Handbook

April 10, 2015

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 [12]. DUNE provides a generic interface to many existing grid management libraries such as UG [24], ALBERTA [2], ALUGrid [3] and a few more. DUNE also extensively uses template programming in order to achieve minimal overhead when accessing the underlying grid libraries[1].

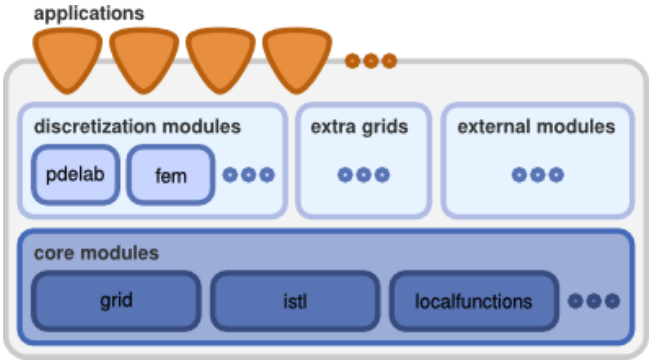


Figure 1.1: A high-level overview of DUNE's design is available on the project's web site [12].

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 [7].

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 [5].

In addition to the grid interface, DUNE also provides quite a few additional modules, of which the `dune-localfunctions` and `dune-istl` modules are the most relevant in the context of this handbook. `dune-localfunctions` provides a set of generic finite element shape functions, while `dune-istl` is the **I**terative **S**olver **T**emplate **L**ibrary and provides generic, highly optimized linear algebra routines for solving the generated systems.

DuMu$^{\text{x}}$ 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$^{\text{x}}$ 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$^{\text{x}}$ includes ready to use numerical models and a few example applications.

2 Getting started

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

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 11.

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 at [11]. 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 SVN account to our servers.

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 11 or on the DUNE website [17].

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 [17] and in much more comprehensive form in the DUNE Buildsystem Howto [13]. 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 the option `-fno-strict-aliasing` to the C++ compiler [26], which is done here via a command-line argument to `dunecontrol`:

```
$ # 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 and which can be possibly updated by subversion later on.

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 `implicit/test_box2p`:

2. Enter the folder `implicit/2p`. 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.

3 Tutorial

In DuMu^x two sorts of models are implemented: Fully-coupled models and decoupled models. In the fully-coupled 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 decoupled 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, coupled and decoupled 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.

In section 7.2.1 a short introduction to the box method is given, in section 7.2.2 the cell centered finite volume method is introduced. The box method is used in the fully-coupled models for the spatial discretization of the system of equations. The decoupled models employ usually a cell centered finite volume scheme. The following two sections of the tutorial demonstrate how to solve problems using, first, a fully-coupled model (section 3.1) and, second, using a decoupled model (section 3.2). Being the easiest case, an isothermal two-phase system (two fluid phases, one solid phase) will be considered.

3.1 Solving a Problem Using a Fully-Coupled 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.

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

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

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

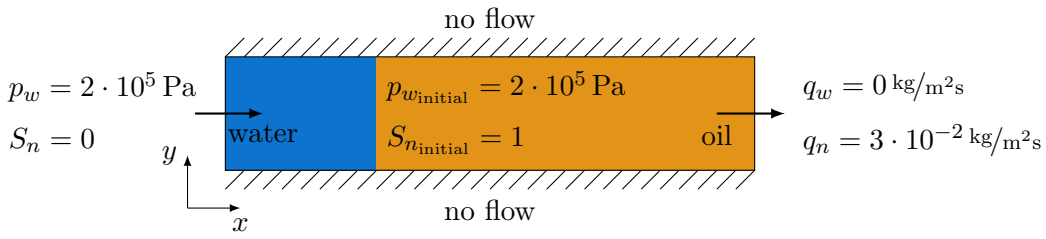


Figure 3.1: Geometry of the tutorial problem with initial and boundary conditions.

3.1.1 The Main File

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

Listing 1 (File tutorial/tutorial_coupled.cc)

```
24 #include "config.h"
25 #include "tutorialproblem_coupled.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.UpperRightX    The x-coordinate of the grid's upper-right corner [m]\n"
41         << "\t-Grid.UpperRightY    The y-coordinate of the grid's upper-right corner [m]\n"
42         << "\t-Grid.NumberOfCellsX The grid's x-resolution\n"
43         << "\t-Grid.NumberOfCellsY The grid's y-resolution\n"
44         << "\n";
45 }
46
47 int main(int argc, char** argv)
48 {
49     typedef TTAG(TutorialProblemCoupled) TypeTag;
50     return Dumux::start<TypeTag>(argc, argv, usage);
51 }
```

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

At line 49 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.

After this, the default startup routine `Dumux::start()` is called on line 50. 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_coupled.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_coupled.input)

```
1  #####################################################################
2  # Parameter file for tutorial_coupled.
3  # Everything behind a '#' is a comment.
4  # Type "./tutorial_coupled --help" for more information.
5  #####################################################################
6
7  #####################################################################
8  # Mandatory arguments
9  #####################################################################
10
11 [TimeManager]
12 TEnd = 500000 # duration of the simulation [s]
13 DtInitial = 10 # initial time step size [s]
14
15 [Grid]
16 UpperRightX = 300 # x-coordinate of the upper-right corner of the grid [m]
17 UpperRightY = 60 # y-coordinate of the upper-right corner of the grid [m]
18 NumberOfCellsX = 100 # x-resolution of the grid
19 NumberOfCellsY = 1 # y-resolution of the grid
20
21 #####################################################################
22 # Simulation restart
23 #
24 # DuMux simulations can be restarted from *.drs files
25 # Set Restart to the value of a specific file,
26 # e.g.:  'Restart = 27184.1' for the restart file
27 # name_time=27184.1_rank=0.drs
28 # Please comment in the two lines below, if restart is desired.
29 #####################################################################
30 # [TimeManager]
31 # Restart = ...
```

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. For more information about the input file please refer to section 4.3.

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_coupled.hh)

```
24 #ifndef DUMUX_TUTORIAL_PROBLEM_COUPLED_HH    // guardian macro
25 #define DUMUX_TUTORIAL_PROBLEM_COUPLED_HH    // guardian macro
26
27 // The numerical model
```

```
28 #include <dumux/implicit/2p/2pmodel.hh>
29
30 // The base porous media box problem
31 #include <dumux/implicit/common/implicitporousmediaproblem.hh>
32
33 // The DUNE grid used
34 #if HAVE_ALUGRID
35 #include <dune/grid/alugrid.hh>
36 #elif HAVE_DUNE_ALUGRID
37 #include <dune/alugrid/grid.hh>
38 #elif HAVE_UG
39 #include <dune/grid/uggrid.hh>
40 #else
41 #include <dune/grid/yaspgrid.hh>
42 #endif // HAVE_ALUGRID, HAVE_UG
43
44 // Spatially dependent parameters
45 #include "tutorialspatialparams_coupled.hh"
46
47 // The components that are used
48 #include <dumux/material/components/h2o.hh>
49 #include <dumux/material/components/lnapl.hh>
50 #include <dumux/io/cubegridcreator.hh>
51
52 namespace Dumux{
53 // Forward declaration of the problem class
54 template <class TypeTag>
55 class TutorialProblemCoupled;
56
57 namespace Properties {
58 // Create a new type tag for the problem
59 NEW_TYPE_TAG(TutorialProblemCoupled, INHERITS_FROM(BoxTwoP, TutorialSpatialParamsCoupled));
60
61 // Set the "Problem" property
62 SET_PROP(TutorialProblemCoupled, Problem)
63 { typedef Dumux::TutorialProblemCoupled<TypeTag> type;};
64
65 // Set grid and the grid creator to be used
66 #if HAVE_ALUGRID || HAVE_DUNE_ALUGRID
67 SET_TYPE_PROP(TutorialProblemCoupled, Grid, Dune::ALUGrid</*dim=*/2, 2, Dune::cube, Dune::
      nonconforming>);
68 #elif HAVE_UG
69 SET_TYPE_PROP(TutorialProblemCoupled, Grid, Dune::UGGrid<2>);
70 #else
71 SET_TYPE_PROP(TutorialProblemCoupled, Grid, Dune::YaspGrid<2>);
72 #endif // HAVE_ALUGRID
73 SET_TYPE_PROP(TutorialProblemCoupled, GridCreator, Dumux::CubeGridCreator<TypeTag>);
74
75 // Set the wetting phase
76 SET_PROP(TutorialProblemCoupled, WettingPhase)
77 {
78 private: typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
79 public: typedef Dumux::LiquidPhase<Scalar, Dumux::H2O<Scalar> > type;
80 };
81
82 // Set the non-wetting phase
83 SET_PROP(TutorialProblemCoupled, NonwettingPhase)
84 {
85 private: typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
86 public: typedef Dumux::LiquidPhase<Scalar, Dumux::LNAPL<Scalar> > type;
87 };
88
89 SET_TYPE_PROP(TutorialProblemCoupled, FluidSystem, Dumux::TwoPImmiscibleFluidSystem<TypeTag>);
```

```
90 // Disable gravity
91 SET_BOOL_PROP(TutorialProblemCoupled, ProblemEnableGravity, false);
92 }
93
94 /*!
95  * \ingroup TwoPBoxModel
96  *
97  * \brief  Tutorial problem for a fully coupled twophase box model.
98  */
99 template <class TypeTag>
100 class TutorialProblemCoupled : public ImplicitPorousMediaProblem<TypeTag>
101 {
102     typedef ImplicitPorousMediaProblem<TypeTag> ParentType;
103     typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
104     typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
105
106     // Grid dimension
107     enum { dim = GridView::dimension,
108            dimWorld = GridView::dimensionworld
109     };
110
111     // Types from DUNE-Grid
112     typedef typename GridView::template Codim<0>::Entity Element;
113     typedef typename GridView::template Codim<dim>::Entity Vertex;
114     typedef typename GridView::Intersection Intersection;
115     typedef Dune::FieldVector<Scalar, dimWorld> GlobalPosition;
116
117     // Dumux specific types
118     typedef typename GET_PROP_TYPE(TypeTag, TimeManager) TimeManager;
119     typedef typename GET_PROP_TYPE(TypeTag, Indices) Indices;
120     typedef typename GET_PROP_TYPE(TypeTag, PrimaryVariables) PrimaryVariables;
121     typedef typename GET_PROP_TYPE(TypeTag, BoundaryTypes) BoundaryTypes;
122     typedef typename GET_PROP_TYPE(TypeTag, FVElementGeometry) FVElementGeometry;
123
124 public:
125     TutorialProblemCoupled(TimeManager &timeManager,
126                            const GridView &gridView)
127         : ParentType(timeManager, gridView)
128         , eps_(3e-6)
129     {
130 #if !(HAVE_ALUGRID || HAVE_DUNE_ALUGRID || HAVE_UG)
131       std::cout << "If you want to use simplices instead of cubes, install and use ALUGrid or
          UGGrid." << std::endl;
132 #endif // !(HAVE_ALUGRID || HAVE_DUNE_ALUGRID || HAVE_UG)
133     }
134
135     //! Specifies the problem name. This is used as a prefix for files
136     //! generated by the simulation.
137     const char *name() const
138     { return "tutorial_coupled"; }
139
140     //! Returns true if a restart file should be written.
141     bool shouldWriteRestartFile() const
142     { return false; }
143
144     //! Returns true if the current solution should be written to disk
145     //! as a VTK file
146     bool shouldWriteOutput() const
147     {
148         return
149             this->timeManager().timeStepIndex() > 0 &&
150             (this->timeManager().timeStepIndex() % 1 == 0);
151     }
```

```
152
153     //! Returns the temperature within a finite volume. We use constant
154     //! 10 degrees Celsius.
155     Scalar temperature() const
156     { return 283.15; };
157
158     //! Specifies which kind of boundary condition should be used for
159     //! which equation for a finite volume on the boundary.
160     void boundaryTypes(BoundaryTypes &bcTypes, const Vertex &vertex) const
161     {
162         const GlobalPosition &globalPos = vertex.geometry().center();
163         if (globalPos[0] < eps_) // Dirichlet conditions on left boundary
164            bcTypes.setAllDirichlet();
165         else // neuman for the remaining boundaries
166            bcTypes.setAllNeumann();
167
168     }
169
170     //! Evaluates the Dirichlet boundary conditions for a finite volume
171     //! on the grid boundary. Here, the 'values' parameter stores
172     //! primary variables.
173     void dirichlet(PrimaryVariables &values, const Vertex &vertex) const
174     {
175         values[Indices::pwIdx] = 200.0e3; // 200 kPa = 2 bar
176         values[Indices::snIdx] = 0.0; // 0 % oil saturation on left boundary
177     }
178
179     //! Evaluates the boundary conditions for a Neumann boundary
180     //! segment. Here, the 'values' parameter stores the mass flux in
181     //! [kg/(m^2 * s)] in normal direction of each phase. Negative
182     //! values mean influx.
183     void neumann(PrimaryVariables &values,
184                  const Element &element,
185                  const FVElementGeometry &fvGeometry,
186                  const Intersection &intersection,
187                  int scvIdx,
188                  int boundaryFaceIdx) const
189     {
190         const GlobalPosition &globalPos =
191             fvGeometry.boundaryFace[boundaryFaceIdx].ipGlobal;
192         Scalar right = this->bBoxMax()[0];
193         // extraction of oil on the right boundary for approx. 1.e6 seconds
194         if (globalPos[0] > right - eps_) {
195             // oil outflux of 30 g/(m * s) on the right boundary.
196             values[Indices::contiWEqIdx] = 0;
197             values[Indices::contiNEqIdx] = 3e-2;
198         } else {
199             // no-flow on the remaining Neumann-boundaries.
200             values[Indices::contiWEqIdx] = 0;
201             values[Indices::contiNEqIdx] = 0;
202         }
203     }
204
205     //! Evaluates the initial value for a control volume. For this
206     //! method, the 'values' parameter stores primary variables.
207     void initial(PrimaryVariables &values,
208                  const Element &element,
209                  const FVElementGeometry &fvGeometry,
210                  int scvIdx) const
211     {
212         values[Indices::pwIdx] = 200.0e3; // 200 kPa = 2 bar
213         values[Indices::snIdx] = 1.0;
214     }
```

```
215
216     //! Evaluates the source term for all phases within a given
217     //! sub-control-volume. In this case, the 'values' parameter
218     //! stores the rate mass generated or annihilated per volume unit
219     //! in [kg / (m^3 * s)]. Positive values mean that mass is created.
220     void source(PrimaryVariables &values,
221                 const Element &element,
222                 const FVElementGeometry &fvGeometry,
223                 int scvIdx) const
224     {
225         values[Indices::contiWEqIdx] = 0.0;
226         values[Indices::contiNEqIdx]= 0.0;
227     }
228
229 private:
230     // small epsilon value
231     Scalar eps_;
232 };
233 }
234
235 #endif
```

First, a new type tag is created for the problem in line 59. 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 62, a problem class is attached to the new type tag, while the grid which is going to be used is defined in line 66 – 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.upperRightX`, `Grid.upperRightY`, `Grid.numberOfCellsX` and `Grid.numberOfCellsY`. 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 [19] is chosen as the wetting phase on line 79 and liquid oil is chosen as the non-wetting phase on line 86. The last property, which is set in line 91, 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 100.

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 codimension `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 in chapter 6.

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_coupled.hh`:

Listing 4 (File tutorial/tutorialspatialparams_coupled.hh)

```
25 #ifndef DUMUX_TUTORIAL_SPATIAL_PARAMS_COUPLED_HH
26 #define DUMUX_TUTORIAL_SPATIAL_PARAMS_COUPLED_HH
27
28 // include parent spatialparameters
29 #include <dumux/material/spatialparams/implicitspatialparams.hh>
30
31 // include material laws
32 #include <dumux/material/fluidmatrixinteractions/2p/regularizedbrookscorey.hh>
33 #include <dumux/material/fluidmatrixinteractions/2p/efftoabslaw.hh>
34 #include <dumux/material/fluidmatrixinteractions/2p/linearmaterial.hh>
35
36 namespace Dumux {
37 //forward declaration
38 template<class TypeTag>
39 class TutorialSpatialParamsCoupled;
40
41 namespace Properties
42 {
43 // The spatial parameters TypeTag
44 NEW_TYPE_TAG(TutorialSpatialParamsCoupled);
45
46 // Set the spatial parameters
47 SET_TYPE_PROP(TutorialSpatialParamsCoupled, SpatialParams,
48         Dumux::TutorialSpatialParamsCoupled<TypeTag>);
49
50 // Set the material law
51 SET_PROP(TutorialSpatialParamsCoupled, MaterialLaw)
52 {
53 private:
```

```
54      // material law typedefs
55      typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
56      // select material law to be used
57      typedef RegularizedBrooksCorey<Scalar> RawMaterialLaw;
58  public:
59      // adapter for absolute law
60      typedef EffToAbsLaw<RawMaterialLaw> type;
61  };
62  }
63
64  /*!
65   * \ingroup TwoPBoxModel
66   *
67   * \brief The spatial parameters for the fully coupled tutorial problem
68   *        which uses the twophase box model.
69   */
70  template<class TypeTag>
71  class TutorialSpatialParamsCoupled: public ImplicitSpatialParams<TypeTag>
72  {
73      // Get informations for current implementation via property system
74      typedef typename GET_PROP_TYPE(TypeTag, Grid) Grid;
75      typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
76      typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
77      enum
78      {
79          dim = Grid::dimension
80      };
81
82      // Get object types for function arguments
83      typedef typename GET_PROP_TYPE(TypeTag, FVElementGeometry) FVElementGeometry;
84      typedef typename Grid::Traits::template Codim<0>::Entity Element;
85
86  public:
87      // get material law from property system
88      typedef typename GET_PROP_TYPE(TypeTag, MaterialLaw) MaterialLaw;
89      // determine appropriate parameters depending on selected materialLaw
90      typedef typename MaterialLaw::Params MaterialLawParams;
91
92      /*! Intrinsic permeability tensor K \f$[m^2]\f$ depending
93       *  on the position in the domain
94       *
95       *  \param element The finite volume element
96       *  \param fvGeometry The finite-volume geometry in the box scheme
97       *  \param scvIdx The local vertex index
98       *
99       *  Alternatively, the function intrinsicPermeabilityAtPos(const GlobalPosition& globalPos)
100      *  could be defined, where globalPos is the vector including the global coordinates
101      *  of the finite volume.
102      */
103     const Dune::FieldMatrix<Scalar, dim, dim> &intrinsicPermeability(const Element &element,
104                                                                      const FVElementGeometry &fvGeometry,
105                                                                      const int scvIdx) const
106     { return K_; }
107
108     /*! Defines the porosity \f$[-]\f$ of the porous medium depending
109      * on the position in the domain
110      *
111      *  \param element The finite volume element
112      *  \param fvGeometry The finite-volume geometry in the box scheme
113      *  \param scvIdx The local vertex index
114      *
115      *  Alternatively, the function porosityAtPos(const GlobalPosition& globalPos)
116      *  could be defined, where globalPos is the vector including the global coordinates
```

```
117      *  of the finite volume.
118      */
119     Scalar porosity(const Element &element,
120                     const FVElementGeometry &fvGeometry,
121                     const int scvIdx) const
122     { return 0.2; }
123
124     /*! Returns the parameter object for the material law (i.e. Brooks-Corey)
125      *  depending on the position in the domain
126      *
127      *  \param element The finite volume element
128      *  \param fvGeometry The finite-volume geometry in the box scheme
129      *  \param scvIdx The local vertex index
130      *
131      *  Alternatively, the function materialLawParamsAtPos(const GlobalPosition& globalPos)
132      *  could be defined, where globalPos is the vector including the global coordinates
133      *  of the finite volume.
134      */
135     const MaterialLawParams& materialLawParams(const Element &element,
136                                                const FVElementGeometry &fvGeometry,
137                                                const int scvIdx) const
138     {
139         return materialParams_;
140     }
141
142     // constructor
143     TutorialSpatialParamsCoupled(const GridView& gridView) :
144         ImplicitSpatialParams<TypeTag>(gridView),
145         K_(0)
146     {
147         //set main diagonal entries of the permeability tensor to a value
148         //setting to one value means: isotropic, homogeneous
149         for (int i = 0; i < dim; i++)
150             K_[i][i] = 1e-7;
151
152         //set residual saturations
153         materialParams_.setSwr(0.0);
154         materialParams_.setSnr(0.0);
155
156         //parameters of Brooks & Corey Law
157         materialParams_.setPe(500.0);
158         materialParams_.setLambda(2);
159     }
160
161 private:
162     Dune::FieldMatrix<Scalar, dim, dim> K_;
163     // Object that holds the values/parameters of the selected material law.
164     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,

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

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 exercises are given in the tutorial folder in the sub-folder `solutions_coupled` 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 coupled tutorial model by typing `make tutorial_coupled` followed by `./tutorial_coupled`. 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 $x = 400\,m$ and $y = 500\,m$ and with discretization lengths of $\Delta x = 20\,m$ and $\Delta y = 20\,m$. For this you have to edit the parameter file (`tutorialproblem_coupled.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_coupled.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_coupled` 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 73 of the problem file the type of gridcreator is chosen. By choosing a different grid creator you can discretize the domain with different elements. Hint: You can find gridcreators in `dumux/io/`, change for example from `cubegridcreator.hh` to `simplexgridcreator.hh`. For ALUGrid you have to change the ALUGrid type in line 67 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 chapter 6.

e) **Use a Full-Fledged Fluid System**
DuMu^x usually describes fluid mixtures via *fluid systems*, see also chapter 6. In order to include

a fluid system, you first have to comment out lines 76 to 87 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/h2oairfluidsystem.hh` in the material folder, and set a type property `FluidSystem` (see line 89) 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. `TutorialProblemCoupled(..)`.
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 $p_e = 0.0\,\text{Pa}$ and maximal capillary pressure of e.g. $p_{c_{max}} = 2000.0\,\text{Pa}$ 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 60) in `tutorialspatialparams_coupled.hh`. Leave the type definition of `Scalar` and remove the type definition of `BrooksAndCorey` in the private section of the property definition. Exchange the `EffToAbsLaw` with the `LinearMaterial` law type in the public section. 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_coupled.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.
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_coupled.hh` (e.g. with the name `ex2_tutorialproblem_coupled.hh` and new spatial parameters `ex2_tutorialspatialparams_coupled.hh`, just like `tutorialspatialparams_coupled.hh`.

