DuMu^X

Handbook

Version 2.11

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, 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 **D**istributed and **U**nified **N**umerics **E**nvironment [8]. DUNE provides a generic interface to many existing grid management libraries such as UG [12], ALUGrid [3, 2] and a few more. DUNE also extensively uses template programming in order to achieve minimal overhead when accessing the underlying grid libraries[1].

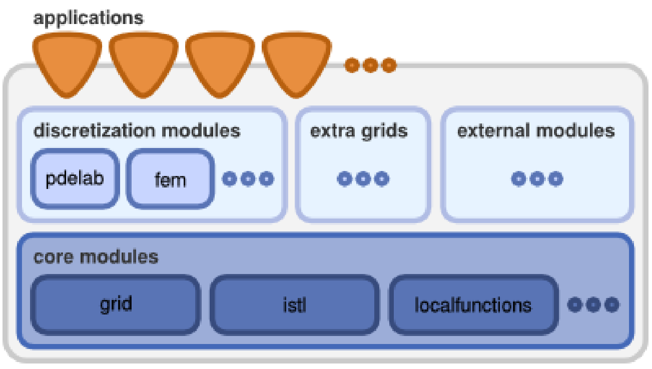


Figure 1.1: A high-level overview of DUNE's design is available on the project's web site [8].

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 1 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 [6].

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. For a more thorough description of DUNE's grid definition, see [4].

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 form of an additional module `dumux`. It depends on the DUNE core modules `dune-common`, `dune-grid`, `dune-istl`, and on `dune-localfunctions`. The main intention of DuMu^x 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^x includes ready to use numerical models and a few example applications.

2 Getting started

First, we briefly describe the installation procedure. Then we provide a quick start guide for the first DuMu^x experience.

2.1 Quick Installation of DuMu^x

This only provides one quick way of installing DuMu^x. You should have a recent working Linux environment, no DUNE core modules should be installed. If you need more information, have DUNE already installed, please have a look at the detailed installation instructions in Section 2.3.

2.1.1 Obtaining the Code with the Script `checkout-dumux`

The shell-script `checkout-dumux` facilitates setting up a DUNE/DuMu^x directory tree. It is available after obtaining a download link via `http://www.dumux.org/download/`. For example the second line below will check out the required DUNE modules and `dumux`, `dumux-devel` and the `external` folder, which contains some useful external software and libraries. Again, `joeuser` needs to be replaced by the actual user name.

```
$ checkout-dumux -h       # show help,
$ checkout-dumux -gme -u joeuser -p password -d DUMUX
```

Be aware that you cannot get `dumux-devel` or the external libraries from `dumux-external` unless you have an GitLab account with the right privileges.

If you want to install DUNE and DuMu^x without the help of `checkout-dumux` script a complete installation guide can be found in chapter 2.3.

2.1.2 Build of DUNE and DuMu^x

Building of DUNE and DuMu^x is done by the command-line script `dunecontrol` as described in DUNE Installation Notes[1]. More details about the build-system can be found in the DUNE buildsystem documentation[2]. If something fails during the execution of `dunecontrol` feel free to report it to the DUNE or DuMu^x developer mailing list, but also try to include error details.

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

[1]`https://www.dune-project.org/doc/installation/`
[2]`https://www.dune-project.org/buildsystem/`

```
$ # make sure you are in the directory DUNE-Root
$ ./dune-common/bin/dunecontrol --configure-opts="CXXFLAGS=-fno-strict-
   aliasing" --use-cmake all
```

Too many options can make life hard. That's why usually option files are being used together with `dunecontrol` and its sub-tools. Larger sets of options are kept in them. If you are going to compile with options suited for debugging the code, the following can be a starting point:

```
$ # make sure you are in the directory DUNE-Root
$ cp dumux/debug.opts my-debug.opts      # create a personal version
$ gedit my-debug.opts                    # optional editing the options file
$ ./dune-common/bin/dunecontrol --opts=my-debug.opts --use-cmake all
```

More optimized code, which is typically not usable for standard debugging tasks, can be produced by

```
$ cp dumux/optim.opts my-optim.opts
$ ./dune-common/bin/dunecontrol --opts=my-optim.opts --use-cmake all
```

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 files above are 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. One avoids confusing it with the option files which came out of the distribution.

2.2 Quick Start Guide: The First Run of a Test Application

The previous section showed how to install and compile DuMu^x. This chapter shall give a very brief introduction how to run a first test application and how to visualize the first output files. A more detailed explanations can be found in the tutorials in the following chapter.
All executables are compiled in the `build` subdirectories of DuMu^x. If not given differently in the input files, this is `build-cmake` as default.

1. Go to the directory `build-cmake/test`. There, various test application folders can be found. Let us consider as example `porousmediumflow/2p/implicit/test_box2p`:

2. Enter the folder `porousmediumflow/2p/implicit`. Type `make test_box2p` in order to compile the application `test_box2p`. To run the simulation, type
   `./test_box2p -parameterFile ./test_box2p.input`
   into the console. The parameter `-parameterFile` specifies that all important parameters (like first timestep size, end of simulation and location of the grid file) can be found in a text file in the same directory with the name `test_box2p.input`.

3. 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>.

4. 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 left hand side, you can choose the desired parameter to be displayed.

2.3 Detailed Installation Instructions

In this section about the installation of DuMu^x it is assumed that you work with a Linux or Apple OS X operating system and that you are familiar with the use of a command line shell. Installation means that you unpack DUNE together with DuMu^x in a certain directory. Then, you compile it in that directory tree in which you do the further work, too. You also should know how to install new software packages or you should have a person on hand who can give you assistance with that. In section 2.3.1 we list some prerequisites for running DUNE and DuMu^x. Please check in said paragraph whether you can fulfill them. In addition, section 2.3.4 provides some details on optional libraries and modules.

In a technical sense DuMu^x is a module of DUNE. Thus, the installation procedure of DuMu^x is the same as that of DUNE. Details regarding the installation of DUNE are provided on the DUNE website [8]. If you are interested in more details about the build system that is used, they can be found in the DUNE buildsystem documentation[3] and CMake's documentation[4].

All DUNE modules, including DuMu^x, get extracted into a common directory, as it is done in an ordinary DUNE installation. We refer to that directory abstractly as DUNE root directory or, in short, as DUNE-Root. If it is used as directory's path of a shell command it is typed as `DUNE-Root`. For the real DUNE root directory on your file system any valid directory name can be chosen.

Source code files for each DUNE module are contained in their own subdirectory within DUNE-Root. We name this directory of a certain module *module root directory* or `module-root-directory` if it is a directory path, e. g. for the module `dumux` these names are *dumux root directory* respective `dumux-root-directory`. The real directory names for the modules can be chosen arbitrarily. In this manual they are the same as the module name or the module name extended by a version number suffix. The name of each DUNE module is defined in the file `dune.module`, which is in the root directory of the respective module. This should not be changed by the user.

After extracting the source code for all relevant DUNE modules, including DuMu^x, DUNE has to be built by the shell-command `dunecontrol` which is part of the DUNE build system.

2.3.1 Prerequisites

A reasonable recent C++ compiler (g++ (4.9), clang++ (3.5), or Intels ICC), CMake (version 2.8.12 or newer) and their dependencies are required. For prerequisite software packages to install see [8].

The building of included documentation like this handbook requires LaTeX and auxiliary tools `bibtex`. One usually chooses a LaTeX distribution like `texlive` for this purpose. It is possible to switch off the building of the documentation by setting the switch `--disable-documentation` in the

[3]`https://www.dune-project.org/buildsystem/`
[4]`https://cmake.org/documentation/`

CONFIGURE_FLAGS of the building options, see Chapter 2.1.2. Additional parts of documentation are contained within the source code files as special formatted comments. Extracting them can be done using doxygen, cf. Section 2.3.3.

Depending on whether you are going to use external libraries and modules for additional DUNE features, additional software packages may be required. Some hints on that are given in Section 2.3.4.

Git clients must be installed to download modules from Git repositories.

2.3.2 Obtaining Source Code for DUNE and DuMu^x

As stated above, the DuMu^x release and trunk (developer tree) are based on the most recent DUNE release 2.4, comprising the core modules dune-common, dune-geometry, dune-grid, dune-istl and dune-localfunctions. For working with DuMu^x, these modules are required. The external module dune-PDELab is recommended and required for several DuMu^x features.

Two possibilities exist to get the source code of DUNE and DuMu^x. Firstly, DUNE and DuMu^x can be downloaded as tar files from the respective DUNE and DuMu^x website. They have to be extracted as described in the next paragraph. Secondly, a method to obtain the most recent source code (or, more generally, any of its previous revisions) by direct access to the software repositories of the revision control system is described in the subsequent part.

However, if a user does not want to use the most recent version, certain version tags or branches (i. e. special names) are means of the software revision control system to provide access to different versions of the software.

Obtaining the software by installing tar files The slightly old-fashionedly named 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 2.4.1) and DuMu^x websites to a certain folder in your file system. Create the DUNE root directory, named `dune` 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 root directory* is `dumux-2.11` (or whatever version you downloaded).

```
$ mkdir dune
$ cd dune
$ tar xzvf path_to_tarball_of/dune-common-2.4.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-geometry-2.4.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-grid-2.4.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-istl-2.4.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-localfunctions-2.4.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-pdelab-2.0.0.tar.gz
$ tar xzvf path_to_tarball_of/dune-typetree-2.4.1.tar.gz
$ tar xzvf path_to_tarball_of/dumux-2.9.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 DUNE root directory, named e.g. `DUNE-ROOT` 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. The DUNE modules of the stable 2.4 release branch are checked out as described on the DUNE website [8]:

```
$ mkdir DUNE-ROOT
$ cd DUNE-ROOT
$ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-common.
   git
$ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-
   geometry.git
$ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-grid.
   git
$ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-istl.
   git
$ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-
   localfunctions.git
$ git clone -b releases/2.3 https://gitlab.dune-project.org/PDELab/dune-
   typetree.git
$ git clone -b releases/2.0 https://gitlab.dune-project.org/PDELab/dune-
   pdelab.git
```

The newest and maybe unstable developments are also provided in these repositories and can be found in the *master* branch. Please check the DUNE website [8] for further information. We always try to keep up with the latest developments of DUNE. However, the current DuMu^x release is based on the stable 2.4 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.

The `dumux` module is checked out as described below (see also the DuMu^x website: `http://www.dumux.org/`). Its file tree has to be created in the DUNE-Root directory, where the DUNE modules have also been checked out to. Subsequently, the next command is executed there, too. The dumux root directory is called `dumux` here.

```
$ # make sure you are in DUNE-Root
$ git clone https://git.iws.uni-stuttgart.de/dumux-repositories/dumux.git
```

Patching DUNE or external libraries Patching of DUNE modules in order to work together with DuMu^x can be necessary for several reasons. Software like a compiler or even a standard library changes at times. But, for example, a certain release of a software component that we depend on, may not reflect that change and thus it has to be modified. In the dynamic developing process of software which depends on other modules it is not always feasible to adapt everything to the most recent version of each module. They may fix problems with a certain module of a certain release without introducing too much structural change.

DuMu^x contains patches and documentation about their usage and application within the directory `dumux/patches`. Please check the README file in that directory for recent information. In general, a patch can be applied as follows (the exact command or the used parameters may be slightly different). We include here an example of a patching dune-grid.

```
$ # make sure you are in DUNE-Root
$ cd dune-grid
$ patch -p0 < ../dumux/patches/grid-2.3.1.patch
```

It can be removed by

```
$ path -p0 -R < ../dumux/patches/grid-2.3.1.patch
```

The `checkout-dumux` script also applies patches, if not explicitly requested not to do so.

Hints for DuMu^x-Developers If you also want to actively participate in the development of DuMu^x, you can allways send patches to the Mailing list.

To get more involved, you can apply either for full developer access or for developer access on certain parts of DuMu^x. Granted developer access means that you are allowed to commit own code and that you can access the `dumux-devel` module. This enhances `dumux` by providing maybe unstable code from the developer group.

2.3.3 Building Documentation

Doxygen

Doxygen documentation is done by especially 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 execute `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.

Handbook

To build the DuMu^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`.

2.3.4 External Libraries and Modules

The libraries described below provide additional functionality but are not generally required to run DuMu^x. If you are going to use an external library check the information provided on the DUNE website[5]. If you are going to use an external DUNE module the website on external modules[6] can be helpful.

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-debug.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 DUNE root directory.

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`

- **dune-foamgrid**: External grid module. One- and two-dimensional grids in a physical space of arbitrary dimension; non-manifold grids, growth, element paramterizations, 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`

- **DUNE-multidomaingrid** and **DUNE-multidomain**: External modules which offer a meta grid that has different sub-domains. Each sub-domain can have a local operator that is coupled by a coupling condition. They are used for multi-physics approaches or domain decomposition methods. Download: `https://github.com/smuething/dune-multidomaingrid` and `https://github.com/smuething/dune-multidomain`

- **DUNE-PDELab**: External module to write more easily discretizations. PDELab provides a sound number of discretizations like FEM or discontinuous Galerkin methods. Download: `https://gitlab.dune-project.org/pdelab/dune-pdelab`

- **PARDISO**: External library for solving linear equations. The package PARDISO is a thread-safe, high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and asymmetric linear systems of equations on shared memory multiprocessors. The

[5]DUNE: External libraries, `https://www.dune-project.org/doc/external-libraries/`
[6]DUNE: External modules, `https://www.dune-project.org/groups/external/`

precompiled binary can be downloaded after personal registration from the PARDISO website: `http://www.pardiso-project.org`

- **SuperLU**: External library for solving linear equations. SuperLU is a general purpose library for the direct solution of large, sparse, non-symmetric systems of linear equations. Download: `http://crd.lbl.gov/~xiaoye/SuperLU`
- **UMFPack**: External library for solving linear equeations. It is part of SuiteSparse.
- **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`

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.

3 Tutorial

In DuMu^x two sorts of models are implemented: implicit models and sequential models. In the implicit models a flow system is described by a system of strongly coupled equations, which can be for example mass balance equations for phases, mass balance equations for components or energy balance equations. In contrast, a sequential model consists of a pressure equation, which is iteratively coupled to a saturation equation, concentration equations, energy balance equations, etc.

Examples for different kinds of both, implicit and sequential models, are isothermal two-phase models, isothermal two-phase two-component models, non-isothermal two-phase models, and non-isothermal two-phase two-component models.

The following two sections demonstrate solving problems using an implicit model 3.1 and a sequential model 3.2. An isothermal two-phase system (two fluid phases, one solid phase) will be considered.

3.1 Solving a Problem Using a Fully-Implicit Model

The process of setting up a problem using DuMu^x can be roughly divided into four parts:

1. A suitable model has to be chosen.
2. The geometry of the problem and correspondingly a grid have to be defined.
3. Material properties and constitutive relationships have to be selected.
4. Boundary conditions and initial conditions have to be specified.

The problem being solved in this tutorial is illustrated in Figure 3.1. A rectangular domain with no-flow boundaries on the top and on the bottom, which is initially saturated with oil, is considered. Water infiltrates from the left side into the domain and replaces the oil. Gravity effects are neglected here.

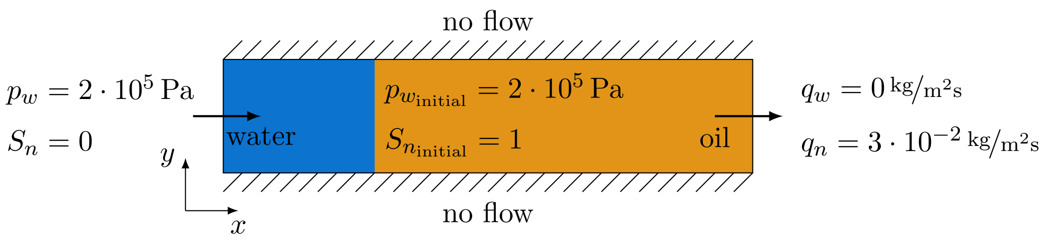

no flow
$p_w = 2 \cdot 10^5$ Pa
$S_n = 0$
water
$p_{w_{\text{initial}}} = 2 \cdot 10^5$ Pa
$S_{n_{\text{initial}}} = 1$
oil
$q_w = 0\,\mathrm{kg/m^2 s}$
$q_n = 3 \cdot 10^{-2}\,\mathrm{kg/m^2 s}$
y
x
no flow

Figure 3.1: Geometry of the tutorial problem with initial and boundary conditions.

The solved equations are the mass balances of water and oil:

$$\frac{\partial(\phi \, S_w \, \varrho_w)}{\partial t} - \nabla \cdot \left(\varrho_w \, \frac{k_{rw}}{\mu_w} \, \mathbf{K} \, \nabla p_w \right) - q_w = 0 \tag{3.1}$$

$$\frac{\partial(\phi \, S_o \, \varrho_o)}{\partial t} - \nabla \cdot \left(\varrho_o \, \frac{k_{ro}}{\mu_o} \, \mathbf{K} \, \nabla p_o \right) - q_o = 0 \tag{3.2}$$

3.1.1 The Main File

Listing 1 shows the main application file `tutorial/tutorial_implicit.cc` for the implicit two-phase model. This file has to be compiled and executed in order to solve the problem described above.

Listing 1 (File tutorial/tutorial_implicit.cc)

```
24 #include <config.h>
25 #include "tutorialproblem_implicit.hh"
26 #include <dumux/common/start.hh>
27
28 //! Prints a usage/help message if something goes wrong or the user asks for help
29 void usage(const char *progName, const std::string &errorMsg)
30 {
31     std::cout
32         <<  "\nUsage: " << progName << " [options]\n";
33     if (errorMsg.size() > 0)
34         std::cout << errorMsg << "\n";
35     std::cout
36         << "\n"
37         << "The list of mandatory arguments for this program is:\n"
38         << "\t-TEnd                  The end of the simulation [s]\n"
39         << "\t-DtInitial             The initial timestep size [s]\n"
40         << "\t-Grid.UpperRight       The x-/y-coordinates of the grid's upper-right corner [m]\n"
41         << "\t-Grid.Cells            The grid's x-/y-resolution\n"
42         << "\n";
43 }
44
45 int main(int argc, char** argv)
46 {
47     typedef TTAG(TutorialProblemImplicit) TypeTag;
48     return Dumux::start<TypeTag>(argc, argv, usage);
49 }
```

From line 24 to line 26 the required headers are included.

At line 47 the type tag of the problem, which is going to be simulated, is specified. All other data types can be retrieved via the DuMu^x property system and only depend on this single type tag. For a more thorough introduction to the DuMu^x property system, see chapter 5.4.

After this, the default startup routine `Dumux::start()` is called on line 48. This function deals with parsing the command line arguments, reading the parameter file, setting up the infrastructure necessary for DUNE, loading the grid, and starting the simulation. Required parameters for the start of the simulation, such as the initial time-step size, the simulation time or details of the grid, can be either specified by command line arguments of the form (`-ParameterName ParameterValue`), in the file specified by the `-ParameterFile` argument, or if the latter is not specified, in the file `tutorial_implicit.input`. If a parameter is specified on the command line as well as in the parameter file, the values provided in the command line have precedence. Listing 2 shows the default parameter file for the tutorial problem.

Listing 2 (File tutorial/tutorial_implicit.input)

```
[TimeManager]
TEnd = 500000 # duration of the simulation [s]
DtInitial = 10 # initial time step size [s]

