v} = \omega_{K,\sigma_1^v} \sigma_1^v (u_K - u_{\sigma_1^v}) + \omega_{K,\sigma_2^v} \sigma_2^v (u_K - u_{\sigma_2^v}),$$ (6.39)

with $(\omega_{K,\sigma_1^v}, \omega_{K,\sigma_2^v})^T = |\sigma_1^v| D_K^{-1} A_K n_{\sigma_1^v}$. These values are calculated in DuMu$^x$ by using the function `computeMpfaTransmissibility`.

To deduce a cell-centered scheme, the introduced face unknowns $u_{\sigma_i^v}$ have to be eliminated. This is done by enforcing flux continuity for each sub-control volume face, i.e.

1. $F_{K,\sigma_1^v} + F_{L,\sigma_1^v} = 0$,
2. $F_{K,\sigma_2^v} + F_{M,\sigma_2^v} = 0$,
3. $F_{L,\sigma_2^v} + F_{M,\sigma_2^v} = 0$.

This results in a system of equations for the face unknowns $u_{\sigma}$

$$A_{3\times3} u_{\sigma} = B_{3\times3} u,$$ (6.43)

where $u$ contains the three cell unknowns $u_K, u_L, u_M$ and $u_{\sigma}$ the three face unknowns $u_{\sigma_1^v}, u_{\sigma_2^v}, u_{\sigma_3^v}$.

Inserting these face unknowns into the flux expression (6.39) yields

$$F_{K,\sigma_1^v} = \sum_{N \in \{K,L,M\}} t^N_{K,\sigma_1^v} u_N = t_{K,\sigma_1^v} \cdot u,$$ (6.44)

for each cell $K$ and sub-control volume face $\sigma_i^v$.

### 6.3.2 Box Method – A Short Introduction

The so called box method unites the advantages of the finite-volume (FV) and finite-element (FE) methods.

First, the model domain $\Omega$ is discretized with a FE mesh consisting of nodes $i$ and corresponding elements $E_k$. Then, a secondary FV mesh is constructed by connecting the midpoints and barycenters of the elements surrounding node $i$ creating a box $B_i$ around node $i$ (see Figure 6.6a).

The FE mesh divides the box $B_i$ into subcontrolvolumes (scv’s) $b_i^k$ (see Figure 6.6b). Figure 6.6c shows the finite element $E_k$ and the scv’s $b_i^k$ inside $E_k$, which belong to four different boxes $B_i$. Also necessary for the discretization are the faces of the subcontrolvolumes (scvf’s) $e_{ij}^k$ between the scv’s $b_i^k$ and $b_j^k$, where $|e_{ij}^k|$ is the length of the scvf. The integration points $x_{ij}^k$ on $e_{ij}^k$ and the outer normal vector $n_{ij}^k$ are also to be defined (see Figure 6.6c).

The advantage of the FE method is that unstructured grids can be used, while the FV method is mass conservative. The idea is to apply the FV method (balance of fluxes across the interfaces) to each FV box $B_i$ and to get the fluxes across the interfaces $e_{ij}^k$ at the integration points $x_{ij}^k$ from the FE approach. Consequently, at each scvf the following expression results:
In the following, the discretization of the balance equation is going to be derived. From the Reynolds transport theorem follows the general balance equation:

\[
\int_{\Omega} \frac{\partial}{\partial t} u \, dx + \int_{\partial\Omega} (vu + w) \cdot n \, d\Gamma = \int_{\Omega} q \, dx \quad (6.46)
\]

\[
f(u) = \int_{\Omega} \frac{\partial u}{\partial t} \, dx + \int_{\Omega} \nabla \cdot [uv + w(u)] \, dx - \int_{\Omega} q \, dx = 0 \quad (6.47)
\]

where term 1 describes the changes of entity \( u \) within a control volume over time, term 2 the advective, diffusive and dispersive fluxes over the interfaces of the control volume and term 3 is the source and sink term. \( \Omega \) denotes the model domain and \( F(u) = F(v, p) = F(v(x, t), p(x, t)) \).