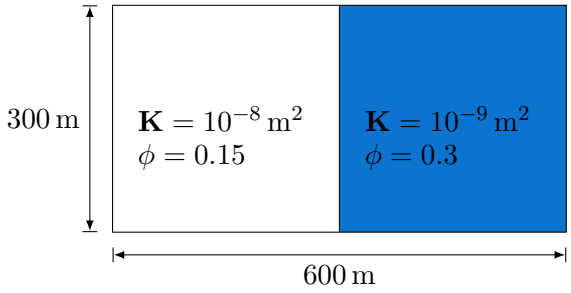


Figure 3.2: Exercise 1g: Set-up of a model domain with a heterogeneity. Grid spacing: $\Delta x = 20\,\mathrm{m}$ $\Delta y = 20\,\mathrm{m}$.

The new files should contain the definition of new classes with names that relate to the file name, such as `Ex2TutorialProblemCoupled`. Make sure that you also adjust the guardian macros in lines 24 and 25 in the header files (e.g. change `DUMUX_TUTORIALPROBLEM_COUPLED_HH` to `DUMUX_EX2_TUTORIALPROBLEM_COUPLED_HH`). Include the new problem file in `tutorial_coupled.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. `Ex2TutorialProblemCoupled`. The type tag definition has to be adjusted in `tutorial_coupled.cc` too (see line 49). 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 s. Then, you can compile the program.

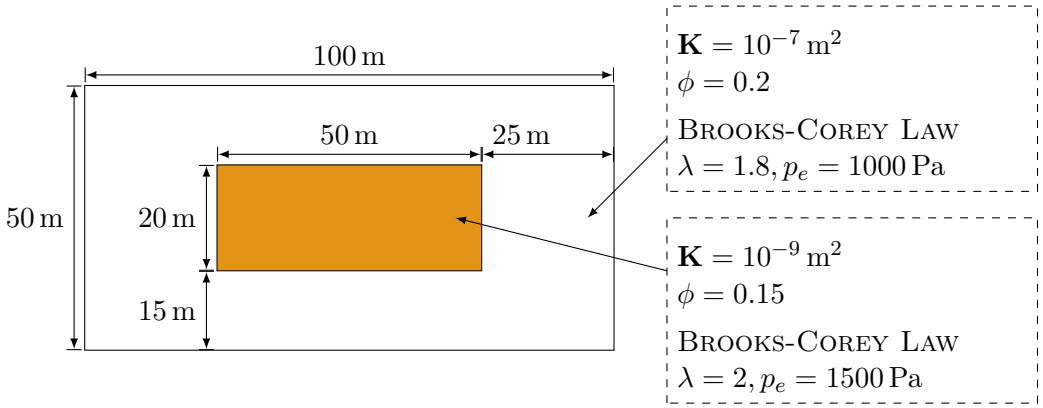


Figure 3.3: Set-up of the model domain and the soil parameters

- Increase the simulation time to e.g. $4 \times 10^7\,\mathrm{s}$. Investigate the saturation: Is the value range

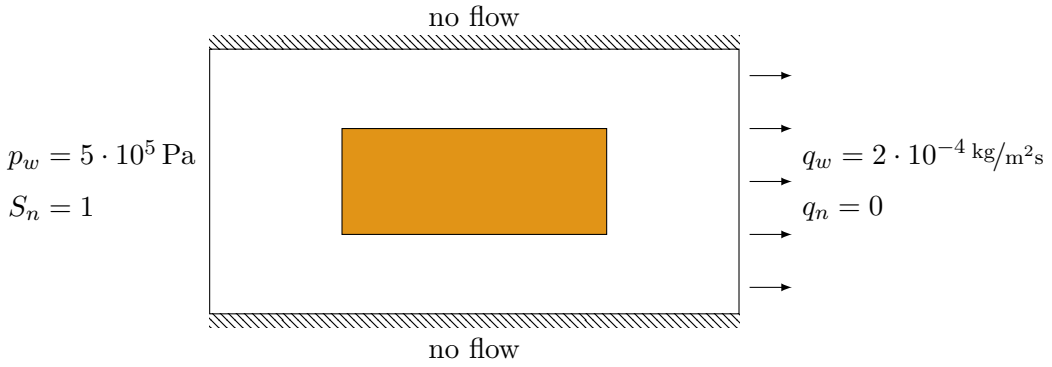


Figure 3.4: Boundary Conditions

reasonable?

- What happens if you increase the resolution of the grid?

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 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_coupled.input` in the `solutions_coupled` 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_coupled.diff` and `ex3_tutorialspatialparams_coupled.diff` in the `solutions_coupled` folder. Add some (for example `Newton.MaxSteps` and `Problem.EnableGravity`) to the parameter file `tutorial_coupled.input` and observe what happens if they are modified. For more information about the input file please refer to section 4.3.

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 $0.00112\,\mathrm{Pa\,s}$.

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.

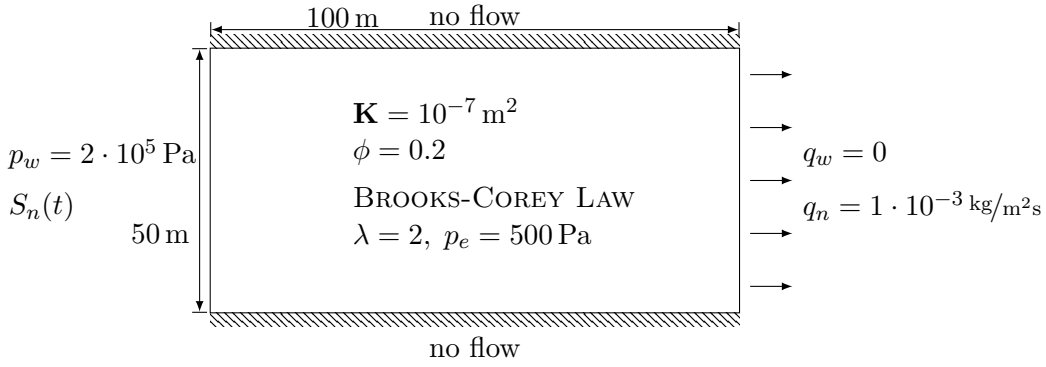


Figure 3.5: Set-up of the model domain and the soil parameters

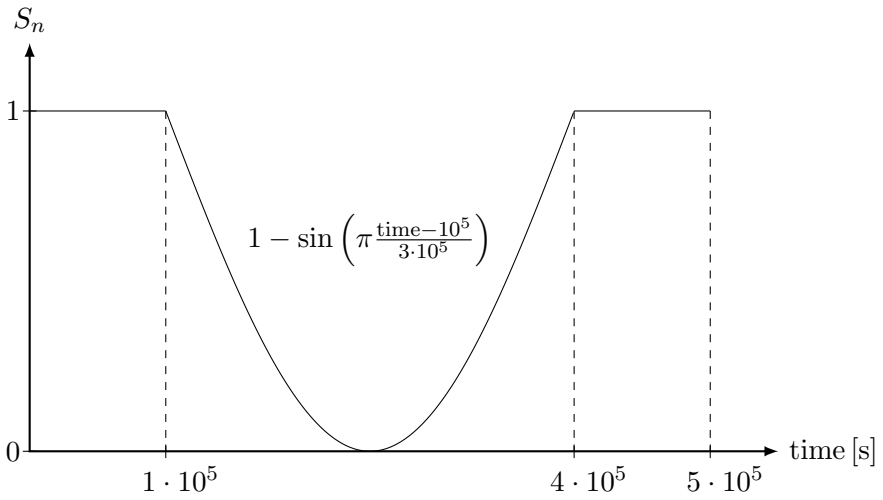


Figure 3.6: Time Dependent Boundary Conditions

- 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?

3.2 Solving a problem using a Decoupled 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 decoupled 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.

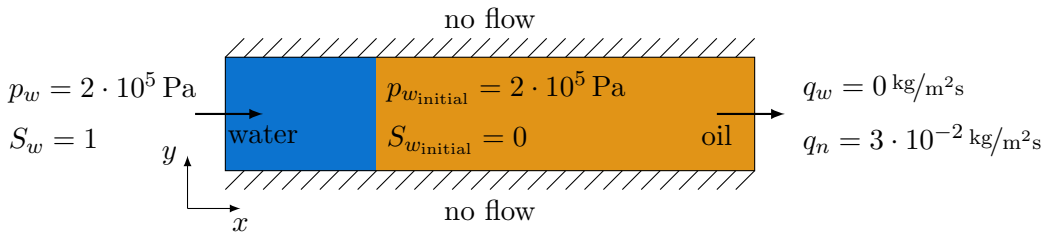


Figure 3.7: Geometry of the tutorial problem with initial and boundary conditions.

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 decoupled model. This main file can be found in the directory `/tutorial` of the stable part of DuMu^x.

Listing 5 (File tutorial/tutorial_decoupled.cc)

```
24 #include "config.h"
25
26 #include "tutorialproblem_decoupled.hh"
27 #include <dumux/common/start.hh>
28
29 //! Prints a usage/help message if something goes wrong or the user asks for help
30 void usage(const char *progName, const std::string &errorMsg)
31 {
32     std::cout
33         <<  "\nUsage: " << progName << " [options]\n";
34     if (errorMsg.size() > 0)
35         std::cout << errorMsg << "\n";
36     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.UpperRightX    The x-coordinate of the grid's upper-right corner [m]\n"
42          << "\t-Grid.UpperRightY    The y-coordinate of the grid's upper-right corner [m]\n"
43          << "\t-Grid.NumberOfCellsX The grid's x-resolution\n"
44          << "\t-Grid.NumberOfCellsY The grid's y-resolution\n"
45          << "\n";
46  }
47
48
49  ////////////////////////
50  // the main function
51  ////////////////////////
52  int main(int argc, char** argv)
53  {
54      typedef TTAG(TutorialProblemDecoupled) TypeTag;
55      return Dumux::start<TypeTag>(argc, argv, usage);
56  }
```

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

At line 54 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.

After this DuMu^x' default startup routine `Dumux::start()` is called in line 55. 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_decoupled.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_decoupled.input)

```
1  #####################################################################
2  # Parameter file for tutorial_decoupled.
3  # Everything behind a '#' is a comment.
4  # Type "./tutorial_decoupled --help" for more information.
5  #####################################################################
6
7  #####################################################################
8  # Mandatory arguments
9  #####################################################################
10
11 [TimeManager]
12 TEnd = 100000 # duration of the simulation [s]
13 DtInitial = 10 # initial time step size [s]
14
15 [Grid]
16 UpperRightX = 300 # x-coordinate of the upper-right corner of the grid [m]
17 UpperRightY = 60 # y-coordinate of the upper-right corner of the grid [m]
18 NumberOfCellsX = 100 # x-resolution of the grid
19 NumberOfCellsY = 1 # y-resolution of the grid
20
```

```
21 ############################################################
22 # Simulation restart
23 #
24 # DuMux simulations can be restarted from *.drs files
25 # Set Restart to the value of a specific file,
26 # e.g.:  'Restart = 27184.1' for the restart file
27 # name_time=27184.1_rank=0.drs
28 # Please comment in the two lines below, if restart is desired.
29 ############################################################
30 # [TimeManager]
31 # Restart = ...
```

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. For more information about the input file please refer to section 4.3.

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_decoupled.hh`.

Listing 7 (File tutorial/tutorialproblem_decoupled.hh)

```
24 #ifndef DUMUX_TUTORIALPROBLEM_DECOUPLED_HH // guardian macro
25 #define DUMUX_TUTORIALPROBLEM_DECOUPLED_HH // guardian macro
26
27 // the grid includes
28 #include <dune/grid/yaspgrid.hh>
29 #include <dumux/io/cubegridcreator.hh>
30
31 // dumux 2p-decoupled environment
32 #include <dumux/decoupled/2p/diffusion/fv/fvpressureproperties2p.hh>
33 #include <dumux/decoupled/2p/transport/fv/fvtransportproperties2p.hh>
34 #include <dumux/decoupled/2p/impes/impesproblem2p.hh>
35
36 // assign parameters dependent on space (e.g. spatial parameters)
37 #include "tutorialspatialparams_decoupled.hh"
38
39 // include cfl-criterion after coats: more suitable if the problem is not advection dominated
40 #include<dumux/decoupled/2p/transport/fv/evalcflfluxcoats.hh>
41
42 // the components that are used
43 #include <dumux/material/components/h2o.hh>
44 #include <dumux/material/components/lnapl.hh>
45
46 namespace Dumux
47 {
48
49 template<class TypeTag>
50 class TutorialProblemDecoupled;
51
52 //////////
53 // Specify the properties for the lens problem
54 //////////
55 namespace Properties
56 {
57 // create a new type tag for the problem
58 NEW_TYPE_TAG(TutorialProblemDecoupled, INHERITS_FROM(FVPressureTwoP, FVTransportTwoP, IMPESTwoP
       ,
```

```
59                                                      TutorialSpatialParamsDecoupled));
60
61 // Set the problem property
62 SET_PROP(TutorialProblemDecoupled, Problem)
63 {
64     typedef Dumux::TutorialProblemDecoupled<TypeTag> type;
65 };
66
67 // Set the grid type
68 SET_TYPE_PROP(TutorialProblemDecoupled, Grid, Dune::YaspGrid<2>);
69
70 //Set the grid creator
71 SET_TYPE_PROP(TutorialProblemDecoupled, GridCreator, Dumux::CubeGridCreator<TypeTag>);
72
73 // Set the wetting phase
74 SET_PROP(TutorialProblemDecoupled, WettingPhase)
75 {
76 private:
77     typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
78 public:
79     typedef Dumux::LiquidPhase<Scalar, Dumux::H2O<Scalar> > type;
80 };
81
82 // Set the non-wetting phase
83 SET_PROP(TutorialProblemDecoupled, NonwettingPhase)
84 {
85 private:
86     typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
87 public:
88     typedef Dumux::LiquidPhase<Scalar, Dumux::LNAPL<Scalar> > type;
89 };
90
91 SET_TYPE_PROP(TutorialProblemDecoupled, EvalCflFluxFunction, Dumux::EvalCflFluxCoats<TypeTag>);

92 SET_SCALAR_PROP(TutorialProblemDecoupled, ImpetCFLFactor, 0.95);
93
94 // Disable gravity
95 SET_BOOL_PROP(TutorialProblemDecoupled, ProblemEnableGravity, false);
96 }
97
98 /*! \ingroup DecoupledProblems
99  * @brief Problem class for the decoupled tutorial
100 */
101 template<class TypeTag>
102 class TutorialProblemDecoupled: public IMPESProblem2P<TypeTag>
103 {
104     typedef IMPESProblem2P<TypeTag> ParentType;
105     typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
106     typedef typename GET_PROP_TYPE(TypeTag, TimeManager) TimeManager;
107     typedef typename GET_PROP_TYPE(TypeTag, Indices) Indices;
108
109     typedef typename GET_PROP_TYPE(TypeTag, BoundaryTypes) BoundaryTypes;
110     typedef typename GET_PROP(TypeTag, SolutionTypes) SolutionTypes;
111     typedef typename SolutionTypes::PrimaryVariables PrimaryVariables;
112
113     enum
114     {
115         dimWorld = GridView::dimensionworld
116     };
117
118     enum
119     {
120         wPhaseIdx = Indices::wPhaseIdx,
```

```
121         nPhaseIdx = Indices::nPhaseIdx,
122         pwIdx = Indices::pwIdx,
123         swIdx = Indices::swIdx,
124         pressEqIdx = Indices::pressureEqIdx,
125         satEqIdx = Indices::satEqIdx
126     };
127
128     typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
129
130     typedef typename GridView::Traits::template Codim<0>::Entity Element;
131     typedef typename GridView::Intersection Intersection;
132     typedef Dune::FieldVector<Scalar, dimWorld> GlobalPosition;
133
134 public:
135     TutorialProblemDecoupled(TimeManager &timeManager, const GridView &gridView)
136         : ParentType(timeManager, gridView), eps_(1e-6)
137     {
138         //write only every 10th time step to output file
139         this->setOutputInterval(10);
140     }
141
142     //! The problem name.
143     /*! This is used as a prefix for files generated by the simulation.
144     */
145     const char *name() const
146     {
147         return "tutorial_decoupled";
148     }
149
150     //!  Returns true if a restart file should be written.
151     /* The default behaviour is to write no restart file.
152      */
153     bool shouldWriteRestartFile() const
154     {
155         return false;
156     }
157
158     //! Returns the temperature within the domain at position globalPos.
159     /*! This problem assumes a temperature of 10 degrees Celsius.
160      *
161      *  \param element The finite volume element
162      *
163      * Alternatively, the function temperatureAtPos(const GlobalPosition& globalPos) could be
164      * defined, where globalPos is the vector including the global coordinates of the finite
           volume.
165      */
166     Scalar temperature(const Element& element) const
167     {
168         return 273.15 + 10; // -> 10Â°C
169     }
170
171     //! Returns a constant pressure to enter material laws at position globalPos.
172     /* For incompressible simulations, a constant pressure is necessary
173      * to enter the material laws to gain a constant density etc. In the compressible
174      * case, the pressure is used for the initialization of material laws.
175      *
176      * \param element The finite volume element
177      *
178      * Alternatively, the function referencePressureAtPos(const GlobalPosition& globalPos)
           could be
179      * defined, where globalPos is the vector including the global coordinates of the finite
           volume.
180      */
```

```
181     Scalar referencePressure(const Element& element) const
182     {
183         return 2e5;
184     }
185
186     //! Source of mass \f$ [\frac{kg}{m^3 \cdot s}] \f$ of a finite volume.
187     /*! Evaluate the source term for all phases within a given
188      *  volume.
189      *
190      *  \param values Includes sources for the two phases
191      *  \param element The finite volume element
192      *
193      *  The method returns the mass generated (positive) or
194      *  annihilated (negative) per volume unit.
195      *
196      * Alternatively, the function sourceAtPos(PrimaryVariables &values, const GlobalPosition&
         globalPos)
197      * could be defined, where globalPos is the vector including the global coordinates of the
         finite volume.
198      */
199     void source(PrimaryVariables &values, const Element& element) const
200     {
201         values = 0;
202     }
203
204     //! Type of boundary conditions at position globalPos.
205     /*! Defines the type the boundary condition for the pressure equation,
206      *  either pressure (dirichlet) or flux (neumann),
207      *  and for the transport equation,
208      *  either saturation (dirichlet) or flux (neumann).
209      *
210      *  \param bcTypes Includes the types of boundary conditions
211      *  \param globalPos The position of the center of the finite volume
212      *
213      *  Alternatively, the function boundaryTypes(PrimaryVariables &values, const Intersection&
214      *  intersection) could be defined, where intersection is the boundary intersection.
215      */
216     void boundaryTypesAtPos(BoundaryTypes &bcTypes, const GlobalPosition& globalPos) const
217     {
218             if (globalPos[0] < this->bBoxMin()[0] + eps_)
219             {
220                 bcTypes.setDirichlet(pressEqIdx);
221                 bcTypes.setDirichlet(satEqIdx);
222 //                bcTypes.setAllDirichlet(); // alternative if the same BC is used for both
    types of equations
223             }
224             // all other boundaries
225             else
226             {
227                 bcTypes.setNeumann(pressEqIdx);
228                 bcTypes.setNeumann(satEqIdx);
229 //                bcTypes.setAllNeumann(); // alternative if the same BC is used for both types
     of equations
230             }
231     }
232     //! Value for dirichlet boundary condition at position globalPos.
233     /*! In case of a dirichlet BC for the pressure equation the pressure \f$ [Pa] \f$, and for
234      *  the transport equation the saturation [-] have to be defined on boundaries.
235      *
236      *  \param values Values of primary variables at the boundary
237      *  \param intersection The boundary intersection
238      *
239      *  Alternatively, the function dirichletAtPos(PrimaryVariables &values, const
```

```
            GlobalPosition& globalPos)
240      *  could be defined, where globalPos is the vector including the global coordinates of the
             finite volume.
241      */
242     void dirichlet(PrimaryVariables &values, const Intersection& intersection) const
243     {
244         values[pwIdx] = 2e5;
245         values[swIdx] = 1.0;
246     }
247     //! Value for neumann boundary condition \f$ [\frac{kg}{m^3 \cdot s}] \f$ at position
            globalPos.
248     /*! In case of a neumann boundary condition, the flux of matter
249      *  is returned as a vector.
250      *
251      *  \param values Boundary flux values for the different phases
252      *  \param globalPos The position of the center of the finite volume
253      *
254      *  Alternatively, the function neumann(PrimaryVariables &values, const Intersection&
             intersection) could be defined,
255      *  where intersection is the boundary intersection.
256      */
257     void neumannAtPos(PrimaryVariables &values, const GlobalPosition& globalPos) const
258     {
259         values = 0;
260         if (globalPos[0] > this->bBoxMax()[0] - eps_)
261         {
262             values[nPhaseIdx] = 3e-2;
263         }
264     }
265     //! Initial condition at position globalPos.
266     /*! Only initial values for saturation have to be given!
267      *
268      *  \param values Values of primary variables
269      *  \param element The finite volume element
270      *
271      *  Alternatively, the function initialAtPos(PrimaryVariables &values, const GlobalPosition
             & globalPos)
272      *  could be defined, where globalPos is the vector including the global coordinates of the
             finite volume.
273      */
274     void initial(PrimaryVariables &values,
275             const Element &element) const
276     {
277         values = 0;
278     }
279
280 private:
281     const Scalar eps_;
282 };
283 } //end namespace
284
285 #endif
```

First, both DUNE grid handlers and the decoupled model of DuMu^x have to be included. Then, a new type tag is created for the problem in line 59. In this case, the new type tag inherits all properties defined for the `DecoupledTwoP` type tag, which means that for this problem the two-phase decoupled approach is chosen as discretization scheme (defined via the include in line 34). On line 62, a problem class is attached to the new type tag, while the grid which is going to be used is defined in line 68 – in this case an `YaspGrid` is created. 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` (line 71) 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.upperRightX`, `Grid.upperRightY`, `Grid.numberOfCellsX` and `Grid.numberOfCellsY`. 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 the *Dune Grid Interface HOWTO* [12].

