DuMu$^\mathsf{X}$

Handbook

Version 3.7,
Handbook version from March 27, 2024

Lehrstuhl für Hydromechanik und Hydrosystemmodellierung,

Universität Stuttgart, Paffenwaldring 61, D-70569 Stuttgart, Germany

`http://dumux.org`

Contents

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, 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 built on top of DUNE, the **D**istributed and **U**nified **N**umerics **E**nvironment [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[1].
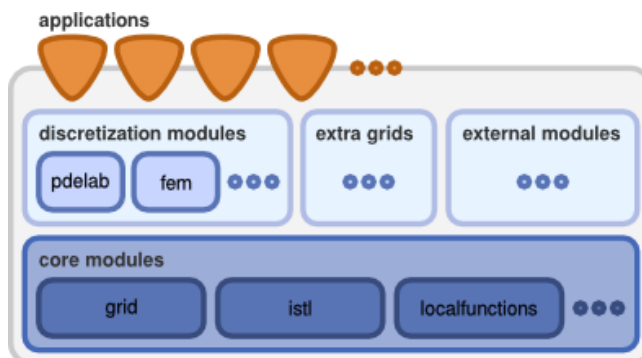

Figure 1.1: A high-level overview of DUNE's design is available on the project's web site [10].

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 that applied hexahedrons as cells,

---

[1]In fact, the performance penalty resulting from the use of DUNE's grid interface is usually negligible [8].

the 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`). 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 **I**terative **S**olver **T**emplate **L**ibrary and provides generic, highly optimized linear algebra routines for solving the generated systems.

DuMu^x comes in the 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^x is to provide a framework for 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^x includes ready-to-use numerical models and a few example applications.

This document is the handbook to a new minor version update of DuMu^x: version 3.7. The release contains improvements and new features compared to version 3.6. The update is backwards compatible with the last release 3.6. To facilitate the transition for our users, we have created a changelog helping to update programs from version 3.6 to version 3.7 and giving an overview of new capabilities. It is available online at `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^x and wish everyone an exciting simulation experience.

2 Overview and Infrastructure

This chapter provides an overview of the general structure in DuMu$^\text{x}$ (2.1) and gives help for basic work with DuMu$^\text{x}$ (2.2-2.5). Further, it presents useful external tools (2.6) and basic concepts (2.8).

2.1 Directory Structure

DuMu$^\text{x}$ 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$^\text{x}$

- `doc`: files necessary for the Doxygen documentation and this handbook, and various logos

- `dumux`: the main folder, containing the source files. See Fig. 2.1 for a visualized structure. For more information on the models, have a look at the Doxygen documentation.

- `examples`: well-documented examples of applying DuMu$^\text{x}$ to typical physical problems. In the `README.md` files, the setup is explained, the used code is presented as well as 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.

2.2 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

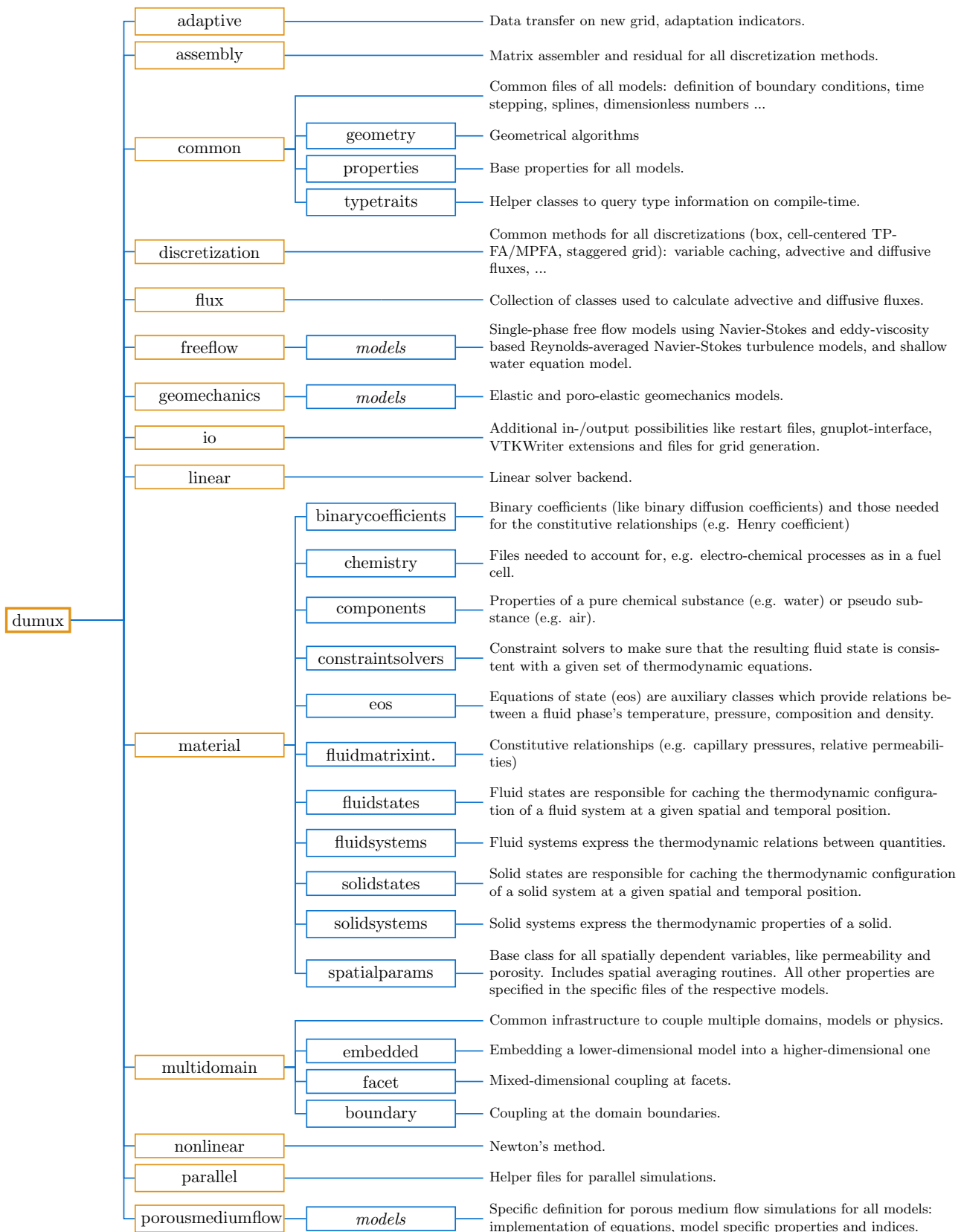|		
adaptive		Data transfer on new grid, adaptation indicators.
assembly		Matrix assembler and residual for all discretization methods.
common		Common files of all models: definition of boundary conditions, time stepping, splines, dimensionless numbers ...
	geometry	Geometrical algorithms
	properties	Base properties for all models.
	typetraits	Helper classes to query type information on compile-time.
discretization		Common methods for all discretizations (box, cell-centered TP-FA/MPFA, staggered grid): variable caching, advective and diffusive fluxes, ...
flux		Collection of classes used to calculate advective and diffusive fluxes.
freeflow	models	Single-phase free flow models using Navier-Stokes and eddy-viscosity based Reynolds-averaged Navier-Stokes turbulence models, and shallow water equation model.
geomechanics	models	Elastic and poro-elastic geomechanics models.
io		Additional in-/output possibilities like restart files, gnuplot-interface, VTKWriter extensions and files for grid generation.
linear		Linear solver backend.
material	binarycoefficients	Binary coefficients (like binary diffusion coefficients) and those needed for the constitutive relationships (e.g. Henry coefficient)
	chemistry	Files needed to account for, e.g. electro-chemical processes as in a fuel cell.
	components	Properties of a pure chemical substance (e.g. water) or pseudo substance (e.g. air).
	constraintsolvers	Constraint solvers to make sure that the resulting fluid state is consistent with a given set of thermodynamic equations.
	eos	Equations of state (eos) are auxiliary classes which provide relations between a fluid phase's temperature, pressure, composition and density.
	fluidmatrixint.	Constitutive relationships (e.g. capillary pressures, relative permeabilities)
	fluidstates	Fluid states are responsible for caching the thermodynamic configuration of a fluid system at a given spatial and temporal position.
	fluidsystems	Fluid systems express the thermodynamic relations between quantities.
	solidstates	Solid states are responsible for caching the thermodynamic configuration of a solid system at a given spatial and temporal position.
	solidsystems	Solid systems express the thermodynamic properties of a solid.
	spatialparams	Base class for all spatially dependent variables, like permeability and porosity. Includes spatial averaging routines. All other properties are specified in the specific files of the respective models.
multidomain		Common infrastructure to couple multiple domains, models or physics.
	embedded	Embedding a lower-dimensional model into a higher-dimensional one
	facet	Mixed-dimensional coupling at facets.
	boundary	Coupling at the domain boundaries.
nonlinear		Newton's method.
parallel		Helper files for parallel simulations.
porousmediumflow	models	Specific definition for porous medium flow simulations for all models: implementation of equations, model specific properties and indices.

dumux

Figure 2.1: Structure of the directory `dumux` containing the DuMu$^\text{x}$ 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:

```
1  dumux_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-2.7/`.

2.3 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 the getting started guide you will get the following output at the end of the simulation run [1]:

```
# 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"
ResidualReduction = "1e-13"
Verbosity = "0"
[ LinarSolver.Preconditioner ]
Iterations = "1"
Relaxation = "1.0"
[ Newton ]
EnableAbsoluteResidualCriterion = "false"
```

---

[1]If you did not get the output, add `Parameters::print();` to your main file.