Like the FE method, the box method follows the principle of weighted residuals. In the function \( f(u) \) the unknown \( u \) is approximated by discrete values at the nodes of the FE mesh \( \tilde{u}_i \) and linear basis functions \( N_i \) yielding an approximate function \( f(\tilde{u}) \). For \( u \in \{v, p, x^k\} \) this means:
\[ \tilde{p} = \sum_i N_i \hat{p}_i \quad (6.48) \]
\[ \nabla \tilde{p} = \sum_i \nabla N_i \hat{p}_i \quad (6.51) \]
\[ \tilde{\nu} = \sum_i N_i \hat{\nu}_i \quad (6.49) \]
\[ \nabla \tilde{\nu} = \sum_i \nabla N_i \hat{\nu}_i \quad (6.52) \]
\[ \tilde{x}^\kappa = \sum_i N_i \hat{x}^\kappa_i \quad (6.50) \]
\[ \nabla \tilde{x}^\kappa = \sum_i \nabla N_i \hat{x}^\kappa_i. \quad (6.53) \]

Due to the approximation with node values and basis functions the differential equations are not exactly fulfilled anymore but a residual \( \varepsilon \) is produced.

\[ f(u) = 0 \quad \Rightarrow \quad f(\tilde{u}) = \varepsilon \quad (6.54) \]

Application of the principle of weighted residuals, meaning the multiplication of the residual \( \varepsilon \) with a weighting function \( W_j \) and claiming that this product has to vanish within the whole domain,

\[ \int_{\Omega} \varepsilon W_j \, dx = 0 \quad \text{with} \quad \sum_j W_j = 1 \quad (6.55) \]

yields the following equation:

\[ \int_{\Omega} \frac{\partial \tilde{u}}{\partial t} W_j \, dx + \int_{\Omega} [\nabla \cdot F(\tilde{u})] W_j \, dx - \int_{\Omega} q W_j \, dx = \int_{\Omega} \varepsilon W_j \, dx = 0. \quad (6.56) \]

For standard Galerkin schemes, the weighting functions \( W_j \) are chosen the same as the ansatz functions \( N_j \). However, this does not yield a locally mass-conservative scheme. Therefore, for the Box method, the weighting functions \( W_j \) are chosen as the piece-wise constant functions over a control volume box \( B_j \), i.e.

\[ W_j(x) = \begin{cases} 1 & x \in B_j \\ 0 & x \notin B_j. \end{cases} \quad (6.57) \]

Thus, the Box method is a Petrov-Galerkin scheme, where the weighting functions do not belong to the same function space than the ansatz functions.

Inserting definition (6.57) into equation (6.56) and using the Green-Gaussian integral theorem results in

\[ \int_{B_j} \frac{\partial \hat{u}}{\partial t} \, dx + \int_{\partial B_j} F(\tilde{u}) \cdot n \, d\Gamma_{B_j} - \int_{B_j} q \, dx = 0, \quad (6.58) \]

which has to hold for every box \( B_j \).

The first term in equation (6.58) can be written as

\[ \int_{B_j} \frac{\partial \hat{u}}{\partial t} \, dx = \frac{d}{dt} \int_{B_j} \hat{u}_i N_i \, dx = \sum_i \frac{\partial \hat{u}_i}{\partial t} \int_{B_j} N_i \, dx. \quad (6.59) \]

Here, a mass lumping technique is applied by assuming that the storage capacity is reduced to the nodes. This means that the integrals \( M_{i,j} = \int_{B_j} N_i \, dx \) are replaced by some mass lumped terms \( M_{i,j}^{\text{lump}} \) which are defined as

\[ M_{i,j}^{\text{lump}} = \begin{cases} |B_j| & j = i \\ 0 & j \neq i, \end{cases} \quad (6.60) \]
where $|B_j|$ is the volume of the FV box $B_j$ associated with node $j$. The application of this assumption yields

$$
|B_j| \frac{\partial \tilde{u}_j}{\partial t} + \int_{\partial B_j} F(\tilde{u}) \cdot n \, d\Gamma_{B_j} - Q_j = 0,
$$

(6.61)

where $Q_j$ is an approximation (using some quadrature rule) of the integrated source/sink term $\int_{B_j} q \, dx$.