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 74 to 89. These are linked to the appropriate chemical species in line 79 and 88. 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 decoupled model. First, in line 91 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 92 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 95 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 96, the problem class is defined in line 102. As its property, the problem class itself is also derived from a parent, `IMPESProblem2P`. The class constructor (line 136) 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 14), 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 145, and line 153 indicates whether restart files are written. As decoupled 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 139, 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 216.

There are the methods for general parameters, source- or sinkterms, boundary conditions (lines 216 to 257) and initial values for the transported quantity in line 275. 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_decoupled.hh`:

Listing 8 (File tutorial/tutorialspatialparams_decoupled.hh)

```
24 #ifndef DUMUX_TUTORIAL_SPATIAL_PARAMS_DECOUPLED_HH
25 #define DUMUX_TUTORIAL_SPATIAL_PARAMS_DECOUPLED_HH
26
```

```
27
28 #include <dumux/material/spatialparams/fvspatialparams.hh>
29 #include <dumux/material/fluidmatrixinteractions/2p/linearmaterial.hh>
30 #include <dumux/material/fluidmatrixinteractions/2p/regularizedbrookscorey.hh>
31 #include <dumux/material/fluidmatrixinteractions/2p/efftoabslaw.hh>
32
33 namespace Dumux
34 {
35
36 //forward declaration
37 template<class TypeTag>
38 class TutorialSpatialParamsDecoupled;
39
40 namespace Properties
41 {
42 // The spatial parameters TypeTag
43 NEW_TYPE_TAG(TutorialSpatialParamsDecoupled);
44
45 // Set the spatial parameters
46 SET_TYPE_PROP(TutorialSpatialParamsDecoupled, SpatialParams,
47         Dumux::TutorialSpatialParamsDecoupled<TypeTag>);
48
49 // Set the material law
50 SET_PROP(TutorialSpatialParamsDecoupled, MaterialLaw)
51 {
52 private:
53     // material law typedefs
54     typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
55     typedef RegularizedBrooksCorey<Scalar> RawMaterialLaw;
56 public:
57     typedef EffToAbsLaw<RawMaterialLaw> type;
58 };
59 }
60
61 //! Definition of the spatial parameters for the decoupled tutorial
62
63 template<class TypeTag>
64 class TutorialSpatialParamsDecoupled: public FVSpatialParams<TypeTag>
65 {
66     typedef FVSpatialParams<TypeTag> ParentType;
67     typedef typename GET_PROP_TYPE(TypeTag, Grid) Grid;
68     typedef typename GET_PROP_TYPE(TypeTag, GridView) GridView;
69     typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
70     typedef typename Grid::ctype CoordScalar;
71
72     enum
73         {dim=Grid::dimension, dimWorld=Grid::dimensionworld};
74     typedef typename Grid::Traits::template Codim<0>::Entity Element;
75
76     typedef Dune::FieldVector<CoordScalar, dimWorld> GlobalPosition;
77     typedef Dune::FieldMatrix<Scalar,dim,dim> FieldMatrix;
78
79 public:
80     typedef typename GET_PROP_TYPE(TypeTag, MaterialLaw) MaterialLaw;
81     typedef typename MaterialLaw::Params MaterialLawParams;
82
83     //! Intrinsic permeability tensor K \f$[m^2]\f$ depending
84     /*! on the position in the domain
85      *
86      *  \param element The finite volume element
87      *
88      *  Alternatively, the function intrinsicPermeabilityAtPos(const GlobalPosition& globalPos)
          could be
```

```
89      *  defined, where globalPos is the vector including the global coordinates of the finite
            volume.
90      */
91     const FieldMatrix& intrinsicPermeability (const Element& element) const
92     {
93             return K_;
94     }
95
96     //! Define the porosity \f$[-]\f$ of the porous medium depending
97     /*! on the position in the domain
98      *
99      *  \param element The finite volume element
100     *
101     *  Alternatively, the function porosityAtPos(const GlobalPosition& globalPos) could be
102     *  defined, where globalPos is the vector including the global coordinates of the finite
            volume.
103     */
104    double porosity(const Element& element) const
105    {
106        return 0.2;
107    }
108
109    /*! Return the parameter object for the material law (i.e. Brooks-Corey)
110     *  depending on the position in the domain
111     *
112     *  \param element The finite volume element
113     *
114     *  Alternatively, the function materialLawParamsAtPos(const GlobalPosition& globalPos)
115     *  could be defined, where globalPos is the vector including the global coordinates of
116     *  the finite volume.
117     */
118    const MaterialLawParams& materialLawParams(const Element &element) const
119    {
120            return materialLawParams_;
121    }
122
123    //! Constructor
124    TutorialSpatialParamsDecoupled(const GridView& gridView)
125    : ParentType(gridView), K_(0)
126    {
127        for (int i = 0; i < dim; i++)
128                K_[i][i] = 1e-7;
129
130        // residual saturations
131        materialLawParams_.setSwr(0);
132        materialLawParams_.setSnr(0);
133
134        // parameters for the Brooks-Corey Law
135        // entry pressures
136        materialLawParams_.setPe(500);
137
138        // Brooks-Corey shape parameters
139        materialLawParams_.setLambda(2);
140    }
141
142 private:
143    MaterialLawParams materialLawParams_;
144    FieldMatrix K_;
145 };
146
147 } // end namespace
148 #endif
```

As this file only slightly differs from the coupled 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 coupled 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 decoupled 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 decoupled tutorial model by typing `./tutorial_decoupled`. 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_decoupled.cc` and `tutorial_decoupled.input`), it is automatically chosen. If the name differs the program has to be started typing `./tutorial_decoupled -parameterFile <filename>.input`. For more options you can also type `./tutorial_decoupled -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 139 in the file `tutorialproblem_decoupled`) as to write an output only every 20 time-steps. Compile the main file by typing `make tutorial_decoupled` 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_decoupled.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

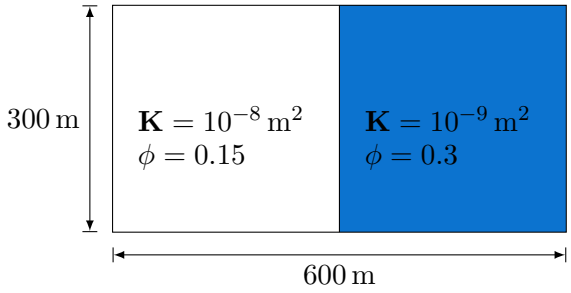


Figure 3.8: Exercise 1d: Set-up of a model domain a heterogeneity. $\Delta x = \Delta y = 20\,\mathrm{m}$.

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^x fluid system**
DuMu^x usually organizes fluid mixtures via a `fluidsystem`, see also chapter 6. In order to include a fluidsystem you first have to comment the lines 74 to 89 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/h2oairfluidsystem.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. `TutorialProblemDecoupled(..)`. 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 decoupled 2p2c model.
The density of the gas is magnitudes smaller than that of oil, so please decrease the outflow rate to $q_n = 3 \times 10^{-4}\,\left[\frac{\mathrm{kg}}{\mathrm{m^2 s}}\right]$. 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 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_decoupled.hh` (e.g. with the name `ex2_tutorialproblem_decoupled.hh` and new spatial parameters just like `tutorialspatialparams_decoupled.hh`. These files need to be included in the file `tutorial_decoupled.cc`.

Each new files should contain the definition of a new class with a name that relates to the file name, such as `Ex2TutorialProblemDecoupled`. Make sure that you also adjust the guardian macros in lines 24 and 25 in the header files (e.g. change `DUMUX_TUTORIALPROBLEM_DECOUPLED_HH` to `DUMUX_EX2_TUTORIALPROBLEM_DECOUPLED_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. `Ex2TutorialProblemDecoupled`. 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 $p_w = 2 \times 10^5\,\text{Pa}$. 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 $1e6\,\text{s}$.

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

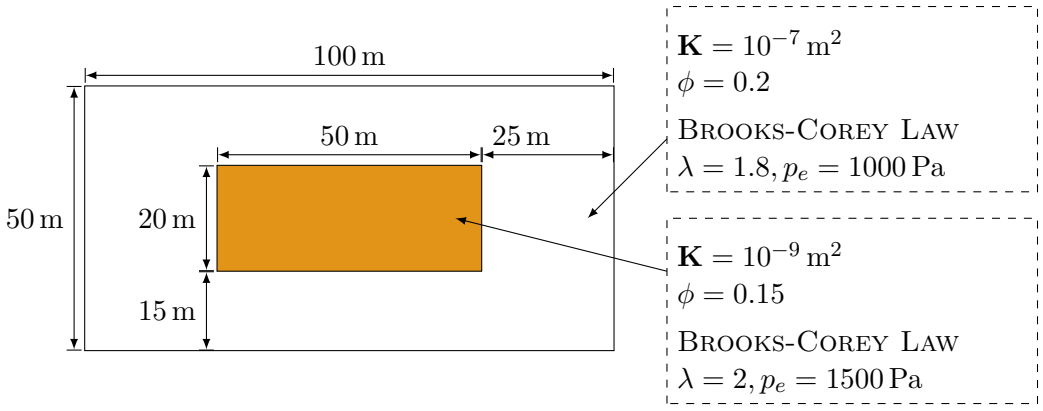


Figure 3.9: Set-up of the model domain and the soil parameters

- 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$^{\text{x}}$ provides a simple method to read in parameters (such as simulation end time or modelling parameters) via `Paramter`

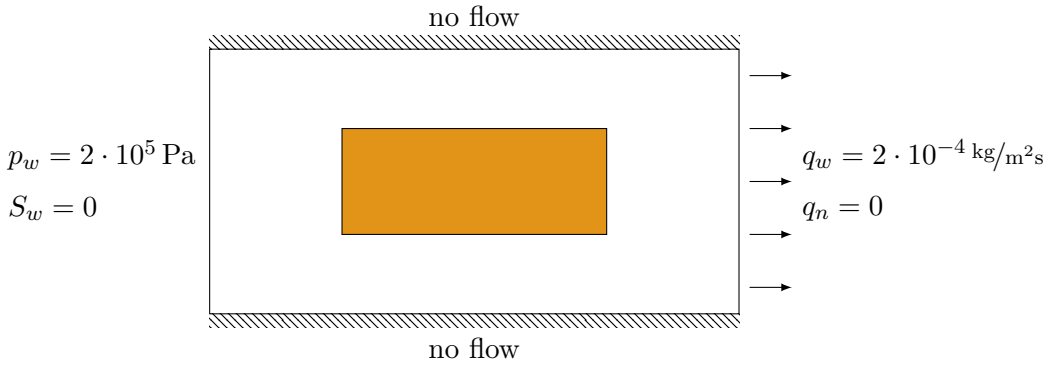


Figure 3.10: Boundary Conditions

`Input Files`. The tests in the Test-folder `/test/` already use this system.
If you look at the Application in `/test/implicit/2p/`, 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 decoupled 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 $889.51\,\mathrm{kg/m^3}$ and a viscosity of $0.00112\,\mathrm{Pa\,s}$.

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 $10^5\,\mathrm{s}$ and $4 \cdot 10^5\,\mathrm{s}$. 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 $p_w = 2 \times 10^5\,\mathrm{Pa}$. 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.

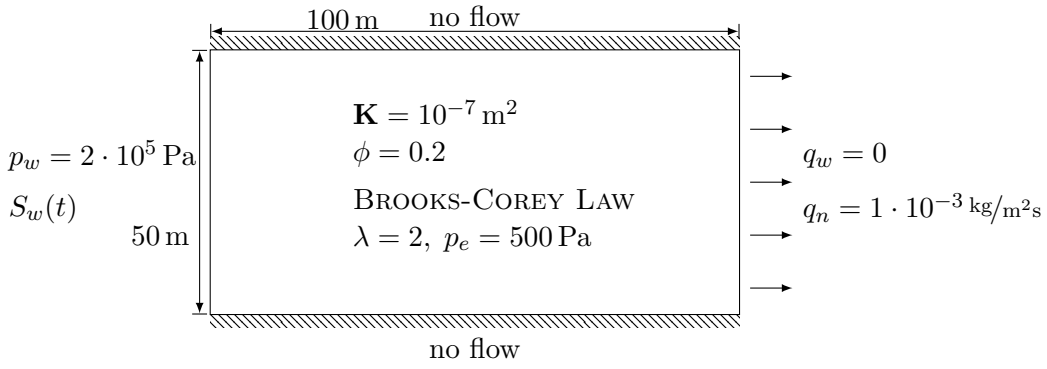


Figure 3.11: Set-up of the model domain and the soil parameters

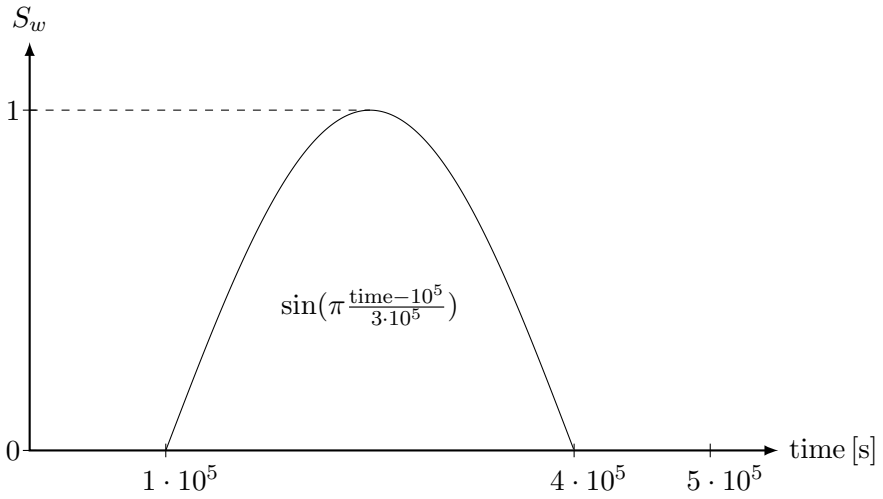


Figure 3.12: Time Dependent Boundary Conditions

- 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 coupled and the decoupled 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 coupled or decoupled. Try to formulate a spatial parameters file that works with both problems, the coupled and the decoupled. Therein, only use functions at the position.

4 Structure, Guidelines, New Folder Setup

4.1 Directory Structure

We briefly describe the directory structure of DuMu^x in terms of subdirectories, source files, and tests. For more details, the Doxygen documentation should be considered. DuMu^x comes in form of a DUNE module `dumux`. It has a similar structure as other DUNE modules like `dune-grid`. The following subdirectories are within the module's root directory, from now on assumed to be `/`:

- `bin`: contains binaries, e.g. used for the automatic testing
- `CMake`: the configuration options for building DuMu^x using CMake. See the file `INSTALL.cmake` in the root directory of `dumux` for details. Of course, it is also possible to use the DUNE buildsystem just like for the other DUNE modules.
- `doc`: contains the Doxygen documentation in `doxygen`, this handbook in `handbook`, and the DuMu^x logo in various formats in `logo`. The html documentation produced by Doxygen can be accessed as usual, namely, by opening `doc/doxygen/html/index.html` with a web browser.
- `dumux`: the DuMu^x source files. See Section 4.1.1 for details.
- `test`: tests for each numerical model and the property system. See Section 4.1.2 for details.
- `tutorial`: contains the tutorials described in Chapter 3.

4.1.1 The directory `dumux`

The directory `dumux` contains the DuMu^x source files. It consists of the following subdirectories (see Figure 4.1):

- `implicit`: the general fully implicit method is contained in the subdirectory `common`. The subdirectories `box` and `cellcentered` contain the code for the according discretization types. They also contain files `..fvelementgeometry.hh` employed by the box or cc method to extract the dual mesh geometry information out of the primal one. Each of the other subdirectories contain a derived specific numerical model.
- `common`: general stuff like the property system and the time management for the fully coupled as well as the decoupled models, and the `start.hh` file that includes the common routine for starting a model called in the main function.
- `decoupled`: numerical models to solve the pressure equation as part of the fractional flow formulation. The specific models are contained in corresponding subdirectories. In each model folder are subdirectories for the implicit pressure equation sorted by the employed discretization

method, and for the explicit transport equation. The general decoupled formulation for the implicit pressure explicit transport formulation can be found in the subdirectory `common`.

- `io`: additional in-/output possibilities like restart files, gnuplot-interface and a VTKWriter extension.
- `material`: everything related to material parameters and constitutive equations. The properties of a pure chemical substance (e.g. water) or pseudo substance (e.g. air) can be found in the subdirectory `components` with the base class `components/component.hh`. The fluidsytem in the folder `fluidsystems` collects the information from the respective component and binary coefficients files, and contains the fluid characteristics of phases (e.g. viscosity, density, enthalpy, diffusion coefficients) for compositional or non-compositional multi-phase flow.

  The base class for all spatially dependend variables – like permeability and porosity – can be found in `spatialparams`. The base class in `implicitspatialparameters.hh` also provides spatial averaging routines. All other spatial properties are specified in the specific files of the respective models. Furthermore, the constitutive relations – e.g. $p_c(S_w)$ – are in `fluidmatrixinteractions`, while the necessary binary coefficients like the Henry coefficient or binary diffusion coefficients are definded in `binarycoefficients`.
- `nonlinear`: Newton's method.

4.1.2 The directory `test`

The directory `test` contains a test for each numerical model and for the property system. The tests for the property system can be found in `common`. The subfolder `implicit` contains tests for the fully coupled models (`1p`, `1p2c`, `2p`, `2p2c`, `2p2cni`, `2pni`, `3p3c`, `3p3cni`, `mpnc` and `richards`), while the subdirectory `decoupled` corresponds to the decoupled models. Each subdirectory contains one or more program files `test_*.cc`, where `*` usually is the name of the folder. Moreover, the problem definitions can be found in the `*problem.hh` files and the definition of the spatially dependent parameters in `*spatialparameters.hh`. Simply executing the tests should either run the full test or give a list of required command line arguments. After test execution, VTK output files should have been generated. For more detailed descriptions of the tests, the problem definitions and their corresponding Doxygen documentation should be considered.

4.2 Setup of a New Folder and New Tests

Setting up a New Folder In this section it is described how to set up a new folder and how to tell the build system, that there is a new one.

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) adapt the `CMakeList.txt` in the newly created folder and add your test (see below for more information)

4) go to your `build`-directory and type `make` to reconfigure the system

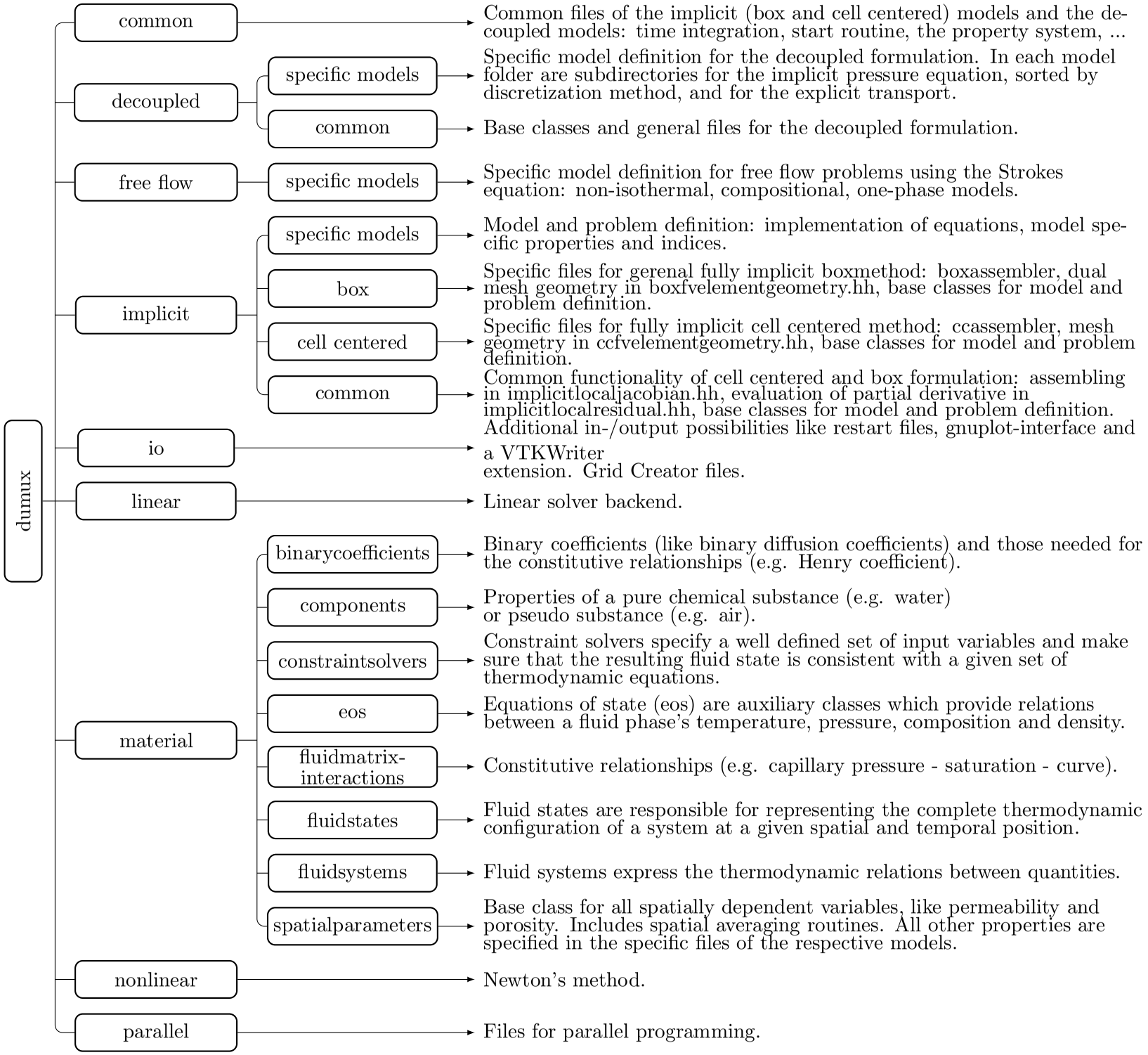


Figure 4.1: Structure of the directory `dumux` containing the DuMu^x source files.