[Grid]
UpperRight = 300 60 # x-/y-coordinates of the upper-right corner of the grid [m]
Cells = 100 1 # x-/y-resolution of the grid
```

To provide an error message, the usage message which is displayed to the user if the simulation is called incorrectly, is printed via the custom function which is defined on line 29 in the main file. In this function the usage message is customized to the problem at hand. This means that at least the necessary parameters are listed here.

3.1.2 The Problem Class

When solving a problem using DuMu^x, the most important file is the so-called *problem file* as shown in listing 3.

Listing 3 (File tutorial/tutorialproblem_implicit.hh)

```
#ifndef DUMUX_TUTORIAL_PROBLEM_IMPLICIT_HH    // guardian macro
#define DUMUX_TUTORIAL_PROBLEM_IMPLICIT_HH    // guardian macro

// The numerical model
#include <dumux/porousmediumflow/2p/implicit/model.hh>

// The base porous media box problem
#include <dumux/porousmediumflow/implicit/problem.hh>

// Spatially dependent parameters
#include "tutorialspatialparams_implicit.hh"

// The components that are used
#include <dumux/material/components/h2o.hh>
#include <dumux/material/components/lnapl.hh>

namespace Dumux{
// Forward declaration of the problem class
template <class TypeTag>
class TutorialProblemImplicit;

namespace Properties {
// Create a new type tag for the problem
NEW_TYPE_TAG(TutorialProblemImplicit, INHERITS_FROM(BoxTwoP, TutorialSpatialParamsImplicit));

// Set the "Problem" property
SET_PROP(TutorialProblemImplicit, Problem)
{ typedef TutorialProblemImplicit<TypeTag> type;};

// Set grid and the grid creator to be used
#if HAVE_DUNE_ALUGRID
SET_TYPE_PROP(TutorialProblemImplicit, Grid, Dune::ALUGrid</*dim=*/2, 2, Dune::cube, Dune::nonconforming>);
#elif HAVE_UG
SET_TYPE_PROP(TutorialProblemImplicit, Grid, Dune::UGGrid<2>);
#else
SET_TYPE_PROP(TutorialProblemImplicit, Grid, Dune::YaspGrid<2>);
#endif // HAVE_DUNE_ALUGRID
```

```

// Set the wetting phase
SET_PROP(TutorialProblemImplicit, WettingPhase)
{
private: typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
public: typedef FluidSystems::LiquidPhase<Scalar, H2O<Scalar> > type;
};

// Set the non-wetting phase
SET_PROP(TutorialProblemImplicit, NonwettingPhase)
{
private: typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
public: typedef FluidSystems::LiquidPhase<Scalar, LNAPL<Scalar> > type;
};

SET_TYPE_PROP(TutorialProblemImplicit, FluidSystem, TwoPImmiscibleFluidSystem<TypeTag>);
// Disable gravity
SET_BOOL_PROP(TutorialProblemImplicit, ProblemEnableGravity, false);
}

/*!
 * \ingroup TwoPBoxModel
 *
 * \brief  Tutorial problem for a fully coupled twophase box model.
 */
template <class TypeTag>
class TutorialProblemImplicit : public ImplicitPorousMediaProblem<TypeTag>
{
    typedef ImplicitPorousMediaProblem<TypeTag> ParentType;
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
    typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;

    // Grid dimension
    enum { dim = GridView::dimension,
           dimWorld = GridView::dimensionworld
    };

    // Types from DUNE-Grid
    typedef typename GridView::template Codim<0>::Entity Element;
    typedef typename GridView::template Codim<dim>::Entity Vertex;
    typedef typename GridView::Intersection Intersection;
    typedef Dune::FieldVector<Scalar, dimWorld> GlobalPosition;

    // Dumux specific types
    typedef typename GET_PROP_TYPE(TypeTag, TimeManager) TimeManager;
    typedef typename GET_PROP_TYPE(TypeTag, Indices) Indices;
    typedef typename GET_PROP_TYPE(TypeTag, PrimaryVariables) PrimaryVariables;
    typedef typename GET_PROP_TYPE(TypeTag, BoundaryTypes) BoundaryTypes;
    typedef typename GET_PROP_TYPE(TypeTag, FVElementGeometry) FVElementGeometry;

public:
    TutorialProblemImplicit(TimeManager &timeManager,
                            const GridView &gridView)
        : ParentType(timeManager, gridView)
        , eps_(3e-6)
    {
#if !(HAVE_DUNE_ALUGRID || HAVE_UG)
      std::cout << "If you want to use simplices instead of cubes, install and use dune-ALUGrid
          or UGGrid." << std::endl;
#endif // !(HAVE_DUNE_ALUGRID || HAVE_UG)
    }

    //! Specifies the problem name. This is used as a prefix for files
    //! generated by the simulation.
    std::string name() const
    { return "tutorial_implicit"; }

    //! Returns true if a restart file should be written.
    bool shouldWriteRestartFile() const
    { return false; }

    //! Returns true if the current solution should be written to disk
    //! as a VTK file
    bool shouldWriteOutput() const
    {
        return
            (this->timeManager().timeStepIndex() > 0
             && (this->timeManager().timeStepIndex() % 1 == 0));
    }

    //! Returns the temperature within a finite volume. We use constant
    //! 10 degrees Celsius.
    Scalar temperature() const
    { return 283.15; }

    //! Specifies which kind of boundary condition should be used for
    //! which equation for a finite volume on the boundary.
    void boundaryTypes(BoundaryTypes &bcTypes, const Vertex &vertex) const
    {
        const GlobalPosition &globalPos = vertex.geometry().center();
        if (globalPos[0] < eps_) // Dirichlet conditions on left boundary
           bcTypes.setAllDirichlet();
        else // neuman for the remaining boundaries
           bcTypes.setAllNeumann();

    }

    //! Evaluates the Dirichlet boundary conditions for a finite volume
    //! on the grid boundary. Here, the 'values' parameter stores
    //! primary variables.
    void dirichlet(PrimaryVariables &values, const Vertex &vertex) const
    {
        values[Indices::pwIdx] = 200.0e3; // 200 kPa = 2 bar
        values[Indices::snIdx] = 0.0; // 0 % oil saturation on left boundary
    }

    //! Evaluates the boundary conditions for a Neumann boundary
    //! segment. Here, the 'values' parameter stores the mass flux in
    //! [kg/(m^2 * s)] in normal direction of each phase. Negative
    //! values mean influx.
    void neumann(PrimaryVariables &values,
                 const Element &element,
                 const FVElementGeometry &fvGeometry,
                 const Intersection &intersection,
                 int scvIdx,
                 int boundaryFaceIdx) const
    {
        const GlobalPosition &globalPos =
            fvGeometry.boundaryFace[boundaryFaceIdx].ipGlobal;
        Scalar right = this->bBoxMax()[0];
        // extraction of oil on the right boundary for approx. 1.e6 seconds
        if (globalPos[0] > right - eps_) {
            // oil outflux of 30 g/(m * s) on the right boundary.
            values[Indices::contiWEqIdx] = 0;
            values[Indices::contiNEqIdx] = 3e-2;
        } else {
```

```
186             // no-flow on the remaining Neumann-boundaries.
187             values[Indices::contiWEqIdx] = 0;
188             values[Indices::contiNEqIdx] = 0;
189         }
190     }
191 
192     //! Evaluates the initial value for a control volume. For this
193     //! method, the 'values' parameter stores primary variables.
194     void initial(PrimaryVariables &values,
195                  const Element &element,
196                  const FVElementGeometry &fvGeometry,
197                  int scvIdx) const
198     {
199         values[Indices::pwIdx] = 200.0e3; // 200 kPa = 2 bar
200         values[Indices::snIdx] = 1.0;
201     }
202 
203     //! Evaluates the source term for all phases within a given
204     //! sub-control-volume. In this case, the 'values' parameter
205     //! stores the rate mass generated or annihilated per volume unit
206     //! in [kg / (m^3 * s)]. Positive values mean that mass is created.
207     void source(PrimaryVariables &values,
208                 const Element &element,
209                 const FVElementGeometry &fvGeometry,
210                 int scvIdx) const
211     {
212         values[Indices::contiWEqIdx] = 0.0;
213         values[Indices::contiNEqIdx]= 0.0;
214     }
215 
216 private:
217     // small epsilon value
218     Scalar eps_;
219 };
220 }
221 
222 #endif
```

First, a new type tag is created for the problem in line 47. In this case, the new type tag inherits all properties from the BoxTwoP type tag, which means that for this problem the two-phase box model is chosen as discretization scheme. Further, it inherits from the spatial parameters type tag, which is defined in line 44 of the problem-dependent spatial parameters file. On line 50, a problem class is attached to the new type tag, while the grid which is going to be used is defined in line 54 – in this case that is Dune::YaspGrid. Since there's no uniform mechanism to allocate grids in DUNE, DuMu^x features the concept of grid creators. In this case the generic CubeGridCreator which creates a structured hexahedron grid of a specified size and resolution. For this grid creator the physical domain of the grid is specified via the run-time parameters Grid.UpperRight and Grid.Cells. These parameters can be specified via the command-line or in a parameter file.

Next, the appropriate fluid system, which specifies the thermodynamic relations of the fluid phases, has to be chosen. By default, the two-phase model uses the TwoPImmiscibleFluidSystem, which assumes immiscibility of the phases, but requires the components used for the wetting and non-wetting phases to be explicitly set. In this case, liquid water which uses the relations from IAPWS'97 [10] is chosen as the wetting phase on line 66 and liquid oil is chosen as the non-wetting phase on line 73. The last property, which is set in line 78, tells the model not to use gravity.

Parameters which are specific to a physical set-up to be simulated, such as boundary and initial conditions, source terms or temperature within the domain, and which are required to solve the differential equations of the models are specified via a *problem* class. This class should be derived from `ImplicitPorousMediaProblem` as done in line 87.

The problem class always has at least five methods:

- A method `boundaryTypes()` specifying the type of boundary conditions at each vertex.
- A method `dirichlet()` specifying the actual values for the Dirichlet conditions at each Dirichlet vertex.
- A method `neumann()` specifying the actual values for the Neumann conditions, which are usually evaluated at the integration points of the Neumann boundary faces.
- A method for source or sink terms called `source()`, usually evaluated at the center of a control volume.
- A method called `initial()` for specifying the initial conditions at each vertex.

For the definition of the boundary condition types and of the values of the Dirichlet boundaries, two parameters are available:

bcTypes/values: A vector which stores the result of the method. What the values in this vector mean is dependent on the method: For `dirichlet()`, `values` contains the actual values of the primary variables, for `boundaryTypes()`, `bcTypes` contains the boundary condition types. It has as many entries as the model has primary variables / equations. For the typical case, in which all equations have the same boundary condition type at a certain position, there are two methods that set the appropriate conditions for all primary variables / equations: `setAllDirichlet()` and `setAllNeumann()`.

vertex: The boundary condition and the Dirichlet values are specified for a vertex, which represents a sub-control volume in the box discretization. This inhibits the specification of two different boundary condition types for one equation at one sub-control volume. Be aware that the second parameter is a Dune grid entity with the co-dimension `dim`.

To ensure that no boundaries are undefined, a small safeguard value `eps_` is usually added when comparing spatial coordinates. The left boundary is hence not detected by checking, if the first coordinate of the global position is equal to zero, but by testing whether it is smaller than a very small value `eps_`.

Methods for box models which make statements about boundary segments of the grid (such as `neumann()`) are called with six arguments:

values: A vector `neumann()`, in which the mass fluxes per area unit over the boundary segment are specified.

element: The element of the grid where the boundary segment is located.

fvGeometry: The finite-volume geometry induced on the finite element by the box scheme.

intersection: The `Intersection` of the boundary segment as given by the grid.

scvIdx: The index of the sub-control volume in `fvGeometry` which is assigned to the boundary segment.

boundaryFaceIdx: The index of the boundary face in `fvGeometry` which represents the boundary segment.

Similarly, the `initial()` and `source()` methods specify properties of control volumes and thus only get `values`, `element`, `fvGeometry` and `scvIdx` as arguments.

In addition to these five methods, there might be some model-specific methods. If the isothermal two-phase model is used, this includes for example a `temperature()` method which returns the temperature in Kelvin of the fluids and the rock matrix in the domain. This temperature is then used by the model to calculate fluid properties which possibly depend on it, e.g. density. The `bBoxMax()` ("*max*imum coordinated of the grid's *b*ounding *box*") method is used here to determine the extend of the physical domain. It returns a vector with the maximum values of each global coordinate of the grid. This method and the analogous `bBoxMin()` method are provided by the base class `Dumux::BoxProblem<TypeTag>`.

3.1.3 Defining Fluid Properties

The DuMu^x distribution includes some common substances which can be used out of the box. The properties of the pure substances (such as the components nitrogen, water, or the pseudo-component air) are provided by header files located in the folder `dumux/material/components`.

Most often, when two or more components are considered, fluid interactions such as solubility effects come into play and properties of mixtures such as density or enthalpy are of interest. These interactions are defined by *fluid systems*, which are located in `dumux/material/fluidsystems`. A more thorough overview of the DuMu^x fluid framework can be found at `http://www.dumux.org/doxygen-stable/html-2.11/modules.php`

3.1.4 Defining Spatially Dependent Parameters

In DuMu^x, many properties of the porous medium can depend on the spatial location. Such properties are the *intrinsic permeability*, the parameters of the *capillary pressure* and the *relative permeability*, the *porosity*, the *heat capacity* as well as the *heat conductivity*. Such parameters are defined using a so-called *spatial parameters* class.

If the box discretization is used, the spatial parameters class should be derived from the base class `Dumux::BoxSpatialParams<TypeTag>`. Listing 4 shows the file `tutorialspatialparams_implicit.hh`:

Listing 4 (File tutorial/tutorialspatialparams_implicit.hh)

```
#ifndef DUMUX_TUTORIAL_SPATIAL_PARAMS_IMPLICIT_HH
#define DUMUX_TUTORIAL_SPATIAL_PARAMS_IMPLICIT_HH

// include parent spatialparameters
#include <dumux/material/spatialparams/implicit.hh>

// include material laws
#include <dumux/material/fluidmatrixinteractions/2p/regularizedbrookscorey.hh>
#include <dumux/material/fluidmatrixinteractions/2p/efftoabslaw.hh>
#include <dumux/material/fluidmatrixinteractions/2p/linearmaterial.hh>

namespace Dumux {
//forward declaration
template<class TypeTag>
```

```
class TutorialSpatialParamsImplicit;

namespace Properties
{
// The spatial parameters TypeTag
NEW_TYPE_TAG(TutorialSpatialParamsImplicit);

// Set the spatial parameters
SET_TYPE_PROP(TutorialSpatialParamsImplicit, SpatialParams,
        TutorialSpatialParamsImplicit<TypeTag>);

// Set the material law
SET_PROP(TutorialSpatialParamsImplicit, MaterialLaw)
{
private:
    // material law typedefs
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
    // select material law to be used
    typedef RegularizedBrooksCorey<Scalar> RawMaterialLaw;
public:
    // adapter for absolute law
    typedef EffToAbsLaw<RawMaterialLaw> type;
};
}

/*!
 * \ingroup TwoPBoxModel
 *
 * \brief The spatial parameters for the fully coupled tutorial problem
 *        which uses the twophase box model.
 */
template<class TypeTag>
class TutorialSpatialParamsImplicit: public ImplicitSpatialParams<TypeTag>
{
    // Get informations for current implementation via property system
    typedef typename GET_PROP_TYPE(TypeTag, Grid) Grid;
    typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
    enum
    {
        dim = Grid::dimension
    };

    // Get object types for function arguments
    typedef typename GET_PROP_TYPE(TypeTag, FVElementGeometry) FVElementGeometry;
    typedef typename Grid::Traits::template Codim<0>::Entity Element;

public:
    // get material law from property system
    typedef typename GET_PROP_TYPE(TypeTag, MaterialLaw) MaterialLaw;
    // determine appropriate parameters depending on selected materialLaw
    typedef typename MaterialLaw::Params MaterialLawParams;

    /*! Intrinsic permeability tensor K \f$[m^2]\f$ depending
     *  on the position in the domain
     *
     *  \param element The finite volume element
     *  \param fvGeometry The finite-volume geometry in the box scheme
     *  \param scvIdx The local vertex index
     *
     *  Alternatively, the function intrinsicPermeabilityAtPos(const GlobalPosition& globalPos)
     *  could be defined, where globalPos is the vector including the global coordinates
     *  of the finite volume.
     */
    const Dune::FieldMatrix<Scalar, dim, dim> &intrinsicPermeability(const Element &element,
                                                                     const FVElementGeometry &fvGeometry,
                                                                     const int scvIdx) const
    { return K_; }

    /*! Defines the porosity \f$[-]\f$ of the porous medium depending
     * on the position in the domain
     *
     *  \param element The finite volume element
     *  \param fvGeometry The finite-volume geometry in the box scheme
     *  \param scvIdx The local vertex index
     *
     *  Alternatively, the function porosityAtPos(const GlobalPosition& globalPos)
     *  could be defined, where globalPos is the vector including the global coordinates
     *  of the finite volume.
     */
    Scalar porosity(const Element &element,
                    const FVElementGeometry &fvGeometry,
                    const int scvIdx) const
    { return 0.2; }

    /*! Returns the parameter object for the material law (i.e. Brooks-Corey)
     *  depending on the position in the domain
     *
     *  \param element The finite volume element
     *  \param fvGeometry The finite-volume geometry in the box scheme
     *  \param scvIdx The local vertex index
     *
     *  Alternatively, the function materialLawParamsAtPos(const GlobalPosition& globalPos)
     *  could be defined, where globalPos is the vector including the global coordinates
     *  of the finite volume.
     */
    const MaterialLawParams& materialLawParams(const Element &element,
                                               const FVElementGeometry &fvGeometry,
                                               const int scvIdx) const
    {
        return materialParams_;
    }

    // constructor
    TutorialSpatialParamsImplicit(const GridView& gridView) :
        ImplicitSpatialParams<TypeTag>(gridView),
        K_(0)
    {
        //set main diagonal entries of the permeability tensor to a value
        //setting to one value means: isotropic, homogeneous
        for (int i = 0; i < dim; i++)
            K_[i][i] = 1e-7;

        //set residual saturations
        materialParams_.setSwr(0.0);
        materialParams_.setSnr(0.0);

        //parameters of Brooks & Corey Law
        materialParams_.setPe(500.0);
        materialParams_.setLambda(2);
    }

private:
    Dune::FieldMatrix<Scalar, dim, dim> K_;
    // Object that holds the values/parameters of the selected material law.
    MaterialLawParams materialParams_;
```