```
EnableChop = "false"
EnableResidualCriterion = "false"
EnableShiftCriterion = "true"
MaxAbsoluteResidual = "1e-5"
MaxRelativeShift = "1e-8"
MaxSteps = "18"
MinSteps = "2"
ResidualReduction = "1e-5"
SatisfyResidualAndShiftCriterion = "false"
TargetSteps = "10"
UseLineSearch = "false"
[ TimeLoop ]
MaxTimeStepSize = "1e300"
[ Vtk ]
AddProcessRank = "true"
AddVelocity = "false"

# Unused parameters:
Grid.LowerLeft = "0 0"
```

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)

2.3.1 Parameter Values

To get the value of an input parameter please use:

```
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:

```
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:

```
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:

```
[ Model0.Grid ]
File = file0.dgf
[ Model1.Grid ]
File = file1.dgf
```

2.4 Restart DuMuˣ Simulations

DuMuˣ has some experimental support for check-pointing (restarting paused/stopped/crashed simulations). You can restart a DuMuˣ 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 DuMuˣ 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.

Writing double-precision floating point numbers to VTK files is available since DUNE release 2.7. If you are using that version, it is now possible to specify output precision in the input file using `Vtk.Precision` followed by either `Float32`, `Float64`, `UInt32`, `UInt8` or `Int32`. `Float32` is set as the default. We especially advice the use of `Float64` when working with restart files.

The restart capabilities will hopefully be improved in future versions of DuMuˣ-3. We are looking forward to any contributions (especially HDF5 / XDMF support, improvement of VTK support).

2.5 Developing DuMuˣ

2.5.1 Communicate with DuMuˣ 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 DuMuˣ 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 DuMuˣ-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.

2.5.2 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 DuMuˣ visit `https://git.iws.uni-stuttgart.de/dumux-repositories/dumux/blob/master/CONTRIBUTING.md`.

2.5.3 Tips and Tricks

DuMu^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^x are built with the help of `dunecontrol`. Per default, DuMu^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:

```
1  set(CMAKE_BUILD_TYPE Debug)
```

Afterwards rerun cmake (run cmake <`path-to-build-dir`>).

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^x. A more complex example would be a case in which you have to configure and build only e.g. DUNE-grid and DuMu^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^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__`:

```
1  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 choose 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:

```
1  // define the minimal debug level somewhere in your code
2  #define DUNE_MINIMAL_DEBUG_LEVEL 4
3  Dune::dgrave << "message"; // will be printed
4  Dune::dwarn  << "message"; // will be printed
5  Dune::dinfo  << "message"; // will NOT be printed
```

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.

2.6 External Tools

2.6.1 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

2.6.2 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.

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

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

```
1  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:

```
1  gnuplot.resetPlot();                          // reset the plot
2  gnuplot.setXRange(0.0, 72000.0);              // specify xmin and xmax
3  gnuplot.setYRange(0.0, 1.0);                  // specify ymin and ymax
4  gnuplot.setXlabel("time [s]");                // set xlabel
5  gnuplot.setYlabel("mole fraction mol/mol");   // set ylabel
6
```

```
7    // set x-values, y-values, the name of the data file and the Gnuplot options
8    gnuplot.addDataSetToPlot(x, y, "N2.dat", options);
9
10   gnuplot.plot("mole_fraction_N2");          // set the name of the output file
```

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`).

2.6.3 Gstat

Gstat is an open source software tool which generates geostatistical random fields (see `www.gstat.org`). In order to use gstat, execute the `bin/installexternal.py` from your DuMu$^\text{x}$ 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).

2.6.4 ParaView

To visualize the simulation data you have produced using DuMu$^\text{x}$, we recommend using Paraview. This open-source software supports DuMu$^\text{x}$'s standard data formats, and can be operated either with a GUI or with batching tools.

2.7 Scripts

A suite of scripts is available within the dumux repository to assist in the performance of common tasks. These scripts can be found in the `dumux/bin/` directory. The majority of these scripts began as bash (*.sh) scripts, but most have been replaced with more portable and user-friendly Python versions. Some of these scripts are used in the installation or the creation of working dumux-dependent repositories; these are outlined in section 2.7.1. Others can be used to perform post-processing tasks after simulation data has been produced; these are outlined in section 2.7.2. In order to maintain the suite of unit tests and examples within this repository, some testing scripts are also required; these are outlined in section 2.7.3. All remaining tests are mentioned in section 2.7.4.

2.7.1 Repository development Scripts:

- Installation Scripts: `installdumux.py` and `installexternal.py` both have been developed to make the installation of the base Dumux suite and any external dependencies as simple as possible.

- Module Development: When developing within the dumux simulation environment, it is recommended to do so within your own module. When sharing this module with others, or installing it on multiple machines, it is often difficult to find exactly which version you are using so that your work will run in different places. To simplify this, the following scripts are available in the `util/` directory:

  - The script `createdockerimage.py` can help to create a docker image to put your work in a container for simple delivery.

– In order to extract the content and dependencies within your model, `extractmodulepart.py` will collect all headers used to run a specified test and collect them all in a functioning external module.

– To document the exact versions that have been used, the scripts `getusedversions.py` is available to search each through an existing simulation dumux suite and find the branch, commit hash, and commit date of the current implementation.

– The script `create_cmakelists.py` can be used to integrate new header files to the dune/-dumux cmake system.

– In order to produce an installscript that will install your repository and its dependencies on another computer, the `makeinstallscript.py` is available. This script will collect all of the dependencies and their versions using the `getmoduleinfo.py` and the other scripts listed above and write a script to install a new version in the same configuration. In the past it has been recommended to run the script `test_dumux.sh` to ensure that the final dunecontrol and compilation pass have worked.

2.7.2 Post-processing Scripts:

Data visualization and post processing is an integral part of any exercise in simulation. A few scripts are available to begin with in the `postprocessing/` subdirectory.

• Paraview data collection scripts: Although there are many tools available within the open-source visualization program paraview (see 2.6.4), performing consistent data collection from your simulations can be tedious when analyzing many output files. The scripts `exportscreenshot2d.py`, `extractlinedata.py`, and `extractpointdataovertime.py` are each scripts that can help to perform paraview data collection commands consistently and each produce easy to use `.csv` files or images.

• Error convergence script: When evaluating errors to a reference solution across grid refinements, the L2 error is typically evaluated. A script to help perform this is available in `l2error.py`.

2.7.3 Testing Scripts:

In order to ensure consistency when developing in dumux, developers are encouraged to introduce unit tests and reference tests that can be run to ensure new code introductions do not break existing functions. While existing simulations should each test something when the `ctest` command is called, a few scripts exist in the `testing/` subdirectory to make this simple to develop and quick to evaluate.

• Run tests: In order to automatically find and run tests that fit a specific criteria, the scripts `findtests.py`, `runtest.py` and `runselectedtests.py` each are available. These are each used to automate the testing process.

• Reference Solution Comparisons: The scripts `fuzzycomparevtu.py` and `fuzzycomparedata.py` are both used to compare simulation results against reference simulation results. When running a test, one can use these files to compare the output from the test against reference simulation results. Should these results differ, the difference for each field is shown. These scripts are typically directly called during testing.

2.7.4 Miscellaneous:

In addition, a few miscellaneous scripts are available.

- Parameter scripts: The script `doc/generate_parameterlist.py` is available for tracking all of the runtime parameters used.