Adding a New Test Program To simply add a new executable use the following macro. The test will *not* be built automatically when running `ctest`. You have to compile it manually by `make test_program`.

```
add_executable_all(test\_program test\_program.cc)
```

To add a test, which should be compiled when running `ctest`, use the `add_dumux_test` macro. You can decide whether, the program should be run after compiling or not. Please note that the name of the test (first argument) must be unique, whereas the name of the executable (second argument) can occur multiple times.

```
add_dumux_test(test\_program test\_program test\_program.cc
  test\_program # add this line, if the program should also be run
  )
```

To add a test which should be run and compared to a reference solution when using `ctest`, please use the following structure. The macro `$CMAKE_SOURCE_DIR` gives the location of your source code. The macro `$CMAKE_CURRENT_BINARY_DIR` gives the current folder with the executable.

```
add_dumux_test(test\_program test\_program test\_program.cc
  ${CMAKE_SOURCE_DIR}/bin/runTest.sh
  ${CMAKE_SOURCE_DIR}/bin/fuzzycomparevtu.py
  LOCATION_TO_THE_REFERNCE_SOLUTION/test\_program-reference.vtu
  ${CMAKE_CURRENT_BINARY_DIR}/test\_program-00009.vtu
  ${CMAKE_CURRENT_BINARY_DIR}/test\_program)
```

Committing a New older to SVN For those who work with Subversion (`svn`) and want to commit a newly setup folder to the repository some basics are given in this paragraph. For further reading please check out the Subversion User Manual found at [4] where you will also find a "High Speed Turorial" in the appendix.
The four most important commands are `svn checkout`, `svn update`, `svn add` and `svn commit`. The first one (`svn checkout`) you probably already know from the DuMu^x installation. It will create a copy of the trunk version from the svn server on your local system. Use `svn update` to get the latest changes in the repository (commits from other users). In order to add a new folder to the repository the following steps have to be taken:

1) `svn update`: The first step is to update your DuMu^x. You should execute this command in your dumux-stable or dumux-devel folder.

2) `svn add --depth=empty YOURFOLDER`: This command adds the folder without its content.

3) In your folder: use `svn add YOURFILES` to add your files. Generally, you should only add your header files (.hh), your source files (.cc), your input file (.input), if required your grid file (.dgf) or if necessary other text-based files. Please do not upload (large) binary files.

4) Type `svn status` in your `dumux`-root directory the see all the file changes. `?` indicates possible forgotten files. Make sure that you include all necessary files in your commit.

5) Use `svn commit` from the directory level containing your folder. This uploads all your changes to the svn server. You will be asked to briefly explain the content of your commit in an editor.

4.3 Parameter Files in DuMu^x

DuMu^x simulations can be run with the use parameter files. Here basic information how to set, extend, and improve your problem by using parameter files. A list of all available parameters is provided in the `doxygen` documentation of the file `parameterfile`, which is accessible via `Modules -> Parameters`.

4.3.1 Advantages of Parameter Files

Parameter files are worth of taking a closer look at, because using then considerably improves the workflow.

- The parameter file is read in by the compiled program. This way you can change values without having to recompile the whole application.
- With a very generic model, you can use different input files for defining different setups and always use the same program.
- You can use the parameter file in order to back up parameters that you used for a certain model run.

4.3.2 Changing 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"
# Compile-time specified parameters:
[ Implicit ]
EnableHints = "0"
MassUpwindWeight = "1"
MaxTimeStepDivisions = "10"
MobilityUpwindWeight = "1"
```

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

```
NumericDifferenceMethod = "1"
UseTwoPointFlux = "0"
[ LinearSolver ]
MaxIterations = "250"
PreconditionerRelaxation = "1"
ResidualReduction = "1e-06"
Verbosity = "0"
[ Newton ]
EnableResidualCriterion = "0"
EnableShiftCriterion = "1"
MaxRelativeShift = "1e-08"
MaxSteps = "18"
ResidualReduction = "1e-05"
SatisfyResidualAndShiftCriterion = "0"
TargetSteps = "10"
UseLineSearch = "0"
WriteConvergence = "0"
[ Problem ]
EnableGravity = "1"
[ TimeManager ]
MaxTimeStepSize = "1.79769e+308"
[ Vtk ]
AddVelocity = "0"
```

A number of things can be learned from this. Most prominently it tells you the parameters, that can easily be added to the input file without having to change anything in the source code. The output will tell you, which parameters are available to the problem and whether they have been specified

- *run-time* via your input file
- *compile-time* and have not been overwritten by the input file
- in your input file, but are *UNUSED* by the simulation

For example by adding

```
[ Newton ]
MaxRelativeShift = "1e-11"
```

to the input file you can specify that the Newton solver considers itself converged for an error a thousand times smaller.

The *UNUSED* warning

```
# UNUSED parameters:
Problem.ImportantVariable = "42"
```

is important, because it shows that the application did not read in this value. Maybe because it was attributed to the wrong group or there was a typo. This feature is *very* useful for debugging or spotting typos, like when you wanted to overwrite one of the parameters listed under `Compile-time specified parameters` and misspelled it in the input file, it will be listed in the `UNUSED parameters` section.

4.3.3 Technical Issues on Parameters

In case you want to learn more about how the input files work, please have a look at the very helpful DUNE documentation, look for `Dune::ParameterTree`.

The parameter tree can also be filled without the help of a text file. Everything that is specified in a DuMu^x input file can also be specified directly on the command line. If there is also an input file, the respective parameter on the command line has precedence.

All applications have a help message which you can read by giving `--help` as a command line argument to the application. A message listing syntax and the mandatory input will be displayed on the command line.

4.4 Restart DuMu^x Simulations

Using the restart capability of DuMu^x can be advantageous for computationally expensive or time consuming simulations, because you can restart the simulation from a specific point in time and e.g. 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 -ParameterFile test_program.input -TimeManager.Restart
   RESTART_TIME
```

To the test restart behavior e.g. use the `test_box1p2cni` problem in the `test/implicit/1p2c` folder. You get the `RESTART_TIME` from the name of your `.drs` file. Please note, that restarting will only work by giving a exact time from an existing restart file. Depending on your type of model, you should get a `.drs` file every 5th or 10th time step. If this not frequently enough, you can change it by using the following function into your problem header:

```
1 /*!
2  * \brief Returns true if a restart file should be written to
3  *        disk.
4  */
5 bool shouldWriteRestartFile() const
6 {
7   return true;
8 }
```

4.5 Coding Guidelines

An important characteristic of source code is that it is written only once but usually it is read many times (e.g. when debugging things, adding features, etc.). For this reason, good programming frameworks always aim to be as readable as possible, even if comes with higher effort to write them in the first place. The remainder of this section is almost a verbatim copy of the DUNE coding guidelines found at `http://www.dune-project.org/doc/devel/codingstyle.html`. These guidelines are also recommended for coding with DuMu^x as developer and user.

In order to keep the code maintainable we have decided upon a set of coding rules. Some of them may seem like splitting hairs to you, but they do make it much easier for everybody to work on code that hasn't been written by oneself.

Documentation: DuMu^x, as any software project of similar complexity, will stand and fall with the quality of its documentation. Therefore it is of paramount importance that you document well everything you do! We use the Doxygen system to extract easily-readable documentation from the source code. Please use its syntax everywhere.
We proclaim the Doc-Me Dogma. It goes like this: Whatever you do, and in whatever hurry you happen to be, please document everything at least with a `/** \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 document freely what each part of your code does. All comments/ documentation is in **English**. In particular, please comment **all**:

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

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

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

Naming: In order to avoid duplicated types or variables a better understanding and usability of the code we have the following naming principles.

- **Variables/Functions...**
  - *use* letters and digits
  - *first letter* is lower case.
  - *CamelCase*: if your variable names consists of several words, then the first letter of each new word should be capital.
  - *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 →
  - *Self-Explaining*: especially abbreviations should be avoided (saturation in stead of S)
- **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.

Exceptions: The use of exceptions for error handling is encouraged. Until further notice, all exceptions thrown are Dune exceptions.

Debugging Code: Global debugging code is switched off by setting the macro NDEBUG or the compiler flag -DNDEBUG. In particular, all asserts are automatically removed. Use those asserts freely!

5 The DuMu^x Property System

This section is dedicated to the DuMu^x property system. First, a high level overview over its design and principle ideas is given, then follows a short reference and a short self-contained example.

5.1 Concepts and Features of the DuMu^x Property System

The DuMu^x property system was designed as an attempt to mitigate the problems of traits classes. In fact, 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, which means that there are no run-time performance penalties associated with it. It is based on the following concepts:

Property: 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 can be seen as a label with its name.

Property Inheritance: 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.

5.2 DuMu^x Property System Reference

All source files which use the DuMu^x 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

```
1 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);
```

```
17
18 SET_BOOL_PROP(MyTypeTag, MyBoolValue, true);
19 SET_INT_PROP(MyTypeTag, MyIntValue, 12345);
20 SET_SCALAR_PROP(MyTypeTag, MyScalarValue, 12345.67890);
21 }}
```

Un-setting Properties

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

```
1 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:

```
1 namespace Dumux {
2 namespace Properties {
3 NEW_TYPE_TAG(BaseTypeTag);
4 NEW_TYPE_TAG(DerivedTypeTag, INHERITS_FROM(BaseTypeTag));
5
6 NEW_PROP_TAG(TestProp);
7
8 SET_TYPE_PROP(BaseTypeTag, TestProp, int);
9 UNSET_PROP(DerivedTypeTag, TestProp);
10 // trying to access the 'TestProp' property for 'DerivedTypeTag'
11 // will trigger a compiler error!
12 }}
```

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

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

or using the convenience macros

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

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

```
1 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

```
1 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;
};
```

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;
};
```

5.3 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 5.1a.

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 5.1b lists a few property names which make sense for at least one of the nodes of Figure 5.1a. 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]
```

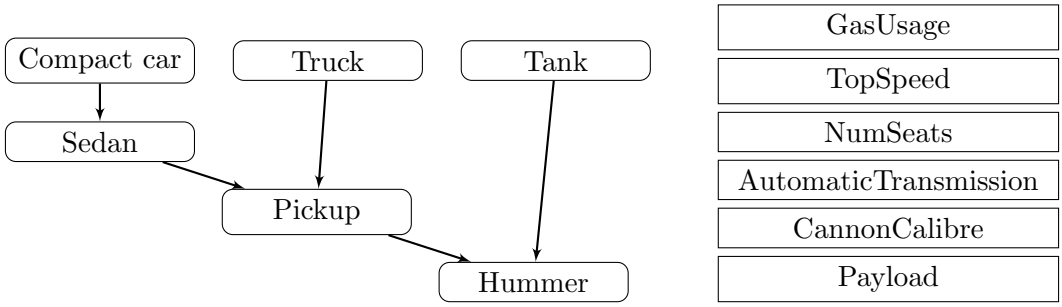


Figure 5.1: **(a)** A possible property inheritance graph for various kinds of cars. The lower nodes inherit from higher ones; Inherited properties from nodes on the right take precedence over the properties defined on the left. **(b)** Property names which make sense for at least one of the car types of (a).

```
16 NEW_PROP_TAG(AutomaticTransmission); // true/false
17 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

```
18 SET_INT_PROP(CompactCar, TopSpeed, GET_PROP_VALUE(TypeTag, GasUsage) * 30);
19 SET_INT_PROP(CompactCar, NumSeats, 5);
20 SET_INT_PROP(CompactCar, GasUsage, 4);
21
22 SET_INT_PROP(Truck, TopSpeed, 100);
23 SET_INT_PROP(Truck, NumSeats, 2);
24 SET_INT_PROP(Truck, GasUsage, 18);
25 SET_INT_PROP(Truck, Payload, 35);
26
```

```
27 SET_INT_PROP(Tank, TopSpeed, 60);
28 SET_INT_PROP(Tank, GasUsage, 65);
29 SET_INT_PROP(Tank, CanonCaliber, 120);
30
31 SET_INT_PROP(Sedan, GasUsage, 7);
32 SET_BOOL_PROP(Sedan, AutomaticTransmission, true);
33
34 SET_INT_PROP(Pickup, TopSpeed, 120);
35 SET_INT_PROP(Pickup, Payload, 5);
36
37 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

```
38 UNSET_PROP(HummerH1, CanonCaliber);
39
40 }} // close namespaces
```

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

```
41 int main()
42 {
43     std::cout << "top speed of sedan: " << GET_PROP_VALUE(TTAG(Sedan), TopSpeed) << "\n";
44     std::cout << "top speed of truck: " << GET_PROP_VALUE(TTAG(Truck), TopSpeed) << "\n";
45
46     std::cout << PROP_DIAGNOSTIC(TTAG(Sedan), TopSpeed);
47     std::cout << PROP_DIAGNOSTIC(TTAG(HummerH1), CanonCaliber);
48
49     Dumux::Properties::print<TTAG(Sedan)>();
50 }
```

will yield the following output:

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

6 The DuMu^x Fluid Framework

This chapter discusses the DuMu^x fluid framework. DuMu^x users who do not want to write new models and who do not need new fluid configurations may skip this chapter.

In the chapter, a high level overview over the the principle concepts is provided first, then some implementation details follow.

6.1 Overview of the Fluid Framework

The DuMu^x fluid framework currently features the following concepts (listed roughly in their order of importance):

Fluid state: Fluid states are responsible for representing the complete thermodynamic configuration of a system at a given spatial and temporal position. A fluid state always provides access methods to **all** thermodynamic quantities, but the concept of a fluid state does not mandate what assumptions are made to store these thermodynamic quantities. What fluid states also do **not** do is to make sure that the thermodynamic state which they represent is physically possible.

Fluid system: Fluid systems express the thermodynamic **relations**[1] between quantities. Since functions do not exhibit any internal state, fluid systems are stateless classes, i.e. all member functions are `static`. This is a conscious decision since the thermodynamic state of the system is expressed by a fluid state!

Parameter cache: Fluid systems sometimes require computationally expensive parameters for multiple relations. Such parameters can be cached using a so-called parameter cache. Parameter cache objects are specific for each fluid system but they must provide a common interface to update the internal parameters depending on the quantities which changed since the last update.

Constraint solver: Constraint solvers are auxiliary tools to make sure that a fluid state is consistent with some thermodynamic constraints. All constraint solvers specify a well defined set of input variables and make sure that the resulting fluid state is consistent with a given set of thermodynamic equations. See section 6.4 for a detailed description of the constraint solvers which are currently available in DuMu^x.

Equation of state: Equations of state (EOS) are auxiliary classes which provide relations between a fluid phase's temperature, pressure, composition and density. Since these classes are only used internally in fluid systems, their programming interface is currently ad-hoc.

Component: Components are fluid systems which provide the thermodynamic relations for the liquid and gas phase of a single chemical species or a fixed mixture of species. Their main purpose is to

[1] Strictly speaking, these relations are functions, mathematically.

provide a convenient way to access these quantities from full-fledged fluid systems. Components are not supposed to be used by models directly.

Binary coefficient: Binary coefficients describe the relations of a mixture of two components. Typical binary coefficients are HENRY coefficients or binary molecular diffusion coefficients. So far, the programming interface for accessing binary coefficients has not been standardized in DuMu^x.

6.2 Fluid States

Fluid state objects express the complete thermodynamic state of a system at a given spatial and temporal position.

6.2.1 Exported Constants

All fluid states **must** export the following constants:

numPhases: The number of fluid phases considered.

numComponents: The number of considered chemical species or pseudo-species.

6.2.2 Accessible Thermodynamic Quantities

Also, **all** fluid states **must** provide the following methods:

temperature(): The absolute temperature T_α of a fluid phase α.

pressure(): The absolute pressure p_α of a fluid phase α.

saturation(): The saturation S_α of a fluid phase α. The saturation is defined as the pore space occupied by the fluid divided by the total pore space:

$$S_\alpha := \frac{\phi \mathcal{V}_\alpha}{\phi \mathcal{V}}$$

moleFraction(): Returns the molar fraction x_α^κ of the component κ in fluid phase α. The molar fraction x_α^κ is defined as the ratio of the number of molecules of component κ and the total number of molecules of the phase α.

massFraction(): Returns the mass fraction X_α^κ of component κ in fluid phase α. The mass fraction X_α^κ is defined as the weight of all molecules of a component divided by the total mass of the fluid phase. It is related with the component's mole fraction by means of the relation

$$X_\alpha^\kappa = x_\alpha^\kappa \frac{M^\kappa}{\overline{M}_\alpha} ,$$

where M^κ is the molar mass of component κ and $\overline{M}_\alpha$ is the mean molar mass of a molecule of phase α.

averageMolarMass(): Returns $\overline{M}_\alpha$, the mean molar mass of a molecule of phase α. For a mixture of $N > 0$ components, $\overline{M}_\alpha$ is defined as

$$\overline{M}_\alpha = \sum_{\kappa=1}^{N} x_\alpha^\kappa M^\kappa$$

density(): Returns the density ρ_α of the fluid phase α.

molarDensity(): Returns the molar density $\rho_{mol,\alpha}$ of a fluid phase α. The molar density is defined by the mass density ρ_α and the mean molar mass $\overline{M}_\alpha$:

$$\rho_{mol,\alpha} = \frac{\rho_\alpha}{\overline{M}_\alpha} \ .$$

molarVolume(): Returns the molar volume $v_{mol,\alpha}$ of a fluid phase α. This quantity is the inverse of the molar density.

molarity(): Returns the molar concentration c_α^κ of component κ in fluid phase α.

fugacity(): Returns the fugacity f_α^κ of component κ in fluid phase α. The fugacity is defined as

$$f_\alpha^\kappa := \Phi_\alpha^\kappa x_\alpha^\kappa p_\alpha \ ,$$

where Φ_α^κ is the *fugacity coefficient* [23]. The physical meaning of fugacity becomes clear from the equation

$$f_\alpha^\kappa = p_\alpha \exp\left\{\frac{\zeta_\alpha^\kappa}{RT_\alpha}\right\} \ ,$$

where ζ_α^κ represents the κ's chemical potential in phase α, R stands for the ideal gas constant, and T_α for the absolute temperature of phase α. Assuming thermal equilibrium, there is a one-to-one mapping between a component's chemical potential ζ_α^κ and its fugacity f_α^κ. In this case chemical equilibrium can thus be expressed by

$$f^\kappa := f_\alpha^\kappa = f_\beta^\kappa \quad \forall \alpha, \beta$$

fugacityCoefficient(): Returns the fugacity coefficient Φ_α^κ of component κ in fluid phase α.

enthalpy(): Returns specific enthalpy h_α of a fluid phase α.

internalEnergy(): Returns specific internal energy u_α of a fluid phase α. The specific internal energy is defined by the relation

$$u_\alpha = h_\alpha - \frac{p_\alpha}{\rho_\alpha}$$

viscosity(): Returns the dynamic viscosity μ_α of fluid phase α.

6.2.3 Available Fluid States

Currently, the following fluid states are provided by DuMu^x:

NonEquilibriumFluidState: This is the most general fluid state supplied. It does not assume thermodynamic equilibrium and thus stores all phase compositions (using mole fractions), fugacity coefficients, phase temperatures, phase pressures, saturations and specific enthalpies.

CompositionalFluidState: This fluid state is very similar to the `NonEquilibriumFluidState` with the difference that the `CompositionalFluidState` assumes thermodynamic equilibrium. In the context of multi-phase flow in porous media, this means that only a single temperature needs to be stored.

ImmisicibleFluidState: This fluid state assumes that the fluid phases are immiscible, which implies that the phase compositions and the fugacity coefficients do not need to be stored explicitly.

PressureOverlayFluidState: This is a so-called *overlay* fluid state. It allows to set the pressure of all fluid phases but forwards everything else to another fluid state.

SaturationOverlayFluidState: This fluid state is like the `PressureOverlayFluidState`, except that the phase saturations are settable instead of the phase pressures.

TempeatureOverlayFluidState: This fluid state is like the `PressureOverlayFluidState`, except that the temperature is settable instead of the phase pressures. Note that this overlay state assumes thermal equilibrium regardless of underlying fluid state.

CompositionOverlayFluidState: This fluid state is like the `PressureOverlayFluidState`, except that the phase composition is settable (in terms of mole fractions) instead of the phase pressures.

6.3 Fluid Systems

Fluid systems express the thermodynamic relations between the quantities of a fluid state.

6.3.1 Parameter Caches

All fluid systems must export a type for their `ParameterCache` objects. Parameter caches can be used to cache parameter that are expensive to compute and are required in multiple thermodynamic relations. For fluid systems which do need to cache parameters, DuMu^x provides a `NullParameterCache` class.

The actual quantities stored by parameter cache objects are specific to the fluid system and no assumptions on what they provide should be made outside of their fluid system. Parameter cache objects provide a well-defined set of methods to make them coherent with a given fluid state, though. These update are:

updateAll(fluidState, except): Update all cached quantities for all phases. The `except` argument contains a bit field of the quantities which have not been modified since the last call to a `update()` method.

updateAllPresures(fluidState): Update all cached quantities which depend on the pressure of any fluid phase.

updateAllTemperatures(fluidState): Update all cached quantities which depend on temperature of any fluid phase.

updatePhase(fluidState, phaseIdx, except): Update all cached quantities for a given phase. The quantities specified by the `except` bit field have not been modified since the last call to an `update()` method.

updateTemperature(fluidState, phaseIdx): Update all cached quantities which depend on the temperature of a given phase.

updatePressure(fluidState, phaseIdx): Update all cached quantities which depend on the pressure of a given phase.

updateComposition(fluidState, phaseIdx): Update all cached quantities which depend on the composition of a given phase.

updateSingleMoleFraction(fluidState, phaseIdx, compIdx): Update all cached quantities which depend on the value of the mole fraction of a component in a phase.

Note, that the parameter cache interface only guarantees that if a more specialized `update()` method is called, it is not slower than the next more-general method (e.g. calling `updateSingleMoleFraction()` may be as expensive as `updateAll()`). It is thus advisable to rather use a more general `update()` method once than multiple calls to specialized `update()` methods.

To make usage of parameter caches easier for the case where all cached quantities ought to be recalculated if a quantity of a phase was changed, it is possible to only define the `updatePhase()` method and derive the parameter cache from `Dumux::ParameterCacheBase`.