Using an implicit Euler time discretization finally leads to the discretized form which will be applied to the mathematical flow and transport equations:

$$
|B_j| \frac{\tilde{u}_j^{n+1} - \tilde{u}_j^n}{\Delta t} + \int_{\partial B_j} F(\tilde{u}_j^{n+1}) \cdot n \, d\Gamma_{B_j} - Q_j^{n+1} = 0.
$$

(6.62)

Equation (6.62) has to be fulfilled for each box $B_j$.

### 6.3.3 Staggered Grid – A Short Introduction

![Discretization of the staggered-grid method](image)

- Cell-centered primary variables ($p_\alpha, x_\alpha, T, k, \ldots$)
- $v_{\alpha,x}$
- $v_{\alpha,y}$
- finite volume mesh and boundaries of the control volumes centered around the scalar variables
- boundaries of the staggered control volumes located around the velocity $x$-component
- boundaries of the staggered control volumes located around the velocity $y$-component

Figure 6.7: Discretization of the staggered-grid method. The figure shows the different control volume arrangements, which are staggered with respect to each other. There are the control volumes centered around the scalar primary variables in black, the control volumes located around the $x$-component of the velocity in blue and the control volumes located around the $y$-components of the velocity in red. The control volume boundaries are given by lines. Additionally, there is one shaded example control volume each.

In the two-dimensional free-flow models, the continuity equation is discretized using the black control volumes, the $x$-component of the momentum equation is discretized using the blue control volumes and the $y$-component is discretized using the red control volumes. In three dimensions this works analogously.
The staggered-grid or marker-and-cell method uses a finite volume method with different control volumes for different equations. There are control volumes centered around the scalar primary variables. They correspond to the finite volume mesh. Additionally, there are control volumes located around the \( x \), \( y \) and (in 3D) \( z \) velocity components which are shifted in the \( x \), \( y \) and \( z \) direction, such that the velocity components are located on the edges of the cell-centered finite volume mesh (see Figure 6.7). As for the cell-centered method, the fluxes are evaluated at the edges of each control volume with a two-point flux approximation, cf. 6.3.1.

The staggered-grid method is robust, mass conservative, and free of pressure oscillations but should, as the cell-centered TPFA method, only be applied for structured grids. Currently, all free-flow models in DuMu\( ^x \) use the staggered-grid discretization.

### 6.4 Steps of a DuMu\( ^x \) Simulation

This chapter is supposed to give a short overview over how things are “handed around” in DuMu\( ^x \). It is not a comprehenisve guide through the modeling framework of DuMu\( ^x \), but hopefully it will help getting to grips with it.

In Section 6.4.1 the structure of DuMu\( ^x \) is shown from a content point of view.

#### 6.4.1 Structure – by Content

In Figure 6.8 the algorithmic representations of a monolithical solution scheme is illustrated down to the element level.

![Algorithmic representation of a monolithical solution scheme](#)

Figure 6.8: Structure of a monolithical solution scheme in DuMu\( ^x \).
6.4.2 Structure – by Implementation

A possible starting point to understand how the above mentioned algorithm is implemented within DuMu\(^x\), is the example main file [https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-course/releases/3.1/exercises/exercise-mainfile/exercise_1p_a.cc](https://git.iws.uni-stuttgart.de/dumux-repositories/dumux-course/releases/3.1/exercises/exercise-mainfile/exercise_1p_a.cc)

6.5 Property System

A high level overview over the property system’s design and principle ideas are given, then follows a reference and a self-contained example.

6.5.1 Motivation and features

The DuMu\(^x\) property system is a traits system which allows easy inheritance.

In the context of the DuMu\(^x\) property system, a property is an arbitrary class body which may contain type definitions, values and methods.

Just like normal classes, properties can be arranged in hierarchies. In the context of the DuMu\(^x\) property system, nodes of the inheritance hierarchy are called type tags.

It also supports property nesting. Property nesting means that the definition of a property can depend on the value of other properties which may be defined for arbitrary levels of the inheritance hierarchy.