- Other: `doc/getcontributors.sh` can be used to find the names and contact information for all commit authors over a given time period.

2.8 Assembling the linear system

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

2.8.1 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 $\mathbf{u}$. The sum of the terms is called residual $\mathbf{r}(\mathbf{u})$ which is a function of the solution. For example:

$$\underbrace{\phi\frac{\partial \varrho_\alpha S_\alpha}{\partial t} - \mathrm{div}\left(\varrho_\alpha \frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}\left(\mathbf{grad}\ p_\alpha - \varrho_\alpha\mathbf{g}\right)\right) - q_\alpha}_{=:\,\mathbf{r}(\mathbf{u})} = 0$$

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

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}}\mathbf{r}\left(\mathbf{u}^i\right) = J_{\mathbf{r}(\mathbf{u}^i)} = \left(\frac{\mathrm{d}}{\mathrm{d}\mathbf{u}_m^i}\mathbf{r}\left(\mathbf{u}^i\right)_n\right)_{m,n}$$

with i denoting the Newton iteration step. Each column is the residual derived with respect to the mth entry of $\mathbf{u}^i$.

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

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

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

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

2.8.2 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 block structure. Each block contains the degrees of freedom (also
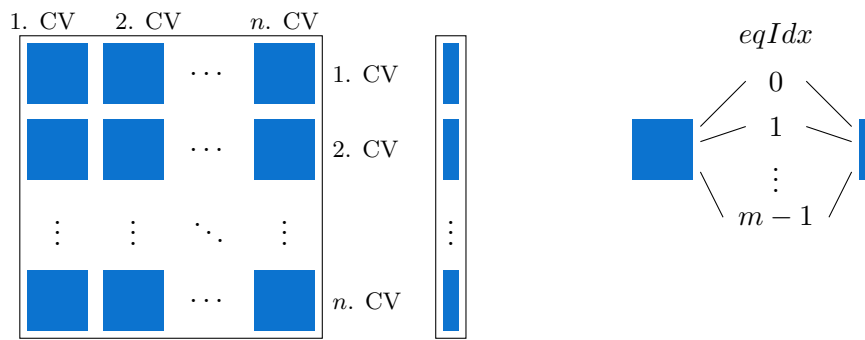
Figure 2.2: Structure of matrix and vector, left: block structure, right: within block

called variables or unknowns) for a control volume. The equation index is used to order the degrees of freedom. For each control volume we have one block. The mapper is used to order the blocks.

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].

3 Advanced DuMuˣ – Detailed Instructions

This chapter contains detailed information for those who are interested in deeper modifications of underlying DuMuˣ models, classes, functions, etc.

3.1 Physical Basics

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

3.1.1 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 nonwetting 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. 3.1 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 α, e.g. w, n and g in the example of Fig. 3.1, refers to the phase, while the superscript κ, e.g. w, a and c in the example of Fig. 3.1, refers to the component.

p_α	phase pressure	ϕ	porosity
T	temperature	K	absolute permeability tensor
S_α	phase saturation	τ	tortuosity
x_α^κ	mole fraction of component κ in phase α	$\boldsymbol{g}$	gravitational acceleration
X_α^κ	mass fraction of component κ in phase α	q_α^κ	volume source term of κ in α
$\varrho_{\text{mol},\alpha}$	molar density of phase α	u_α	specific internal energy
ϱ_α	mass density of phase α	h_α	specific enthalpy
M	molar mass of a phase or component	c_{s}	specific heat enthalpy
$k_{\text{r}\alpha}$	relative permeability	λ_{pm}	heat conductivity
μ_α	phase viscosity	q^h	heat source term
D_α^κ	diffusivity of component κ in phase α	$\boldsymbol{v}_{a,\alpha}$	advective velocity
$\boldsymbol{v}_\alpha$	velocity (Darcy or free flow)		

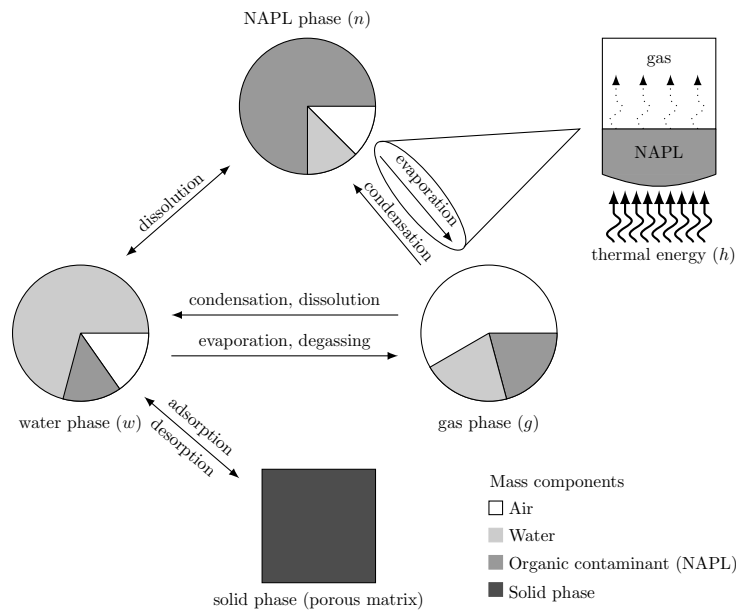Table 3.1: Notation list for most of the variables and indices used in DuMu$^{\text{x}}$.

Figure 3.1: Mass and energy transfer between the phases in a water-NAPL-gas system [9]

3.1.2 Scale[1]

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$^{\text{x}}$.

3.1.3 Porous medium properties[1]

Porosity

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

$$\phi = \frac{V_{\text{fluid}}}{V_{\text{total}}} = 1 - \frac{V_{\text{solid}}}{V_{\text{total}}}. \tag{3.1}$$

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 $\mathbf{K}$. For isotropic porous media, it can be reduced to a scalar quantity K.

---

[1]This subsection is taken from [13] in a slightly adapted form.

3.1.4 Mass fraction, mole fraction[1]

The composition of a phase is described by mass or mole fractions of the components. The mole fraction x_α^κ of component κ in phase α is defined as:

$$x_\alpha^\kappa = \frac{n_\alpha^\kappa}{\sum_i n_\alpha^i},$$

(3.2)

where n_α^κ is the number of moles of component κ in phase α. The mass fraction X_α^κ is defined similarly using the mass of component κ in phase α instead of n_α^κ, $X_\alpha^\kappa = \text{mass}_\alpha^\kappa/\text{mass}_\alpha^\text{total}$. The molar mass M^κ of the component κ relates the mass fraction to the mole fraction and vice versa.

3.1.5 Fluid properties[1]

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

Density

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

Viscosity

The dynamic viscosity μ_α characterizes the resistance of a fluid to flow. As density, it is a fluid phase property. For Newtonian fluids, it relates the shear stress τ_s to the velocity gradient $dv_{\alpha,x}/dy$:

$$\tau_\text{s} = \mu_\alpha \frac{dv_{\alpha,x}}{dy}.$$

(3.3)

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

3.1.6 Fluid phase interactions in porous media[1]

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_α of a phase α 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}:

$$p_{\mathrm{c}} = p_{\mathrm{n}} - p_{\mathrm{w}}. \tag{3.4}$$

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 λ describing the pore-size distribution:

$$p_{\mathrm{c}} = p_{\mathrm{d}} S_{\mathrm{e}}^{-\frac{1}{\lambda}}, \tag{3.5}$$

with

$$S_{\mathrm{e}} = \frac{S_{\mathrm{w}} - S_{\mathrm{w,r}}}{1 - S_{\mathrm{w,r}}}, \tag{3.6}$$

where $S_{\mathrm{w,r}}$ 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 $k_{\mathrm{r},\alpha}$, 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 nonwetting phase are different as the wetting phase predominantly occupies small pores and the edges of larger pores while the nonwetting phases occupies large pores. The relative permeabilities for the wetting phase $k_{\mathrm{r},\mathrm{w}}$ and the nonwetting phase are e.g. calculated as (also by Brooks and Corey [7]):

$$k_{\mathrm{r},\mathrm{w}} = S_{\mathrm{e}}^{\frac{2+3\lambda}{\lambda}} \tag{3.7}$$

and

$$k_{\mathrm{r},\mathrm{n}} = (1 - S_{\mathrm{e}})^2 \left(1 - S_{\mathrm{e}}^{\frac{2+\lambda}{\lambda}} \right). \tag{3.8}$$

3.1.7 Transport processes in porous media [1]

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

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

$$\mathbf{v} = -\frac{\mathbf{K}}{\mu}(\nabla p - \rho \mathbf{g}). \tag{3.9}$$

$\mathbf{v}$ is proportional to $(\nabla p - \rho \mathbf{g})$ with the proportional factor $\mathbf{K}/\mu$. This equation can be extended to calculate the velocity $\mathbf{v}_\alpha$ of phase α in the case of two-phase flow by considering the relative permeability $k_{\mathrm{r},\alpha}$ (Section 3.1.6):

$$\mathbf{v}_\alpha = -\frac{k_{\mathrm{r},\alpha}\mathbf{K}}{\mu_\alpha}(\nabla p_\alpha - \rho_\alpha \mathbf{g}) \tag{3.10}$$

Diffusion

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

$$\mathbf{j_d} = -\rho_\alpha D_\alpha^\kappa \nabla X_\alpha^\kappa, \tag{3.11}$$

where D_α^κ is the molecular diffusion coefficient of component κ in phase α. 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 $D_{\mathrm{pm},\alpha}^\kappa$:

$$D_{\mathrm{pm},\alpha}^\kappa = \phi \tau_\alpha S_\alpha D_\alpha^\kappa, \tag{3.12}$$

where τ_α is the tortuosity of phase α.