6.3.2 Exported Constants and Capabilities

Besides providing the type of their `ParameterCache` objects, fluid systems need to export the following constants and auxiliary methods:

numPhases: The number of considered fluid phases.

numComponents: The number of considered chemical (pseudo-) species.

init(): Initialize the fluid system. This is usually used to tabulate some quantities

phaseName(): Given the index of a fluid phase, return its name as human-readable string.

componentName(): Given the index of a component, return its name as human-readable string.

isLiquid(): Return whether the phase is a liquid, given the index of a phase.

isIdealMixture(): Return whether the phase is an ideal mixture, given the phase index. In the context of the DuMu^x fluid framework a phase α is an ideal mixture if, and only if, all its fugacity coefficients Φ_α^κ do not depend on the phase composition. (Although they might very well depend on temperature and pressure.)

isIdealGas(): Return whether a phase α is an ideal gas, i.e. it adheres to the relation

$$p_\alpha v_{mol,\alpha} = RT_\alpha \ ,$$

with R being the ideal gas constant.

isCompressible(): Return whether a phase α is compressible, i.e. its density depends on pressure p_α.

molarMass(): Given a component index, return the molar mass of the corresponding component.

6.3.3 Thermodynamic Relations

Fluid systems have been explicitly designed to provide as few thermodynamic relations as possible. A full-fledged fluid system thus only needs to provide the following thermodynamic relations:

density(): Given a fluid state, an up-to-date parameter cache and a phase index, return the mass density ρ_α of the phase.

fugacityCoefficient(): Given a fluid state, an up-to-date parameter cache as well as a phase and a component index, return the fugacity coefficient Φ_α^κ of a the component for the phase.

viscosity(): Given a fluid state, an up-to-date parameter cache and a phase index, return the dynamic viscosity μ_α of the phase.

diffusionCoefficient(): Given a fluid state, an up-to-date parameter cache, a phase and a component index, return the calculate the molecular diffusion coefficient for the component in the fluid phase.

Molecular diffusion of a component κ in phase α is caused by a gradient of the chemical potential and follows the law

$$J_\alpha^\kappa = -D_\alpha^\kappa \ \mathbf{grad} \zeta_\alpha^\kappa \ ,$$

where ζ_α^κ is the component's chemical potential, D_α^κ is the diffusion coefficient and J_α^κ is the diffusive flux. ζ_α^κ is connected to the component's fugacity f_α^κ by the relation

$$\zeta_\alpha^\kappa = RT_\alpha \ln \frac{f_\alpha^\kappa}{p_\alpha} \ .$$

binaryDiffusionCoefficient(): Given a fluid state, an up-to-date parameter cache, a phase index and two component indices, return the binary diffusion coefficient for the binary mixture. This method is less general than `diffusionCoefficient` method, but relations can only be found for binary diffusion coefficients in the literature.

enthalpy(): Given a fluid state, an up-to-date parameter cache and a phase index, this method calulates the specific enthalpy h_α of the phase.

thermalConductivity: Given a fluid state, an up-to-date parameter cache and a phase index, this method returns the thermal conductivity λ_α of the fluid phase. The thermal conductivity is defined by means of the relation

$$\dot{Q} = \lambda_\alpha \mathbf{grad} \ T_\alpha \ ,$$

where $\dot{Q}$ is the heat flux caused by the temperature gradient $\mathbf{grad} \ T_\alpha$.

heatCapacity(): Given a fluid state, an up-to-date parameter cache and a phase index, this method computes the isobaric heat capacity $c_{p,\alpha}$ of the fluid phase. The isobaric heat capacity is defined as the partial derivative of the specific enthalpy h_α to the fluid pressure:

$$c_{p,\alpha} = \frac{\partial h_\alpha}{\partial p_\alpha}$$

Fluid systems may chose not to implement some of these methods and throw an exception of type `Dune::NotImplemented` instead. Obviously, such fluid systems cannot be used for models that depend on those methods.

6.3.4 Available Fluid Systems

Currently, the following fluid systems are available in DuMu^x:

Dumux::FluidSystems::TwoPImmiscible: A two-phase fluid system which assumes immiscibility of the fluid phases. The fluid phases are thus completely specified by means of their constituting components. This fluid system is intended to be used with models that assume immiscibility.

Dumux::FluidSystems::H2ON2: A two-phase fluid system featuring gas and liquid phases and distilled water (H_2O) and pure molecular Nitrogen (N_2) as components.

Dumux::FluidSystems::H2OAir: A two-phase fluid system featuring gas and liquid phases and distilled water (H_2O) and air (Pseudo component composed of 79% N_2, 20% O_2 and 1% Ar) as components.

Dumux::FluidSystems::H2OAirMesitylene: A three-phase fluid system featuring gas, NAPL and water phases and distilled water, air and Mesitylene ($C_6H_3(CH_3)_3$) as components. This fluid system assumes all phases to be ideal mixtures.

Dumux::FluidSystems::H2OAirXylene: A three-phase fluid system featuring gas, NAPL and water as phases and distilled water, air and Xylene (C_8H_{10}) as components. This fluid system assumes all phases to be ideal mixtures.

Dumux::FluidSystems::Spe5: A three-phase fluid system featuring gas, oil and water as phases and the seven components distilled water, Methane (C_1), Propane (C_3), Pentane (C_5), Heptane (C_7), Decane (C_{10}), Pentadecane (C_{15}) and Icosane (C_{20}). For the water phase the IAPWS-97 formulation is used as equation of state, while for the gas and oil phases a Peng-Robinson equation of state with slightly modified parameters is used. This fluid system is highly non-linear, and the gas and oil phases also cannot be considered ideal mixtures[21].

6.4 Constraint Solvers

Constraint solvers connect the thermodynamic relations expressed by fluid systems with the thermodynamic quantities stored by fluid states. Using them is not mandatory for models, but given the fact that some thermodynamic constraints can be quite complex to solve, sharing this code between models makes sense. Currently, DuMu^x provides the following constraint solvers:

CompositionFromFugacities: This constraint solver takes all component fugacities, the temperature and pressure of a phase as input and calculates the composition of the fluid phase. This means that the thermodynamic constraints used by this solver are

$$f^\kappa = \Phi^\kappa_\alpha(\{x^\beta_\alpha\}, T_\alpha, p_\alpha) p_\alpha x^\kappa_\alpha \ ,$$

where f^κ, T_α and p_α are fixed values.

ComputeFromReferencePhase: This solver brings all fluid phases into thermodynamic equilibrium with a reference phase β, assuming that all phase temperatures and saturations have already been set. The constraints used by this solver are thus

$$\begin{aligned} f^\kappa_\beta = f^\kappa_\alpha = \Phi^\kappa_\alpha(\{x^\beta_\alpha\}, T_\alpha, p_\alpha) p_\alpha x^\kappa_\alpha \ , \\ p_\alpha = p_\beta + p_{c\beta\alpha} \ , \end{aligned}$$

where $p_{c\beta\alpha}$ is the capillary pressure between the fluid phases β and α.

CompositionalFlash: A compositional 2p2c flash solver for the sequential models in DuMu^x. Input is temperature, phase pressures and feed mass fraction, the solver computes the compositional variables and saturations.

NcpFlash: This is a so-called flash solver. A flash solver takes the total mass of all components per volume unit and the phase temperatures as input and calculates all phase pressures, saturations and compositions. This flash solver works for an arbitrary number of phases $M > 0$ and components $N \geq M - 1$. In this case, the unknown quantities are the following:

- M pressures p_α
- M saturations S_α
- $M \cdot N$ mole fractions x^κ_α

This sums up to $M \cdot (N + 2)$. The equations side of things provides:

- $(M - 1) \cdot N$ equations stemming from the fact that the fugacity of any component is the same in all phases, i.e.
$$f^\kappa_\alpha = f^\kappa_\beta$$
holds for all phases α, β and all components κ.
- 1 equation comes from the fact that the whole pore space is filled by some fluid, i.e.
$$\sum_{\alpha=1}^{M} S_\alpha = 1$$
- $M - 1$ constraints are given by the capillary pressures:
$$p_\beta = p_\alpha + p_{c\beta\alpha} \ ,$$
for all phases α, β

- N constraints come the fact that the total mass of each component is given:

$$c_{tot}^{\kappa} = \sum_{\alpha=1}^{M} x_{\alpha}^{\kappa} \, \rho_{mol,\alpha} = const$$

- And finally M model assumptions are used. This solver uses the NCP constraints proposed in [22]:

$$0 = \min\{S_{\alpha}, 1 - \sum_{\kappa=1}^{N} x_{\alpha}^{\kappa}\}$$

The number of equations also sums up to $M \cdot (N + 2)$. Thus, the system of equations is closed.

ImmiscibleFlash: This is a flash solver assuming immiscibility of the phases. It is similar to the `NcpFlash` solver but a lot simpler.

MiscibleMultiphaseComposition: This solver calculates the composition of all phases provided that each of the phases is potentially present. Currently, this solver does not support non-ideal mixtures.

7 Physical and Numerical Models Available in DuMu^x

7.1 Physical and Mathematical Description

Characteristic of compositional multiphase models is that the phases are not only matter of a single chemical substance. Instead, their composition in general includes several species, and for the mass transfer, the component behavior is quite different from the phase behavior. In the following, we give some basic definitions and assumptions that are required for the formulation of the model concept below. As an example, we take a three-phase three-component system water-NAPL-gas [8]. The modification for other multicomponent systems is straightforward and can be found, e. g., in [6, 1].

7.1.1 Basic Definitions and Assumptions for the Compositional Model Concept

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), air (a), and the organic contaminant (c) (see Fig. 7.1).

Equilibrium: For the non-isothermal multiphase processes in porous media under consideration, we state that the assumption of local thermal equilibrium is valid since flow velocities are small. We neglect chemical reactions and biological decomposition and assume chemical equilibrium. 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 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.

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)		

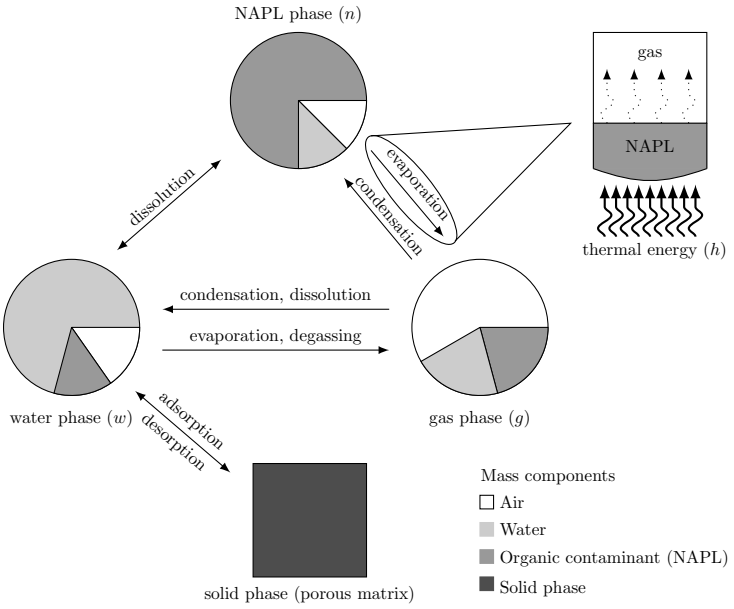


Figure 7.1: Mass and energy transfer between the phases

7.1.2 Balance Equations

For the balance equations for multicomponent systems, it is in many cases convenient to use a molar formulation of the continuity equation. Considering the mass conservation for each component allows us to drop source/sink terms for describing the mass transfer between phases. Then, the molar mass balance can be written as:

$$\phi \frac{\partial(\sum_\alpha \varrho_{\text{mol},\alpha} x_\alpha^\kappa S_\alpha)}{\partial t} - \sum_\alpha \operatorname{div}\left\{\frac{k_{\text{r}\alpha}}{\mu_\alpha}\varrho_{\text{mol},\alpha} x_\alpha^\kappa \mathbf{K}(\mathbf{grad}\, p_\alpha - \varrho_\alpha \boldsymbol{g})\right\} - \sum_\alpha \operatorname{div}\left\{\tau\phi S_\alpha D_\alpha^\kappa \varrho_{\text{mol},\alpha}\, \mathbf{grad}\, x_\alpha^\kappa\right\} - q^\kappa = 0, \qquad \kappa \in \{\text{w,a,c}\}. \quad (7.1)$$

The component mass balance can also be written in terms of mass fractions by replacing molar densities by mass densities and mole by mass fractions. To obtain a single conserved quantity in the temporal derivative, the total concentration, representing the mass of one component per unit volume, is defined as

$$C^\kappa = \sum_\alpha \phi S_\alpha \varrho_{\text{mass},\alpha} X_\alpha^\kappa \,.$$

Using this definition, the component mass balance is written as:

$$\frac{\partial C^\kappa}{\partial t} = \sum_\alpha \operatorname{div}\left\{\frac{k_{\text{r}\alpha}}{\mu_\alpha}\varrho_{\text{mass},\alpha} X_\alpha^\kappa \mathbf{K}(\mathbf{grad}\, p_\alpha + \varrho_{\text{mass},\alpha}\boldsymbol{g})\right\} + \sum_\alpha \operatorname{div}\left\{\tau\phi S_\alpha D_\alpha^\kappa \varrho_{\text{mass},\alpha}\frac{M^\kappa}{M_\alpha}\,\mathbf{grad}\, x_\alpha^\kappa\right\} + q^\kappa = 0, \qquad \kappa \in \{\text{w,a,c}\}. \quad (7.2)$$

In the case of non-isothermal systems, we further have to balance the thermal energy. We assume fully reversible processes, such that entropy is not needed as a model parameter. Furthermore, we neglect dissipative effects and the heat transport due to molecular diffusion. The energy balance can then be formulated as:

$$\phi \frac{\partial \left(\sum_\alpha \varrho_\alpha u_\alpha S_\alpha\right)}{\partial t} + (1-\phi)\frac{\partial \varrho_s c_s T}{\partial t} - \operatorname{div}\left(\lambda_{\text{pm}} \operatorname{\mathbf{grad}} T\right) - \sum_\alpha \operatorname{div}\left\{\frac{k_{r\alpha}}{\mu_\alpha}\varrho_\alpha h_\alpha K\left(\operatorname{\mathbf{grad}} p_\alpha - \varrho_\alpha \boldsymbol{g}\right)\right\} - q^h = 0. \quad (7.3)$$

In order to close the system, supplementary constraints for capillary pressure, saturations and mole fractions are needed, [20]. According to the Gibbsian phase rule, the number of degrees of freedom in a non-isothermal compositional multiphase system is equal to the number of components plus one. This means we need as many independent unknowns in the system description. The available primary variables are, e. g., saturations, mole/mass fractions, temperature, pressures, etc.

7.2 Implicit 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 section.

7.2.1 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 7.2a).

The FE mesh divides the box B_i into subcontrolvolumes (scv's) b_i^k (see Figure 7.2b). Figure 7.2c 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 7.2c).

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 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:

$$f(\tilde{u}(x_{ij}^k)) \cdot \mathbf{n}_{ij}^k \, |e_{ij}^k| \qquad \text{with} \qquad \tilde{u}(x_{ij}^k) = \sum_i N_i(x_{ij}^k) \cdot \hat{u}_i. \quad (7.4)$$

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

$$\underbrace{\int_G \frac{\partial}{\partial t} u \, dG}_{1} + \underbrace{\int_{\partial G} (\mathbf{v}u + \mathbf{w}) \cdot \mathbf{n} \, d\Gamma}_{2} = \underbrace{\int_G q \, dG}_{3} \quad (7.5)$$

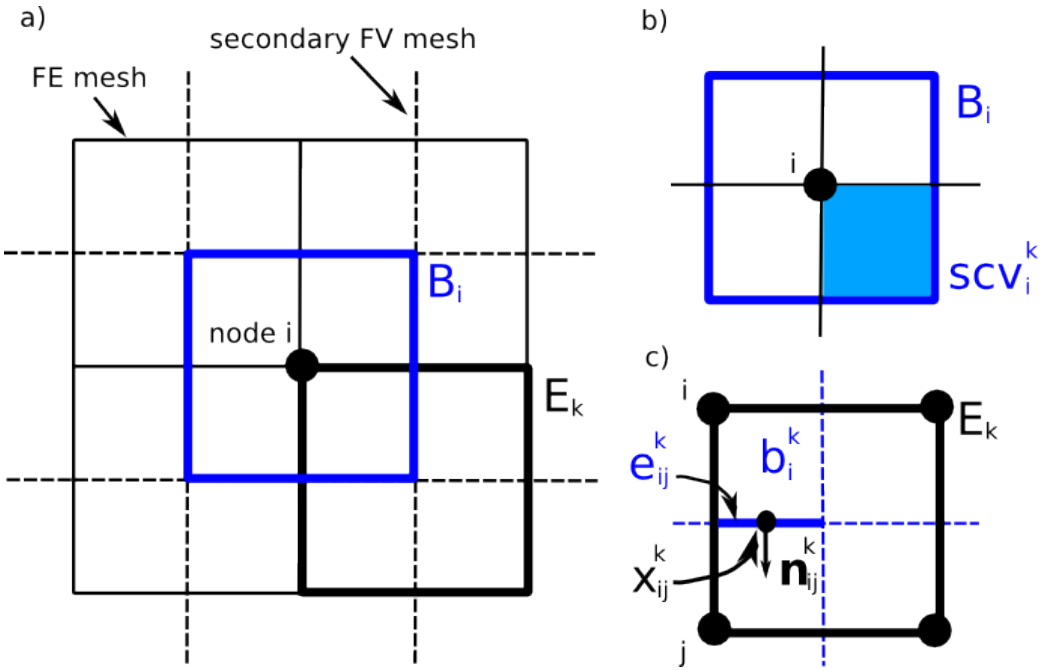


Figure 7.2: Discretization of the box method

$$f(u) = \int_G \frac{\partial u}{\partial t}\, dG + \int_G \nabla \cdot \underbrace{[\mathbf{v}u + \mathbf{w}(u)]}_{F(u)}\, dG - \int_G q\, dG = 0 \tag{7.6}$$

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

$$\tilde{p} = \sum_i N_i \hat{p}_i \tag{7.7}$$

$$\tilde{\mathbf{v}} = \sum_i N_i \hat{\mathbf{v}} \tag{7.8}$$

$$\tilde{x}^\kappa = \sum_i N_i \hat{x}^\kappa \tag{7.9}$$

$$\nabla\tilde{p} = \sum_i \nabla N_i \hat{p}_i \tag{7.10}$$

$$\nabla\tilde{\mathbf{v}} = \sum_i \nabla N_i \hat{\mathbf{v}} \tag{7.11}$$

$$\nabla\tilde{x}^\kappa = \sum_i \nabla N_i \hat{x}^\kappa. \tag{7.12}$$

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

$$f(u) = 0 \qquad \Rightarrow \qquad f(\tilde{u}) = \varepsilon \tag{7.13}$$

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,

$$\int_G W_j \cdot \varepsilon \stackrel{!}{=} 0 \qquad \text{with} \qquad \sum_j W_j = 1 \tag{7.14}$$

yields the following equation:

$$\int_G W_j \frac{\partial \tilde{u}}{\partial t}\, dG + \int_G W_j \cdot [\nabla \cdot F(\tilde{u})]\; dG - \int_G W_j \cdot q\, dG = \int_G W_j \cdot \varepsilon\, dG \stackrel{!}{=} 0. \tag{7.15}$$

Then, the chain rule and the GREEN-GAUSSIAN integral theorem are applied.

$$\int_G W_j \frac{\partial \sum_i N_i \hat{u}_i}{\partial t}\, dG + \int_{\partial G} [W_j \cdot F(\tilde{u})] \cdot \mathbf{n}\, d\Gamma_G + \int_G \nabla W_j \cdot F(\tilde{u})\, dG - \int_G W_j \cdot q\, dG = 0 \tag{7.16}$$

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 $M_{i,j}^{lump}$ which is defined as:

$$M_{i,j}^{lump} = \begin{cases} \int_G W_j\, dG = \int_G N_i\, dG = V_i & i = j \\ 0 & i \neq j \end{cases} \tag{7.17}$$

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

$$V_i \frac{\partial \hat{u}_i}{\partial t} + \int_{\partial G} [W_j \cdot F(\tilde{u})] \cdot \mathbf{n}\, d\Gamma_G + \int_G \nabla W_j \cdot F(\tilde{u})\, dG - V_i \cdot q = 0\,. \tag{7.18}$$

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

$$W_j(x) = \begin{cases} 1 & x \in B_i \\ 0 & x \notin B_i \end{cases} \tag{7.19}$$

causes $\nabla W_j = 0$:

$$V_i \frac{\partial \hat{u}_i}{\partial t} + \int_{\partial B_i} [W_j \cdot F(\tilde{u})] \cdot \mathbf{n}\, d\Gamma_{B_i} - V_i \cdot q = 0. \tag{7.20}$$

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

$$V_i \frac{\hat{u}_i^{n+1} - \hat{u}_i^n}{\Delta t} + \int_{\partial B_i} F(\tilde{u}^{n+1}) \cdot \mathbf{n}\, d\Gamma_{B_i} - V_i\, q^{n+1} = 0 \tag{7.21}$$

7.2.2 Cell Centered Finite Volume Method – A Short Introduction

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 7.3). 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

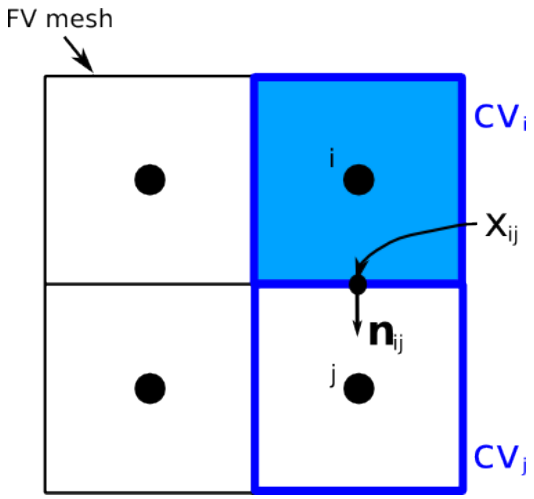


Figure 7.3: Discretization of the cell centered finite volume method

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).

7.3 Available Models

The following description of the available models is automatically extracted from the Doxygen documentation.