6.5.2 How-to

All source files which use the property system should include the header file `dumux/common/properties/propertysystem.hh`. Declaration of type tags and property tags as well as defining properties must be done inside the namespace Dumux::Properties.

Defining Type Tags

New nodes in the type tag hierarchy can be defined in the TTag namespace using

```cpp
1 // Create new type tags
2 namespace TTag {
3 struct NewTypeTagName { using InheritsFrom = std::tuple<BaseTagName1, BaseTagName2, ...>; };
4 } // end namespace TTag
```

where the `InheritsFrom` alias is optional. To avoid inconsistencies in the hierarchy, each type tag may be defined only once for a program. If you call `GetProp` the property system will first look for the properties defined in `BaseTagName1` in the `InheritsFrom` list. If a defined property is found this property is returned. If no defined property is found the search will continue in the ancestors of `BaseTagName1`. If again no defined property is found the search will continue in the second `BaseTagName2` in the list, and so on. If no defined property is found at all, a compiler error is triggered.

Example:

```cpp
1 namespace Dumux {
2 namespace Properties {
3 namespace TTag {
4 struct MyBaseTypeTag1 {};
5 struct MyBaseTypeTag2 {};
```
Defining new Property Tags

New property tags are defined using

```cpp
template<class TypeTag, class MyTypeTag> struct NewPropTagName { using type = UndefinedProperty; };
```

Example:
```cpp
namespace Dumux {
    namespace Properties {
        template<class TypeTag, class MyTypeTag> struct MyPropertyTag { using type = UndefinedProperty; };
    }
}
```

If you need to forward declare a property use
```cpp
// forward declaration
template<class TypeTag, class MyTypeTag> struct NewPropTagName;
```

Defining Properties

The value of a property on a given node of the type tag hierarchy is defined using

```cpp
template<class TypeTag> struct PropertyTagName<TypeTag, TTag::TypeTagName> {
    // arbitrary body of a struct
};
```

This means a property is defined for a specific type tag node `TTag::TypeTagName` by providing a partial template specialization of `PropertyTagName`. The body typically contains either the alias `type`, or a data member `value`. However, you can of course write in the body whatever you like.

```cpp
template<class TypeTag> struct PropertyTagName<TypeTag, TTag::TypeTagName> { using type = type; };
template<class TypeTag> struct PropertyTagName<TypeTag, TTag::TypeTagName> { static constexpr bool value = booleanValue; };
template<class TypeTag> struct PropertyTagName<TypeTag, TTag::TypeTagName> { static constexpr int value = integerValue; };
```

Example:
```cpp
namespace Dumux {
    namespace Properties {
        // Create new type tag
        namespace TTag {
            struct MyTypeTag {};
        }
        // Define some properties
    }
}
```
Retrieving Property Values

The type of a property can be retrieved using

```
using Prop = GetProp<TypeTag, Properties::PropertyTag>;
```

There is a helper struct and a helper function to retrieve the `type` and `value` members of a property

```
using PropType = GetPropType<TypeTag, Properties::PropertyTag>;
constexpr auto propValue = getPropValue<TypeTag, Properties::PropertyTag>();
```

Example:

```
template <TypeTag>
class MyClass {

    // retrieve the `::value` attribute of the `UseMoles` property
    static constexpr bool useMoles = getPropValue<TypeTag, Properties::UseMoles>();
    static constexpr bool useMoles2 = GetProp<TypeTag, Properties::UseMoles>::value; // equivalent

    // retrieve the `::type` attribute of the `Scalar` property
    using Scalar = GetPropType<TypeTag, Properties::Scalar>;
    using Scalar2 = GetProp<TypeTag, Properties::Scalar>::type; // equivalent
};
```

Nesting Property Definitions

Inside property definitions there is access to all other properties which are defined somewhere on the type tag hierarchy. The node for which the current property is requested is available via the template argument `TypeTag`. Inside property class bodies `GetPropType` can be used to retrieve other properties and create aliases.