3.1.8 Gas mixing laws

Prediction of the $p - \varrho - T$ 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:

$$p = \sum_i p_i. \tag{3.13}$$

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 3.2). The density of the mixture, ϱ, can be calculated as follows:

Gas A   Gas B   Mixture AB

V $+$ V $\longrightarrow$ V

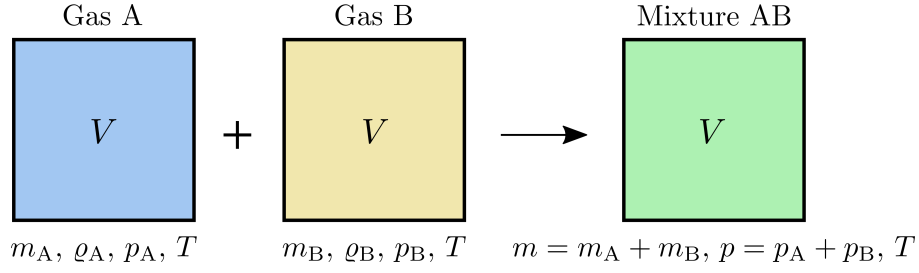$m_A,\ \varrho_A,\ p_A,\ T$ $m_B,\ \varrho_B,\ p_B,\ T$ $m = m_A + m_B,\ p = p_A + p_B,\ T$

Figure 3.2: Dalton's law visualized

$$\varrho = \frac{m}{V} = \frac{m_A + m_B}{V} = \frac{\varrho_A V + \varrho_B V}{V} = \varrho_A + \varrho_B, \tag{3.14}$$

or for an arbitrary number of gases:

$$\varrho = \sum_i \varrho_i; \quad \varrho_m = \sum_i \varrho_{m,i}. \tag{3.15}$$

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:

$$V = \sum_i V_i. \tag{3.16}$$

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 3.3). The density of the mixture, ϱ, can be

Gas A   Gas B   Mixture AB

V_A $+$ V_B $\longrightarrow$ $V = V_A + V_B$

$m_A,\ \varrho_A,\ p,\ T$ $m_B,\ \varrho_B,\ p,\ T$ $m = m_A + m_B,\ p,\ T$
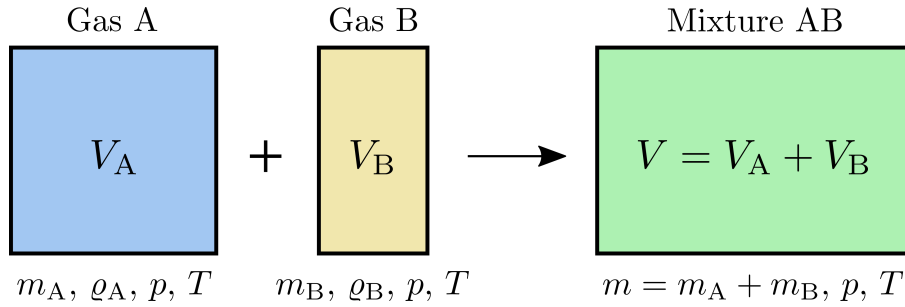
Figure 3.3: Amagat's law visualized

calculated as follows:

$$\varrho = \frac{m}{V} = \frac{m}{V_A + V_B} = \frac{m}{\frac{m_A}{\varrho_A} + \frac{m_B}{\varrho_B}} = \frac{m}{\frac{X_A m}{\varrho_A} + \frac{X_B m}{\varrho_B}} = \frac{1}{\frac{X_A}{\varrho_A} + \frac{X_B}{\varrho_B}}, \tag{3.17}$$

or for an arbitrary number of gases:

$$\varrho = \frac{1}{\sum_i \frac{X_i}{\varrho_i}}; \quad \varrho_m = \frac{1}{\sum_i \frac{x_i}{\varrho_{m,i}}}. \tag{3.18}$$

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:

$$p = \varrho \frac{RT}{M}; \quad p = \varrho_m RT. \tag{3.19}$$

Using the assumption of ideal gases and either Dalton's law or Amagat's law lead to the density of the mixture, ϱ, as:

$$\varrho = \frac{p}{RT} \sum_i M_i x_i; \quad \varrho_m = \frac{p}{RT}. \tag{3.20}$$

3.1.9 Available Models

A list of all available models can be found in the Doxygen documentation at `https://dumux.org/docs/doxygen/releases/3.7/modules.html`. The documentation includes a detailed description for every model.

3.2 Temporal Discretization and Solution Strategies

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

3.2.1 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

$$\frac{\partial m(u)}{\partial t} + \nabla \cdot \mathbf{f}(u, \nabla u) + q(u) = 0, \tag{3.21}$$

seeking an unknown quantity u in terms of storage m, flux $\mathbf{f}$ and source q. All available Dumux models can be written mathematically in form of (3.21) with possibly vector-valued quantities u, m, q and a tensor-valued flux $\mathbf{f}$. For the sake of simplicity, we assume scalar quantities u, m, q and a vector-valued flux $\mathbf{f}$ in the notation below.

For discretizing (3.21), we need to choose an approximation for the temporal derivative $\partial m(u)/\partial t$. While many elaborate methods for this approximation exist, we focus on the simplest one of a first order difference quotient

$$\frac{\partial m(u_{k/k+1})}{\partial t} \approx \frac{m(u_{k+1}) - m(u_k)}{\Delta t_{k+1}} \tag{3.22}$$

for approximating the solution u at time t_k (forward) or t_{k+1} (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 (3.22) in (3.21) at time t_k leads to

$$\frac{m(u_{k+1}) - m(u_k)}{\Delta t_{k+1}} + \nabla \cdot \mathbf{f}(u_k, \nabla u_k) + q(u_k) = 0, \tag{3.23}$$

whereas the implicit Euler method is described as

$$\frac{m(u_{k+1}) - m(u_k)}{\Delta t_{k+1}} + \nabla \cdot \mathbf{f}(u_{k+1}, \nabla u_{k+1}) + q(u_{k+1}) = 0. \tag{3.24}$$

Once the solution u_k at time t_k is known, it is straightforward to determine $m(u_{k+1})$ from (3.23), while attempting to do the same based on (3.24) involves the solution of a system of equations. On the other hand, the explicit method (3.23) is stable only if the time step size Δt_{k+1} is below a certain limit that depends on the specific balance equation, whereas the implicit method (3.24) is unconditionally stable.

3.2.2 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.

3.3 Spatial Discretization

We discretize space with cell-centered finite volume methods (3.3.1), the box method (3.3.2) or a staggered grid scheme (3.3.3). 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.

3.3.1 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

$$\begin{aligned} \nabla \cdot (-\mathbf{\Lambda}\nabla u) &= q & &\text{in } \Omega \\ (-\mathbf{\Lambda}\nabla u) \cdot \mathbf{n} &= v_N & &\text{on } \Gamma_N \\ u &= u_D & &\text{on } \Gamma_D. \end{aligned} \tag{3.25}$$

Here, $\mathbf{\Lambda} = \mathbf{\Lambda}(\mathbf{x}, \mathbf{u})$ is a symmetric and positive definite tensor of second rank (e.g. permeability, diffusivity, etc.), $u = u(\mathbf{x})$ is unknown and $q = q(\mathbf{x}, \mathbf{u})$ is a source/sink. We denote by $\mathcal{M}$ the mesh

that results from the division of the domain Ω into n_e control volumes $K \subset \Omega$. Each K is a polygonal open set such that $K \cap L = \emptyset, \forall K \neq L$ and $\overline{\Omega} = \cup_{K \in \mathcal{M}} \overline{K}$.