7.3.1 Fully-Implicit Models

The fully-implicit models described in this section are using the box or the cell centered finite volume method as described in section 7.2.1 and 7.2.2 for spatial and the implicit Euler method as temporal discretization. The models themselves are located in subdirectories of `dumux/implicit` of the DuMu^x distribution.

The Single-Phase Model: OnePModel

Single-phase, isothermal flow model, which uses a standard Darcy approach as the equation for the conservation of momentum:

$$v = -\frac{\mathbf{K}}{\mu}\left(\mathbf{grad}\, p - \varrho \mathbf{g}\right)$$

and solves the mass continuity equation:

$$\phi \frac{\partial \varrho}{\partial t} + \operatorname{div} \left\{ -\varrho \frac{\mathbf{K}}{\mu} \left(\mathbf{grad}\, p - \varrho \mathbf{g} \right) \right\} = q,$$

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization. The model supports compressible as well as incompressible fluids.

The Single-Phase,Two-Component Model: OnePTwoCModel

This model implements a one-phase flow of a compressible fluid, that consists of two components, using a standard Darcy approach as the equation for the conservation of momentum:

$$v = -\frac{\mathbf{K}}{\mu} \left(\mathbf{grad}\, p - \varrho \mathbf{g} \right)$$

Gravity can be enabled or disabled via the property system. By inserting this into the continuity equation, one gets

$$\phi \frac{\partial \varrho}{\partial t} - \operatorname{div} \left\{ \varrho \frac{\mathbf{K}}{\mu} \left(\mathbf{grad}\, p - \varrho \mathbf{g} \right) \right\} = q \ ,$$

The transport of the components $\kappa \in \{w, a\}$ is described by the following equation:

$$\phi \frac{\partial \varrho X^\kappa}{\partial t} - \operatorname{div} \left\{ \varrho X^\kappa \frac{\mathbf{K}}{\mu} \left(\mathbf{grad}\, p - \varrho \mathbf{g} \right) + \varrho D^\kappa_{\mathrm{pm}} \frac{M^\kappa}{M_\alpha} \mathbf{grad} x^\kappa \right\} = q.$$

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization. The model is able to use either mole or mass fractions. The property useMoles can be set to either true or false in the problem file. Make sure that the according units are used in the problem setup. useMoles is set to true by default.

The primary variables are the pressure p and the mole or mass fraction of dissolved component x.

The Two-Phase Model Using the Richards Assumption: RichardsModel

In the unsaturated zone, Richards' equation

$$\frac{\partial\ \phi S_w \varrho_w}{\partial t} - \operatorname{div} \left\{ \varrho_w \frac{k_{rw}}{\mu_w}\ \mathbf{K}\ \left(\mathbf{grad} p_w - \varrho_w \mathbf{g} \right) \right\} = q_w,$$

is frequently used to approximate the water distribution above the groundwater level.

It can be derived from the two-phase equations, i.e.

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

where $\alpha \in \{w, n\}$ is the fluid phase, $\kappa \in \{w, a\}$ are the components, ρ_α is the fluid density, S_α is the fluid saturation, ϕ is the porosity of the soil, $k_{r\alpha}$ is the relative permeability for the fluid, μ_α is the fluid's dynamic viscosity, $\mathbf{K}$ is the intrinsic permeability, p_α is the fluid pressure and g is the potential of the gravity field.

In contrast to the full two-phase model, the Richards model assumes gas as the non-wetting fluid and that it exhibits a much lower viscosity than the (liquid) wetting phase. (For example at atmospheric pressure and at room temperature, the viscosity of air is only about 1% of the viscosity of liquid water.) As a consequence, the $\frac{k_{r\alpha}}{\mu_\alpha}$ term typically is much larger for the gas phase than for the wetting phase. For this reason, the Richards model assumes that $\frac{k_{rn}}{\mu_n}$ is infinitly large. This implies that the pressure of the gas phase is equivalent to the static pressure distribution and that therefore, mass conservation only needs to be considered for the wetting phase.

The model thus choses the absolute pressure of the wetting phase p_w as its only primary variable. The wetting phase saturation is calculated using the inverse of the capillary pressure, i.e.

$$S_w = p_c^{-1}(p_n - p_w)$$

holds, where p_n is a given reference pressure. Nota bene, that the last step is assumes that the capillary pressure-saturation curve can be uniquely inverted, so it is not possible to set the capillary pressure to zero when using the Richards model!

The Two-Phase MOdel: TwoPModel

This model implements two-phase flow of two immiscible fluids $\alpha \in \{w, n\}$ using a standard multiphase Darcy approach as the equation for the conservation of momentum, i.e.

$$v_\alpha = -\frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}\left(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g}\right)$$

By inserting this into the equation for the conservation of the phase mass, one gets

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

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization.

By using constitutive relations for the capillary pressure $p_c = p_n - p_w$ and relative permeability $k_{r\alpha}$ and taking advantage of the fact that $S_w + S_n = 1$, the number of unknowns can be reduced to two. Currently the model supports choosing either p_w and S_n or p_n and S_w as primary variables. The formulation which ought to be used can be specified by setting the `Formulation` property to either `TwoPCommonIndices::pWsN` or `TwoPCommonIndices::pNsW`. By default, the model uses p_w and S_n.

The Two-Phase, Two-Component Model: TwoPTwoCModel

This model implements two-phase two-component flow of two compressible and partially miscible fluids $\alpha \in \{w, n\}$ composed of the two components $\kappa \in \{w, a\}$. The standard multiphase Darcy approach is used as the equation for the conservation of momentum:

$$v_\alpha = -\frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}\left(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g}\right)$$

By inserting this into the equations for the conservation of the components, one gets one transport equation for each component

$$\phi\frac{\partial(\sum_\alpha \varrho_\alpha \frac{M^\kappa}{M_\alpha} x_\alpha^\kappa S_\alpha)}{\partial t} - \sum_\alpha \text{div}\left\{\varrho_\alpha \frac{M^\kappa}{M_\alpha} x_\alpha^\kappa \frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g})\right\}$$

$$- \sum_\alpha \text{div}\left\{D_{\alpha,\text{pm}}^\kappa \varrho_\alpha \frac{M^\kappa}{M_\alpha}\mathbf{grad} x_\alpha^\kappa\right\} - \sum_\alpha q_\alpha^\kappa = 0 \qquad \kappa \in \{w,a\}, \alpha \in \{w,g\}$$

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization.

By using constitutive relations for the capillary pressure $p_c = p_n - p_w$ and relative permeability $k_{r\alpha}$ and taking advantage of the fact that $S_w + S_n = 1$ and $x_w^\kappa + x_n^\kappa = 1$, the number of unknowns can be reduced to two. The used primary variables are, like in the two-phase model, either p_w and S_n or p_n and S_w. The formulation which ought to be used can be specified by setting the `Formulation` property to either TwoPTwoCIndices::pWsN or TwoPTwoCIndices::pNsW. By default, the model uses p_w and S_n. Moreover, the second primary variable depends on the phase state, since a primary variable switch is included. The phase state is stored for all nodes of the system. The model is able to use either mole or mass fractions. The property useMoles can be set to either true or false in the problem file. Make sure that the according units are used in the problem setup. useMoles is set to true by default. Following cases can be distinguished:

- Both phases are present: The saturation is used (either S_n or S_w, dependent on the chosen `Formulation`), as long as $0 < S_\alpha < 1$.
- Only wetting phase is present: The mole fraction of, e.g., air in the wetting phase x_w^a is used, as long as the maximum mole fraction is not exceeded ($x_w^a < x_{w,max}^a$)
- Only non-wetting phase is present: The mole fraction of, e.g., water in the non-wetting phase, x_n^w, is used, as long as the maximum mole fraction is not exceeded ($x_n^w < x_{n,max}^w$)

The CO2 Model: CO2Model

See TwoPTwoCModel for reference to the equations used. The CO2 model is derived from the 2p2c model. In the CO2 model the phase switch criterion is different from the 2p2c model. The phase switch occurs when the equilibrium concentration of a component in a phase is exceeded, instead of the sum of the components in the virtual phase (the phase which is not present) being greater that unity as done in the 2p2c model. The CO2VolumeVariables do not use a constraint solver for calculating the mole fractions as is the case in the 2p2c model. Instead mole fractions are calculated in the FluidSystem with a given temperature, pressurem and salinity. The model is able to use either mole or mass fractions. The property useMoles can be set to either true or false in the problem file. Make sure that the according units are used in the problem setup. useMoles is set to false by default.

The Three-Phase Model: ThreePModel

This model implements three-phase flow of three fluid phases $\alpha \in \{water, gas, NAPL\}$ The standard multiphase Darcy approach is used as the equation for the conservation of momentum.

By inserting this into the equations for the conservation of the components, the well-known multiphase flow equation is obtained.

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization.

The model uses commonly applied auxiliary conditions like $S_w + S_n + S_g = 1$ for the saturations. Furthermore, the phase pressures are related to each other via capillary pressures between the fluid phases, which are functions of the saturation, e.g. according to the approach of Parker et al.

The used primary variables are gas phase pressure p_g, water saturation S_w and NAPL saturation S_n.

The three-Phase, Three-Component Model: ThreePThreeCModel

This model implements three-phase three-component flow of three fluid phases $\alpha \in \{water, gas, NAPL\}$ each composed of up to three components $\kappa \in \{water, air, contaminant\}$. The standard multiphase Darcy approach is used as the equation for the conservation of momentum:

$$v_\alpha = -\frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}\left(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g}\right)$$

By inserting this into the equations for the conservation of the components, one transport equation for each component is obtained as

$$\phi\frac{\partial(\sum_\alpha \varrho_\alpha X_\alpha^\kappa S_\alpha)}{\partial t} - \sum_\alpha \mathrm{div}\left\{\frac{k_{r\alpha}}{\mu_\alpha}\varrho_\alpha x_\alpha^\kappa \mathbf{K}(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g})\right\}$$
$$-\sum_\alpha \mathrm{div}\left\{D_{\mathrm{pm}}^\kappa \varrho_\alpha \frac{M^\kappa}{M_\alpha}\mathbf{grad} x_\alpha^\kappa\right\} - q^\kappa = 0 \qquad \forall\kappa,\ \forall\alpha$$

Note that these balance equations are molar.

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization.

The model uses commonly applied auxiliary conditions like $S_w + S_n + S_g = 1$ for the saturations and $x_\alpha^w + x_\alpha^a + x_\alpha^c = 1$ for the mole fractions. Furthermore, the phase pressures are related to each other via capillary pressures between the fluid phases, which are functions of the saturation, e.g. according to the approach of Parker et al.

The used primary variables are dependent on the locally present fluid phases. An adaptive primary variable switch is included. The phase state is stored for all nodes of the system. The following cases can be distinguished:

- All three phases are present: Primary variables are two saturations (S_w and S_n), and a pressure, in this case p_g.
- Only the water phase is present: Primary variables are now the mole fractions of air and contaminant in the water phase (x_w^a and x_w^c), as well as the gas pressure, which is, of course, in a case where only the water phase is present, just the same as the water pressure.
- Gas and NAPL phases are present: Primary variables (S_n, x_g^w, p_g).

- Water and NAPL phases are present: Primary variables (S_n, x_w^a, p_g).
- Only gas phase is present: Primary variables (x_g^w, x_g^c, p_g).
- Water and gas phases are present: Primary variables (S_w, x_w^g, p_g).

The Non-Isothermal Model: NIModel

This model implements a generic energy balance for single and multi-phase transport problems. Currently the non-isothermal model can be used on top of the 1p2c, 2p, 2p2c and 3p3c models. Comparison to simple analytical solutions for pure convective and conductive problems are found in the 1p2c test. Also refer to this test for details on how to activate the non-isothermal model.

For the energy balance, local thermal equilibrium is assumed. This results in one energy conservation equation for the porous solid matrix and the fluids:

$$\phi \frac{\partial \sum_\alpha \varrho_\alpha u_\alpha S_\alpha}{\partial t} + (1-\phi)\frac{\partial (\varrho_s c_s T)}{\partial t} - \sum_\alpha \operatorname{div}\left\{ \varrho_\alpha h_\alpha \frac{k_{r\alpha}}{\mu_\alpha} \mathbf{K} \left(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g}\right)\right\} - \operatorname{div}\left(\lambda_{pm} \mathbf{grad}\, T\right) - q^h = 0.$$

where h_α is the specific enthalpy of a fluid phase α and $u_\alpha = h_\alpha - p_\alpha/\varrho_\alpha$ is the specific internal energy of the phase.

The M-Phase, N-Component Model: MpNcModel

This model implements a M-phase flow of a fluid mixture composed of N chemical species. The phases are denoted by lower index $\alpha \in \{1, \ldots, M\}$. All fluid phases are mixtures of $N \geq M-1$ chemical species which are denoted by the upper index $\kappa \in \{1, \ldots, N\}$.

The momentum approximation can be selected via "BaseFluxVariables": Darcy (ImplicitDarcyFluxVariables) and Forchheimer (ImplicitForchheimerFluxVariables) relations are available for all Box models.

By inserting this into the equations for the conservation of the mass of each component, one gets one mass-continuity equation for each component κ

$$\sum_\kappa \left(\phi \frac{\partial \left(\varrho_\alpha x_\alpha^\kappa S_\alpha\right)}{\partial t} + \operatorname{div}\, \left\{ v_\alpha \frac{\varrho_\alpha}{\overline{M}_\alpha} x_\alpha^\kappa \right\} \right) = q^\kappa$$

with $\overline{M}_\alpha$ being the average molar mass of the phase α:

$$\overline{M}_\alpha = \sum_\kappa M^\kappa \, x_\alpha^\kappa$$

For the missing M model assumptions, the model assumes that if a fluid phase is not present, the sum of the mole fractions of this fluid phase is smaller than 1, i.e.

$$\forall \alpha : S_\alpha = 0 \implies \sum_\kappa x_\alpha^\kappa \leq 1$$

Also, if a fluid phase may be present at a given spatial location its saturation must be positive:

$$\forall \alpha : \sum_\kappa x_\alpha^\kappa = 1 \implies S_\alpha \geq 0$$

Since at any given spatial location, a phase is always either present or not present, one of the strict equalities on the right hand side is always true, i.e.

$$\forall \alpha : S_\alpha \left(\sum_\kappa x_\alpha^\kappa - 1 \right) = 0$$

always holds.

These three equations constitute a non-linear complementarity problem, which can be solved using so-called non-linear complementarity functions $\Phi(a, b)$ which have the property

$$\Phi(a, b) = 0 \iff a \geq 0 \wedge b \geq 0 \wedge a \cdot b = 0$$

Several non-linear complementarity functions have been suggested, e.g. the Fischer-Burmeister function

$$\Phi(a, b) = a + b - \sqrt{a^2 + b^2} \, .$$

This model uses

$$\Phi(a, b) = \min\{a, b\} \, ,$$

because of its piecewise linearity.

These equations are then discretized using a fully-implicit vertex centered finite volume scheme (often known as 'box'-scheme) for spatial discretization and the implicit Euler method as temporal discretization.

The model assumes local thermodynamic equilibrium and uses the following primary variables:

- The component fugacities $f^1, \ldots, f^N$
- The pressure of the first phase p_1
- The saturations of the first $M - 1$ phases $S_1, \ldots, S_{M-1}$
- Temperature T if the energy equation is enabled

The Two-Phase, Discrete Fracture Model: TwoPDFMModel

This model implements two-phase flow of two immiscible fluids $\alpha \in \{w, n\}$ using a standard multiphase Darcy approach as the equation for the conservation of momentum, i.e.

$$v_\alpha = -\frac{k_{r\alpha}}{\mu_\alpha} \mathbf{K} \left(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g} \right)$$

By inserting this into the equation for the conservation of the phase mass, one gets

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

These equations are discretized by a fully-coupled vertex centered finite volume (box) scheme as spatial and the implicit Euler method as time discretization.

By using constitutive relations for the capillary pressure $p_c = p_n - p_w$ and relative permeability $k_{r\alpha}$ and taking advantage of the fact that $S_w + S_n = 1$, the number of unknowns can be reduced to two. Currently the model supports choosing either p_w and S_n or p_n and S_w as primary variables. The formulation which ought to be used can be specified by setting the `Formulation` property to either `TwoPCommonIndices::pWsN` or `TwoPCommonIndices::pNsW`. By default, the model uses p_w and S_n.

The Stokes Model: StokesModel

This model implements laminar Stokes flow of a single fluid, solving the momentum balance equation

$$\frac{\partial \left(\varrho_g \boldsymbol{v}_g\right)}{\partial t} + \boldsymbol{\nabla} \cdot \left(p_g \mathbf{I} - \mu_g \left(\boldsymbol{\nabla} \boldsymbol{v}_g + \boldsymbol{\nabla} \boldsymbol{v}_g^T\right)\right) - \varrho_g \mathbf{g} = 0,$$

and the mass balance equation

$$\frac{\partial \varrho_g}{\partial t} + \boldsymbol{\nabla} \cdot \left(\varrho_g \boldsymbol{v}_g\right) - q_g = 0.$$

By setting the property `EnableNavierStokes` to `true` the Navier-Stokes equation can be solved. In this case an additional term

$$+\varrho_g \left(\boldsymbol{v}_g \cdot \boldsymbol{\nabla}\right) \boldsymbol{v}_g$$

is added to the momentum balance equation.

This is discretized by a fully-coupled vertex-centered finite volume (box) scheme in space and by the implicit Euler method in time.

The Isothermal N-Component Stokes Model: StokesNcModel

This model implements an isothermal n-component Stokes flow of a fluid solving a momentum balance, a mass balance and a conservation equation for each component. When using mole fractions naturally the densities represent molar densites

Momentum Balance:

$$\frac{\partial \left(\varrho_g \boldsymbol{v}_g\right)}{\partial t} + \boldsymbol{\nabla} \cdot \left(p_g \mathbf{I} - \mu_g \left(\boldsymbol{\nabla} \boldsymbol{v}_g + \boldsymbol{\nabla} \boldsymbol{v}_g^T\right)\right) - \varrho_g \mathbf{g} = 0,$$

Mass balance equation:

$$\frac{\partial \varrho_g}{\partial t} + \boldsymbol{\nabla} \cdot \left(\varrho_g \boldsymbol{v}_g\right) - q_g = 0$$

Component mass balance equations:

$$\frac{\partial \left(\varrho_g X_g^\kappa\right)}{\partial t} + \boldsymbol{\nabla} \cdot \left(\varrho_g \boldsymbol{v}_g X_g^\kappa - D_g^\kappa \varrho_g \frac{M^\kappa}{M_g} \boldsymbol{\nabla} X_g^\kappa\right) - q_g^\kappa = 0$$

This is discretized using a fully-coupled vertex centered finite volume (box) scheme as spatial and the implicit Euler method in time.

The Non-Isothermal N-Component Stokes Model: StokesNcNIModel

This model implements a non-isothermal n-component Stokes flow of a fluid solving a momentum balance, a mass balance, a conservation equation for one component, and one balance equation for the energy.

Momentum Balance:

$$\frac{\partial \left(\varrho_g \boldsymbol{v}_g\right)}{\partial t} + \boldsymbol{\nabla} \cdot \left(p_g \mathbf{I} - \mu_g \left(\boldsymbol{\nabla} \boldsymbol{v}_g + \boldsymbol{\nabla} \boldsymbol{v}_g^T\right)\right) - \varrho_g \mathbf{g} = 0,$$

Mass balance equation:

$$\frac{\partial \varrho_g}{\partial t} + \boldsymbol{\nabla} \cdot \left(\varrho_g \boldsymbol{v}_g\right) - q_g = 0$$

Component mass balance equation:

$$\frac{\partial \left(\varrho_g X_g^\kappa\right)}{\partial t} + \boldsymbol{\nabla} \cdot \left(\varrho_g \boldsymbol{v}_g X_g^\kappa - D_g^\kappa \varrho_g \frac{M^\kappa}{M_g} \boldsymbol{\nabla} x_g^\kappa\right) - q_g^\kappa = 0$$

Energy balance equation:

$$\frac{\partial (\varrho_g u_g)}{\partial t} + \boldsymbol{\nabla} \cdot \left(\varrho_g h_g \boldsymbol{v}_g - \sum_\kappa \left[h_g^\kappa D_g^\kappa \varrho_g \frac{M^\kappa}{M_g} \nabla x_g^\kappa\right] - \lambda_g \boldsymbol{\nabla} T\right) - q_T = 0$$

This is discretized using a fully-coupled vertex centered finite volume (box) scheme as spatial and the implicit Euler method as temporal discretization.

The Linear Elastic Model: ElasticModel

This model implements a linear elastic solid using Hooke's law as stress-strain relation and a quasi-stationary momentum balance equation:

$$\boldsymbol{\sigma} = 2\, G\, \boldsymbol{\epsilon} + \lambda \,\mathrm{tr}(\boldsymbol{\epsilon})\, \boldsymbol{I}.$$

with the strain tensor $\boldsymbol{\epsilon}$ as a function of the solid displacement gradient $\mathbf{grad}\boldsymbol{u}$:

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\mathbf{grad}\boldsymbol{u} + \mathbf{grad}^T \boldsymbol{u}\right).$$

Gravity can be enabled or disabled via the property system. By inserting this into the momentum balance equation, one gets

$$\mathrm{div}\boldsymbol{\sigma} + \varrho \mathbf{g} = 0\ ,$$

The equation is discretized using a vertex-centered finite volume (box) scheme as spatial discretization.

The Linear Elastic One-Phase Two-Component Model: ElOnePTwoCModel

This model implements a one-phase flow of an incompressible fluid, that consists of two components. The deformation of the solid matrix is described with a quasi-stationary momentum balance equation. The influence of the pore fluid is accounted for through the effective stress concept (Biot 1941). The total stress acting on a rock is partially supported by the rock matrix and partially supported by the pore fluid. The effective stress represents the share of the total stress which is supported by the solid rock matrix and can be determined as a function of the strain according to Hooke's law.