```
165 };
166 } // end namespace
167 #endif
```

First, the spatial parameters type tag is created on line 44. The type tag for the problem is then derived from it. The DuMu^x properties defined on the type tag for the spatial parameters are, for example, the spatial parameters class itself (line 48) or the capillary pressure/relative permeability relations[1] which ought to be used by the simulation (line 57). DuMu^x provides several material laws in the folder `dumux/material/fluidmatrixinteractions`. The selected one – here it is a relation according to a regularized version of Brooks & Corey – is included in line 32. After the selection, an adapter class is specified in line 60 to translate between effective and absolute saturations. Like this, residual saturations can be specified in a generic way. As only the employed material law knows the names of the parameters which it requires, it provides a parameter class `RegularizedBrooksCoreyParams` which has the type `Params` and which is defined in line 90. In this case, the spatial parameters only require a single set of parameters which means that it only requires a single material parameter object as can be seen in line 164.

In line 103, a method returning the intrinsic permeability is specified. As can be seen, the method has to be called with three arguments:

`element`: Just like for the problem itself, this parameter describes the considered element by means of a DUNE entity. Elements provide information about their geometry and position and can be mapped to a global index.

`fvGeometry`: It holds information about the finite-volume geometry of the element induced by the box method.

`scvIdx`: This is the index of the sub-control volume of the element which is considered. It is equivalent to the local index of the vertex which corresponds to the considered control volume in the element.

The intrinsic permeability is usually a tensor. Thus the method returns a `dim` $\times$ `dim`-matrix, where `dim` is the dimension of the grid.

The method `porosity()` defined in line 119 is called with the same arguments as `intrinsicPermeability()` and returns a scalar value for porosity dependent on the position in the domain.

Next, the method `materialLawParams()`, defined in line 135, returns the `materialParams_` object that is applied at the specified position. Although in this case only one object is returned, in general, the problem may be heterogeneous, which necessitates returning different objects at different positions in space. While the selection of the type of this object was already explained (line 32), some specific parameter values of the used material law, such as the Brooks & Corey parameters, are still needed. This is done in the constructor at line 153. Depending on the type of the `materialLaw` object, the `set`-methods might be different than those given in this example. The name of the access / set functions as well as the rest of the implementation of the material description can be found in `dumux/material/fluidmatrixinteractions/2p`.

3.1.5 Exercises

The following exercises will give you the opportunity to learn how you can change soil parameters, boundary conditions, run-time parameters and fluid properties in DuMu^x. Possible solutions to these

[1]Taken together, the capillary pressure and the relative permeability relations are called *material law*.

exercises are given in the tutorial folder in the sub-folder `solutions_implicit` as `.diff` files. In these files only the lines that are different from the original file are listed. They can be opened using the program `kompare`, simply type `kompare SOLUTIONFILE` into the terminal.

Exercise 1

For Exercise 1 you have to make only some small changes in the tutorial files.

a) **Running the Program**
To get an impression what the results should look like you can compile and run the original version of the implicit tutorial model by typing `make tutorial_implicit` followed by `./tutorial_implicit`. Note, that the time-step size is automatically adapted during the simulation. For the visualization of the results using ParaView, please refer to section 2.2.

b) **Changing the Model Domain and the Boundary Conditions**
Change the size of the model domain so that you get a rectangle with edge lengths of $\mathrm{x} = 400\,\mathrm{m}$ and $\mathrm{y} = 500\,\mathrm{m}$ and with discretization lengths of $\Delta\mathrm{x} = 20\,\mathrm{m}$ and $\Delta\mathrm{y} = 20\,\mathrm{m}$. For this you have to edit the parameter file (`tutorial_implicit.input`) and run the program again.
Note, that you do not have to recompile the program if you make changes to the parameter file.

Change the boundary conditions in the file `tutorialproblem_implicit.hh` so that water enters from the bottom and oil is extracted from the top boundary. The right and the left boundary should be closed for water and oil fluxes.
The Neumannn Boundary conditions are multiplied by the normal (pointing outwards), so an influx is negative, an outflux always positive. Such information can easily be found in the documentation of the functions (also look into base classes). Compile the main file by typing `make tutorial_implicit` and run the model as explained above.

c) **Changing the Shape of the Discrete Elements**
In order to complete this exercise you need an external grid manager capable of handling simplex grids, like `ALUGrid` or `UGGrid`. If this is not the case, please skip this exercise. Change the types of elements used for discretizing the domain. In line 54 of the problem file the grid is chosen. For ALUGrid you have to change the ALUGrid type in line 55 from `Dune::cube` to `Dune::simplex`. The shape of the employed elements can be visualized in ParaView by choosing `Surface with Edges`.

d) **Changing Fluids**
Now you can change the fluids. Use DNAPL instead of Oil and Brine instead of Water. To do that, you have to select different components via the property system in the problem file:

   a) Brine: Brine is thermodynamically very similar to pure water but also considers a fixed amount of salt in the liquid phase. Hence, the class `Dumux::Brine` uses a pure water class, such as `Dumux::H2O<Scalar>`, as a second template argument after the data type `<Scalar>`, i.e. `Dumux::Brine<Scalar, Dumux::H2O<Scalar>>`. The file is located in the folder `dumux/material/components/`. Try to include the file and select the component as the wetting phase via the property system.

   b) DNAPL: Now let's include a DNAPL (**d**ense **n**on-**a**queous **p**hase **l**iquid) which is located in the folder `dumux/material/components/`. Try to include the file and select the component as the non-wetting phase via the property system.

   If you want to take a closer look on how the fluid classes are defined and which substances are already available please browse through the files in the directory `/dumux/material/components` and read the doxygen documentation `http://www.dumux.org/doxygen-stable/html-2.11/modules.php`.

e) **Use a Full-Fledged Fluid System**
   DuMu^x usually describes fluid mixtures via *fluid systems*, see also chapter **??**. In order to include a fluid system, you first have to comment out lines 63 to 74 in the problem file. If you use eclipse, this can easily be done by pressing *Ctrl + Shift + 7* – the same as to cancel the comment later on.
   Now include the file `fluidsystems/h2oair.hh` in the material folder, and set a type property `FluidSystem` (see line 76) with the appropriate type, which is either:
   `Dumux::FluidSystems::H2OAir<typename GET_PROP_TYPE(TypeTag, Scalar)>`
   or in the DuMu^x tongue
   `Dumux::H2OAirFluidSystem<TypeTag>`
   However, this is a rather complicated fluid system which considers mixtures of components and also uses tabulated components that need to be initialized – i.e. the tables need to be filled with values. The initialization of the fluid system is normally done in the constructor of the problem by calling `GET_PROP_TYPE(TypeTag, FluidSystem)::init();`. Remember that the constructor function always has the same name as the respective class, i.e. `TutorialProblemImplicit(..)`.
   As water flow replacing a gas is much faster, test your simulation only until 2000 seconds and start with a time-step of 1 second.
   Please reverse the changes made in this part of the exercise, as we will continue to use immiscible phases from here on and hence do not need a complex fluid system.

f) **Changing Constitutive Relations**
   Use an unregularized linear law with an entry pressure of $p_e = 0.0\,\mathrm{Pa}$ and maximal capillary pressure of e.g. $p_{c_{max}} = 2000.0\,\mathrm{Pa}$ instead of using a regularized Brooks-Corey law for the relative permeability and for the capillary pressure saturation relationship. To do that you have to change the material law property (line 57) in `tutorialspatialparams_implicit.hh`. Exchange the `RegularizedBrooksCorey` material law with the `LinearMaterial` law type in the private section of the property definition. You can find the material laws in the folder `dumux/material/fluidmatrixinteractions`. The necessary parameters of the linear law and the respective `set`-functions can be found in the file
   `dumux/material/fluidmatrixinteractions/2p/linearmaterialparams.hh`.
   Call the `set`-functions from the constructor of the `tutorialspatialparams_implicit.hh`.

g) **Heterogeneities**
   Set up a model domain with the soil properties given in Figure 3.2. Adjust the boundary conditions so that water is again flowing from the left to the right of the domain. You can use the fluids of exercise 1b.
   Hint: The current position of the control volume can be obtained using `element.geometry().corner(scvIdx)`, which returns a vector of the global coordinates of the current position.

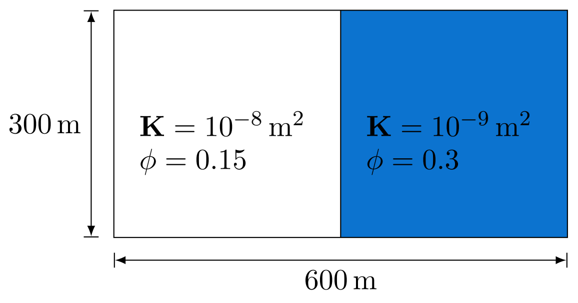

300 m
$\mathbf{K} = 10^{-8}\,\mathrm{m}^2$
$\phi = 0.15$
$\mathbf{K} = 10^{-9}\,\mathrm{m}^2$
$\phi = 0.3$
600 m

Figure 3.2: Exercise 1g: Set-up of a model domain with a heterogeneity. Grid spacing: $\Delta x = 20\,\mathrm{m}$ $\Delta y = 20\,\mathrm{m}$.

When does the front cross the material border? In ParaView, the animation view (*View* → *Animation View*) is a convenient way to get a rough feeling of the time-step sizes.

Exercise 2

For this exercise you should create a new problem file analogous to the file `tutorialproblem_implicit.hh` (e.g. with the name `ex2_tutorialproblem_implicit.hh` and new spatial parameters `ex2_tutorial spatialparams_implicit.hh`, just like `tutorialspatialparams_implicit.hh`.

The new files should contain the definition of new classes with names that relate to the file name, such as `Ex2TutorialProblemImplicit`. Make sure that you also adjust the guardian macros in lines 24 and 25 in the header files (e.g. change `DUMUX_TUTORIALPROBLEM_IMPLICIT_HH` to `DUMUX_EX2_TUTORIALPROBLEM_IMPLICIT_HH`). Include the new problem file in `tutorial_implicit.cc`. Besides adjusting the guardian macros, the new problem file should define and use a new type tag for the problem as well as a new problem class e.g. `Ex2TutorialProblemImplicit`. The type tag definition has to be adjusted in `tutorial_implicit.cc` too (see line 47). Similarly adjust your new spatial parameters file. If you are using Eclipse there is a very helpful function called `Refactor` which you can use to change all similar variables or types in your current file in one go. Just place the cursor at the variable or type you want to change and use the `Refactor` → `Rename` functionality. Make sure to assign your newly defined spatial parameter class to the `SpatialParams` property for the new type tag.

After this, change the run-time parameters so that they match the domain described by figure 3.3. Adapt the problem class so that the boundary conditions are consistent with figure 3.4. Initially, the domain is fully saturated with water and the pressure is $p_w = 5 \times 10^5\,\mathrm{Pa}$. Oil infiltrates from the left side. Create a grid with 20 cells in x-direction and 10 cells in y-direction. The simulation time should be set to $10^6\,\mathrm{s}$ with an initial time-step size of $100\,\mathrm{s}$. Then, you can compile the program.

- Increase the simulation time to e.g. $4 \times 10^7\,\mathrm{s}$. Investigate the saturation: Is the value range reasonable?
- What happens if you increase the resolution of the grid?

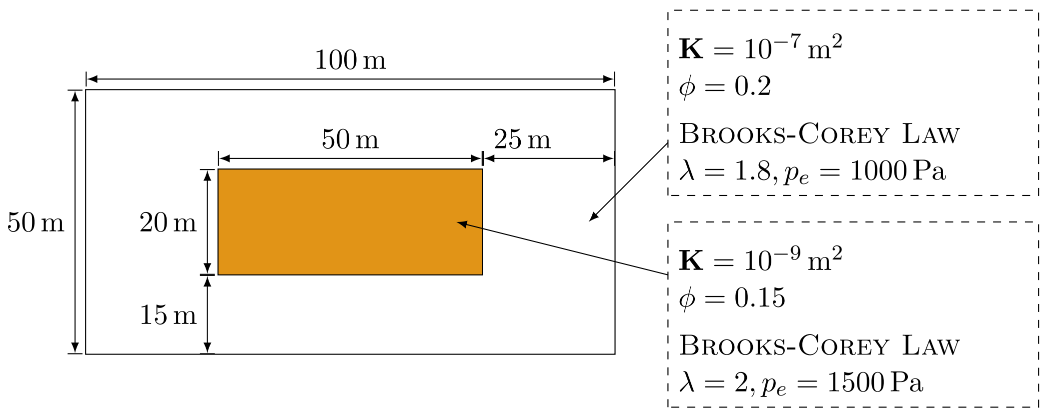

100 m
50 m
25 m
50 m
20 m
15 m
$\mathbf{K} = 10^{-7}\,\mathrm{m}^2$
$\phi = 0.2$
Brooks-Corey Law
$\lambda = 1.8, p_e = 1000\,\mathrm{Pa}$
$\mathbf{K} = 10^{-9}\,\mathrm{m}^2$
$\phi = 0.15$
Brooks-Corey Law
$\lambda = 2, p_e = 1500\,\mathrm{Pa}$

Figure 3.3: Set-up of the model domain and the soil parameters

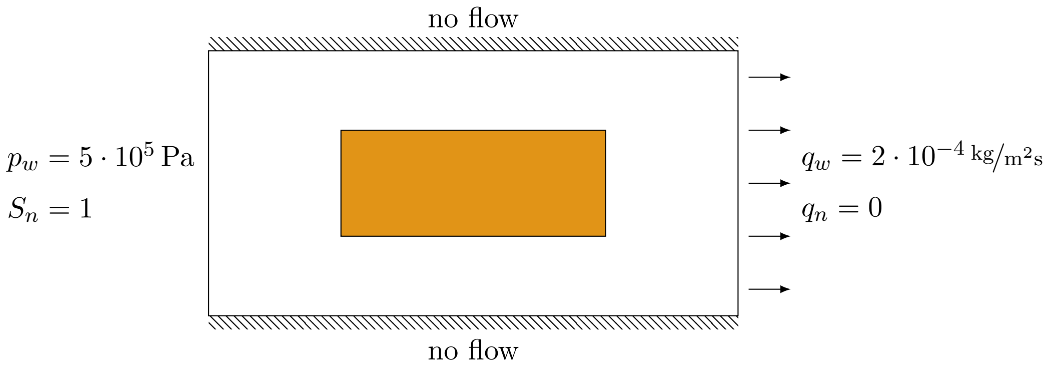

no flow
$p_w = 5 \cdot 10^5\,\mathrm{Pa}$
$S_n = 1$
$q_w = 2 \cdot 10^{-4}\,\mathrm{kg/m^2s}$
$q_n = 0$
no flow

Figure 3.4: Boundary Conditions

Exercise 3: Parameter File Input

As you have experienced, compilation takes quite some time. Therefore, DuMu^x provides a simple method to read in parameters at run-time via *parameter input files*.

In the code, parameters can be read via the macro `GET_RUNTIME_PARAM(TypeTag, Scalar, MyWonderfulGroup.MyWonderfulParameter);`. In this exercise we will explore the possibilities of the parameter file. For this we take a look at the file `ex3_tutorial_implicit.input` in the `solutions_implicit` folder. Besides the parameters which you already used in the parameter file above, there are parameters which can be used to control the Newton and the Linear solver (groups: `Newton` and `LinearSolver`). Run-time parameters used in the problem or spatial parameters classes can also be set with the respective group names (`Problem` and `SpatialParams`) in the parameter file. For the latter parameters to be included in the program they have to be assigned in the problem or spatial parameters constructor. This can be done as shown in the files `ex3_tutorialproblem_implicit.diff` and `ex3_tutorialspatialparams_implicit.diff` in the `solutions_implicit` folder. Add some (for example `Newton.MaxSteps` and `Problem.EnableGravity`) to the parameter file `tutorial_implicit.input` and observe what happens if they are modified.

Exercise 4: Create a New Component

Create a new file for the benzene component called `benzene.hh` and implement a new component. (You may get a hint by looking at existing components in the directory `/dumux/material/components`). Use benzene as a new fluid and run the model of Exercise 2 with water and benzene. Benzene has a density of 889.51 $\mathrm{kg/m^3}$ and a viscosity of 0.00112 Pa s.

Exercise 5: Time Dependent Boundary Conditions

In this exercise we want to investigate the influence of time dependent boundary conditions. For this, redo the steps of exercise 2 and create a new problem and spatial parameters file.

After this, change the run-time parameters so that they match the domain described by figure 3.5. Adapt the problem class so that the boundary conditions are consistent with figure 3.6. Here you can see the time dependence of the nonwetting saturation, where water infiltrates only during 10^5 s and $4 \cdot 10^5$ s. To implement these time dependencies you need the actual time $t_{n+1} = t_n + \Delta t$ and the endtime of the simulation. For this you can use the methods `this->timeManager().time()`, `this->timeManager().timeStepSize()` and `this->timeManager().endTime()`.

Initially, the domain is fully saturated with oil and the pressure is $p_w = 2 \times 10^5$ Pa. Water infiltrates from the left side. Create a grid with 100 cells in x-direction and 10 cells in y-direction. The simulation time should be set to $5 \cdot 10^5$ s with an initial time-step size of 10 s. To avoid too big time-step sizes you should set the parameter `MaxTimeStepSize` for the group `TimeManager` (in your input file) to 1000 s. Then, you can compile the program.

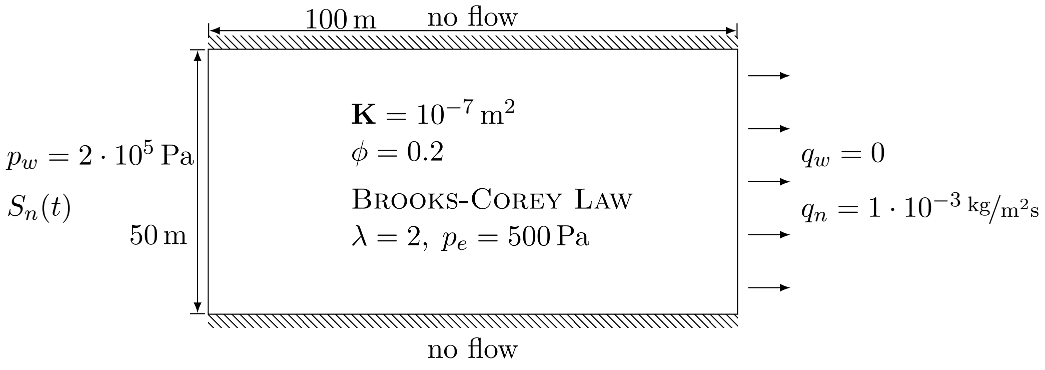

100 m
no flow
$\mathbf{K} = 10^{-7}\,\mathrm{m^2}$
$\phi = 0.2$
Brooks-Corey Law
$\lambda = 2,\; p_e = 500\,\mathrm{Pa}$
$p_w = 2 \cdot 10^5\,\mathrm{Pa}$
$S_n(t)$
50 m
$q_w = 0$
$q_n = 1 \cdot 10^{-3}\,\mathrm{kg/m^2s}$
no flow

Figure 3.5: Set-up of the model domain and the soil parameters

- Open ParaView and plot the values of S_n at time $5 \cdot 10^5$ s over the x-axis. (`Filter->Data Analysis->Plot Over Line`)
- What happens without any time-step restriction?

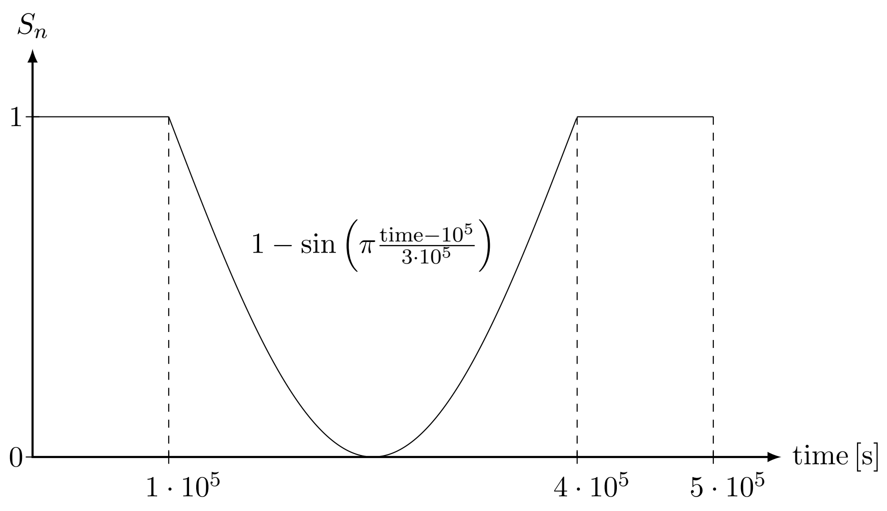


Figure 3.6: Time Dependent Boundary Conditions

3.2 Solving a problem using a Sequential Model

The process of solving a problem using DuMu^x can be roughly divided into four parts:

a) The geometry of the problem and correspondingly a grid have to be defined.

b) Material properties and constitutive relationships have to be defined.

c) Boundary conditions as well as initial conditions have to be defined.

d) A suitable model has to be chosen.

In contrast to the last section, we now apply a sequential solution procedure, a so-called *IMPET* (*IM*plicit *P*ressure *E*xplicit *T*ransport) algorithm. This means that the pressure equation is first solved using an implicit method. The resulting velocities are then used to solve a transport equation explicitly.
In this tutorial, pure fluid phases are solved with a finite volume discretization of both pressure- and transport step. Primary variables, according to default settings of the model, are the pressure and the saturation of the wetting phase.

The problem which is solved in this tutorial is illustrated in figure 3.7. A rectangular domain with no flow boundaries on the top and at the bottom, which is initially saturated with oil, is considered. Water infiltrates from the left side into the domain. Gravity effects are neglected.

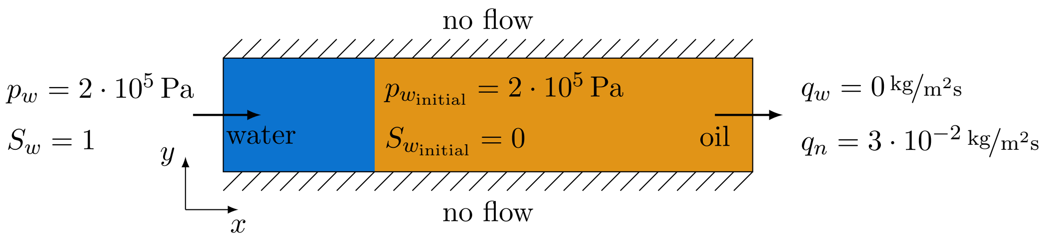

no flow
$p_w = 2 \cdot 10^5$ Pa
$S_w = 1$
water
$p_{w_{\text{initial}}} = 2 \cdot 10^5$ Pa
$S_{w_{\text{initial}}} = 0$
oil
$q_w = 0\,\text{kg}/\text{m}^2\text{s}$
$q_n = 3 \cdot 10^{-2}\,\text{kg}/\text{m}^2\text{s}$
y
x
no flow

Figure 3.7: Geometry of the tutorial problem with initial and boundary conditions.

Listing 5 shows how the main file, which has to be executed, has to be set up, if the problem described above is to be solved using a sequential model. This main file can be found in the directory `/tutorial` of the stable part of DuMu^x.

Listing 5 (File tutorial/tutorial_sequential.cc)