For the derivation of the finite-volume formulation we integrate the first equation of (3.25) over a control volume K and apply the Gauss divergence theorem:

$$\int_{\partial K} (-\mathbf{\Lambda} \nabla u) \cdot \mathbf{n} \, d\Gamma = \int_K q \, dx. \tag{3.26}$$

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

$$\sum_{\sigma \subset \partial K} F_{K,\sigma} = Q_K, \quad \forall K \in \mathcal{M}, \tag{3.27}$$

where $F_{K,\sigma}$ is the discrete flux through face σ flowing out of cell K and $Q_K := \int_K q \, dx$ is the integrated source/sink term. Equation (3.27) is the typical cell-centered finite-volume formulation. Finite-volume schemes differ in the way how the term $(\mathbf{\Lambda}_K \nabla u) \cdot \mathbf{n}$ is approximated (i.e. the choice of the fluxes $F_{K,\sigma}$). Using the symmetry of the tensor $\mathbf{\Lambda}_K$, this term can be rewritten as $\nabla u \cdot \mathbf{\Lambda}_K \mathbf{n}$, which corresponds to the directional derivative of u in co-normal direction $\mathbf{\Lambda}_K \mathbf{n}$. 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 (3.25).

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

$$\mathbf{\Lambda}_K \mathbf{n}_{K,\sigma} = t_{K,\sigma} \mathbf{d}_{K,\sigma} + \mathbf{d}_{K,\sigma}^\perp, \quad t_{K,\sigma} = \frac{\mathbf{n}_{K,\sigma}^T \mathbf{\Lambda}_K \mathbf{d}_{K,\sigma}}{\mathbf{d}_{K,\sigma}^T \mathbf{d}_{K,\sigma}}, \ \mathbf{d}_{K,\sigma}^\perp = \mathbf{\Lambda}_K \mathbf{n}_{K,\sigma} - t_{K,\sigma} \mathbf{d}_{K,\sigma}, \tag{3.28}$$

with the tensor $\mathbf{\Lambda}_K$ associated with control volume K, the distance vector $\mathbf{d}_{K,\sigma} := \mathbf{x}_\sigma - \mathbf{x}_K$ and $\mathbf{d}_{K,\sigma}^T \mathbf{d}_{K,\sigma}^\perp = 0$, see Figure 3.4 for the used notations. The same can be done for the conormal $\mathbf{\Lambda}_L \mathbf{n}_{L,\sigma}$. The $t_{K,\sigma}$ and $t_{L,\sigma}$ are the transmissibilities associated with the face σ. These transmissibilities are calculated in DuMu$^\text{x}$ by using the function `computeTpfaTransmissibility`.

With these notations, it follows that for each cell K and face σ

$$\nabla u \cdot \mathbf{\Lambda}_K \mathbf{n}_{K,\sigma} = t_{K,\sigma} \nabla u \cdot \mathbf{d}_{K,\sigma} + \nabla u \cdot \mathbf{d}_{K,\sigma}^\perp. \tag{3.29}$$

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

$$F_{K,\sigma} = -|\sigma| t_{K,\sigma} (u_\sigma - u_K), \qquad F_{L,\sigma} = -|\sigma| t_{L,\sigma} (u_\sigma - u_L). \tag{3.30}$$
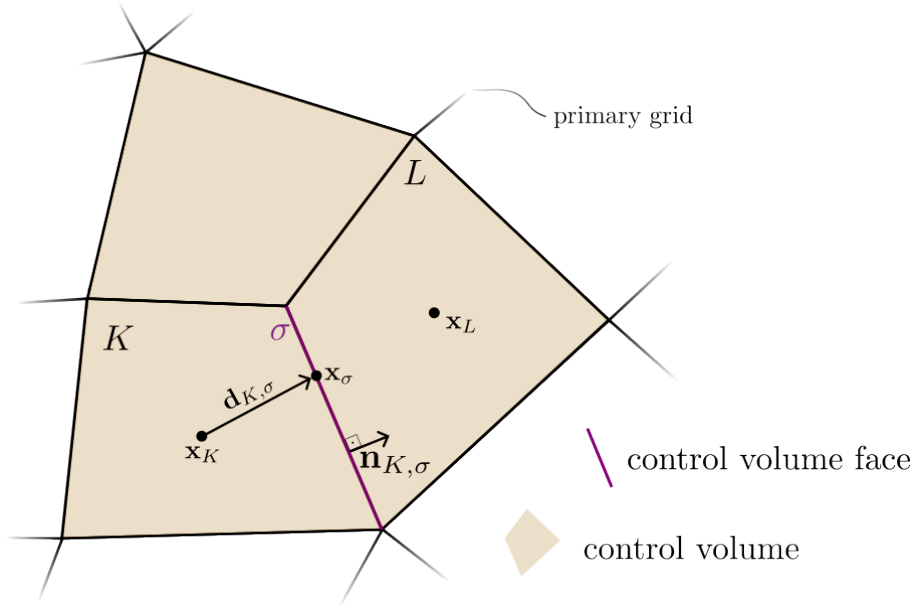
Figure 3.4: Two neighboring control volumes sharing the face σ.

Enforcing local flux conservation, i.e. $F_{K,\sigma} + F_{L,\sigma} = 0$, results in

$$u_\sigma = \frac{t_{K,\sigma} u_K + t_{L,\sigma} u_L}{t_{K,\sigma} + t_{L,\sigma}}. \tag{3.31}$$

With this, the fluxes (3.30) are rewritten as

$$F_{K,\sigma} = |\sigma| \frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}} (u_K - u_L), \quad F_{L,\sigma} = |\sigma| \frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}} (u_L - u_K). \tag{3.32}$$

By neglecting the orthogonal term, the consistency of the scheme is lost for general grids, where $\nabla u \cdot \mathbf{d}_{K,\sigma}^\perp \neq 0$. The consistency is achieved only for so-called K-orthogonal grids for which $\mathbf{d}_{K,\sigma}^\perp = 0$. For such grids we deduce that

$$\frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}} = \frac{\tau_{K,\sigma} \tau_{L,\sigma}}{\tau_{K,\sigma} d_{L,\sigma} + \tau_{L,\sigma} d_{K,\sigma}}, \tag{3.33}$$

with $\tau_{K,\sigma} := \mathbf{n}_{K,\sigma} \mathbf{\Lambda}_K \mathbf{n}_{K,\sigma}, \tau_{L,\sigma} := \mathbf{n}_{L,\sigma} \mathbf{\Lambda}_L \mathbf{n}_{L,\sigma}, d_{K,\sigma} := \mathbf{n}_{K,\sigma} \cdot \mathbf{d}_{K,\sigma}$, 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 u_σ in addition to the cell unknowns u_K and enforcing the physically motivated continuity of fluxes and continuity of the solution across the faces. For a face σ between the two polygons K and L these conditions read:

$$\begin{aligned} F_{K,\sigma} + F_{L,\sigma} &= 0 \\ u_{K,\sigma} = u_{L,\sigma} &= u_\sigma. \end{aligned} \tag{3.34}$$

Using these conditions, the intermediate face unknowns u_σ can be eliminated and the fluxes are expressed as a function of the cell unknowns u_N and associated transmissibilities $t^N_{K,\sigma}$:

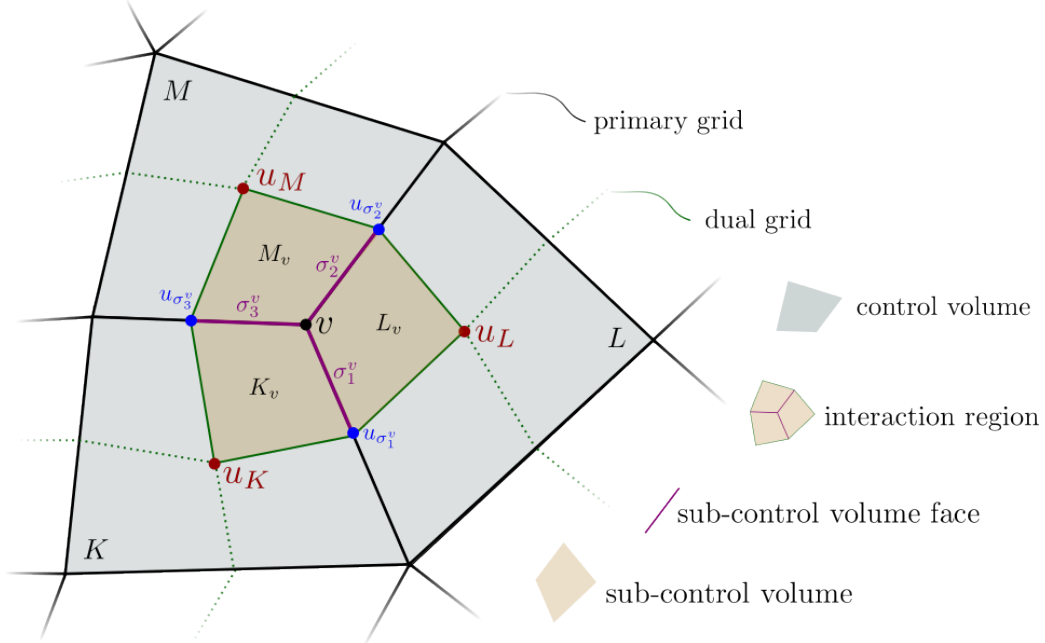$$F_{K,\sigma} = \sum_{N \in \mathcal{S}_{K,\sigma}} t^N_{K,\sigma} u_N. \tag{3.35}$$

Figure 3.5: Interaction region for the Mpfa-O method. The graphic on the right illustrates how the sub-control volume L_v and face σ^v_2 are embedded in cell L. Note that the face stencils for all sub-control volume faces in the depicted interaction region are $\mathcal{S}_{\sigma^v_i} = \{K, L, M\}$, meaning that the fluxes over the sub-control volume faces depend on the three cell unknowns u_K, u_L, u_M.

The main difference between the various finite-volume schemes available is the assembly of the face fluxes, i.e. the computation of the $t^N_{K,\sigma}$ and the size of $\mathcal{S}_{K,\sigma}$. For the Tpfa, that has been presented in the last section, the stencil and transmissibilities are given as

$$\mathcal{S}_{K,\sigma} = \{K, L\}, \quad t^K_{K,\sigma} = |\sigma| \frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}}, \ t^L_{K,\sigma} = -|\sigma| \frac{t_{K,\sigma} t_{L,\sigma}}{t_{K,\sigma} + t_{L,\sigma}},$$

with $t_{K,\sigma}, t_{L,\sigma}$ as defined in equation (3.28).

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 $\nabla u \cdot \mathbf{d}^\perp_{K,\sigma}$ 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_v. Analogously, each face is sub-divided into sub-control volume faces σ^v, see Figure 3.5. Note that for the Tpfa method sub-control volumes and control volumes coincide. Also, we allow for piecewise constant $\mathbf{\Lambda}$ (denoted as $\mathbf{\Lambda}_K$ for each cell K) and construct discrete gradients $\nabla^{K_v}_\mathcal{D} u$ (per

sub-control volume K_v). In the following, we restrict our discussion to the two-dimensional setup that is shown in Figure 3.5. Here, the discrete gradients are constructed to be consistent such that the following conditions hold:

$$\nabla_{\mathcal{D}}^{K_v} u \cdot (\mathbf{x}_{\sigma_1^v} - \mathbf{x}_K) = u_{\sigma_1^v} - u_K, \quad \nabla_{\mathcal{D}}^{K_v} u \cdot (\mathbf{x}_{\sigma_3^v} - \mathbf{x}_K) = u_{\sigma_3^v} - u_K. \tag{3.36}$$

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

$$\nabla_{\mathcal{D}}^{K_v} u = \mathbb{D}_{K_v}^{-T} \begin{bmatrix} u_{\sigma_1^v} - u_K \\ u_{\sigma_3^v} - u_K \end{bmatrix}, \qquad \text{with} \ \ \mathbb{D}_{K_v} := \begin{bmatrix} \mathbf{x}_{\sigma_1^v} - \mathbf{x}_K & \mathbf{x}_{\sigma_3^v} - \mathbf{x}_K \end{bmatrix}. \tag{3.37}$$

This enables us to write the discrete flux across σ_1^v from cell K as follows:

$$F_{K,\sigma_1^v} := -|\sigma_1^v| \mathbf{n}_{\sigma_1^v}^T \mathbf{\Lambda}_K \nabla_{\mathcal{D}}^{K_v} u. \tag{3.38}$$

Inserting the discrete gradient, yields

$$F_{K,\sigma_1^v} = \omega_{K,\sigma_1^v \sigma_1^v}(u_K - u_{\sigma_1^v}) + \omega_{K,\sigma_1^v \sigma_3^v}(u_K - u_{\sigma_3^v}), \tag{3.39}$$

with $(\omega_{K,\sigma_1^v \sigma_1^v}, \omega_{K,\sigma_1^v \sigma_3^v})^T = |\sigma_1^v| \mathbb{D}_{K_v}^{-1} \mathbf{\Lambda}_K \mathbf{n}_{\sigma_1^v}$. These values are calculated in DuMu$^{\mathrm{x}}$ 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.

$$F_{K,\sigma_1^v} + F_{L,\sigma_1^v} = 0, \tag{3.40}$$

$$F_{K,\sigma_3^v} + F_{M,\sigma_3^v} = 0, \tag{3.41}$$

$$F_{L,\sigma_2^v} + F_{M,\sigma_2^v} = 0. \tag{3.42}$$

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

$$\mathbb{A}^{3\times 3} \mathbf{u}_\sigma = \mathbb{B}^{3\times 3} \mathbf{u}, \tag{3.43}$$

where $\mathbf{u}$ contains the three cell unknowns u_K, u_L, u_M and $\mathbf{u}_\sigma$ the three face unknowns $u_{\sigma_1^v}, u_{\sigma_2^v}, u_{\sigma_3^v}$. Inserting these face unknowns into the flux expression (3.39) yields

$$F_{K,\sigma_i^v} = \sum_{N \in \{K,L,M\}} t_{K,\sigma_i^v}^N u_N = \mathbf{t}_{K,\sigma_i^v} \cdot \mathbf{u}, \tag{3.44}$$

for each cell K and sub-control volume face σ_i^v.

3.3.2 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 Ω is discretized using a FE mesh consisting of nodes and elements, see the primary grid in Figure 3.6. Then, a secondary FV mesh is constructed by connecting the face barycenters and element barycenters, thus creating a control volume B_v, also called box, with node v
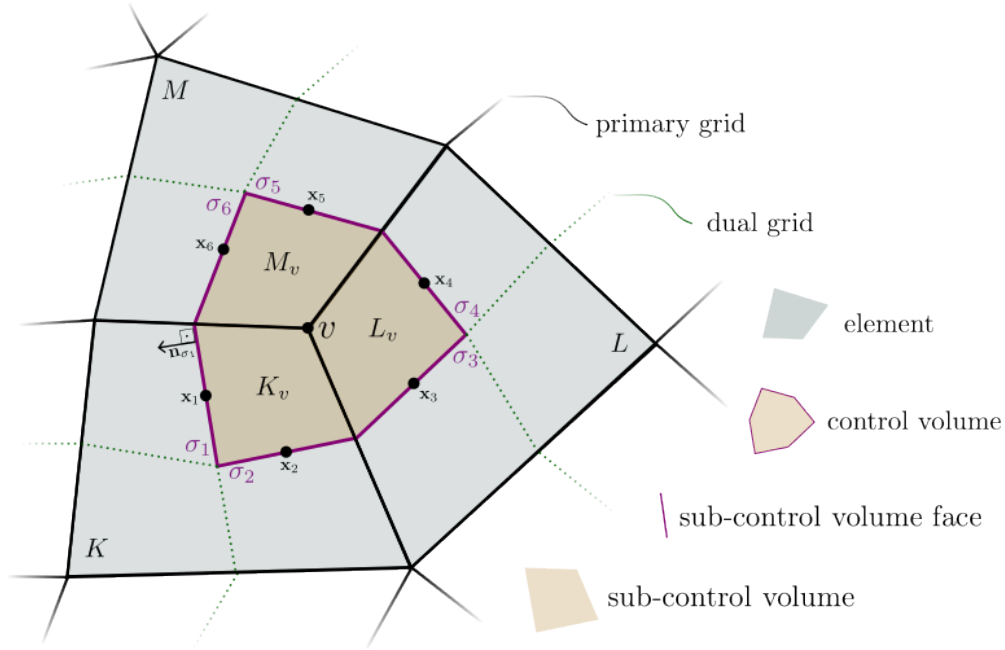
Figure 3.6: Control volume for the box method.

in the center (see Figure 3.6). When referring to faces, we mean entities of codimension 1 with respect to the element dimension.