Example:
6 Advanced DuMuX – Detailed Instructions

Figure 6.9: (a) A possible property inheritance graph for various kinds of cars. The lower nodes inherit from higher ones; Inherited properties from nodes on the right take precedence over the properties defined on the left. (b) Property names which make sense for at least one of the car types of (a).

```cpp
template<class TypeTag>
struct Vector<TypeTag, TTag::MyModelTypeTag>
{
    using Scalar = GetPropType<TypeTag, Properties::Scalar>;
    using type = std::vector<Scalar>;
};
```

### 6.5.3 A Self-Contained Example

As a concrete example, let us consider some kinds of cars: Compact cars, sedans, trucks, pickups, military tanks and the Hummer-H1 sports utility vehicle. Since all these cars share some characteristics, it makes sense to inherit those from the closest matching car type and only specify the properties which are different. Thus, an inheritance diagram for the car types above might look like outlined in Figure 6.9a.

Using the DuMuX property system, this inheritance hierarchy is defined by:

```cpp
#include <dumux/common/propertysystem.hh>
#include <iostream>
namespace Dumux {
namespace Properties {
namespace TTag {
    struct CompactCar {}; 
    struct Truck {}; 
    struct Tank {}; 
    struct Sedan { using InheritsFrom = std::tuple<CompactCar>; }; 
    struct Pickup { using InheritsFrom = std::tuple<Truck, Sedan>; }; 
    struct HummerH1 { using InheritsFrom = std::tuple<Tank, Pickup>; }; 
}}
```  

Figure 6.9b lists a few property names which make sense for at least one of the nodes of Figure 6.9a. These property names can be defined as follows:

```cpp
template<class TypeTag, class MyTypeTag>
struct GasUsage { using type = UndefinedProperty; }; // [l/100km]
template<class TypeTag, class MyTypeTag>
```
So far, the inheritance hierarchy and the property names are completely separate. What is missing is setting some values for the property names on specific nodes of the inheritance hierarchy. Let us assume the following:

- For a compact car, the top speed is the gas usage in \( \frac{1}{100 \text{km}} \) times 30, the number of seats is 5 and the gas usage is \( 4 \frac{1}{100 \text{km}} \).
- A truck is by law limited to \( 100 \frac{\text{km}}{\text{h}} \) top speed, the number of seats is 2, it uses \( 18 \frac{1}{100 \text{km}} \) and has a cargo payload of 35 t.
- A tank exhibits a top speed of \( 60 \frac{\text{km}}{\text{h}} \), uses \( 65 \frac{1}{100 \text{km}} \) and features a 120 mm diameter cannon.
- A sedan has a gas usage of \( 7 \frac{1}{100 \text{km}} \), as well as an automatic transmission, in every other aspect it is like a compact car.
- A pick-up truck has a top speed of \( 120 \frac{\text{km}}{\text{h}} \) and a payload of 5 t. In every other aspect it is like a sedan or a truck but if in doubt, it is more like a truck.
- The Hummer-H1 SUV exhibits the same top speed as a pick-up truck. In all other aspects it is similar to a pickup and a tank, but, if in doubt, more like a tank.

Using the DuMu\(^x\) property system, these assumptions are formulated using

```cpp
struct TopSpeed { using type = UndefinedProperty; }; // [km/h]
template<class TypeTag, class MyTypeTag>
struct NumSeats { using type = UndefinedProperty; }; // []
template<class TypeTag, class MyTypeTag>
struct AutomaticTransmission { using type = UndefinedProperty; }; // true/false
template<class TypeTag, class MyTypeTag>
struct CanonCaliber { using type = UndefinedProperty; }; // [mm]
template<class TypeTag, class MyTypeTag>
struct Payload { using type = UndefinedProperty; }; // [t]
```

```cpp
// For a compact car, the top speed is the gas usage in \( \frac{1}{100 \text{km}} \) times 30, the number of seats is 5 and the gas usage is \( 4 \frac{1}{100 \text{km}} \).