```
#include <config.h>

#include "tutorialproblem_sequential.hh"
#include <dumux/common/start.hh>

//! Prints a usage/help message if something goes wrong or the user asks for help
void usage(const char *progName, const std::string &errorMsg)
{
    std::cout
        <<  "\nUsage: " << progName << " [options]\n";
    if (errorMsg.size() > 0)
        std::cout << errorMsg << "\n";
    std::cout
37          << "\n"
38          << "The list of mandatory arguments for this program is:\n"
39          << "\t-TEnd                   The end of the simulation [s]\n"
40          << "\t-DtInitial              The initial timestep size [s]\n"
41          << "\t-Grid.UpperRight        The x-/y-coordinates of the grid's upper-right corner [m]\n"
42          << "\t-Grid.Cells             The grid's x-/y-resolution\n"
43          << "\n";
44 }
45
46
47 ////////////////////////
48 // the main function
49 ////////////////////////
50 int main(int argc, char** argv)
51 {
52     typedef TTAG(TutorialProblemSequential) TypeTag;
53     return Dumux::start<TypeTag>(argc, argv, usage);
54 }
```

First, from line 24 to line 27 the DUNE and DuMu^x files containing essential functions and classes are included.

At line 52 the type tag of the problem which is going to be simulated is set. All other data types can be retrieved by the DuMu^x property system and only depend on this single type tag. For an introduction to the property system, see section 5.4.

After this DuMu^x' default startup routine `Dumux::start()` is called in line 53. This function deals with parsing the command line arguments, reading the parameter file, setting up the infrastructure necessary for DUNE, loading the grid, and starting the simulation. All parameters can be either specified by command line arguments of the form (`-ParameterName ParameterValue`), in the file specified by the `-parameterFile` argument, or if the latter is not specified, in the file `tutorial_sequential.input`. If a parameter is specified on the command line as well as in the parameter file, the values provided in the command line have precedence. Listing 6 shows the default parameter file for the tutorial problem.

Listing 6 (File tutorial/tutorial_sequential.input)

```
[TimeManager]
TEnd = 100000 # duration of the simulation [s]
DtInitial = 10 # initial time step size [s]

[Grid]
UpperRight = 300 60 # x-/y-coordinate of the upper-right corner of the grid [m]
Cells = 100 1 # x-/y-resolution of the grid
```

To provide an error message, the usage message which is displayed to the user if the simulation is called incorrectly, is printed via the custom function which is defined on line 30. In this function the usage message is customized to the problem at hand. This means that at least the necessary parameters are listed here.

3.2.1 The Problem Class

When solving a problem using DuMu^x, the most important file is the so-called *problem file* as shown in listing 7 of `tutorialproblem_sequential.hh`.

Listing 7 (File tutorial/tutorialproblem_sequential.hh)

```
#ifndef DUMUX_TUTORIALPROBLEM_SEQUENTIAL_HH // guardian macro
#define DUMUX_TUTORIALPROBLEM_SEQUENTIAL_HH // guardian macro

// dumux 2p-sequential environment
#include <dumux/porousmediumflow/2p/sequential/diffusion/cellcentered/pressureproperties.hh>
#include <dumux/porousmediumflow/2p/sequential/transport/cellcentered/properties.hh>
#include <dumux/porousmediumflow/2p/sequential/impes/problem.hh>

// assign parameters dependent on space (e.g. spatial parameters)
#include "tutorialspatialparams_sequential.hh"

// include cfl-criterion after coats: more suitable if the problem is not advection dominated
#include<dumux/porousmediumflow/2p/sequential/transport/cellcentered/evalcflfluxcoats.hh>

// the components that are used
#include <dumux/material/components/h2o.hh>
#include <dumux/material/components/lnapl.hh>

namespace Dumux
{

template<class TypeTag>
class TutorialProblemSequential;

//////////
// Specify the properties for the lens problem
//////////
namespace Properties
{
// create a new type tag for the problem
NEW_TYPE_TAG(TutorialProblemSequential, INHERITS_FROM(FVPressureTwoP, FVTransportTwoP,
    IMPESTwoP,
                                                     TutorialSpatialParamsSequential));

// Set the problem property
SET_PROP(TutorialProblemSequential, Problem)
{
    typedef TutorialProblemSequential<TypeTag> type;
};

// Set the grid type
SET_TYPE_PROP(TutorialProblemSequential, Grid, Dune::YaspGrid<2>);

// Set the wetting phase
SET_PROP(TutorialProblemSequential, WettingPhase)
{
private:
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
public:
    typedef FluidSystems::LiquidPhase<Scalar, H2O<Scalar> > type;
};

// Set the non-wetting phase
SET_PROP(TutorialProblemSequential, NonwettingPhase)
{
private:
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
public:
    typedef FluidSystems::LiquidPhase<Scalar, LNAPL<Scalar> > type;
};

SET_TYPE_PROP(TutorialProblemSequential, EvalCflFluxFunction, EvalCflFluxCoats<TypeTag>);
SET_SCALAR_PROP(TutorialProblemSequential, ImpetCFLFactor, 0.95);
```

```

// Disable gravity
SET_BOOL_PROP(TutorialProblemSequential, ProblemEnableGravity, false);
}

/*! \ingroup SequentialProblems
 * @brief Problem class for the sequential tutorial
*/
template<class TypeTag>
class TutorialProblemSequential: public IMPESProblem2P<TypeTag>
{
    typedef IMPESProblem2P<TypeTag> ParentType;
    typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
    typedef typename GET_PROP_TYPE(TypeTag, TimeManager) TimeManager;
    typedef typename GET_PROP_TYPE(TypeTag, Indices) Indices;

    typedef typename GET_PROP_TYPE(TypeTag, BoundaryTypes) BoundaryTypes;
    typedef typename GET_PROP(TypeTag, SolutionTypes) SolutionTypes;
    typedef typename SolutionTypes::PrimaryVariables PrimaryVariables;

    enum
    {
        dimWorld = GridView::dimensionworld
    };

    enum
    {
        wPhaseIdx = Indices::wPhaseIdx,
        nPhaseIdx = Indices::nPhaseIdx,
        pwIdx = Indices::pwIdx,
        swIdx = Indices::swIdx,
        pressEqIdx = Indices::pressureEqIdx,
        satEqIdx = Indices::satEqIdx
    };

    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;

    typedef typename GridView::Traits::template Codim<0>::Entity Element;
    typedef typename GridView::Intersection Intersection;
    typedef Dune::FieldVector<Scalar, dimWorld> GlobalPosition;

public:
    TutorialProblemSequential(TimeManager &timeManager, const GridView &gridView)
        : ParentType(timeManager, gridView), eps_(1e-6)
    {
        //write only every 10th time step to output file
        this->setOutputInterval(10);
    }

    //! The problem name.
    /*! This is used as a prefix for files generated by the simulation.
    */
    std::string name() const
    {
        return "tutorial_sequential";
    }

    //!  Returns true if a restart file should be written.
    /* The default behaviour is to write no restart file.
     */
    bool shouldWriteRestartFile() const
    {
        return false;
    }

    //! Returns the temperature within the domain at position globalPos.
    /*! This problem assumes a temperature of 10 degrees Celsius.
     *
     *  \param element The finite volume element
     *
     * Alternatively, the function temperatureAtPos(const GlobalPosition& globalPos) could be
     * defined, where globalPos is the vector including the global coordinates of the finite
       volume.
     */
    Scalar temperature(const Element& element) const
    {
        return 273.15 + 10; // -> 10Â°C
    }

    //! Returns a constant pressure to enter material laws at position globalPos.
    /* For incrompressible simulations, a constant pressure is necessary
     * to enter the material laws to gain a constant density etc. In the compressible
     * case, the pressure is used for the initialization of material laws.
     *
     * \param element The finite volume element
     *
     * Alternatively, the function referencePressureAtPos(const GlobalPosition& globalPos)
       could be
     * defined, where globalPos is the vector including the global coordinates of the finite
       volume.
     */
    Scalar referencePressure(const Element& element) const
    {
        return 2e5;
    }

    //! Source of mass \f$ [\frac{kg}{m^3 \cdot s}] \f$ of a finite volume.
    /*! Evaluate the source term for all phases within a given
     *  volume.
     *
     *  \param values Includes sources for the two phases
     *  \param element The finite volume element
     *
     *  The method returns the mass generated (positive) or
     *  annihilated (negative) per volume unit.
     *
     * Alternatively, the function sourceAtPos(PrimaryVariables &values, const GlobalPosition&
       globalPos)
     * could be defined, where globalPos is the vector including the global coordinates of the
       finite volume.
     */
    void source(PrimaryVariables &values, const Element& element) const
    {
        values = 0;
    }

    //! Type of boundary conditions at position globalPos.
    /*! Defines the type the boundary condition for the pressure equation,
     *  either pressure (dirichlet) or flux (neumann),
     *  and for the transport equation,
     *  either saturation (dirichlet) or flux (neumann).
     *
     *  \param bcTypes Includes the types of boundary conditions
     *  \param globalPos The position of the center of the finite volume
     *
     *  Alternatively, the function boundaryTypes(PrimaryVariables &values, const Intersection&
     *  intersection) could be defined, where intersection is the boundary intersection.
     */
    void boundaryTypesAtPos(BoundaryTypes &bcTypes, const GlobalPosition& globalPos) const
    {
            if (globalPos[0] < this->bBoxMin()[0] + eps_)
            {
                bcTypes.setDirichlet(pwIdx);
                bcTypes.setDirichlet(swIdx);
//                bcTypes.setAllDirichlet(); // alternative if the same BC is used for all
    primary variables
            }
            // all other boundaries
            else
            {
                bcTypes.setNeumann(pressEqIdx);
                bcTypes.setNeumann(satEqIdx);
//                bcTypes.setAllNeumann(); // alternative if the same BC is used for all
    equations
            }
    }
    //! Value for dirichlet boundary condition at position globalPos.
    /*! In case of a dirichlet BC for the pressure equation the pressure \f$ [Pa] \f$, and for
     *  the transport equation the saturation [-] have to be defined on boundaries.
     *
     *  \param values Values of primary variables at the boundary
     *  \param intersection The boundary intersection
     *
     *  Alternatively, the function dirichletAtPos(PrimaryVariables &values, const
         GlobalPosition& globalPos)
     *  could be defined, where globalPos is the vector including the global coordinates of the
          finite volume.
     */
    void dirichlet(PrimaryVariables &values, const Intersection& intersection) const
    {
        values[pwIdx] = 2e5;
        values[swIdx] = 1.0;
    }
    //! Value for neumann boundary condition \f$ [\frac{kg}{m^3 \cdot s}] \f$ at position
        globalPos.
    /*! In case of a neumann boundary condition, the flux of matter
     *  is returned as a vector.
     *
     *  \param values Boundary flux values for the different phases
     *  \param globalPos The position of the center of the finite volume
     *
     *  Alternatively, the function neumann(PrimaryVariables &values, const Intersection&
         intersection) could be defined,
     *  where intersection is the boundary intersection.
     */
    void neumannAtPos(PrimaryVariables &values, const GlobalPosition& globalPos) const
    {
        values = 0;
        if (globalPos[0] > this->bBoxMax()[0] - eps_)
        {
            values[nPhaseIdx] = 3e-2;
        }
    }
    //! Initial condition at position globalPos.
    /*! Only initial values for saturation have to be given!
     *
     *  \param values Values of primary variables
     *  \param element The finite volume element
     *
```

```
264      * Alternatively, the function initialAtPos(PrimaryVariables &values, const GlobalPosition
           & globalPos)
265      * could be defined, where globalPos is the vector including the global coordinates of the
           finite volume.
266      */
267     void initial(PrimaryVariables &values,
268             const Element &element) const
269     {
270         values = 0;
271     }
272
273 private:
274     const Scalar eps_;
275 };
276 } //end namespace
277
278 #endif
```

First, both DUNE grid handlers and the sequential model of DuMu$^{\mathrm{x}}$ have to be included. Then, a new type tag is created for the problem in line 55. In this case, the new type tag inherits all properties defined for the `SequentialTwoP` type tag, which means that for this problem the two-phase sequential approach is chosen as discretization scheme (defined via the include in line 30). On line 58, a problem class is attached to the new type tag, while the grid which is going to be used is defined in line 64 – in this case an `YaspGrid` is created. Since there's no uniform mechanism to allocate grids in DUNE, DuMu$^{\mathrm{x}}$ features the concept of grid creators. In this case the generic `CubeGridCreator` which creates a structured hexahedron grid of a specified size and resolution. For this grid creator the physical domain of the grid is specified via the run-time parameters `Grid.UpperRight` and `Grid.Cells`. These parameters can be specified via the command-line or in a parameter file. For more information about the DUNE grid interface, the different grid types that are supported and the generation of different grids, consult chapter 5.5 of this document and the *Dune Grid Interface HOWTO* [8].

Next, we select the material of the simulation: In the case of a pure two-phase model, each phase is a bulk fluid, and the complex (compositional) fluidsystems do not need to be used. However, they can be used (see exercise 1 4). Instead, we use a simplified fluidsystem container that provides classes for liquid and gas phases, line 67 to 82. These are linked to the appropriate chemical species in line 72 and 81. For all parameters that depend on space, such as the properties of the soil, the specific spatial parameters for the problem of interest are specified in line 47.

Now we arrive at some model parameters of the applied two-phase sequential model. First, in line 84 a flux function for the evaluation of the cfl-criterion is defined. This is optional as there exists also a default flux function. The choice depends on the problem which has to be solved. For cases which are not advection dominated the one chosen here is more reasonable. Line 85 assigns the CFL-factor to be used in the simulation run, which scales the time-step size (kind of security factor). The last property in line 88 is optional and tells the model not to use gravity.

After all necessary information is written into the property system and its namespace is closed in line 89, the problem class is defined in line 95. As its property, the problem class itself is also derived from a parent, `IMPESProblem2P`. The class constructor (line 129) is able to hold two vectors, which is not needed in this tutorial.

Beside the definition of the boundary and initial conditions (discussed in subsection 3.1.2 from 4^{th} paragraph on page 18), the problem class also contains general information about the current simulation. First, the name used by the `VTK-writer` to generate output is defined in the method of line 138, and line 146 indicates whether restart files are written. As sequential schemes usually feature small time-steps, it can be usefull to set an output interval larger than 1. The respective function is called in line 132, which gets the output interval as argument.

The following methods all have in common that they may be dependent on space. Hence, they all have either an `element` or an `intersection` as their function argument: Both are DUNE entities, depending on whether the parameter of the method is defined in an element, such as initial values, or on an intersection, such as a boundary condition. As it may be sufficient to return values only based on a position, DuMu^x models can also access functions in the problem with the form `...AtPos(GlobalPosition& globalPos)`, without an DUNE entity, as one can see in line 209.

There are the methods for general parameters, source- or sinkterms, boundary conditions (lines 209 to 250) and initial values for the transported quantity in line 268. For more information on the functions, consult the documentation in the code.

3.2.2 The Definition of the Parameters that are Dependent on Space

Listing 8 shows the file `tutorialspatialparams_sequential.hh`:

Listing 8 (File tutorial/tutorialspatialparams_sequential.hh)

```
#ifndef DUMUX_TUTORIAL_SPATIAL_PARAMS_SEQUENTIAL_HH
#define DUMUX_TUTORIAL_SPATIAL_PARAMS_SEQUENTIAL_HH


#include <dumux/material/spatialparams/fv.hh>
#include <dumux/material/fluidmatrixinteractions/2p/linearmaterial.hh>
#include <dumux/material/fluidmatrixinteractions/2p/regularizedbrookscorey.hh>
#include <dumux/material/fluidmatrixinteractions/2p/efftoabslaw.hh>

namespace Dumux
{

//forward declaration
template<class TypeTag>
class TutorialSpatialParamsSequential;

namespace Properties
{
// The spatial parameters TypeTag
NEW_TYPE_TAG(TutorialSpatialParamsSequential);

// Set the spatial parameters
SET_TYPE_PROP(TutorialSpatialParamsSequential, SpatialParams,
        TutorialSpatialParamsSequential<TypeTag>);

// Set the material law
SET_PROP(TutorialSpatialParamsSequential, MaterialLaw)
{
private:
    // material law typedefs
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
    typedef RegularizedBrooksCorey<Scalar> RawMaterialLaw;
public:
    typedef EffToAbsLaw<RawMaterialLaw> type;
};
}