As an equation for the conservation of momentum within the fluid phase Darcy's approach is used:

$$v = -\frac{\mathbf{K}}{\mu}\left(\mathbf{grad}\, p - \varrho_w \mathbf{g}\right)$$

Gravity can be enabled or disabled via the property system. By inserting this into the volume balance of the solid-fluid mixture, one gets

$$\frac{\partial \mathrm{div}\mathbf{u}}{\partial t} - \mathrm{div}\left\{\frac{\mathbf{K}}{\mu}\left(\mathbf{grad}\, p - \varrho_w \mathbf{g}\right)\right\} = q\ ,$$

The transport of the components $\kappa \in \{w, a\}$ is described by the following equation:

$$\frac{\partial \phi_{eff} X^\kappa}{\partial t} - \mathrm{div}\left\{X^\kappa \frac{\mathbf{K}}{\mu}\left(\mathbf{grad}\, p - \varrho_w \mathbf{g}\right) + D^\kappa_{\mathrm{pm}} \frac{M^\kappa}{M_\alpha} \mathbf{grad} x^\kappa - \phi_{eff} X^\kappa \frac{\partial \boldsymbol{u}}{\partial t}\right\} = q.$$

If the model encounters stability problems, a stabilization term can be switched on. The stabilization term is defined in Aguilar et al (2008):

$$\beta \mathrm{div}\mathbf{grad}\frac{\partial p}{\partial t}$$

with β:

$$\beta = h^2/4(\lambda + 2\mu)$$

where h is the discretization length.

The balance equations with the stabilization term are given below:

$$\frac{\partial \mathrm{div}\mathbf{u}}{\partial t} - \mathrm{div}\left\{\frac{\mathbf{K}}{\mu}\left(\mathbf{grad}\, p - \varrho_w \mathbf{g}\right) + \varrho_w \beta \mathbf{grad}\frac{\partial p}{\partial t}\right\} = q\ ,$$

The transport of the components $\kappa \in \{w, a\}$ is described by the following equation:

$$\frac{\partial \phi_{eff} X^\kappa}{\partial t} - \mathrm{div}\left\{X^\kappa \frac{\mathbf{K}}{\mu}\left(\mathbf{grad}\, p - \varrho_w \mathbf{g}\right) + \varrho_w X^\kappa \beta \mathbf{grad}\frac{\partial p}{\partial t} + D^\kappa_{\mathrm{pm}} \frac{M^\kappa}{M_\alpha} \mathbf{grad} x^\kappa - \phi_{eff} X^\kappa \frac{\partial \boldsymbol{u}}{\partial t}\right\} = q.$$

The quasi-stationary momentum balance equation is:

$$\mathrm{div}\left(\boldsymbol{\sigma'} - p\boldsymbol{I}\right) + \left(\phi_{eff}\varrho_w + (1 - \phi_{eff}) * \varrho_s\right)\mathbf{g} = 0\ ,$$

with the effective stress:

$$\boldsymbol{\sigma'} = 2\,G\,\boldsymbol{\epsilon} + \lambda\,\mathrm{tr}(\boldsymbol{\epsilon})\,\boldsymbol{I}.$$

and the strain tensor $\boldsymbol{\epsilon}$ as a function of the solid displacement gradient $\mathbf{grad}\boldsymbol{u}$:

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\mathbf{grad}\boldsymbol{u} + \mathbf{grad}^T\boldsymbol{u}\right).$$

Here, the rock mechanics sign convention is switch off which means compressive stresses are < 0 and tensile stresses are > 0. The rock mechanics sign convention can be switched on for the vtk output via the property system.

The effective porosity is calculated as a function of the solid displacement:

$$\phi_{eff} = \frac{\phi_{init} + \mathrm{div}\boldsymbol{u}}{1 + \mathrm{div}}$$

All equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization.

The primary variables are the pressure p and the mole or mass fraction of dissolved component x and the solid displacement vector $\boldsymbol{u}$.

The Linear Elastic Two-Phase Model: ElTwoPModel

This model implements a two-phase flow of compressible immiscible fluids $\alpha \in \{w, n\}$. The deformation of the solid matrix is described with a quasi-stationary momentum balance equation. The influence of the pore fluid is accounted for through the effective stress concept (Biot 1941). The total stress acting on a rock is partially supported by the rock matrix and partially supported by the pore fluid. The effective stress represents the share of the total stress which is supported by the solid rock matrix and can be determined as a function of the strain according to Hooke's law.

As an equation for the conservation of momentum within the fluid phases the standard multiphase Darcy's approach is used:

$$v_\alpha = -\frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}\left(\mathbf{grad}\, p_\alpha - \varrho_\alpha \mathbf{g}\right)$$

Gravity can be enabled or disabled via the property system. By inserting this into the continuity equation, one gets

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

A quasi-stationary momentum balance equation is solved for the changes with respect to the initial conditions (Darcis 2012), note that this implementation assumes the soil mechanics sign convention (i.e. compressive stresses are negative):

$$\mathrm{div}\left(\boldsymbol{\Delta\sigma'} - \Delta p_{eff}\boldsymbol{I}\right) + \Delta\varrho_b \mathbf{g} = 0\ ,$$

with the effective stress:

$$\boldsymbol{\sigma'} = 2\,G\,\boldsymbol{\epsilon} + \lambda\,\mathrm{tr}(\boldsymbol{\epsilon})\,\mathbf{I}.$$

and the strain tensor $\boldsymbol{\epsilon}$ as a function of the solid displacement gradient $\mathbf{gradu}$:

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\mathbf{gradu} + \mathbf{grad}^T\mathbf{u}\right).$$

Here, the rock mechanics sign convention is switch off which means compressive stresses are < 0 and tensile stresses are > 0. The rock mechanics sign convention can be switched on for the vtk output via the property system.

The effective porosity and the effective permeability are calculated as a function of the solid displacement-:

$$\phi_{eff} = \frac{\phi_{init} + \mathrm{div}\mathbf{u}}{1 + \mathrm{div}\mathbf{u}}$$

$$K_{eff} = K_{init}\exp\left(22.2(\phi_{eff}/\phi_{init} - 1)\right)$$

The mass balance equations are discretized using a vertex-centered finite volume (box) or cell-centered finite volume scheme as spatial and the implicit Euler method as time discretization. The momentum balance equations are discretized using a standard Galerkin Finite Element method as spatial discretization scheme.

The primary variables are the wetting phase pressure p_w, the nonwetting phase saturation S_n and the solid displacement vector $\mathbf{u}$ (changes in solid displacement with respect to initial conditions).

7.3.2 Decoupled Models

The basic idea the so-called decoupled models have in common is to reformulate the equations of multi-phase flow (e.g. Eq. 7.1) into one equation for pressure and equations for phase-/component-/etc. 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 decoupled model is the so-called 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 decoupled structure allows the use of different discretization methods for the different equations. The standard method used in the decoupled 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 The two-phase model and The Two-Phase, Two-Component Model is provided for two dimensions.

The one-phase model

This model solves equations of the form

$$\mathrm{div}\,\boldsymbol{v} = q.$$

The velocity $\boldsymbol{v}$ is the single phase Darcy velocity:

$$\boldsymbol{v} = -\frac{1}{\mu}\boldsymbol{K}\left(\mathbf{grad}\,p + \rho\, g\,\mathbf{grad}\,z\right),$$

where p is the pressure, $\boldsymbol{K}$ the absolute permeability, μ the viscosity, ρ the density, and g the gravity constant, and q is the source term. At the boundary, $p = p_D$ on $\Gamma_{Dirichlet}$, and $\boldsymbol{v}\cdot\boldsymbol{n} = q_N$ on $\Gamma_{Neumann}$.

The two-phase model

Pressure Model This model solves equations of the form

$$\phi \left(\rho_w \frac{\partial S_w}{\partial t} + \rho_n \frac{\partial S_n}{\partial t} \right) + \operatorname{div} \boldsymbol{v}_{total} = q.$$

The definition of the total velocity $\boldsymbol{v}_{total}$ depends on the choice of the primary pressure variable. Further, fluids can be assumed to be compressible or incompressible (Property: `EnableCompressibility`). In the incompressible case a wetting (w) phase pressure as primary variable leads to

$$-\operatorname{div} \left[\lambda \boldsymbol{K} \left(\mathbf{grad}\, p_w + f_n \mathbf{grad}\, p_c + \sum f_\alpha \rho_\alpha\, g\, \mathbf{grad}\, z \right) \right] = q,$$

a non-wetting (n) phase pressure yields

$$-\operatorname{div} \left[\lambda \boldsymbol{K} \left(\mathbf{grad}\, p_n - f_w \mathbf{grad}\, p_c + \sum f_\alpha \rho_\alpha\, g\, \mathbf{grad}\, z \right) \right] = q,$$

and a global pressure leads to

$$-\operatorname{div} \left[\lambda \boldsymbol{K} \left(\mathbf{grad}\, p_{global} + \sum f_\alpha \rho_\alpha\, g\, \mathbf{grad}\, z \right) \right] = q.$$

Here, p_α is a phase pressure, p_{global} the global pressure of a classical fractional flow formulation (see e.g. P. Binning and M. A. Celia, "Practical implementation of the fractional flow approach to multi-phase flow simulation", Advances in water resources, vol. 22, no. 5, pp. 461-478, 1999.), $p_c = p_n - p_w$ is the capillary pressure, $\boldsymbol{K}$ the absolute permeability, $\lambda = \lambda_w + \lambda_n$ the total mobility depending on the saturation ($\lambda_\alpha = k_{r_\alpha}/\mu_\alpha$), $f_\alpha = \lambda_\alpha/\lambda$ the fractional flow function of a phase, ρ_α a phase density, g the gravity constant and q the source term.

For all cases, $p = p_D$ on $\Gamma_{Dirichlet}$, and $\boldsymbol{v}_{total} \cdot \boldsymbol{n} = q_N$ on $\Gamma_{Neumann}$.

The slightly compressible case is only implemented for phase pressures! In this case for a wetting (w) phase pressure as primary variable the equations are formulated as

$$\phi \left(\rho_w \frac{\partial S_w}{\partial t} + \rho_n \frac{\partial S_n}{\partial t} \right) - \operatorname{div} \left[\lambda \boldsymbol{K} \left(\mathbf{grad}\, p_w + f_n\, \mathbf{grad}\, p_c + \sum f_\alpha \rho_\alpha\, g\, \mathbf{grad}\, z \right) \right] = q,$$

and for a non-wetting (n) phase pressure as

$$\phi \left(\rho_w \frac{\partial S_w}{\partial t} + \rho_n \frac{\partial S_n}{\partial t} \right) - \operatorname{div} \left[\lambda \boldsymbol{K} \left(\mathbf{grad}\, p_n - f_w \mathbf{grad}\, p_c + \sum f_\alpha \rho_\alpha\, g\, \mathbf{grad}\, z \right) \right] = q,$$

In this slightly compressible case the following definitions are valid: $\lambda = \rho_w \lambda_w + \rho_n \lambda_n$, $f_\alpha = (\rho_\alpha \lambda_\alpha)/\lambda$ This model assumes that temporal changes in density are very small and thus terms of temporal derivatives are negligible in the pressure equation. Depending on the formulation the terms including time derivatives of saturations are simplified by inserting $S_w + S_n = 1$.

In the IMPES models the default setting is:

- formulation: $p_w - S_w$ (Property: *Formulation* defined as *DecoupledTwoPCommonIndices::pwsw*)
- compressibility: disabled (Property: *EnableCompressibility* set to *false*)

Saturation Model This model solves equations of the form

$$\phi \frac{\partial(\rho_\alpha S_\alpha)}{\partial t} + \operatorname{div}(\rho_\alpha \boldsymbol{v_\alpha}) = q_\alpha,$$

where S_α is the saturation of phase α (wetting (w), non-wetting (n)) and $\boldsymbol{v}_\alpha$ is the phase velocity defined by the multi-phase Darcy equation. If a phase velocity is reconstructed from the pressure solution it can be directly inserted into the previous equation. In the incompressible case the equation is further divided by the phase density ρ_α. If a total velocity is reconstructed the saturation equation is reformulated into:

$$\phi \frac{\partial S_w}{\partial t} + f_w \operatorname{div} \boldsymbol{v}_t + f_w \lambda_n \boldsymbol{K} (\mathbf{grad}\, p_c + (\rho_n - \rho_w)\, g\, \mathbf{grad} z) = q_\alpha,$$

to get a wetting phase saturation or

$$\phi \frac{\partial S_n}{\partial t} + f_n \operatorname{div} \boldsymbol{v}_t - f_n \lambda_w \boldsymbol{K} (\mathbf{grad}\, p_c + (\rho_n - \rho_w)\, g\, \mathbf{grad} z) = q_\alpha,$$

if the non-wetting phase saturation is the primary transport variable.

The total velocity formulation is only implemented for incompressible fluids and f_α is the fractional flow function, λ_α is the mobility, $\boldsymbol{K}$ the absolute permeability, p_c the capillary pressure, ρ the fluid density, g the gravity constant, and q the source term.

In the IMPES models the default setting is:
formulation: p_w - S_w (Property: *Formulation* defined as *DecoupledTwoPCommonIndices::pwsw*)
compressibility: disabled (Property: *EnableCompressibility* set to *false*)

The Two-Phase, Two-Component Model

Provides a Finite Volume implementation for the pressure equation of a compressible system with two components. An IMPES-like method is used for the sequential solution of the problem. Diffusion is neglected, capillarity can be regarded. Isothermal conditions and local thermodynamic equilibrium are assumed. Gravity is included.

$$c_{total} \frac{\partial p}{\partial t} + \sum_\kappa \frac{\partial v_{total}}{\partial C^\kappa} \nabla \cdot \left(\sum_\alpha X_\alpha^\kappa \varrho_\alpha \mathbf{v}_\alpha \right) = \sum_\kappa \frac{\partial v_{total}}{\partial C^\kappa} q^\kappa,$$

where $\mathbf{v}_\alpha = -\lambda_\alpha \mathbf{K} (\nabla \mathbf{p}_\alpha + \rho_\alpha \mathbf{g})$. c_{total} represents the total compressibility, for constant porosity this yields $-\frac{\partial V_{total}}{\partial p_\alpha}$, p_α denotes the phase pressure, $\mathbf{K}$ the absolute permeability, λ_α the phase mobility, ρ_α the phase density and $\mathbf{g}$ the gravity constant and C^κ the total Component concentration. See paper SPE 99619 or "Analysis of a Compositional Model for Fluid Flow in Porous Media" by Chen, Qin and Ewing for derivation.

The pressure base class FVPressure assembles the matrix and right-hand-side vector and solves for the pressure vector, whereas this class provides the actual entries for the matrix and RHS vector. The partial derivatives of the actual fluid volume v_{total} are gained by using a secant method.

The transport step is described by the finite volume model for the solution of the transport equation for compositional two-phase flow.

$$\frac{\partial C^\kappa}{\partial t} = -\nabla \cdot \left(\sum_\alpha X_\alpha^\kappa \varrho_\alpha \mathbf{v}_\alpha \right) + q^\kappa,$$

where $\mathbf{v}_\alpha = -\lambda_\alpha \mathbf{K} (\nabla \mathbf{p}_\alpha + \rho_\alpha \mathbf{g})$. p_α denotes the phase pressure, $\mathbf{K}$ the absolute permeability, λ_α the phase mobility, ρ_α the phase density and $\mathbf{g}$ the gravity constant and C^κ the total Component concentration. The whole flux contribution for each cell is subdivided into a storage term, a flux term and a source term. Corresponding functions (`getFlux()` and `getFluxOnBoundary()`) are provided, internal sources are directly treated.

8 The flow of things in DuMu^x

This chapter is supposed to show how things are "handed around" in DuMu^x. This is not a comprehenisve guide through the modeling framework of DuMu^x, but hopefully it will help getting to grips with it.

In Section 8.1 the structure of DuMu^x is shown from a *content* point of view. Section 8.2 is written from the point of view of the *implementation*. These two approaches are linked by the circled numbers (like ①) in the flowchart of Section 8.2 corresponding to the enumeration of the list of Section 8.1. This is supposed to demonstrate at which point of the program-flow you are content- and implementation-wise.

Section 8.2 is structured by boxes and $\overrightarrow{\text{arrows}}$. Boxes stand for more or less important points in the programm. They may may be reckoned "step stones". Likewise, the arrows connect the boxes. If important things happen in between, it is written under the arrows.

Plain boxes stand for generic parts of the program. double {{boundings}} stand for the implementatin specific part of the program, like `2p, 2p2c...`. This will be the most important part for most users. snakelike lines tell you that this part is specific to the components considered.

For keeping things simple, the program flow of a `2p` model is shown. There are extensive comments regarding the formating in the tex file: so feel free, to enhance this description.

8.1 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 8.1, the algorithmic representations of both approaches down to the element level are illustrated.

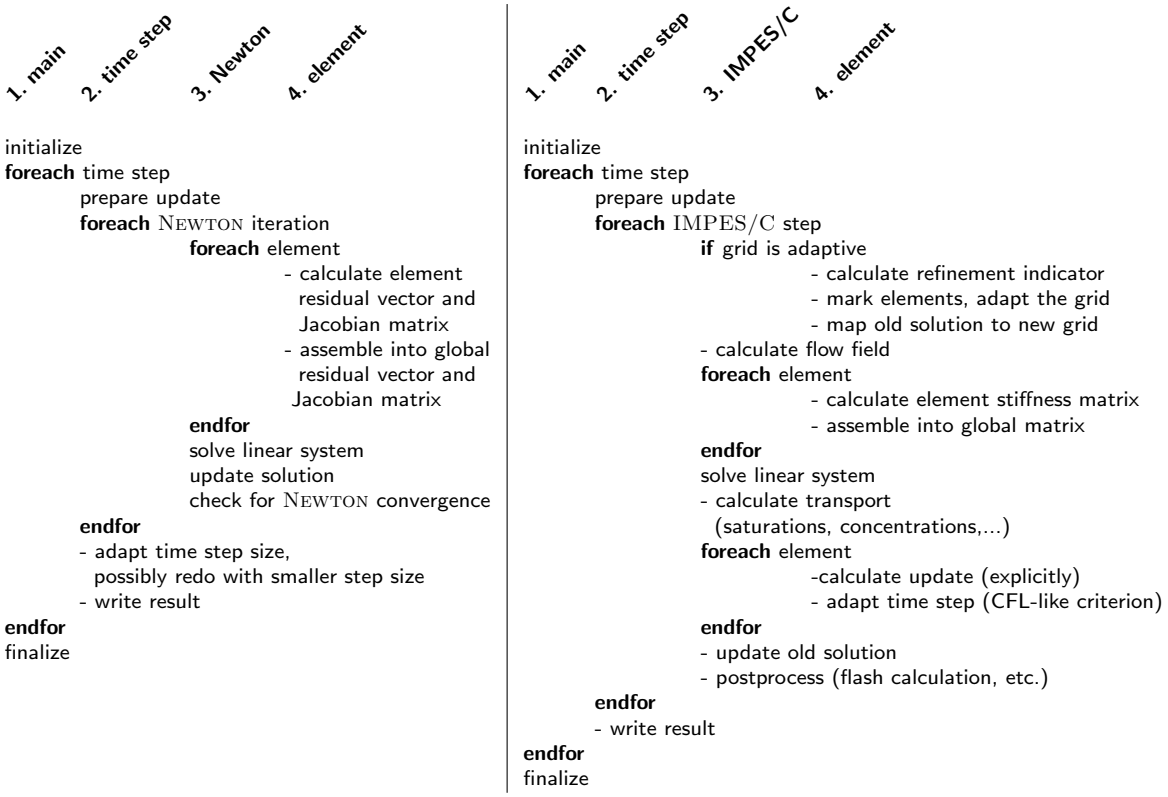


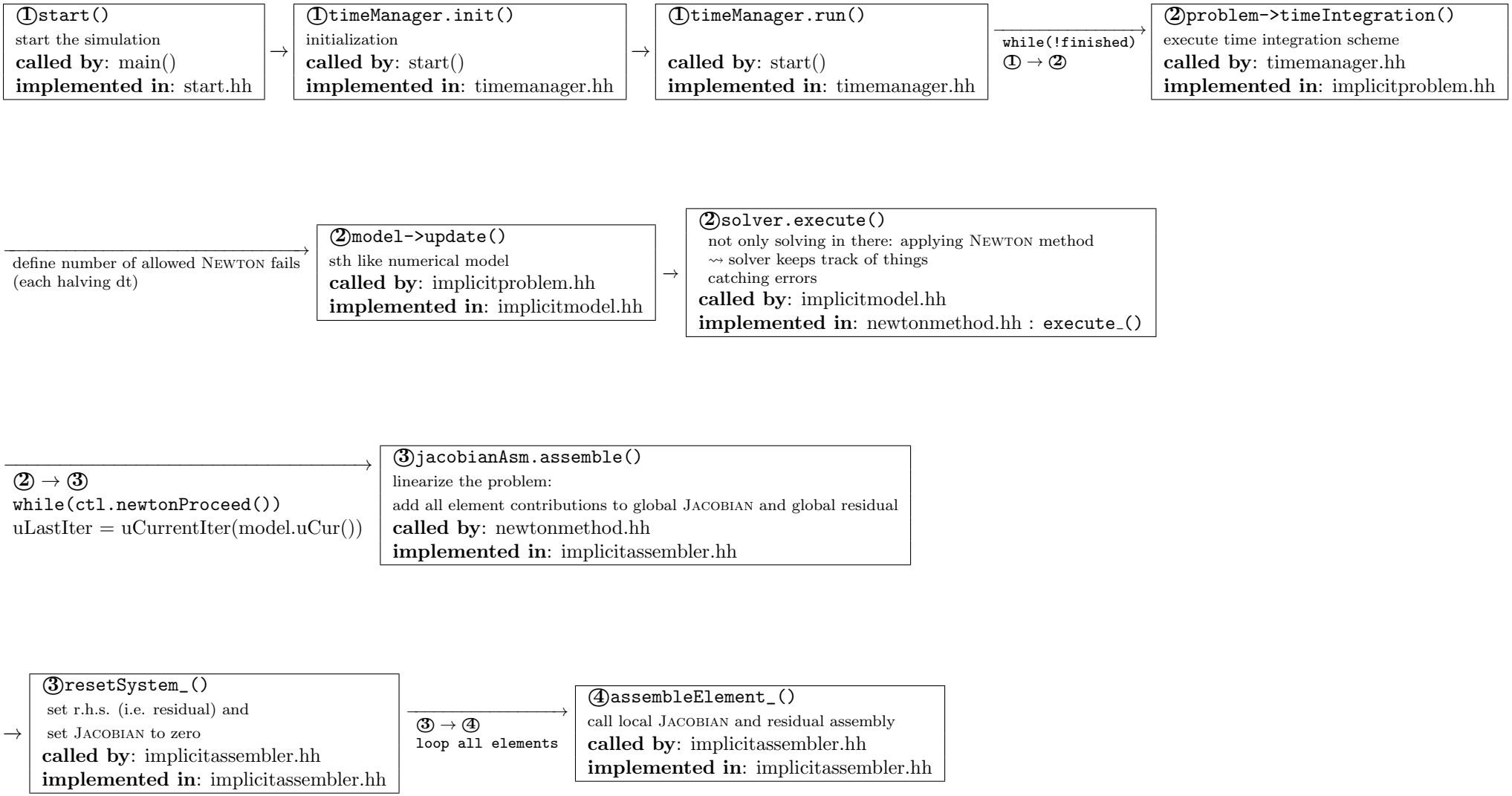
Figure 8.1: Structure of a coupled fully-implicit (**left**) and a decoupled semi-implicit (**right**) scheme in DuMu^x.