Control volumes B_v are partitioned into sub-control volumes (scvs) such that each sub-control volume is the intersection of the control volume with a different primal grid element, $M_v = B_v \cap M$, see Figure 3.6. The faces of B_v are partitioned into sub-control volume faces σ_k analogously and $|\sigma_k|$ denotes the measure of sub-control volume face k. Finally, the integration points x_k which lie on the scvfs and the outer normal vectors $\mathbf{n}_{\sigma_k}$ also need to be defined, see Figure 3.6 for reference.

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 control volume and to get the fluxes across the sub-control volume faces at the integration points x_k from the FE approach. Consequently, at each scvf the following expression results:

$$ f(\tilde{u}(x_k)) \cdot \mathbf{n}_{\sigma_k} \, |\sigma_k| \qquad \text{with} \qquad \tilde{u}(x_k) = \sum_i N_i(x_k) \cdot \hat{u}_i, \tag{3.45} $$

where N_i represents the basis function of the finite element ansatz at node i. The basis functions are defined such that $N_i(x_j) = \delta_{ij}$ with δ_{ij} being the Kronecker delta. In the following, the discretization of the balance equation is going to be derived. From the REYNOLDS transport theorem follows the general balance equation:

$$ \underbrace{\int_\Omega \frac{\partial}{\partial t} u \, \mathrm{d}x}_{1} + \underbrace{\int_{\partial \Omega} (\mathbf{v}u + \mathbf{w}) \cdot \mathbf{n} \, \mathrm{d}\Gamma}_{2} = \underbrace{\int_\Omega q \, \mathrm{d}x}_{3} \tag{3.46} $$

29

$$f(u) = \int_\Omega \frac{\partial u}{\partial t}\, \mathrm{d}x + \int_\Omega \nabla \cdot \underbrace{[\mathbf{v}u + \mathbf{w}(u)]}_{F(u)}\, \mathrm{d}x - \int_\Omega q\, \mathrm{d}x = 0 \tag{3.47}$$

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. Ω denotes the model domain and $F(u) = F(\mathbf{v}, p) = F(\mathbf{v}(x,t), p(x,t))$.

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 $\hat{u}_i$ and linear basis functions N_i yielding an approximate function $f(\tilde{u})$. For $u \in \{\mathbf{v}, p, x^\kappa\}$, this means:

$$\tilde{p} = \sum_i N_i \hat{p}_i \tag{3.48} \qquad\qquad \nabla \tilde{p} = \sum_i \nabla N_i \hat{p}_i \tag{3.51}$$

$$\tilde{\mathbf{v}} = \sum_i N_i \hat{\mathbf{v}}_i \tag{3.49} \qquad\qquad \nabla \tilde{\mathbf{v}} = \sum_i \nabla N_i \hat{\mathbf{v}}_i \tag{3.52}$$

$$\tilde{x}^\kappa = \sum_i N_i \hat{x}_i^\kappa \tag{3.50} \qquad\qquad \nabla \tilde{x}^\kappa = \sum_i \nabla N_i \hat{x}_i^\kappa. \tag{3.53}$$

Due to the approximation with node values and basis functions, the differential equations are not exactly fulfilled anymore but a residual ε is produced.

$$f(u) = 0 \qquad \Rightarrow \qquad f(\tilde{u}) = \varepsilon \tag{3.54}$$

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

$$\int_\Omega \varepsilon W_j\, \mathrm{d}x \stackrel{!}{=} 0 \qquad \text{with} \qquad \sum_j W_j = 1 \tag{3.55}$$

yields the following equation:

$$\int_\Omega \frac{\partial \tilde{u}}{\partial t} W_j\, \mathrm{d}x + \int_\Omega [\nabla \cdot F(\tilde{u})]\, W_j\, \mathrm{d}x - \int_\Omega q W_j\, \mathrm{d}x = \int_\Omega \varepsilon W_j\, \mathrm{d}x \stackrel{!}{=} 0. \tag{3.56}$$

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.

$$W_j(x) = \begin{cases} 1 & x \in B_j \\ 0 & x \notin B_j. \end{cases} \tag{3.57}$$

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

Inserting definition (3.57) into equation (3.56) and using the GREEN-GAUSSIAN integral theorem results in

$$\int_{B_j} \frac{\partial \tilde{u}}{\partial t}\, \mathrm{d}x + \int_{\partial B_j} F(\tilde{u}) \cdot \mathbf{n}\, \mathrm{d}\Gamma_{B_j} - \int_{B_j} q\, \mathrm{d}x \stackrel{!}{=} 0, \tag{3.58}$$

which has to hold for every control volume B_j.

The first term in equation (3.58) can be written as

$$\int_{B_j} \frac{\partial \tilde{u}}{\partial t}\,\mathrm{d}x = \frac{d}{dt} \int_{B_j} \sum_i \hat{u}_i N_i \,\mathrm{d}x = \sum_i \frac{\partial \hat{u}_i}{\partial t} \int_{B_j} N_i \,\mathrm{d}x. \tag{3.59}$$

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

$$M_{i,j}^{lump} = \begin{cases} |B_j| & j = i \\ 0 & j \neq i, \end{cases} \tag{3.60}$$

where $|B_j|$ is the volume of the FV control volume B_j associated with node j. The application of this assumption yields

$$|B_j|\frac{\partial \hat{u}_j}{\partial t} + \int_{\partial B_j} F(\tilde{u}) \cdot \mathbf{n}\,\mathrm{d}\Gamma_{B_j} - Q_j = 0, \tag{3.61}$$

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

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

$$|B_j|\frac{\hat{u}_j^{n+1} - \hat{u}_j^n}{\Delta t} + \int_{\partial B_j} F(\tilde{u}^{n+1}) \cdot \mathbf{n}\,\mathrm{d}\Gamma_{B_j} - Q_j^{n+1} = 0. \tag{3.62}$$

Equation (3.62) has to be fulfilled for each box B_j.

3.3.3 Staggered Grid – A Short Introduction

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 3.7). As for the cell-centered method, the fluxes are evaluated at the edges of each control volume with a two-point flux approximation, cf. 3.3.1.

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$^{\text{x}}$ use the staggered-grid discretization.

3.4 Steps of a DuMu$^{\text{x}}$ Simulation

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

In Section 3.4.1 the structure of DuMu$^{\text{x}}$ is shown from a *content* point of view.

■ Cell-centered primary variables $(p_\alpha, x_\alpha^\kappa, T, k, \epsilon...)$

→ $v_{\alpha,x}$

↑ $v_{\alpha,y}$

□ 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
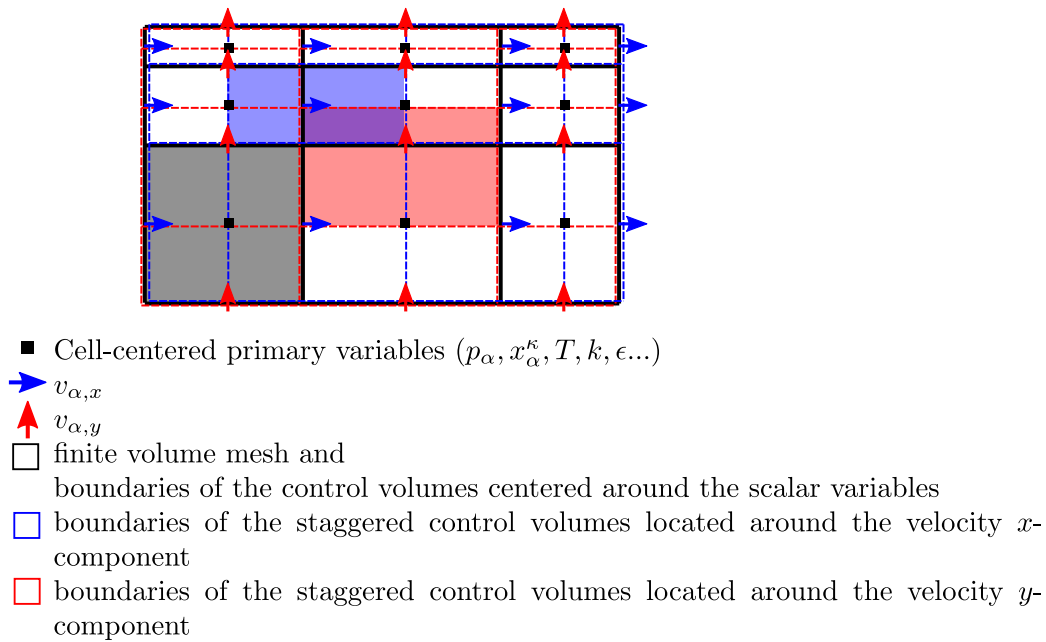
Figure 3.7: Discretization of the staggered-grid method. The figure shows the different control volume arrangements, which are staggered with respect to each other. There are the control volumes centered around the scalar primary variables in black, the control volumes located around the x-component 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.