//! Definition of the spatial parameters for the sequential tutorial

template<class TypeTag>
```

```
class TutorialSpatialParamsSequential: public FVSpatialParams<TypeTag>
{
    typedef FVSpatialParams<TypeTag> ParentType;
    typedef typename GET_PROP_TYPE(TypeTag, Grid) Grid;
    typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
    typedef typename Grid::ctype CoordScalar;

    enum
        {dim=Grid::dimension, dimWorld=Grid::dimensionworld};
    typedef typename Grid::Traits::template Codim<0>::Entity Element;

    typedef Dune::FieldVector<CoordScalar, dimWorld> GlobalPosition;
    typedef Dune::FieldMatrix<Scalar,dim,dim> FieldMatrix;

public:
    typedef typename GET_PROP_TYPE(TypeTag, MaterialLaw) MaterialLaw;
    typedef typename MaterialLaw::Params MaterialLawParams;

    //! Intrinsic permeability tensor K \f$[m^2]\f$ depending
    /*! on the position in the domain
     *
     *  \param element The finite volume element
     *
     *  Alternatively, the function intrinsicPermeabilityAtPos(const GlobalPosition& globalPos)
          could be
     *  defined, where globalPos is the vector including the global coordinates of the finite
         volume.
     */
    const FieldMatrix& intrinsicPermeability (const Element& element) const
    {
            return K_;
    }

    //! Define the porosity \f$[-]\f$ of the porous medium depending
    /*! on the position in the domain
     *
     *  \param element The finite volume element
     *
     *  Alternatively, the function porosityAtPos(const GlobalPosition& globalPos) could be
     *  defined, where globalPos is the vector including the global coordinates of the finite
         volume.
     */
    double porosity(const Element& element) const
    {
        return 0.2;
    }

    /*! Return the parameter object for the material law (i.e. Brooks-Corey)
     *  depending on the position in the domain
     *
     *  \param element The finite volume element
     *
     *  Alternatively, the function materialLawParamsAtPos(const GlobalPosition& globalPos)
     *  could be defined, where globalPos is the vector including the global coordinates of
     *  the finite volume.
     */
    const MaterialLawParams& materialLawParams(const Element &element) const
    {
            return materialLawParams_;
    }

    //! Constructor
    TutorialSpatialParamsSequential(const GridView& gridView)
    : ParentType(gridView), K_(0)
    {
        for (int i = 0; i < dim; i++)
                K_[i][i] = 1e-7;

        // residual saturations
        materialLawParams_.setSwr(0);
        materialLawParams_.setSnr(0);

        // parameters for the Brooks-Corey Law
        // entry pressures
        materialLawParams_.setPe(500);

        // Brooks-Corey shape parameters
        materialLawParams_.setLambda(2);
    }

private:
    MaterialLawParams materialLawParams_;
    FieldMatrix K_;
};

} // end namespace
#endif
```

As this file only slightly differs from the implicit version, consult chapter 3.1.4 for explanations. However, as a standard Finite Volume scheme is used, in contrast to the box-method in the implicit case, the argument list here is the same as for the problem functions: Either an `element`, or only the global position if the function is called `...AtPos(...)`.

3.2.3 Exercises

The following exercises will give you the opportunity to learn how you can change soil parameters, boundary conditions and fluid properties in DuMu^x and to play along with the sequential modelling framework.

Exercise 1

For Exercise 1 you only have to make some small changes in the tutorial files.

a) **Altering output**

To get an impression what the results should look like you can first run the original version of the sequential tutorial model by typing `./tutorial_sequential`. The runtime parameters which are set can be found in the input file (listing 6). If the input file has the same name than the main file (e.g. `tutorial_sequential.cc` and `tutorial_sequential.input`), it is automatically chosen. If the name differs the program has to be started typing `./tutorial_sequential -parameterFile <filename>.input`. For more options you can also type `./tutorial_sequential -h`. For the visualisation with paraview please refer to 2.2.
As you can see, the simulation creates many output files. To reduce these in order to perform longer simulations, change the method responsible for output (line 132 in the file `tutorialproblem_sequential`) as to write an output only every 20 time-steps. Compile the main file by typing `make tutorial_sequential` and run the model. Now, run the simulation for 5e5 seconds.

b) **Changing the Model Domain and the Boundary Conditions**
Change the size of the model domain so that you get a rectangle with edge lengths of x = 300 m and y = 300 m and with discretisation lengths of Δx = 20 m and Δy = 10 m.
Change the boundary conditions in the file `tutorialproblem_sequential.hh` so that water enters from the bottom and oil flows out at the top boundary. The right and the left boundary should be closed for water and oil fluxes. The Neumannn Boundary conditions are multiplied by the normal (pointing outwards), so an influx is negative, an outflux always positive. Such information can easily be found in the documentation of the functions (also look into base classes).

c) **Changing Fluids**
Now you can change the fluids. Use DNAPL instead of Oil and Brine instead of Water. To do that you have to select different components via the property system in the problem file:

   a) Brine: The class `Dumux::Brine` acts as an adapter to the fluid system that alters a pure water class by adding some salt. Hence, the class `Dumux::Brine` uses a pure water class, such as `Dumux::H2O`, as a second template argument after the data type `<Scalar>` as a template argument (be sure to use the complete water class with its own template parameter).

   b) DNAPL: A standard set of chemical substances, such as Water and Brine, is already included (via a list of `#include ..` commandos) and hence easily accessible by default. This is not the case for the class `Dumux::DNAPL`, however, which is located in the folder `dumux/material/components/`. Try to include the file as well as select the component via the property system.

If you want to take a closer look at how the fluid classes are defined and which substances are already available please browse through the files in the directory `/dumux/material/components`.

d) **Use the DuMu$^\text{x}$ fluid system**
DuMu$^\text{x}$ usually organizes fluid mixtures via a `fluidsystem`, see also chapter ??. In order to include a fluidsystem you first have to comment the lines 67 to 82 in the problem file. If you use eclipse, this can easily be done by pressing *str + shift + 7* – the same as to cancel the comment later on.
Now include the file `fluidsystems/h2oair.hh` in the material folder, and set a property `FluidSystem` with the appropriate type, `Dumux::H2OAirFluidSystem<TypeTag>`. However, this rather complicated fluidsystem uses tabularized fluid data, which need to be initialized (i.e. the tables need to be filled with values) in the constructor body of the current problem by adding `GET_PROP_TYPE(TypeTag, FluidSystem)::init();`. Remember that the constructor function always has the same name as the respective class, i.e. `TutorialProblemSequential(..)`.
To avoid the initialization, use the simpler version of water `Dumux::SimpleH2O` or a non-tabulated version `Dumux::H2O`. This can be done by setting the property `Components` type `H2O`, as is done in all the test problems of the sequential 2p2c model.
The density of the gas is magnitudes smaller than that of oil, so please decrease the outflow rate to $q_n = 3 \times 10^{-4} \left[\frac{\text{kg}}{\text{m}^2\text{s}}\right]$. Also reduce the simulation duration to 2e4 seconds.
Please reverse the changes of this example, as we still use bulk phases and hence do not need such an extensive fluid system.

e) **Heterogeneities**
Set up a model domain with the soil properties given in figure 3.8. Adjust the boundary conditions

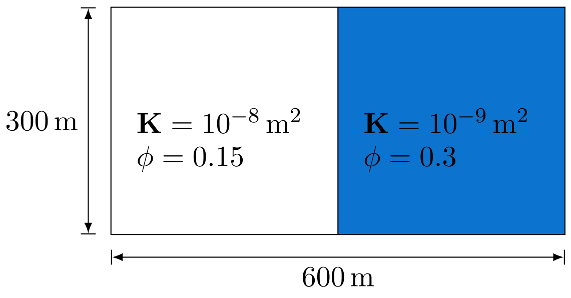

300 m
$\mathbf{K} = 10^{-8}\,\mathrm{m}^2$
$\phi = 0.15$
$\mathbf{K} = 10^{-9}\,\mathrm{m}^2$
$\phi = 0.3$
600 m

Figure 3.8: Exercise 1d: Set-up of a model domain a heterogeneity. $\Delta x = \Delta y = 20\,\mathrm{m}$.

so that water is again flowing from left to right. When does the front cross the material border? In paraview, the option *View* → *Animation View* is nice to get a rough feeling of the time-step sizes.

Exercise 2

For this exercise you should create a new problem file analogous to the file `tutorialproblem_sequential.hh` (e.g. with the name `ex2_tutorialproblem_sequential.hh` and new spatial parameters just like `tutorialspatialparams_sequential.hh`. These files need to be included in the file `tutorial_sequential.cc`.

Each new files should contain the definition of a new class with a name that relates to the file name, such as `Ex2TutorialProblemSequential`. Make sure that you also adjust the guardian macros in lines 24 and 25 in the header files (e.g. change `DUMUX_TUTORIALPROBLEM_SEQUENTIAL_HH` to `DUMUX_EX2_TUTORIALPROBLEM_SEQUENTIAL_HH`). Beside also adjusting the guardian macros, the new problem file should define and use a new type tag for the problem as well as a new problem class e.g. `Ex2TutorialProblemSequential`. Make sure to assign your newly defined spatial parameter class to the `SpatialParams` property for the new type tag.

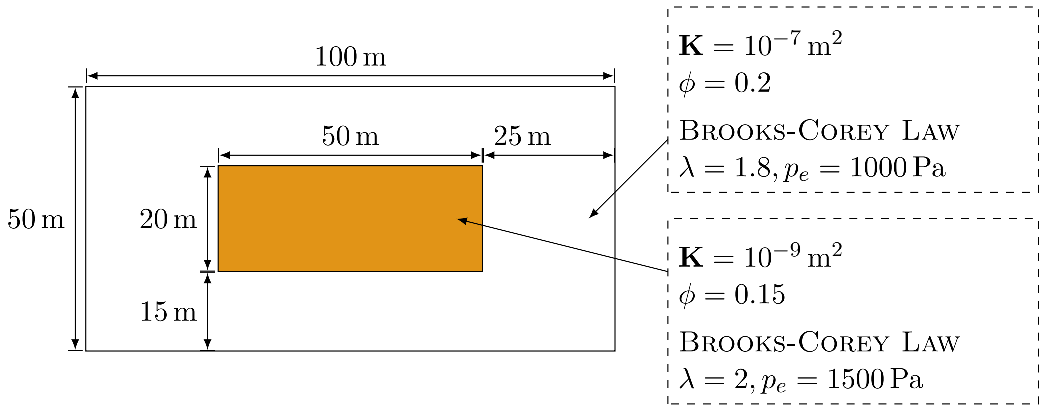
After this, change the domain size (parameter input file) to match the domain described by figure 3.9. Adapt the problem class so that the boundary conditions are consistent with figure 3.10. Initially, the domain is fully saturated with water and the pressure is $p_w = 2 \times 10^5\,\mathrm{Pa}$. Oil infiltrates from the left side. Create a grid with 20 cells in x-direction and 10 cells in y-direction. The simulation time should be set to $1e6\,\mathrm{s}$.

Now include your new problem file in the main file and replace the `TutorialProblemSequential` type tag by the one you've created and compile the program.

- What happens if you increase the resolution of the grid? Hint: Paraview can visualize the time-steps via the "Animation View" (to be enabled unter the button *View*).
- Set the CFL-factor to 1 and investigate the saturation: Is the value range reasonable?
- Further increase the CFL-factor to 2 and investigate the saturation.

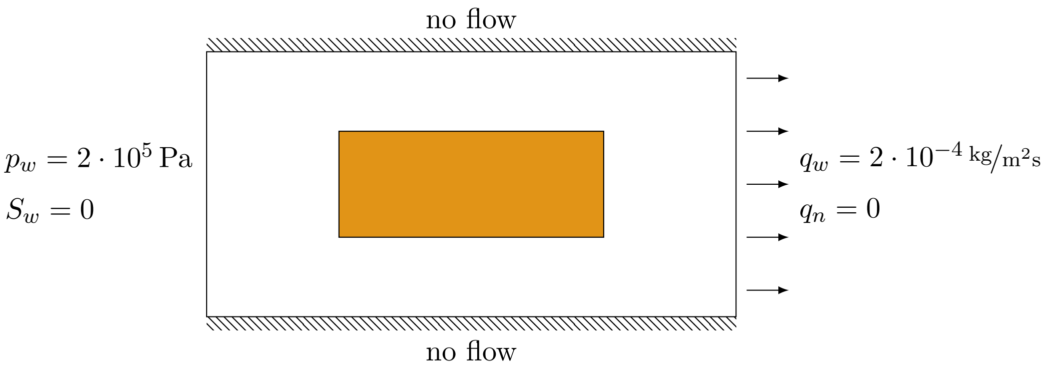
Exercise 3: Parameter file input

As you have experienced, compilation takes quite some time. Therefore, DuMu$^\mathrm{x}$ provides a simple method to read in parameters (such as simulation end time or modelling parameters) via `Paramter`



100 m
50 m
25 m
50 m
20 m
15 m
$\mathbf{K} = 10^{-7}\,\mathrm{m}^2$
$\phi = 0.2$
Brooks-Corey Law
$\lambda = 1.8, p_e = 1000\,\mathrm{Pa}$
$\mathbf{K} = 10^{-9}\,\mathrm{m}^2$
$\phi = 0.15$
Brooks-Corey Law
$\lambda = 2, p_e = 1500\,\mathrm{Pa}$

Figure 3.9: Set-up of the model domain and the soil parameters



no flow
$p_w = 2 \cdot 10^5\,\mathrm{Pa}$
$S_w = 0$
$q_w = 2 \cdot 10^{-4}\,\mathrm{kg/m^2s}$
$q_n = 0$
no flow

Figure 3.10: Boundary Conditions

`Input Files`. The tests in the Test-folder `/test/` already use this system.
If you look at the Application in `/test/porousmediumflow/2p/sequential/`, you see that the main file looks rather empty: The parameter file `test_box2p.input` is read by a standard start procedure, which is called in the main function. This should be adapted for your problem at hand. The program run has to be called with the parameter file as argument. As this is a basic DuMu^x feature, the procedure is the equivalent in the sequential as in the box models. In the code, parameters can be read via the macro `GET_RUNTIME_PARAM(TypeTag, Scalar, MyWonderfulGroup.MyWonderfulParameter);`. In `test_2p`, `MyWonderfulGroup` is the group `SpatialParams` - any type of groups is applicable, if the group definition in the parameter file is enclosed in square brackets. The parameters are then listed thereafter. Try and use as much parameters as possible via the input file, such as lens dimension, grid resolution, soil properties etc. In addition, certain parameters that are specific to the model, such as the `CFL`-factor, can be assigned in the parameter file without any further action.

Exercise 4

Create a new file for benzene called `benzene.hh` and implement a new fluid system. (You may get a hint by looking at existing fluid systems in the directory `/dumux/material/fluidsystems`.)

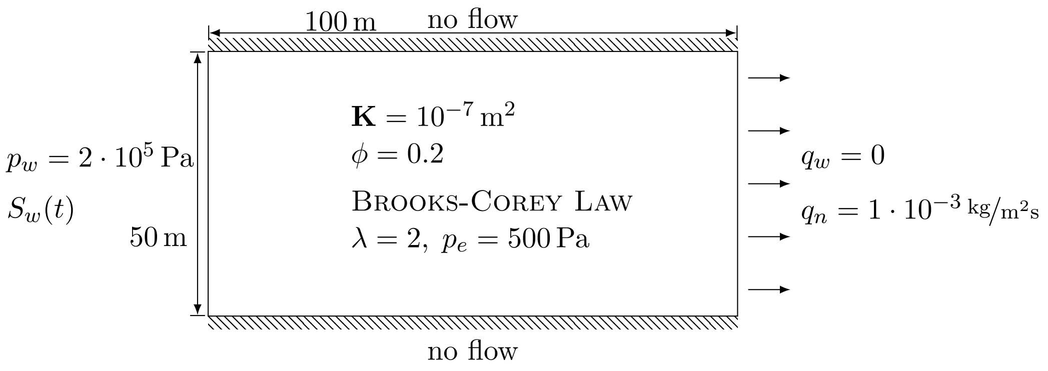
Use benzene as a new fluid and run the model of Exercise 2 with water and benzene. Benzene has a density of 889.51 kg/m^3 and a viscosity of 0.00112 Pa s.

Exercise 5: Time Dependent Boundary Conditions

In this exercise we want to investigate the influence of time dependent boundary conditions. For this, redo the steps of exercise 2 and create a new problem and spatial parameters file.

After this, change the run-time parameters so that they match the domain described by figure 3.11. Adapt the problem class so that the boundary conditions are consistent with figure 3.12. Here you can see the time dependence of the wetting saturation, where water infiltrates only during 10^5 s and $4 \cdot 10^5$ s. To implement these time dependencies you need the actual time $t_{n+1} = t_n + \Delta t$ and the endtime of the simulation. For this you can use the methods `this->timeManager().time()`, `this->timeManager().timeStepSize()` and `this->timeManager().endTime()`.

Initially, the domain is fully saturated with oil and the pressure is $p_w = 2 \times 10^5$ Pa. Water infiltrates from the left side. Create a grid with 100 cells in x-direction and 10 cells in y-direction. The simulation time should be set to $5 \cdot 10^5$ s with an initial time-step size of 10 s. To avoid too big time-step sizes you should set the parameter `MaxTimeStepSize` for the group `TimeManager` (in your input file) to 100 s. You should only create output files every 100^{th} time-step (see exercise 1a). Then, you can compile the program.



100 m
no flow
$\mathbf{K} = 10^{-7}\,\mathrm{m}^2$
$\phi = 0.2$
Brooks-Corey Law
$\lambda = 2,\ p_e = 500\,\mathrm{Pa}$
$p_w = 2 \cdot 10^5$ Pa
$S_w(t)$
50 m
$q_w = 0$
$q_n = 1 \cdot 10^{-3}\,\mathrm{kg/m^2s}$
no flow

Figure 3.11: Set-up of the model domain and the soil parameters

- Open paraview and plot the values of S_w at time $5 \cdot 10^5$ s over the $x-$axis. (`Filter->Data Analysis->Plot Over Line`)
- What happens without any time-step restriction?

Exercise 6

If both the implicit and the sequential tutorial are completed, one should have noticed that the function arguments in the problem function differ slighty, as the numerical models differ. However, both are functions that depend on space, so both models can also work with functions based ond `...AtPos(GlobalPosition & globalPos)`, no matter if we model implicit or sequential. Try to formulate a spatial parameters file that works with both problems, the implicit and the sequential. Therein, only use functions at the position.

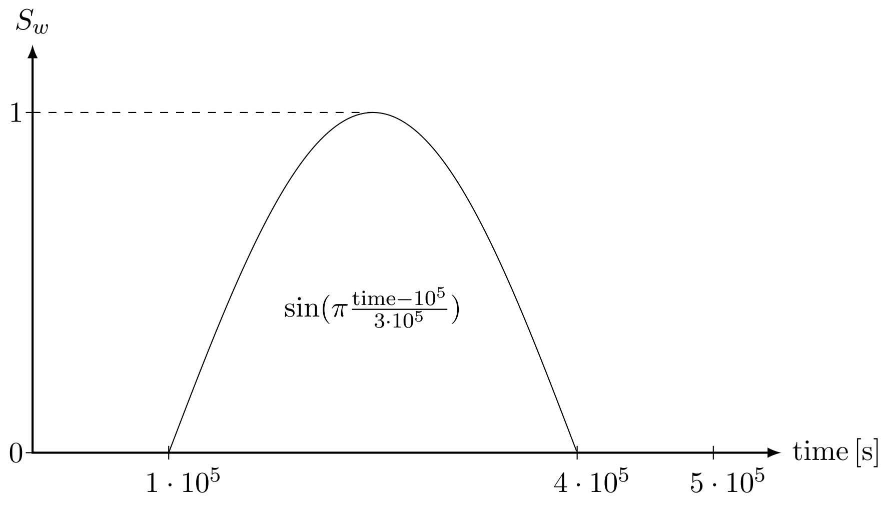


Figure 3.12: Time Dependent Boundary Conditions

3.3 Further Practice

If there is a need for further practice, we refer here to the test problems, that are already implemented in DuMu^x. Several examples for implicit and sequential models can be found in the `test`-directory. An overview over the available tests cases can be found on the class documentation `http://www.dumux.org/documentation.php`. There you also find a *feature-list* for the individual tests.

Another possibility to gain more experience with DuMu^x is the `dumux-lecture` module, where a number of DuMu^x applications are provided and explained. It is structured according to the use of these applications in different lecture classes at the Department of Hydromechanics and Modelling of Hydrosystems. The majority of applications belongs accordingly to the course in Multiphase Modelling (mm), while there are also some basic examples from the courses Environmental Fluid Mechanics (efm) and Modelling of Hydrosystems (mhs). These applications contain were primarily designed to enhance the understanding of conceptualizing the governing physical processes and their implementation in a numerical simulator. Different aspects of modelling multi-phase multi-component flow and transport processes are approached. In the focus are 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, etc. You can find, for example, a comparison of different two-phase flow problems considering in the more simple approach two immiscible fluids while components in both phases with interphase mass transfer are considered in the more complex case. All the scenarios and their physical background are explained in additional .tex-files, which are provided in sub-directories names *description*. The following test cases are contained:

- `Buckley-Leverett Problem - Classical porous media flow show case`
- `CO`$_2$ `plume - The influence of the Gravitational Number`
- `Column Xylene - A VEGAS experiment`
- `Convective Mixing - Related to CO`$_2$ `storage`

- `Fuel Cell`
- `Heatpipe - A show case for two-phase two-component flow with heat fluxes`
- `Heavy Oil - SAGD (steam assisted gravity drainage)`
- `Henry Problem - A show case related to salt water intrusion`
- `McWorther Problem - Classical porous media flow show case`
- `NAPL Infiltration`
- `Remediation Scenarios - For NAPL contaminated unsaturated soils`
- `Groundwater - Simple groundwater flow case for MHS lecture`
- `Different single/two-phase single/two-component problems - Example from lecture EFM`

Dumux-lecture can be obtained as follows:

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

4 Overview and Infrastructure

This chapter provides an overview of the general structure in DuMu^x 4.1 and gives help for basic work with DuMu^x (4.2,4.3,4.4,4.5,4.6). Further it presents useful external tools 4.7 and basic concepts 4.8.

4.1 Directory Structure

DuMu^x has the following folder structure, which is similar to other DUNE modules.

- `bin`: binaries, e.g. used for the automatic testing, postprocessing, installation
- `cmake`: the configuration options for building DuMu^x
- `doc`: files necessary for the Doxygen documentation and this handbook, and various logos
- `dumux`: the main folder, containing the source files, see 4.1 for a visualized structure. For more information on the models have a look at the Doxygen documentation.
- `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 source `*.cc`, the problem definition `*problem.hh`, and an input file `*.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.
- `tutorial`: contains the tutorials described in Chapter 3.

4.2 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 and add 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

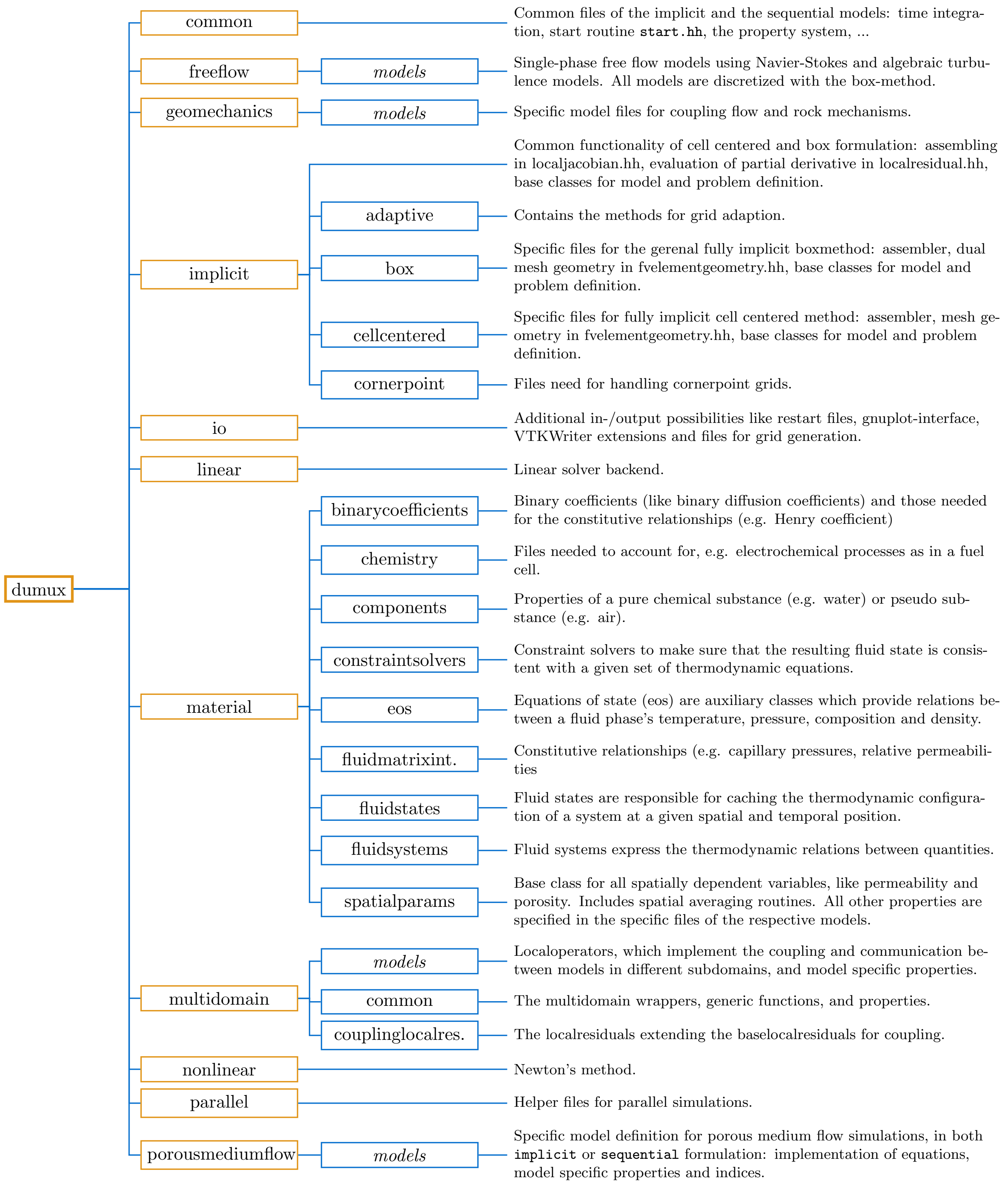

dumux
common
Common files of the implicit and the sequential models: time integration, start routine start.hh, the property system, ...
freeflow
models
Single-phase free flow models using Navier-Stokes and algebraic turbulence models. All models are discretized with the box-method.
geomechanics
models
Specific model files for coupling flow and rock mechanisms.
implicit
Common functionality of cell centered and box formulation: assembling in localjacobian.hh, evaluation of partial derivative in localresidual.hh, base classes for model and problem definition.
adaptive
Contains the methods for grid adaption.
box
Specific files for the gerenal fully implicit boxmethod: assembler, dual mesh geometry in fvelementgeometry.hh, base classes for model and problem definition.
cellcentered
Specific files for fully implicit cell centered method: assembler, mesh geometry in fvelementgeometry.hh, base classes for model and problem definition.
cornerpoint
Files need for handling cornerpoint grids.
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. electrochemical 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 system at a given spatial and temporal position.
fluidsystems
Fluid systems express the thermodynamic relations between quantities.
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
models
Localoperators, which implement the coupling and communication between models in different subdomains, and model specific properties.
common
The multidomain wrappers, generic functions, and properties.
couplinglocalres.
The localresiduals extending the baselocalresiduals for coupling.
nonlinear
Newton's method.
parallel
Helper files for parallel simulations.
porousmediumflow
models
Specific model definition for porous medium flow simulations, in both implicit or sequential formulation: implementation of equations, model specific properties and indices.

Figure 4.1: Structure of the directory `dumux` containing the DuMu$^\mathrm{x}$ source files.

Adding new Test Programs To add a test use the `add_dumux_test` macro. The command has four arguments:

1) name of test (has to be unique)
2) name of executable
3) source file (*.cc)
4) command to be executed as test - either the executable or a some helper script with arguments

4.3 Parameters in DuMu^x

Simulation parameters can be parsed to the program via a parameter file or the command line. A list of all available parameters is provided in the Doxygen documentation of the file `parameterfile`, which is accessible via `Modules -> Parameters`.

After having run the example application from section 2.2 you will get the following output at the end of the simulation run [1]:

```
# Run-time specified parameters:
[ Grid ]
File = "./grids/test_2p.dgf"
[ Implicit ]
EnableJacobianRecycling = "1"
EnablePartialReassemble = "1"
[ Problem ]
Name = "lensbox"
[ SpatialParams ]
LensLowerLeftX = "1.0"
LensLowerLeftY = "2.0"
LensUpperRightX = "4.0"
LensUpperRightY = "3.0"
[ TimeManager ]
DtInitial = "250"
TEnd = "3000"
# DEPRECATED run-time specified parameters:
PrintParameters = "1"
# Replace by:
[ TimeManager ]
PrintParameters = "1"
# Compile-time specified parameters:
[ Implicit ]
EnableHints = "0"
MassUpwindWeight = "1"
MaxTimeStepDivisions = "10"
MobilityUpwindWeight = "1"
NumericDifferenceMethod = "1"
```

[1] If you did not get the output, restart the application the following way: `./test_box2p -PrintParameters true`, this will print the parameters once your simulation is finished

```
UseTwoPointFlux = "0"
[ LinearSolver ]
MaxIterations = "250"
PreconditionerRelaxation = "1"
ResidualReduction = "1e-06"
Verbosity = "0"
[ Newton ]
WriteConvergence = "0"
[ Problem ]
EnableGravity = "1"
[ TimeManager ]
MaxTimeStepSize = "1.79769e+308"
[ Vtk ]
AddVelocity = "0"
# UNUSED parameters:
ImportantVariable = "1"
```

A number of things can be learned:

- *run-time* parameters can be changed without re-compiling
- *deprecated run-time* parameters will be removed in the next release
- *compile-time* parameters cannot be overwritten by the input file
- *unused* are not used by the simulation (maybe typo or wrong group)

All applications have a help message which you can read by giving `--help` as a command line argument to the application.

For further details, please have a look for `Dune::ParameterTree` in the DUNE documentation.

4.4 Restart DuMu^x Simulations

You can restart the simulation from a specific point in time or extend the simulation beyond the originally end of simulation. What you need is a `*.drs` file (which contains the all necessary restart information. Then you can simply restart a simulation via

```
./test_program -TimeManager.Restart RESTART_TIME
```

To test restart behavior, use the `test_box1p2cni` problem in the `test/implicit/1p2c` folder. You get the `RESTART_TIME` from the name of your `.drs` file. Restarting will only work when the exact time from an existing restart file is given. If you need more restart files, you can change the frequency by including the function into your problem:

```
// Writes a restart file every 5th time step
bool shouldWriteRestartFile() const
{
  return (this->timeManager().timeStepIndex() % 5 == 0);
}
```

4.5 Coding Guidelines

Writing code in a readable manner is very important, especially for future code developers (e.g. for adding features, debugging, etc.). This section is inspired by the DUNE coding guidelines `http://www.dune-project.org/doc/devel/codingstyle.html`, which is strongly recommended.

Documentation: Please document freely what each part of your code does. All comments/ documentation is in **English**. We proclaim the Doc-Me Dogma, which means whatever you do, please document it least with

```
/*! \todo Please doc me! */
```

That way at least the absence of documentation is documented, and it is easier to get rid of it systematically. Please comment **all**:

- Method Parameters (in / out)
- Method parameters which are not self-explanatory should always have a meaningful comment at calling sites. Example:

  ```
      localResidual.eval(/*includeBoundaries=*/true);
  ```

- Template Parameters
- Return Values
- Exceptions thrown by a method

Naming: To avoid duplicated types or variables and for a better understanding and usability of the code we have the following naming conventions:

- **Variables/Functions . . .**
  - start in *lower* case and contain letters.
  - *CamelCase*: if the name consists of several words, then the first letter of a new word is capital.
  - *Self-Explaining*: in general abbreviations should be avoided (write saturation in stead of S)
  - *Abbreviations*: If and only if a single letter that represents an abbreviation or index is followed by a single letter (abbreviation or index), CamelCase is **not** used. An inner-word underscore is only permitted if it symbolizes a fraction line. Afterwards, we continue with lower case, i.e. the common rules apply to both enumerator and denominator. Examples:
    - `pw` but `pressureW` → because “pressure” is a word.
    - `srnw` but `sReg` → because “reg” is not an abbreviation of a single letter. “n” abbreviates “non”, “w” is “wetting”, so not CamelCase.
    - `pcgw` but `dTauDPi` → Both “Tau” and “Pi” are words, plus longer than a letter.
    - **But:** `CaCO3` The only exception: chemical formulas are written in their chemically correct way → `CaCO3`

- **Private Data Variables:** Names of private data variables end with an underscore and are the only variables that contain underscores.
- **Type names:** For type names, the same rules as for variables apply. The only difference is that the *first letter is capital.*
- **Files:** File names should consist of *lower case* letters exclusively. Header files get the suffix `.hh`, implementation files the suffix `.cc`
- **The Exclusive-Access Macro:** Every header file traditionally begins with the definition of a preprocessor constant that is used to make sure that each header file is only included once. If your header file is called 'myheaderfile.hh', this constant should be DUMUX_MYHEADERFILE_HH.
- **Macros:** The use of preprocessor macros is strongly discouraged. If you have to use them for whatever reason, please use capital letters only.

4.6 Developing DuMu^x

4.6.1 Communicate with DuMu^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 a *Git* account already) and to constribute 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^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^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.

4.6.2 Tips and Tricks

DuMu^x users and developers at the LH2 are also referred to the internal Wiki for more information.

Option Files optim.opts and debug.opts DUNE and DuMu^x are built with the help of `dunecontrol`, as explained on page 5. The options needed to be specified for that are provided using option files like `debug.opts` and `optim.opts`. These two compile DUNE and DuMu^x either for debugging or for fast simulation. Programs compiled with optimization options can lead to a speedup of factor up to ten! In contrast programs that are compiled with optimization can hardly be debugged. You can modify the files and change the compiler, the name of the build director, add third-party dependencies, add additional compiler flags, ...

```
BUILDDIR=build-clang
CMAKE_FLAGS="\
  -DCMAKE_C_COMPILER=/usr/bin/clang \
  -DCMAKE_CXX_COMPILER=/usr/bin/clang++ \
  -DUG_DIR=./externals/ug-3.12.1"
```

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 the use of an additional module. Then you have to configure and build only DUNE-grid and DuMu^x 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
```

See 2.3.2 if you need to apply patches to DuMu^x or DUNE.