```cpp
template<class TypeTag, class TTag::CompactCar>
{ static constexpr int value = getPropValue<TypeTag, Properties::GasUsage>() * 30; }
```

```cpp
// A truck is by law limited to \( 100 \frac{\text{km}}{\text{h}} \) top speed, the number of seats is 2, it uses \( 18 \frac{1}{100 \text{km}} \) and has a cargo payload of 35 t.

```cpp
template<class TypeTag, class TTag::Truck>
{ static constexpr int value = 100; }
```

```cpp
// A tank exhibits a top speed of \( 60 \frac{\text{km}}{\text{h}} \), uses \( 65 \frac{1}{100 \text{km}} \) and features a 120 mm diameter cannon.

```cpp
template<class TypeTag, class TTag::Tank>
{ static constexpr int value = 60; }
```

```cpp
// A sedan has a gas usage of \( 7 \frac{1}{100 \text{km}} \), as well as an automatic transmission, in every other aspect it is like a compact car.

```cpp
template<class TypeTag, class TTag::Sedan>
{ static constexpr int value = 7; }
```

```cpp
// A pick-up truck has a top speed of \( 120 \frac{\text{km}}{\text{h}} \) and a payload of 5 t. In every other aspect it is like a sedan or a truck but if in doubt, it is more like a truck.

```cpp
template<class TypeTag, class TTag::Pickup>
{ static constexpr int value = 120; }
```

```cpp
// The Hummer-H1 SUV exhibits the same top speed as a pick-up truck. In all other aspects it is similar to a pickup and a tank, but, if in doubt, more like a tank.

```cpp
template<class TypeTag, class TTag::HummerH1>
{ static constexpr int value = 60; }
```
```
Now property values can be retrieved and some diagnostic messages can be generated. For example

```cpp
int main()
{
    std::cout << "top speed of sedan: " << getPropValue<Properties::TTag::Sedan, Properties::TopSpeed>() << "\n";
    std::cout << "top speed of truck: " << getPropValue<Properties::TTag::Truck, Properties::TopSpeed>() << "\n";
}
```

will yield the following output:

```
$ top speed of sedan: 210
$ top speed of truck: 100
```
Gmsh Mesh Format

Gmsh is an open-source flexible grid generator for unstructured finite-element meshes ([11], http://geuz.org/gmsh/). DuMuX supports the default Gmsh mesh format (MSH). For the format specifics and how to create grids with Gmsh, e.g., using the provided GUI, we refer to the Gmsh documentation (http://geuz.org/gmsh/doc/texinfo/gmsh.html).

The MSH format can contain element and boundary markers defined on the grid. Thus, boundaries can be easily marked, e.g., inflow boundaries using Gmsh. Further, the format supports higher order elements. They can be used to create boundary parametrization supported by, e.g., the grid manager UGGrid. An example can be found in dumux/test/io/gridmanager.

Eclipse Grid Format

The Eclipse Grid Format (GRDECL) is commonly used for corner-point grids. Such grids consist of hexahedra, which are described by eight points on so-called pillars. A special feature of corner-point geometries is that points on pillars can degenerate, meaning that two neighboring points on a pillar can coincide. Furthermore, faces are, in general, bi-linear and cells can be non-convex. This allows for the accurate description of faults, layers, or wells, occurring in geological environments.

Furthermore, petrophysical properties can be defined (for each cell), by using eclipse-specific keywords, e.g., PORO, PERMX, PERMY.

DuMuX supports the Eclipse Grid Format by using the opm-grid module (see https://opm-project.org). An example can be found in dumux/test/porousmediumflow/2p/implicit/cornerpoint.

VTK File Format

VTK format uses ASCII or XML format. It is mostly used by DuMuX for output purposes and can be visualized by programs such as Paraview, ViIt or Tecplot. Using VTK files to input grid and parameter data is also possible. An example can be found in dumux/test/io/gridmanager.

Other Grid Formats

Grid formats other than DGF, MSH, GRDECL, or VTK will have to be converted to the DGF, MSH, GRDECL, or VTK format before they can be used in DuMuX. If conversion is not an option, another possibility would be to write your own GridManagers. Examples of other grid formats, which have previously been either converted or custom-created in DuMuX, are ArtMesh grids (fractured network grids), and ICEM grids (CAD developed grids).

6.6.2 The DuMuX GridManager

The Dumux::GridManager class constructs the grid from information in the input file and handles the data. Currently, supported Dune grid interface implementations are YaspGrid, OneDGrid, dune-uggrid, dune-alugrid, dune-foamgrid, dune-subgrid, opm-grid (cornerpoint grids) and dune-spgrid. Grids can be constructed from a DGF, VTK or MSH file by simply providing the filename to the grid in the Grid group of the input file:

\[ \text{Group Name: Grid} \]

\[ \text{Property: GridParameterGroup} \]

This way, it is possible, e.g., for problems with more than one grid, to set different
If you are using an unstructured grid interface like UGGrid or FOAMGrid, constructing a grid from a VTK or MSH is just changing a line:

```
[Grid]
File = myuggrid.msh
```

DuMu\textsuperscript{x} will tell you in case your selected grid manager does not support reading such files.

You want to initially refine your grid? It’s just adding a line:

```
[Grid]
File = mydgfgrid.dgf
Refinement = 4
```

When reading a MSH or VTK file, further parameters are recognized. \texttt{Verbose} enables verbose output on grid construction when set to 1. \texttt{BoundarySegments} enables reading parameterized boundaries. \texttt{PhysicalEntities} enables reading boundary and element flags.

### Parameters specific to the grid implementation

The \texttt{Dumux::GridManager} supports also a selection of parameters that are specific to the chosen grid manager. To give an example, we take a look at the unstructured grid UGGrid. UGGrid supports red-green refinement per default. One can turn off the green closure by setting the grid’s closure type

```
[Grid]
File = mydgfgrid.dgf
ClosureType = None # or Green
```

For all available parameters see the Doxygen documentation.

### Structured grids

If you want to construct a structured grid without using a specific grid file, insert the following into the input file:

```
[Grid]
LowerLeft = 00 0
UpperRight = 11 1
Cells = 1010 20
```

where \texttt{LowerLeft} is a vector to the lower left corner of the grid and \texttt{UpperRight} a vector to the upper right corner. \texttt{Cells} is a vector with the number of cells in each coordinate direction. Note, that for a grid in a two-dimensional world, the vectors only have two entries.

Depending on the grid manager, further parameters are recognized. UGGrid, for example, supports simplex elements as well as hexahedral elements (called “cube” in DUNE). When creating a structured grid, we can select the cell type as follows

```
[Grid]
LowerLeft = 00 0
UpperRight = 11 1
Cells = 1010 20
CellType = Cube # or Simplex
```

For all available parameters see the Doxygen documentation.
Other DuMu\textsuperscript{x} GridManagers

- **CakeGridManager**: Provides a method to create a piece of cake grid.
- **CpGridManager**: Reads the GRDECL file and generates a corner-point grid.
- **SubgridGridManager**: Creates a dune-subgrid for some given host grid.

6.6.3 Input and Output formats

**VTK file format**

DuMu\textsuperscript{x} allows to write out simulation results via the `VtkOutputModule`. For every print-out step, a single VTU file is created. For parallel simulations one file per print-out step is generated for each processor. The PVD file groups the single VTU files and contains additionally the time step information. The VTK file format is supported by common visualisation programs like ParaView, VisIt, and Tecplot.

**Customize the VTK output**

Using the respective `initOutputModule` function of the model `IOFields`, a default set of variables is defined for the output into the VTK files. It is also possible to add further variables, using the member function `addField` of the `VtkOutputModule`. For example, to add a variable called `temperatureExact`:

```cpp
vtkWriter.addField(problem->getExactTemperature(), "temperatureExact");
```

The first input argument of this method is the value of the additional variable, provided by a method of the corresponding problem. If it does not already exists, the user has to provide this method.