3.4.1 Structure – by Content

In Figure 3.8, the algorithmic representations of a monolithical solution scheme is illustrated down to the element level.

3.4.2 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/-/blob/releases/3.7/exercises/exercise-mainfile/exercise1pamain.cc`

3.5 Input and Output

This section briefly explains grid generation in DuMu^x, summarizes the grid formats that can be used by DuMu^x and introduces the DuMu^x `GridManager`. Finally, this section informs about handling
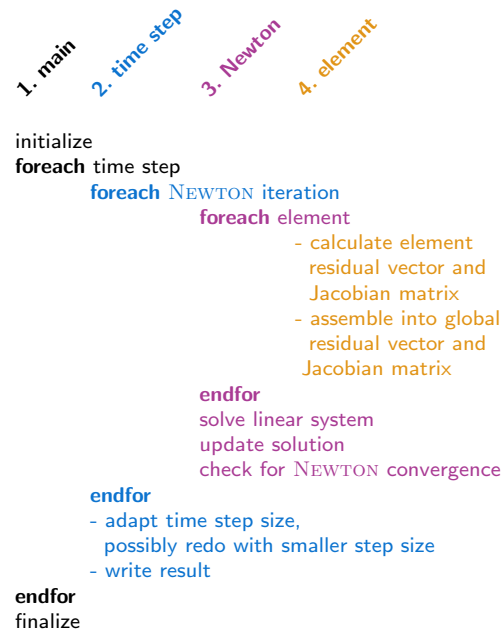
```
                    1. main      2. time step    3. Newton      4. element

initialize
foreach time step
          foreach NEWTON iteration
                    foreach element
                              - calculate element
                                residual vector and
                                Jacobian matrix
                              - assemble into global
                                residual vector and
                                Jacobian matrix
                    endfor
                    solve linear system
                    update solution
                    check for NEWTON convergence
          endfor
          - adapt time step size,
            possibly redo with smaller step size
          - write result
endfor
finalize
```

Figure 3.8: Structure of a monolithical solution scheme in DuMu*.

output in DuMu*.

3.5.1 Supported grid file formats

DuMu* can read grids from file using the Dune Grid Format (DGF), the Gmsh mesh format (MSH), the Eclipse grid format (GRDECL), or the Visualization ToolKit (VTK/VTU/VTP) format.

Dune Grid Format

Most of our DuMu* tests use the Dune Grid Format (DGF) to read in grids. A detailed description of the DGF format and some examples can be found in the DUNE doxygen documentation (**Modules →I/O → Dune Grid Format (DGF)**). To generate larger or more complex DGF files, we recommend to write your own scripts, e.g, in C++, Matlab or Python.

The DGF format can also be used to read in spatial parameters defined on the grid. These parameters can be defined on nodes as well as on the elements. An example for predefined parameters on a grid can be found in `dumux/test/porousmediumflow/co2/`.

Gmsh Mesh Format

Gmsh is an open-source flexible grid generator for unstructured finite-element meshes ([11], `http://geuz.org/gmsh/`). DuMu* 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 as, 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`.

DuMu^x 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/cornerpoint`.

VTK File Format

VTK format uses ASCII or XML format. It is mostly used by DuMu^x 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 DuMu^x. If conversion is not an option, another possibility would be to write your own `GridManager`s. Examples of other grid formats, which have previously been either converted or custom-created in DuMu^x, are ArtMesh grids (fractured network grids), and ICEM grids (CAD developed grids).

3.5.2 The DuMu^x `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 [2] of the input file:

```
1  [Grid]
2  File = mydgfgrid.dgf
```

---

[2]Note, that group name `Grid` is the default group name and can be customized in your problem changing the string property `GridParameterGroup`. This way, it is possible, e.g., for problems with more than one grid, to set different group names for each grid, thus configuring them separately.

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:

```
1  [Grid]
2  File = mygmshgrid.msh
```

DuMu^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:

```
1  [Grid]
2  File = mydgfgrid.dgf
3  Refinement = 4
```

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

Parameters specific to the grid implementation

The `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

```
1  [Grid]
2  File = mydgfgrid.dgf
3  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:

```
1  [Grid]
2  LowerLeft = 0 0 0
3  UpperRight = 1 1 1
4  Cells = 10 10 20
```

where `LowerLeft` is a vector to the lower left corner of the grid and `UpperRight` a vector to the upper right corner. `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

```
1  [Grid]
2  LowerLeft = 0 0 0
3  UpperRight = 1 1 1
4  Cells = 10 10 20
5  CellType = Cube # or Simplex
```

For all available parameters see the Doxygen documentation.

Other DuMu^x `GridManager`s

- `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.

3.5.3 Input and Output formats

VTK file format

Dumux 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`:

```
1  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.

```
1  //! get the analytical temperature
2  const std::vector<Scalar>& getExactTemperature()
3  {
4      return temperatureExact_;
5  }
```

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 3.5.1.

Gnuplot interface

DuMu^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 2.6.2.

Container I/O

DuMu^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

`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 `dune/istl/io.hh`).

Bibliography

[1] Ivar Aavatsmark. An introduction to multipoint flux approximations for quadrilateral grids. *Computational Geosciences*, 6:405–432, 2002.

[2] M. Acosta, C. Merten, G. Eigenberger, H. Class, R. Helmig, B. Thoben, and H. Müller-Steinhagen. Modeling non-isothermal two-phase multicomponent flow in the cathode of PEM fuel cells. *Journal of Power Sources*, page in print, 2006. URL `https://dx.doi.org/10.1016/j.jpowsour.2005.12.068`.

[3] Martin Alkämper, Andreas Dedner, Robert Klöfkorn, and Martin Nolte. The DUNE-ALUGrid Module. *Archive of Numerical Software*, 4(1):1–28, 2016. ISSN 2197-8263. doi: 10.11588/ans.2016.1.23252. URL `http://journals.ub.uni-heidelberg.de/index.php/ans/article/view/23252`.

[4] ALUGrid Homepage. URL `https://www.dune-project.org/modules/dune-alugrid/`.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A Generic Grid Interface For Parallel and Adaptive Scientific Computing. Part II: implementation and tests in DUNE. *Computing*, 82(2):121–138, 2008. doi: 10.1007/s00607-008-0004-9. URL `https://dx.doi.org/10.1007/s00607-008-0004-9`.

[6] A. Bielinski. *Numerical Simulation of CO_2 Sequestration in Geological Formations*. PhD thesis, Institut für Wasserbau, Universität Stuttgart, 2006. URL `http://dx.doi.org/10.18419/opus-252`.

[7] R Brooks and T Corey. Hydrau uc properties of porous media. *Hydrology Papers, Colorado State University*, 24:37, 1964.

[8] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger. An efficient implementation of an adaptive and parallel grid in DUNE. In *Computational Science and High Performance Computing II*, volume 91, pages 67–82. Springer, 2006.

[9] H. Class, R. Helmig, and P. Bastian. Numerical Simulation of Nonisothermal Multiphase Multicomponent Processes in Porous Media – 1. An Efficient Solution Technique. *Advances in Water Resources*, 25:533–550, 2002. doi: 10.1016/S0309-1708(02)00014-3. URL `http://dx.doi.org/10.1016/S0309-1708(02)00014-3`.

[10] DUNE Homepage. URL `http://www.dune-project.org`.

[11] C. Geuzaine and J. F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11): 1309–1331, 2009. doi: 10.1002/nme.2579. URL `http://dx.doi.org/10.1002/nme.2579`.

[12] Rainer Helmig et al. *Multiphase flow and transport processes in the subsurface: a contribution to the modeling of hydrosystems.* Springer-Verlag, 1997.

[13] Johannes Hommel. *Modeling biogeochemical and mass transport processes in the subsurface: Investigation of microbially induced calcite precipitation.* Promotionsschrift, Universität Stuttgart, TASK, 2 2016. URL `https://elib.uni-stuttgart.de/handle/11682/8787`.

[14] UG Homepage. URL `https://www.dune-project.org/modules/dune-uggrid/`.