File Name and Line Number by Predefined Macro If you want to 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:

```
// define the minimal debug level somewhere in your code
#define DUNE_MINIMAL_DEBUG_LEVEL 4
Dune::dgrave << "message"; // will be printed
Dune::dwarn << "message"; // will be printed
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.

Naming conventions General guidelines for naming conventions are specified in Section 4.5. However, in order to avoid ambiguity a list of proposed names for variables, types, functions etc is provided where users and mainly DuMu^x developers can refer for standards (check `dumux-devel/doc/naminglist/naming-conventions.odt`).

4.7 External Tools

4.7.1 Eclipse

There is an Eclipse style file which can be used for DuMu^x.

a) open in eclipse: `Window` → `Preferences` → `C/C++` → `Code Style` → `Formatter`

b) press the `Import` button

c) choose the file `eclipse_profile.xml` from your dumux-devel directory

d) make sure that now DuMu^x is chosen in `Select a profile`

4.7.2 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 comitted 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

4.7.3 Gnuplot

A gnuplot interface is available to plot or visualize results during a simulation run. This is achieved with the help of the class provided in `io/gnuplotinterface.hh`. Have a look at tests including this header for examples how to use this interface.

4.7.4 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.sh` from your DuMu^{x} root directory or donwload, 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).

4.7.5 Kate

For kate there is syntax highlighting style for DuMu^{x} input files. Simply copy the file `dumux-devel/dumux-InputFiles.xml` to the `syntax` folder in your kate configuration directory (e.g. `HOME/.kde4/share/apps-/katepart/syntax/dumuxInputFiles.xml`).

4.7.6 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` $\rightarrow$ `Add new macro`.

Guide: Since ParaView 4.3.1 The ParaView Guide is partly available for free download, see `http://www.paraview.org/documentation/`. It corresponds to the ParaView book, only without three application chapters. Attention, its size is 180 MiB.

4.8 Assembling the linear system

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

4.8.1 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:

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

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

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

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 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.

4.8.2 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 blocking structure. Each block contains the degrees of freedom (also called variable or unknown) for a sub-control volume. The equation index is used to order of the degrees of freedom. For each sub-control volume we have one block. The mapper is used to order the blocks.

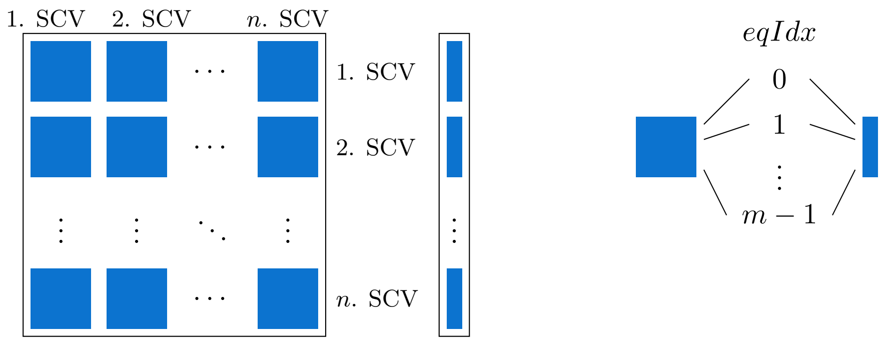


Figure 4.2: Structure of matrix and vector, left blocking structure, right within block

Accessing entries follows this structure. You can access the pressure value in the third sub-control volume in a vector sol with sol[2][pressureIdx].

5 Advanced DuMu^x – Detailed Instructions

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

5.1 Models

Here the basic definitions, the general models concept, and a list of models available in DuMu^x are given.

5.1.1 Basic Definitions and Assumptions

The basic definitions and assumptions are made, using the example of a three-phase three-component system water-NAPL-gas [7]. The modification for other multicomponent systems is straightforward and can be found, e. g., in [5, 1].

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. In this work, we assume 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 the organic contaminant (c) (see Fig. 5.1).

Phases: For compositional multi-phase models, *phases* are not only matter of a single chemical substance. Instead, their composition in general includes several species/components. For mass transfer, the component behavior is quite different from the phase behavior.

Equilibrium: For the non-isothermal, multi-phase, multi-component processes in porous media we state that 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 $\alpha \in \{\mathrm{w}, \mathrm{n}, \mathrm{g}\}$ refers to the phase, while the superscript $\kappa \in \{\mathrm{w}, \mathrm{a}, \mathrm{c}\}$ refers to the component.

5.1.2 Available Models

We distinguish fully-implicit and sequential models. A list of all available models can be found in the Doxygen documentation at `http://www.dumux.org/doxygen-stable/html-2.11/modules.php`. The documentation includes a detailed description for every model.

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
$\varrho_{\mathrm{mol},\alpha}$	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
$k_{\mathrm{r}\alpha}$	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 5.1: Notation list for most of the variables and indices used in DuMu^x.

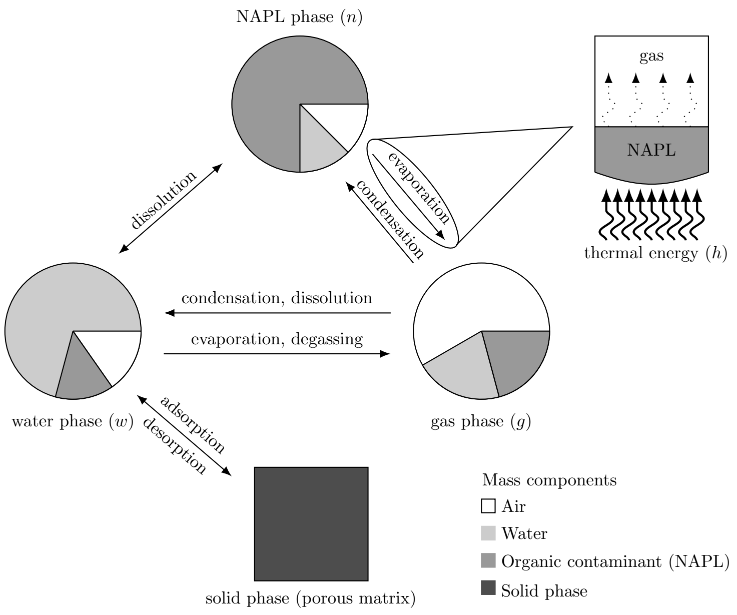


Figure 5.1: Mass and energy transfer between the phases

Fully-Implicit Models

The fully-implicit models are using the box or the cell-centered finite volume method as described in section 5.2.1 and 5.2.2 for spatial and the implicit Euler method as temporal discretization. The fully-implicit models are located in subdirectories of `dumux/freeflow`, `dumux/geomechanics`, and `dumux/porousmediumflow`.

Grid adaption is available for both discretization schemes, box and cc. The `adaptionhelper`-method has to be adjusted to the respective model. Note that the current implementation only ensures mass conservation for incompressible fluids. In general, the spatial parameters, especially the porosity, have to be assigned on the coarsest level of discretization.

Sequential Models

The basic idea of the sequential models 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 a fully implicit model, the sequential structure allows the use of different discretization methods for the different equations. The standard method used in the sequential models 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 models is provided for two dimensions. The sequential models are located in subdirectories of `dumux/porousmediumflow`.

5.2 Spatial Discretization Schemes

For the implicit models there are two spatial discretization schemes (box and Cell Centered Finite Volume Method) available which are shortly introduced in this subsection.

5.2.1 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 G 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 5.2a).

The FE mesh divides the box B_i into subcontrolvolumes (scv's) b_i^k (see Figure 5.2b). Figure 5.2c 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 $\mathbf{n}_{ij}^k$ are also to be defined (see Figure 5.2c).

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

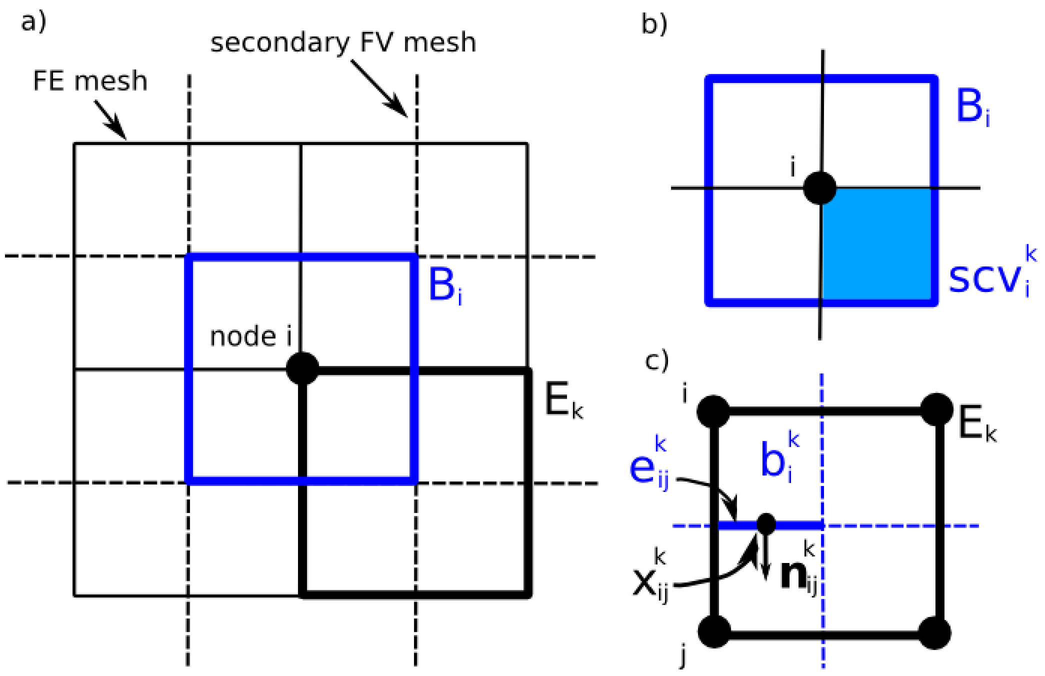


Figure 5.2: Discretization of the box method

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:

$$f(\tilde{u}(x_{ij}^k)) \cdot \mathbf{n}_{ij}^k \, |e_{ij}^k| \qquad \text{with} \qquad \tilde{u}(x_{ij}^k) = \sum_i N_i(x_{ij}^k) \cdot \hat{u}_i. \tag{5.1}$$

In the following, the discretization of the balance equation is going to be derived. From the Reynolds transport theorem follows the general balance equation:

$$\underbrace{\int_G \frac{\partial}{\partial t} u \, dG}_{1} + \underbrace{\int_{\partial G} (\mathbf{v}u + \mathbf{w}) \cdot \mathbf{n} \, d\Gamma}_{2} = \underbrace{\int_G q \, dG}_{3} \tag{5.2}$$

$$f(u) = \int_G \frac{\partial u}{\partial t} \, dG + \int_G \nabla \cdot \underbrace{[\mathbf{v}u + \mathbf{w}(u)]}_{F(u)} \, dG - \int_G q \, dG = 0 \tag{5.3}$$

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. G 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:

$$\tilde{p} = \sum_i N_i \hat{p}_i \tag{5.4}$$

$$\tilde{\mathbf{v}} = \sum_i N_i \hat{\mathbf{v}} \tag{5.5}$$

$$\tilde{x}^\kappa = \sum_i N_i \hat{x}^\kappa \tag{5.6}$$

$$\nabla\tilde{p} = \sum_i \nabla N_i \hat{p}_i \tag{5.7}$$

$$\nabla\tilde{\mathbf{v}} = \sum_i \nabla N_i \hat{\mathbf{v}} \tag{5.8}$$

$$\nabla\tilde{x}^\kappa = \sum_i \nabla N_i \hat{x}^\kappa . \tag{5.9}$$

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

$$f(u) = 0 \qquad \Rightarrow \qquad f(\tilde{u}) = \varepsilon \tag{5.10}$$

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,

$$\int_G W_j \cdot \varepsilon \overset{!}{=} 0 \qquad \text{with} \qquad \sum_j W_j = 1 \tag{5.11}$$

yields the following equation:

$$\int_G W_j \frac{\partial \tilde{u}}{\partial t}\, dG + \int_G W_j \cdot \left[\nabla \cdot F(\tilde{u})\right]\, dG - \int_G W_j \cdot q\, dG = \int_G W_j \cdot \varepsilon\, dG \overset{!}{=} 0. \tag{5.12}$$

Then, the chain rule and the Green-Gaussian integral theorem are applied.

$$\int_G W_j \frac{\partial \sum_i N_i \hat{u}_i}{\partial t}\, dG + \int_{\partial G} \left[W_j \cdot F(\tilde{u})\right] \cdot \mathbf{n}\, d\Gamma_G + \int_G \nabla W_j \cdot F(\tilde{u})\, dG - \int_G W_j \cdot q\, dG = 0 \tag{5.13}$$

A mass lumping technique is applied by assuming that the storage capacity is reduced to the nodes. This means that the integrals $M_{i,j} = \int_G W_j\, N_i\, dG$ are replaced by the mass lumping term $M^{lump}_{i,j}$ which is defined as:

$$M^{lump}_{i,j} = \begin{cases} \int_G W_j\, dG = \int_G N_i\, dG = V_i & i = j \\ 0 & i \neq j \end{cases} \tag{5.14}$$

where V_i is the volume of the FV box B_i associated with node i. The application of this assumption in combination with $\int_G W_j\, q\, dG = V_i\, q$ yields

$$V_i \frac{\partial \hat{u}_i}{\partial t} + \int_{\partial G} \left[W_j \cdot F(\tilde{u})\right] \cdot \mathbf{n}\, d\Gamma_G + \int_G \nabla W_j \cdot F(\tilde{u})\, dG - V_i \cdot q = 0\,. \tag{5.15}$$

Defining the weighting function W_j to be piecewisely constant over a control volume box B_i

$$W_j(x) = \begin{cases} 1 & x \in B_i \\ 0 & x \notin B_i \end{cases} \tag{5.16}$$

causes $\nabla W_j = 0$:

$$V_i \frac{\partial \hat{u}_i}{\partial t} + \int_{\partial B_i} \left[W_j \cdot F(\tilde{u})\right] \cdot \mathbf{n}\, d\Gamma_{B_i} - V_i \cdot q = 0. \tag{5.17}$$

The consideration of the time discretization and inserting $W_j = 1$ finally leads to the discretized form which will be applied to the mathematical flow and transport equations:

$$V_i \frac{\hat{u}_i^{n+1} - \hat{u}_i^n}{\Delta t} + \int_{\partial B_i} F(\tilde{u}^{n+1}) \cdot \mathbf{n} \, d\Gamma_{B_i} - V_i \, q^{n+1} = 0 \tag{5.18}$$

5.2.2 Cell Centered Finite Volume Method – A Short Introduction

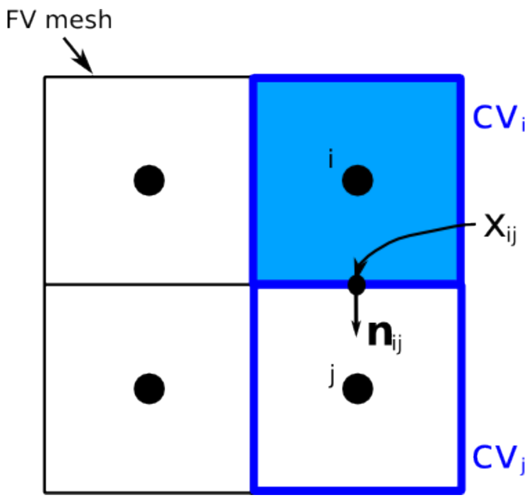


Figure 5.3: Discretization of the cell centered finite volume method

The cell centered finite volume method uses the elements of the grid as control volumes. For each control volume all discrete values are determined at the element/control volume center (see Figure 5.3). The mass or energy fluxes are evaluated at the integration points (x_{ij}), which are located at the midpoints of the control volume faces. This is a two point flux approximation since the flux between the element/control volume centers i and j is calculated only with information from these two points. In contrast the box method uses a multi-point flux approximation where all nodes of the element influence the flux between two specific nodes.
Neumann boundary conditions are applied at the boundary control volume faces and Dirichlet boundary conditions at the boundary control volumes.
The cell centered finite volume method is robust and mass conservative but should only be applied for structured grids (the control volume face normal vector (n_{ij}) should be parallel to the direction of the gradient between the two element/control volume centers).

5.3 Steps of a DuMu^x Simulation

This chapter is supposed to show 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 5.3.1 the structure of DuMu^x is shown from a *content* point of view. Section 5.3.2 however is written from the point of view of the *implementation*. The same coloration in the flowcharts of both sections refers to the same level of calculation. For keeping things simple, the program flow of a `2p` model is shown in section 5.3.2. There are extensive comments regarding the formating in the tex file: so feel free, to enhance this description.

5.3.1 Structure – by Content

This list shows the algorithmic outline of a typical DuMu^x run. Each item stands for a characteristic step within the modeling framework.

In Figure 5.4, the algorithmic representations of both approaches, the fully implicit and the sequential one are illustrated down to the element level.

```
1. main   2. time step   3. Newton   4. element

initialize
foreach time step
    prepare update
    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
```

```
1. main   2. time step   3. IMPES/C   4. element

initialize
foreach time step
    prepare update
    foreach IMPES/C step
        if grid is adaptive
            - calculate refinement indicator
            - mark elements, adapt the grid
            - map old solution to new grid
        - calculate flow field
        foreach element
            - calculate element stiffness matrix
            - assemble into global matrix
        endfor
        solve linear system
        - calculate transport
          (saturations, concentrations,...)
        foreach element
            -calculate update (explicitly)
            - adapt time step (CFL-like criterion)
        endfor
        - update old solution
        - postprocess (flash calculation, etc.)
    endfor
    - write result
endfor
finalize
```

Figure 5.4: Structure of a fully implicit (**left**) and a sequential (**right**) scheme in DuMu^x.

5.3.2 Structure – by Implementation

This section is supposed to help you in getting an idea how things are handled in DuMu^x and in which files things are written down. This is not intuitivly clear, therefore it is mentioned for each $\boxed{\text{step-stone}}$.
called by tells you from which file a function is accessed. **implemented in** tells you in which file the function is written down. The name of the function is set in `typewriter`. Being a function is indicated by round brackets `()` but only the function name is given and not the full signature (arguments...) . Comments regarding the events within one step-stone are set smaller.

`main() ⇒ Dumux::start<ProblemTypeTag>() ⇒ start_()`

`start_()` in: start.hh
start the simulation

→

`start_() ⇒ timeManager.init()`

`init()` in: timemanager.hh
initialization

→

`start_() ⇒ timeManager.run()`

`run()` in: timemanager.hh
time step management

⟶

`while(!finished)`
main → time step

`run() ⇒ problem->timeIntegration()`

`timeIntegration()` in: implicitproblem.hh
execute time integration scheme

⟶

define number of allowed NEWTON fails
(each halving dt)

`timeIntegration() ⇒ model->update()`

`update()` in: implicitmodel.hh
sth like numerical model

→

`update() ⇒ solver.execute()`

`execute()` in: newtonmethod.hh
applying NEWTON method
keeps track of things, catching errors

⟶

time step → Newton step
`while(ctl.newtonProceed()`
uLastIter = uCurrentIter(model.uCur())

`execute() ⇒ execute_() ⇒ jacobianAsm.assemble()`

`assemble()` in: implicitassembler.hh
linearize the problem:
add all element contributions to global JACOBIAN and global residual

→

`assemble() ⇒ asImp_().assemble_() ⇒ resetSystem_()`

`resetSystem_()` in: implicitassembler.hh
set r.h.s. (i.e. residual)
set JACOBIAN to zero

⟶

Newton step → element
`loop all elements`

`assemble() ⇒ asImp_().assemble_() ⇒ asImp_().assembleElement_()`

`assembleElement_()` in: e.g. boxassembler.hh
call local JACOBIAN and residual assembly

→

`assembleElement_() ⇒ model_().localJacobian().assemble()`

`assemble()` in: implicitlocaljacobian.hh
set curr. element, update element's fin.vol.geom.
reset local JACOBIAN to 0
update types of boundaries on this element

→

`assemble() ⇒ prevVolVars_.update(),curVolVars_.update()`

`update()` in: e.g. 2pvolumevariables.hh
call model (e.g. 2p)specific update of quantities defined for the volume:
variables for the *current* and *previous* timestep

`update() ⇒ completeFluidState()`

`completeFluidState()` in: e.g. 2pvolumevariables.hh
calculate all required fluid properties from the primary variables, here the fluid system does the real work:
calculates, saves, and provides: densities, etc.

→

`assemble() ⇒ localResidual().eval()⇒asImp_().eval()`

`eval()` in: e.g. implicitlocalresidual.hh
the element's local residual is calculated:
see the next two stepstones

→

`eval() ⇒ asImp_().evalFluxes_()`

`evalFluxes_()` in: e.g. boxlocalresidual.hh
evaluate the fluxes going into each finite volume, this is *model specific*

evalFluxes_() ⇒ this→asImp_().computeFlux()

`computeFlux()` in: e.g. 2plocalresidual.hh
this calculate the *model specific* fluxes (e.g. advective and diffusive) using the `FluxVariables`

→

eval() ⇒ asImp_().evalVolumeTerms_()

`evalVolumeTerms_()` in: implicitlocalresidual.hh
evaluate the *model specific* storage and source terms for each finite volume

→

eval() ⇒ asImp_().evalBoundary_()

`evalBoundary_()` in: implicitlocalresidual.hh
evaluate the *model specific* boundary conditions

assemble() ⇒ asImp_().evalPartialDerivative_()

`evalPartialDerivative_()` in: e.g. implicitlocaljacobian.hh
actually calculate the element's (local) JACOBIAN matrix a property chooses backward/central/foward differences. here: central differences

→

approximation of partial derivatives: numerical differentiation
add $\pm\epsilon$ solution, divide difference of residual by 2ϵ
all partial derivatives for the element from the local JACOBIAN matrix

`priVars[pvIdx]+=eps`
this is adding eps to the current solution
`curVolVars_[scvIdx].update(+eps)`
recalculate volume variables, having ϵ added
`localResidual().eval(+eps)`
calculate local residual for modified solution as before: involves
- **computeFlux**
- **computeStorage**
- **computeSource**

`store the residual()`
`repeat for priVars[pvIdx]-=eps`
`derivative is (residual(+eps) - residual(-eps))/2eps`

assemble_() ⇒ asImp_().assembleElement_()

`assembleElement_()` in: implicitassembler.hh
Residual of the current solution is now "numerically differentiated", for the element i.e. the local JACOBIAN matrix is calculated.

$\longrightarrow$
The contribution of a single element is done.
Now, it needs to be added to the global quantities:
Add to global residual and global JACOBIAN.

assemble_() ⇒ asImp_().assembleElement_()

`assembleElement_()` in: e.g. boxassembler.hh
Add to global residual.:
`resdidual_[globI+=`
`model_().globalJacobian().resdidual(i)]`

$\longrightarrow$
loop vertices
of an element

assemble_() ⇒ asImp_().assembleElement_()

`assembleElement_()` in: e.g. boxassembler.hh
Add to global residual:
`(*matrix_)[globI][globJ] +=`
`model_().localJacobian().mat(i,j)`

$\longrightarrow$
element → **Newton step**
Assembling of elements to global quantities is done.

execute_() ⇒ while(ctl.newtonProceed())

`newtonProceed()` in: newtoncontroller.hh
Print information.
Start/ stop timer.

$\longrightarrow$
set delta Vector to zero
(this is what is
solved for later)

`execute_() ⇒ ctl.newtonSolveLinear()`

`newtonSolveLinear()` in: newtoncontroller.hh
Catching errors.
Ask the linear solver to solve the system.
i.e.: give JACOBIAN(matrix), delta(x), r.h.s.(residual) to linear solver
$\nabla r(x^k) \cdot \Delta x^k = r(x^k)$
tricky: each NEWTON step solves a linear system of equations.

$\rightarrow$

`newtonSolveLinear() ⇒ int converged = linearSolver_.solve()`

`solve()` in: boxlinearsolver.hh
Solve the linear system with the chosen backend.

`execute_() ⇒ ctl.newtonUpdate()`

`newtonUpdate()` in: newtoncontroller.hh
We solved for the change in solution, but need the solution:
Calculate current (this iteration) solution
from last (iteration) solution and current (iteration) change in solution:
$x^{k+1} = x^k - \Delta x^k$ where $\Delta x^k = (\nabla r(x^k))^{-1} \cdot r(x^k)$

$\rightarrow$

`execute_() ⇒ ctl.newtonEndStep()`

`newtonEndStep()` in: newtoncontroller.hh
Increase counter for number of NEWTON steps.
Print info.

$\longrightarrow$
check whether to do another NEWTON iteration:
that is: check if the error is below tolerance or
maximum number of iterations was reached.

$\longrightarrow$
Newton step $\rightarrow$ **Time step**
NEWTON done
if failed $\rightsquigarrow$ halve timestep size, restart loop

`execute_() ⇒ ctl.newtonEnd()`

`newtonEnd()` in: newtoncontroller.hh
Tell the controller we are done

$\rightarrow$

`update() ⇒ asImp_().updateSuccessful()`

`updateSuccessful()` in: e.g. implicitmodel.hh
can be filled *model specific*

$\longrightarrow$
in while(!finished)

`run() ⇒ problem_->postTimeStep()`

`postTimeStep(),writeOutput()` in: implicitproblem.hh
Give the problem the chance to post-process the solution.

$\longrightarrow$
write output
uPrev $\leftarrow$ uCur
time += dt, timestepIdx++
deal with restart and episodes

`run()⇒setTimeStepSize(problem_->nextTimeStepSize(dt))`
`⇒nextTimeStepSize() ⇒ newtonCtl_.suggestTimestepSize()`

`suggestTimestepSize()` in: newtoncontroller.hh
Determine new time step size from number of NEWTON steps.

$\longrightarrow$
Time step $\rightarrow$ **main**
loop until simulation is finished

5.4 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.

5.4.1 Motivation and features

The DuMu^x property system was designed as an attempt to mitigate the problems of traits classes. It can be seen as a traits system which allows easy inheritance and any acyclic dependency of parameter definitions. Just like traits, the DuMu^x property system is a compile time mechanism, thus there is no run-time performance penalty associated with it.

In the context of the DuMu^x property system, a property is an arbitrary class body which may contain type definitions, values and methods. Each property has a so-called *property tag* which labels its name.

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* and *introspection*. 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. The term introspection denotes the ability to generate diagnostic messages which can be used to find out where a certain property was defined and how it was inherited.

5.4.2 How-to

All source files which use the property system should include the header file `dumux/common/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 using