```cpp
const std::vector<Scalar>& getExactTemperature()
{
  return temperatureExact_;}
```

It is important that the life-time of the added field exceeds the life-time of the writer. That means you can’t pass temporaries to the `addField` function. The vector has to be stored somewhere, e.g. in the program main file.

The second input argument is the name of the additional variable (as it should be written in the VTK files). The example above is taken from:

test/porousmediumflow/1pnc/implicit/1p2c/nonisothermal/convection/main.cc

**VTK as input format**

There is support for reading data and grids from VTK files, see subsection 6.6.1

**Gnuplot interface**

DuMu\textsuperscript{x} provides a small interface to GNUPlot, which can be used to plot results and generate image files (e.g., png). To use the gnuplot, gnuplot has to be installed. For more information see 5.6.2
Container I/O
DuMu\textsuperscript{x} supports writing to file from and reading to some standard C++ containers like std::vector<double> or std::vector<Dune::FieldVector>. If you want to read and write simple vectors, have a look at the header dumux/io/container.hh.

Matrix and Vector I/O

\texttt{dune-istl} supports writing and reading vectors and matrices to/from different format. For example you can write a matrix in a sparse matrix format that can be read by Matlab (see \texttt{dune/istl/io.hh}).

6.7 Parallel Computation

This section explains how DuMu\textsuperscript{x} can be used on multicore / multinode systems.

There are different concepts and methods for parallel programming, which are often grouped in \textit{shared-memory} and \textit{distributed-memory} approaches. The parallelization in DuMu\textsuperscript{x} is based on the model supported by Dune which is currently based on \textit{Message Passing Interface} (MPI) (distributed-memory approach).

The main idea behind the MPI parallelization is the concept of \textit{domain decomposition}. For parallel simulations, the computational domain is split into subdomains and one process (\textit{rank}) is used to solve the local problem of each subdomain. During the global solution process, some data exchange between the ranks/subdomains is needed. MPI is used to send data to other ranks and to receive data from other ranks. The domain decomposition in Dune is handled by the grid managers. The grid is partitioned and distributed on several nodes. Most grid managers contain own domain decomposition methods to split the computational domain into subdomains. Some grid managers also support external tools like METIS, ParMETIS, PTScotch or ZOLTAN for partitioning. On the other hand, linear algebra types such as matrices and vectors do not know that they are in a parallel environment. Communication is then handled by the components of the parallel solvers. Currently, the only parallel solver backend is \texttt{Dumux::AMGBackend}, a parallel AMG-preconditioned BiCGSTAB solver.

In order for DuMu\textsuperscript{x} simulation to run in parallel, an MPI library (e.g. OpenMPI, MPICH or IntelMPI) implementation must be installed on the system.

6.7.1 Prepare a Parallel Application

Not all parts of DuMu\textsuperscript{x} can be used in parallel. In order to switch to the parallel \texttt{Dumux::AMGBackend} solver backend include the respective header

\begin{verbatim}
#include <dumux/linear/amgbackend.hh>
\end{verbatim}

Second, the linear solver must be switched to the AMG backend

\begin{verbatim}
using LinearSolver = Dumux::AMGBackend<TypeTag>;
\end{verbatim}

and the application must be recompiled. The parallel \texttt{Dumux::AMGBackend} instance has to be constructed with a \texttt{Dune::GridView} object and a mapper, in order to construct the parallel index set needed for communication.

\begin{verbatim}
auto linearSolver = std::make_shared<LinearSolver>(leafGridView, gridGeometry->dofMapper());
\end{verbatim}
6.7.2 Run a Parallel Application

The starting procedure for parallel simulations depends on the chosen MPI library. Most MPI implementations use the `mpirun` command

```
mpirun -np <n_cores> <executable_name>
```

where `-np` sets the number of cores (`n_cores`) that should be used for the computation. On a cluster you usually have to use a queuing system (e.g. `slurm`) to submit a job. Check with your cluster administrator how to run parallel applications on the cluster.

6.7.3 Handling Parallel Results

For serial computations, DuMu\textsuperscript{x} produces single vtu-files as default output format. During a simulation, one VTU file is written for every output step. In the parallel case, one VTU file for each step and processor is created. For parallel computations, an additional variable "process rank" is written into the file. The process rank allows the user to inspect the subdomains after the computation. The parallel VTU files are combined in a single pvd file like in sequential simulations that can be opened with e.g. ParaView.
Bibliography