```
NEW_TYPE_TAG(NewTypeTagName, INHERITS_FROM(BaseTagName1, BaseTagName2, ...));
```

where the `INHERITS_FROM` part is optional. To avoid inconsistencies in the hierarchy, each type tag may be defined only once for a program.

Example:

```
namespace Dumux {
namespace Properties {
NEW_TYPE_TAG(MyBaseTypeTag1);
NEW_TYPE_TAG(MyBaseTypeTag2);

NEW_TYPE_TAG(MyDerivedTypeTag, INHERITS_FROM(MyBaseTypeTag1, MyBaseTypeTag2));
}}
```

Declaring Property Tags

New property tags, i.e. labels for properties, are declared using

```
NEW_PROP_TAG(NewPropTagName);
```

A property tag can be declared arbitrarily often, in fact it is recommended that all properties are declared in each file where they are used.

Example:

```
namespace Dumux {
namespace Properties {
NEW_PROP_TAG(MyPropertyTag);
}}
```

Defining Properties

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

```
SET_PROP(TypeTagName, PropertyTagName)
{
  // arbitrary body of a struct
};
```

For each program, a property itself can be declared at most once, although properties may be overwritten for derived type tags.

Also, the following convenience macros are available to define simple properties:

```
SET_TYPE_PROP(TypeTagName, PropertyTagName, type);
SET_BOOL_PROP(TypeTagName, PropertyTagName, booleanValue);
SET_INT_PROP(TypeTagName, PropertyTagName, integerValue);
SET_SCALAR_PROP(TypeTagName, PropertyTagName, floatingPointValue);
```

Example:

```
namespace Dumux {
namespace Properties {
NEW_TYPE_TAG(MyTypeTag);

NEW_PROP_TAG(MyCustomProperty);
NEW_PROP_TAG(MyType);

NEW_PROP_TAG(MyBoolValue);
NEW_PROP_TAG(MyIntValue);
NEW_PROP_TAG(MyScalarValue);

SET_PROP(MyTypeTag, MyCustomProperty)
{
  static void print() { std::cout << "Hello, World!\n"; }
};
SET_TYPE_PROP(MyTypeTag, MyType, unsigned int);

SET_BOOL_PROP(MyTypeTag, MyBoolValue, true);
SET_INT_PROP(MyTypeTag, MyIntValue, 12345);
SET_SCALAR_PROP(MyTypeTag, MyScalarValue, 12345.67890);
}}
```

Un-setting Properties

Sometimes an inherited properties do not make sense for a certain node in the type tag hierarchy. These properties can be explicitly un-set using

```
UNSET_PROP(TypeTagName, PropertyTagName);
```

The un-set property can not be set for the same type tag, but of course derived type tags may set it again.

Example:

```
namespace Dumux {
namespace Properties {
NEW_TYPE_TAG(BaseTypeTag);
NEW_TYPE_TAG(DerivedTypeTag, INHERITS_FROM(BaseTypeTag));

NEW_PROP_TAG(TestProp);

SET_TYPE_PROP(BaseTypeTag, TestProp, int);
UNSET_PROP(DerivedTypeTag, TestProp);
// trying to access the 'TestProp' property for 'DerivedTypeTag'
// will trigger a compiler error!
}}
```

Converting Tag Names to Tag Types

For the C++ compiler, property and type tags are like ordinary types. Both can thus be used as template arguments. To convert a property tag name or a type tag name into the corresponding type, the macros `TTAG(TypeTagName)` and `PTAG(PropertyTagName)` ought to be used.

Retrieving Property Values

The value of a property can be retrieved using

```
GET_PROP(TypeTag, PropertyTag)
```

or using the convenience macros

```
GET_PROP_TYPE(TypeTag, PropertyTag)
GET_PROP_VALUE(TypeTag, PropertyTag)
```

The first convenience macro retrieves the type defined using `SET_TYPE_PROP` and is equivalent to

```
GET_PROP(TypeTag, PropertyTag)::type
```

while the second convenience macro retrieves the value of any property defined using one of the macros `SET_{INT,BOOL,SCALAR}_PROP` and is equivalent to

```
GET_PROP(TypeTag, PropertyTag)::value
```

Example:

```
template <TypeTag>
class MyClass {
  // retrieve the ::value attribute of the 'NumEq' property
  enum { numEq = GET_PROP(TypeTag, NumEq)::value };
  // retrieve the ::value attribute of the 'NumPhases' property using the convenience macro
  enum { numPhases = GET_PROP_VALUE(TypeTag, NumPhases) };

  // retrieve the ::type attribute of the 'Scalar' property
  typedef typename GET_PROP(TypeTag, Scalar)::type Scalar;
  // retrieve the ::type attribute of the 'Vector' property using the convenience macro
  typedef typename GET_PROP_TYPE(TypeTag, Vector) Vector;
};
```

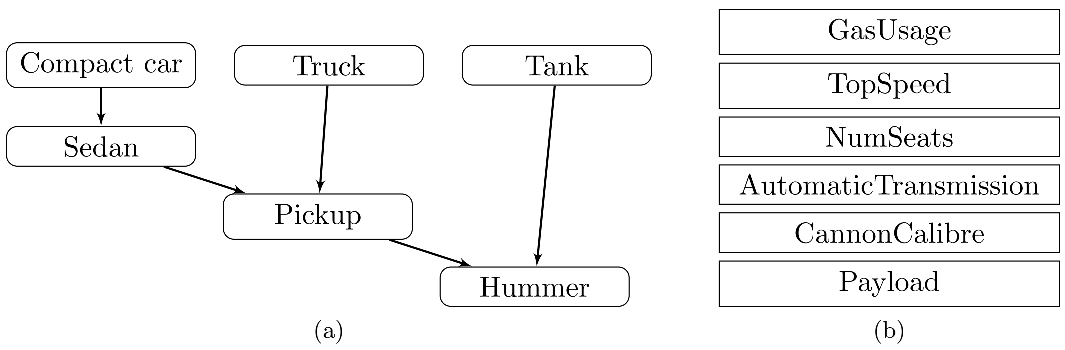


Figure 5.5: **(a)** A possible property inheritance graph for various kinds of cars. The lower nodes inherit from higher ones; Inherited properties from nodes on the right take precedence over the properties defined on the left. **(b)** Property names which make sense for at least one of the car types of (a).

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 keyword `TypeTag`. Inside property class bodies this can be used to retrieve other properties using the `GET_PROP` macros.

Example:

```
SET_PROP(MyModelTypeTag, Vector)
{
private: typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
public: typedef std::vector<Scalar> type;
};
```

5.4.3 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 5.5a.

Using the DuMu^x property system, this inheritance hierarchy is defined by:

```
#include <dumux/common/propertysystem.hh>
#include <iostream>

namespace Dumux {
namespace Properties {
NEW_TYPE_TAG(CompactCar);
NEW_TYPE_TAG(Truck);
NEW_TYPE_TAG(Tank);
NEW_TYPE_TAG(Sedan, INHERITS_FROM(CompactCar));
NEW_TYPE_TAG(Pickup, INHERITS_FROM(Sedan, Truck));
NEW_TYPE_TAG(HummerH1, INHERITS_FROM(Pickup, Tank));
```

Figure 5.5b lists a few property names which make sense for at least one of the nodes of Figure 5.5a. These property names can be declared as follows:

```
NEW_PROP_TAG(TopSpeed); // [km/h]
NEW_PROP_TAG(NumSeats); // []
NEW_PROP_TAG(CanonCaliber); // [mm]
NEW_PROP_TAG(GasUsage); // [l/100km]
NEW_PROP_TAG(AutomaticTransmission); // true/false
NEW_PROP_TAG(Payload); // [t]
```

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 l/100km times 30, the number of seats is 5 and the gas usage is 4 l/100km.
- A truck is by law limited to 100 km/h top speed, the number of seats is 2, it uses 18 l/100km and has a cargo payload of 35 t.
- A tank exhibits a top speed of 60 km/h, uses 65 l/100km and features a 120 mm diameter canon
- A sedan has a gas usage of 7 l/100km, 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 km/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

```
SET_INT_PROP(CompactCar, TopSpeed, GET_PROP_VALUE(TypeTag, GasUsage) * 30);
SET_INT_PROP(CompactCar, NumSeats, 5);
SET_INT_PROP(CompactCar, GasUsage, 4);

SET_INT_PROP(Truck, TopSpeed, 100);
SET_INT_PROP(Truck, NumSeats, 2);
SET_INT_PROP(Truck, GasUsage, 18);
SET_INT_PROP(Truck, Payload, 35);

SET_INT_PROP(Tank, TopSpeed, 60);
SET_INT_PROP(Tank, GasUsage, 65);
SET_INT_PROP(Tank, CanonCaliber, 120);

SET_INT_PROP(Sedan, GasUsage, 7);
SET_BOOL_PROP(Sedan, AutomaticTransmission, true);

SET_INT_PROP(Pickup, TopSpeed, 120);
SET_INT_PROP(Pickup, Payload, 5);

SET_INT_PROP(HummerH1, TopSpeed, GET_PROP_VALUE(TTAG(Pickup), TopSpeed));
```

At this point, the Hummer-H1 has a 120 mm canon which it inherited from its military ancestor. It can be removed by

```
UNSET_PROP(HummerH1, CanonCaliber);

}} // close namespaces
```

Now property values can be retrieved and some diagnostic messages can be generated. For example

```
int main()
{
    std::cout << "top speed of sedan: " << GET_PROP_VALUE(TTAG(Sedan), TopSpeed) << "\n";
    std::cout << "top speed of truck: " << GET_PROP_VALUE(TTAG(Truck), TopSpeed) << "\n";

    std::cout << PROP_DIAGNOSTIC(TTAG(Sedan), TopSpeed);
    std::cout << PROP_DIAGNOSTIC(TTAG(HummerH1), CanonCaliber);

    Dumux::Properties::print<TTAG(Sedan)>();
}
```

will yield the following output:

```
$ top speed of sedan: 210
$ top speed of truck: 100
$ Properties for Sedan:
$   bool    AutomaticTransmission = 'true' defined at test_propertysystem.cc:68
$   int     GasUsage = '7' defined at test_propertysystem.cc:67
$   Inherited from CompactCar:
$     int     NumSeats = '5' defined at test_propertysystem.cc:55
$     int     TopSpeed = '::Dumux::Properties::GetProperty<TypeTag, ::Dumux::Properties::PTag::
    GasUsage>::p::value * 30' defined at test_propertysystem.cc:54
```

5.4.4 Property and Parameter Values

In DuMu^x three different ways to obtain the value of a property are available:

`GET_PROP_VALUE`: Always returns the *compile-time* specified value of the property. This is needed for properties, which are not intended to be changed by parameter files.

`GET_PARAM_FROM_GROUP`: Returns the compile-time specified value, if this value is not be overwritten by the parameter input file.

`GET_RUNTIME_PARAM_FROM_GROUP`: Always returns a *run-time* specified value. If the value is not specified at run-time an error is thrown. This is needed for problem specific properties or properties, which do not have a meaningful default value.

5.5 Grid Handling

This section summarizes some ideas about grid generation and grid formats that can be used by DuMu^x. In general, DuMu^x can read grids from file, or, construct grids inside the code. All grids are constructed inside a so called `GridCreator` which is a DuMu^x property. Note that some `GridCreators` are already available in DuMu^x, so e.g. construction of a structured grid is fairly easy. We will subsequently introduce the supported file formats, the standard grid creator and its capabilities, and briefly mention how to customize and deal with common other grid formats.

5.5.1 Supported file formats

DuMu^x can read grids from file using the Dune Grid Format (DGF) or the Gmsh mesh format.

Dune Grid Format

Most of our DuMu^x tests and tutorials 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 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 is the `test_boxco2` or `test_cco2` in the `dumux/test/porousmediumflow/co2/implicit/` folder.

Gmsh Mesh Format

Gmsh is an open-source flexible grid generator for unstructured finite-element meshes ([9], `http://geuz.org/gmsh/`). DuMu^x 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 in 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 parameterization supported by e.g. the grid manager `UGGrid`. An example can be found in `dumux/test/io/gridcreator`.

5.5.2 The default `GridCreator`

The default `GridCreator` is called `GridCreator` and is automatically avaible in all problems. It can construct grids from a DGF file (*.dgf) by simply providing the filename to the grid in the `Grid` group [1] of the input file:

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

If you are using an unstructured grid manager like `UGGrid` or `ALUGrid`, constructing a grid from a Gmsh mesh file (*.msh) is just changing a line:

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

DuMu^x will tell you in case your selected grid manager does not support reading Gmsh files. You want to intially refine your grid? It's just adding a line:

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

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

[1]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.

Grid manager specific parameters

The default `GridCreator` supports also a selection of grid specific parameters. To give an example we look at the commonly used unstructured grid manager `UGGrid`. `UGGrid`s support 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 with the default grid creator instead of the `File` key supply

```
[Grid]
LowerLeft = 0 0 0
UpperRight = 1 1 1
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`s, for example, supports simplex elements as well as hexahedral elements (called simplified "cube" in DUNE). When creating a structured grid, we can select the cell type as follows

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

For all available parameters see the Doxygen documentation.

5.5.3 Other grid formats and customized grid creators

Other grid formats than DGF and MSH have to be converted to DGF or MSH to be read into DuMu^x. A second possiblity (advanced C++) is to write your own `GridCreator`. For examples have a look at the `CubeGridCreator` for a simple and the `ArtGridCreator` for a more complex example. It follows a (non-comprehensive) list of hints for some other common grid formats.

Petrel

Grids from Petrel (in ASCII format with the extension *.GRDECL) can be imported into DuMu^x in two ways:

a) Using the GRDECL format directly with the help of the grid-manager `dune-cornerpoint`.

b) Converting the GRDECL file into the DGF format.

The fist options requires the installation of `dune-cornerpoint` along with its dependencies. Set the property `Grid` to `Dune::CpGrid` in your problem file.

The second option has the advantage that you end up with a DGF which can then be used with any grid-manager (`dune-alugrid`, `UG` etc.) You also have to install `dune-cornerpoint`. Additionally you have to modify the converter `grdecl2vtu` found in `dune-cornerpoint/examples` to also write a DGF. To do so you have to:

- Include the `dgfwriter.hh` found in `dune-grid/dune/grid/io/file/dgfparser`
- Create an object of the `Dune::DGFWriter` and call the its function `write()` within the `main` function for example after the `vtkwriter()` is called:

```
Dune::DGFWriterParam<CpGrid::LeafGridView> dgfWriter(grid.leafView()))
dgfWriter.write(fnamebase + ".dgf")
```

Material parameters for elements with Petrel specific keywords like `PORO` are parsed by the converter `grdecl2vtu` (see the `main` function). They are available as vectors within the `main` function. The main GRDECL file with the coordinates must include the GRDECL files of the parameters, if for example the parameters are not already included, include the file bearing your parameter in your main GRDECL file:

```
INCLUDE
 'PARAMETER_X.GRDECL'
/
```

To add the parameters to your DGF you have to make changes to the header `dgfwriter.hh` such that they are passed as arguments of the `write()` function and written after each element (modify `writeElement()` and internal `write()` functions accordingly). Take caution that you stick to the correct DGF syntax (see **Modules → I/O → Dune Grid Format (DGF)** for reference).

ArtMesh

ArtMesh is a 3D mesh generation software. It has its own mesh file format which can be read by DuMu^x via the `ArtGridCreator`. Traditionally it was used within DuMu^x for fracture simulations with the discrete fracture matrix model (`2pdfm`). A detailed description of the fracture network creation and gridding can be found for example in [11], pp. 68.

ICEM

For complex geometries a graphical tool to create grids might be appropriate. One possibility to mesh for example CAD geometry data is the commercial software ANSYS ICEM CFD. A very detailed, but outdated description can be found at the LH2 internal wiki. A more recent best practice guide is available in dumux-devel at dumux-devel/util/gridconverters/Documentation_ICEM_CFD_create_mesh.odt. At LH2 exists a script which converts the ICEM mesh into the DGF.

Bibliography

[1] 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.

[2] 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.

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

[4] 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 http://link.springer.com/article/10.1007%2Fs00607-008-0004-9.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] IAPWS Homepage. URL http://www.iapws.org/.

[11] Alexandru-Bogdan Tatomir. *From discrete to continuum concepts of flow in fractured porous media.* PhD thesis, Universität Stuttgart, 2012. URL http://dx.doi.org/10.18419/opus-476.

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