8.1.1 Levels

① main
② time step
③ NEWTON step
④ Element-wise assembly

8.2 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 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.


①start()
start the simulation
called by: main()
implemented in: start.hh
①timeManager.init()
initialization
called by: start()
implemented in: timemanager.hh
①timeManager.run()
called by: start()
implemented in: timemanager.hh
while(!finished)
① → ②
②problem->timeIntegration()
execute time integration scheme
called by: timemanager.hh
implemented in: implicitproblem.hh
define number of allowed NEWTON fails
(each halving dt)
②model->update()
sth like numerical model
called by: implicitproblem.hh
implemented in: implicitmodel.hh
②solver.execute()
not only solving in there: applying NEWTON method
⇝ solver keeps track of things
catching errors
called by: implicitmodel.hh
implemented in: newtonmethod.hh : execute_()
② → ③
while(ctl.newtonProceed())
uLastIter = uCurrentIter(model.uCur())
③jacobianAsm.assemble()
linearize the problem:
add all element contributions to global JACOBIAN and global residual
called by: newtonmethod.hh
implemented in: implicitassembler.hh
③resetSystem_()
set r.h.s. (i.e. residual) and
set JACOBIAN to zero
called by: implicitassembler.hh
implemented in: implicitassembler.hh
③ → ④
loop all elements
④assembleElement_()
call local JACOBIAN and residual assembly
called by: implicitassembler.hh
implemented in: implicitassembler.hh

→ ④`model_().localJacobian().assemble()`
set curr. element, update element's fin.vol.geom.
reset local JACOBIAN to 0
update types of boundaries on this element
called by: implicitassembler.hh
implemented in: implicitlocaljacobian.hh

→ ④`prevVolVars_.update()`, ④`curVolVars_.update()`
call model specific update of quantities defined for the volume:
variables for the *current and previous timestep...!!*
called by: implicitlocaljacobian.hh
implemented in: implicitelementvolumevariables.hh

→ ④`update()`
calculate all two-phase specific quantites defined in the volume
called by: boxelementvolumevariables.hh
implemented in: 2pvolumevariables.hh

→ ④`completeFluidState()`
calculate all required fluid properties from the primary variables
called by: 2pvolumevariables.hh
implemented in: 2pvolumevariables.hh

→ ④ `e.g: rho = Fluidsystem::density()`
`fluidState.setDensity(phaseIdx,rho)`
The fluid system does the real work:
calculates densities, diffusivities ...
The fluidstate save and provides them.
called by: 2pvolumevariables.hh
implemented in: 2pvolumevariables.hh

→ ④`localResidual().eval()`
the element's local residual is calculated:
see the next two stepstones
called by: implicitlocaljacobian.hh
implemented in: boxlocalresidual.hh

→ ④`asImp_().evalFluxes_()`
evaluate the fluxes going into each finite volume
how this is done is [model specific] (see below)
called by: boxlocalresidual.hh
implemented in: boxlocalresidual.hh

→ ④`computeFluxes()`
model specific flux computation
called by: boxlocalresidual.hh
implemented in: 2plocalresidual.hh

→ ④`FluxVariables fluxVars()`
this is a call to a constructor:
calculate the velocities
called by: 2plocalresidual.hh
implemented in: boxdarcyfluxvariables.hh

→ ④`computeAdvectiveFlux()` (other models: also diffusive)
upwinding decision via `massUpwindWeight_`
called by: 2plocalresidual.hh
implemented in: 2plocalresidual.hh

→ ④`asImp_().evalVolumeTerms_()`
evaluate the storage and source terms for each finite volume
how this is done is [model specific] (see below)
called by: boxlocalresidual.hh
implemented in: boxlocalresidual.hh

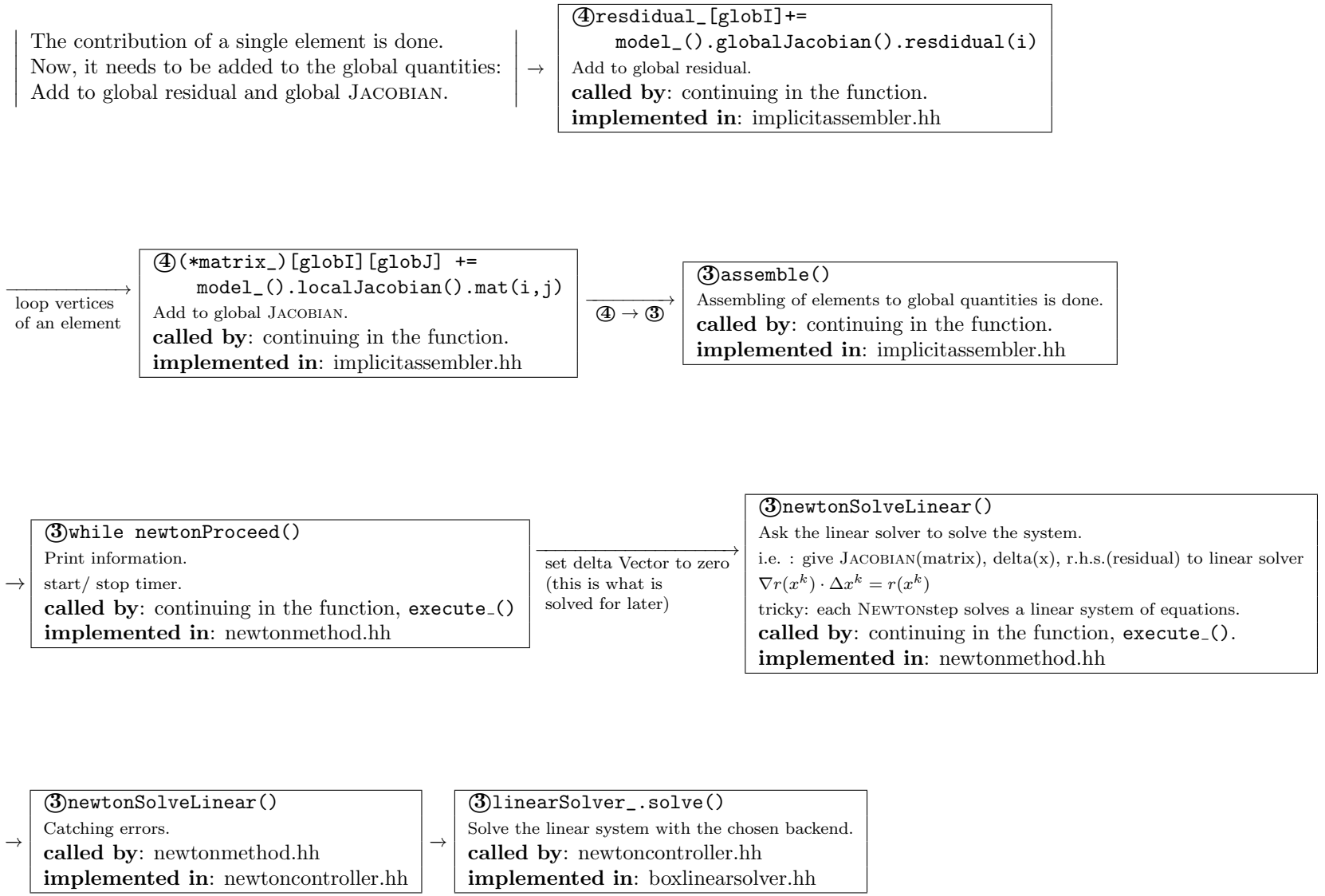
→ ④`asImp_().evalBoundary_()`
deal with the boundary conditions
may be [model specific]
called by: boxlocalresidual.hh
implemented in: boxlocalresidual.hh (or modelspecific)

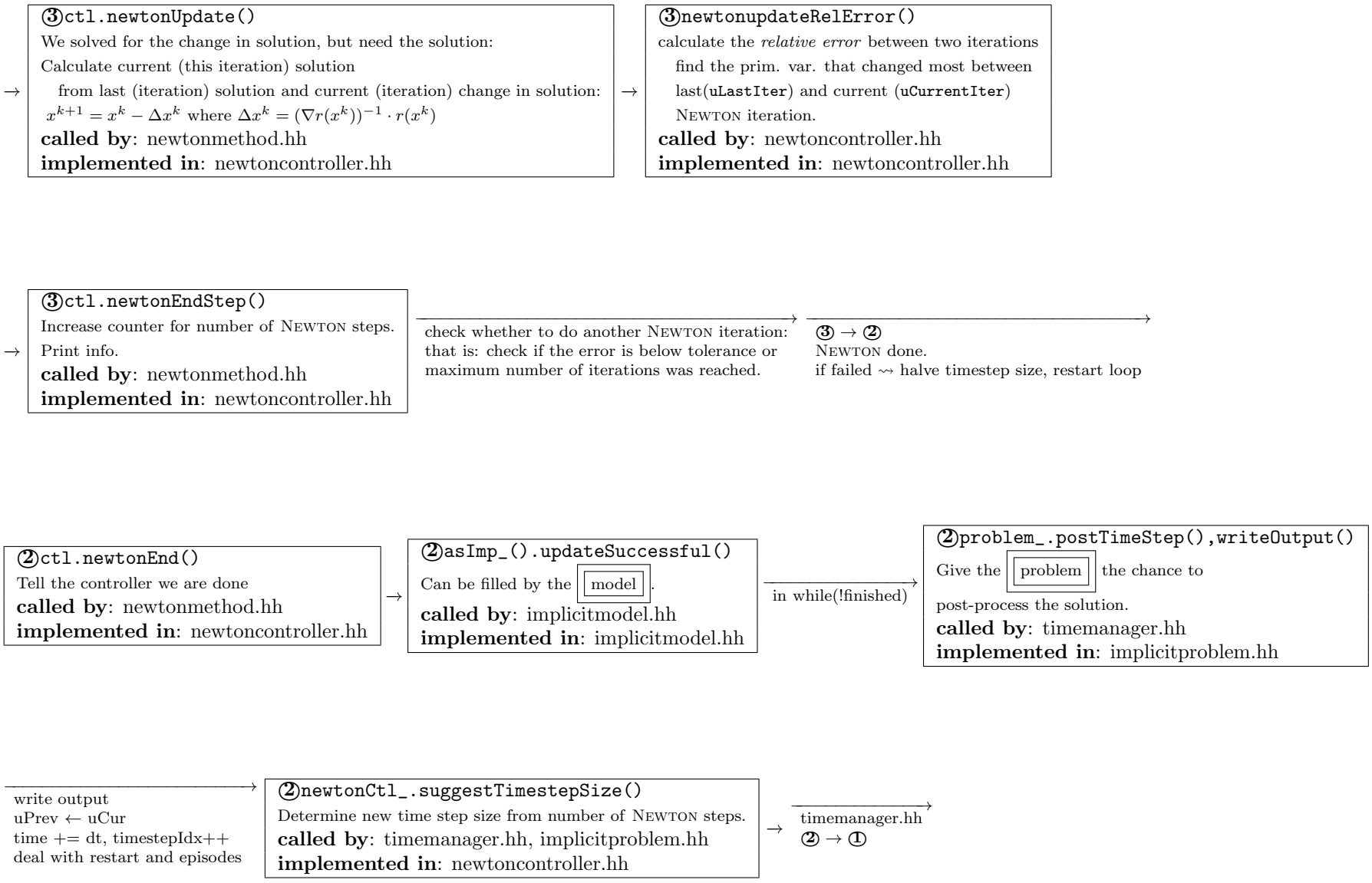
→ ④`asImp_().evalPartialDerivative()`
actually calculate the element's (local) JACOBIAN matrix
a property chooses backward/central/foward differences
here: central differences
called by: implicitlocaljacobian.hh
implemented in: implicitlocaljacobian.hh

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`

→ ④`assembleElement_()`
`model_().localJacobian().assemble()`
Residual of the current solution is now "numerically differentiated", for the element i.e. the local JACOBIAN matrix is calculated.
called by: implicitassembler.hh
implemented in: implicitassembler.hh


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.
④resdidual_[globI]+=
model_().globalJacobian().resdidual(i)
Add to global residual.
called by: continuing in the function.
implemented in: implicitassembler.hh
loop vertices
of an element
④(*matrix_)[globI][globJ] +=
model_().localJacobian().mat(i,j)
Add to global JACOBIAN.
called by: continuing in the function.
implemented in: implicitassembler.hh
④ → ③
③assemble()
Assembling of elements to global quantities is done.
called by: continuing in the function.
implemented in: implicitassembler.hh
③while newtonProceed()
Print information.
start/ stop timer.
called by: continuing in the function, execute_()
implemented in: newtonmethod.hh
set delta Vector to zero
(this is what is
solved for later)
③newtonSolveLinear()
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 NEWTONstep solves a linear system of equations.
called by: continuing in the function, execute_().
implemented in: newtonmethod.hh
③newtonSolveLinear()
Catching errors.
called by: newtonmethod.hh
implemented in: newtoncontroller.hh
③linearSolver_.solve()
Solve the linear system with the chosen backend.
called by: newtoncontroller.hh
implemented in: boxlinearsolver.hh


③ctl.newtonUpdate()
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 $\Delta x^k = (\nabla r(x^k))^{-1} \cdot r(x^k)$
called by: newtonmethod.hh
implemented in: newtoncontroller.hh
③newtonupdateRelError()
calculate the relative error between two iterations
find the prim. var. that changed most between
last(uLastIter) and current (uCurrentIter)
Newton iteration.
called by: newtoncontroller.hh
implemented in: newtoncontroller.hh
③ctl.newtonEndStep()
Increase counter for number of Newton steps.
Print info.
called by: newtonmethod.hh
implemented in: newtoncontroller.hh
check whether to do another Newton iteration:
that is: check if the error is below tolerance or
maximum number of iterations was reached.
③ → ②
Newton done.
if failed ⇝ halve timestep size, restart loop
②ctl.newtonEnd()
Tell the controller we are done
called by: newtonmethod.hh
implemented in: newtoncontroller.hh
②asImp_().updateSuccessful()
Can be filled by the model.
called by: implicitmodel.hh
implemented in: implicitmodel.hh
in while(!finished)
②problem_.postTimeStep(),writeOutput()
Give the problem the chance to
post-process the solution.
called by: timemanager.hh
implemented in: implicitproblem.hh
write output
uPrev ← uCur
time += dt, timestepIdx++
deal with restart and episodes
②newtonCtl_.suggestTimestepSize()
Determine new time step size from number of Newton steps.
called by: timemanager.hh, implicitproblem.hh
implemented in: newtoncontroller.hh
timemanager.hh
② → ①

9 Newton in a Nutshell

Coming back to the example of chapter 8 the following mass conservation equation is to be solved:

$$\underbrace{\phi\frac{\partial \varrho_\alpha S_\alpha}{\partial t} - \operatorname{div}\left\{\varrho_\alpha \frac{k_{r\alpha}}{\mu_\alpha}\mathbf{K}\left(\mathbf{grad}\ p_\alpha - \varrho_\alpha \mathbf{g}\right)\right\} - q_\alpha}_{\mathbf{f}(\mathbf{u}^r)} = 0\ . \tag{9.1}$$

Because of the nonlinear dependencies (even in this comparativly simple equation) in there, this is a really difficult task. However, for finding roots of of difficult equations there is a really handy method out there: NEWTON's method.

When using a fully coupled numerical model, one timestep essentially consists of the application of the NEWTON algorithm to solve the nonlinear system.

One step of the NEWTON method can be formalized as follows:

NEWTON method:

$$\mathbf{u}^{r+1} = \mathbf{u}^r - \left(\mathbf{f}'(\mathbf{u}^r)\right)^{-1}\mathbf{f}(\mathbf{u}^r) \tag{9.2a}$$

$$\Leftrightarrow \mathbf{f}'(\mathbf{u}^r)(\mathbf{u}^{r+1} - \mathbf{u}^r) = -\mathbf{f}(\mathbf{u}^r) \tag{9.2b}$$

$$\Leftrightarrow \underbrace{\mathbf{f}'(\mathbf{u}^r)}_{\text{Jacobian}}(\mathbf{u}^r - \mathbf{u}^{r+1}) = \mathbf{f}(\mathbf{u}^r) \tag{9.2c}$$

with

- ^{r}: last iteration, $^{r+1}$: current iteration,
- $'$: derivative
- $\mathbf{u}$: vector of unknowns, the actual primary variables
- $\mathbf{f}(\mathbf{u}^r)$: function of vector of unknowns

1-D example with slope m:

$$m = \frac{y(u^{r+1}) - y(u^r)}{u^{r+1} - u^r} \text{ for a root of a function: } m = -\frac{y(u^r)}{u^{r+1} - u^r} \tag{9.3}$$

The value of u (generally a vector of unknowns) for which f becomes zero is searched for. Therefore the quantity of interest is $\mathbf{u}^{r+1}$.

But the (BiCGSTAB / Pardiso / ...) linear solver solves systems of the form:

$$A\mathbf{x} = \mathbf{b}. \tag{9.4}$$

Comparing (9.4) with (9.2c) leads to:

- $\mathbf{b} = \mathbf{f}(\mathbf{u}^r)$ r.h.s. as it is known from the last iteration. Here, $\mathbf{f}(\mathbf{u}^r)$ is called residual. It is obtained by evaluating the balance equations with the primary variables, as obtained from the last iteration step.
- $A = \mathbf{f}'(\mathbf{u}^r)$ coefficient matrix or JACOBIAN. It is obtained by numerical differentiation. Evaluating the balance equations at the last solution + eps, then evaluating the balance equations at the last solution - eps, division by 2eps: numerical differentiation complete.
- $\mathbf{x} = (\mathbf{u}^{r+1} - \mathbf{u}^r)$ this is what the linear solver finds as an solution.

This is equivalent to stating that the implemented algorithm solves for the change of the solution. Or in other words: until the $\mathbf{u}$ does not change with one more NEWTON-iteration (do not confuse with timestep!).

In the rest of Dumux (everywhere besides in the solver), not the change of the solution is looked for, but the actual solution is used. Therefore the outcome of the linear solver needs to be reformulated as done in `updateMethod.update(*this, u, *uOld, model);`. In this function the "change in solution" is changed to "solution". Afterwards the quantity `*u` stands for the solution.

10 Tips & Tricks

This chapter tries to be a useful collection of tips and tricks that can be handy when working with DuMu^x. One of the most prominent ideas for developing DUNE/ DuMu^x is that reinventing the wheel in terms of FEM code should be avoided. We try to follow this idea also in the day-to-day work by stating the *tell us dogma*: "If you found something useful, handy or for other reasons helping when working with DuMu^x: put it into this chapter." or inform other developers and write to

10.1 DuMu^x- General Remarks

Flyspray Flyspray or bug-tracking system is a software application mainly designed to keep track of reported software development requests. This includes reported bugs and development requests for new or improved features. The main benefit of a bug-tracking system is to provide a clear centralized overview of all recorded requests and their state. DuMu^x users and developers can submit development requests at `http://www.dumux.org/flyspray/`.

Dashboard The testing-dashboard is a tool to constantly check the DuMu^x problems for compiling or running correctly. It is a useful tool to check the impacts of your commits and for quality management. The dashboard is available at `http://www.dumux.org/dashboard.php`.

The DuMu^x Mailing List: If you have questions concerning DuMu^x, hints for the DuMu^x-developers or specific problems, which you really struggle to solve on your own, you can contact the mailing list `dumux@iws.uni-stuttgart.de`. You can also subscribed to the mailing list via `https://listserv.uni-stuttgart.de/mailman/listinfo/dumux`, then you will be informed about upcoming releases or events.

The Commit Mailing List: If you further want to be informed about commits to the dumux can subscribe to the commit mailing list: `https://listserv.uni-stuttgart.de/mailman/listinfo/dumux-commits`.

10.2 Developing DuMu^x

Checking Your Commits: DuMu^x is developed with the help of Subversion (`svn`). This means that at some point you will commit your new and/or advanced features to the repository. In the following some additional guidelines are shown which are are good practice. Especially if you plan on committing to the stable part of DuMu^x you must follow these steps.

- add files and folders to your repository
- run `make doc` in your build-directory

- run `make headercheck` in your build-directory
- run `ctest` in your build-directory
- double-check whether the test are working. **If not** please investigate whether an update of the reference solution or a review of your changes is necessary
- double-check whether you include all necessary files to your commit
- commit
- check-out dumux in new folder and test if everything is still working (this is necessary to keep bump in the workflow small).
- check the testing-dashboard (see above), whether everything is still working

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`).

Errors Messages Related to the Property System The property system is a powerful tool and internally does some template and macro magic. The price for it are sometimes unintuitive compiler error messages. For example if the definition of a property could not be found, the error is:

```
error: no type named 'p' in 'struct Dumux::Properties::GetProperty<Dumux::Properties::TTag::
    TestProblem, Dumux::Properties::PTag::Scalar, Dumux::Properties::TTag::TestProblem, -1000>'
```

So check whether you did not misspelled any name related to the property system in the line of the error message and whether you really declared the type tag you use there.
Further check, if the `Problem` property was defined and spelled correctly.

Patching Files or Modules See 11.2 if you need to apply patches to DuMu^x or DUNE. 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:

```
$ svn diff > PATCHFILE
```

which creates a text file containing all your changes to the files in the current folder or its subdirectories. Other developers can now simply apply this patch by

```
$ path -p0 < PATCHFILE
```

Using DUNE Debug Streams DUNE provides a helpful feature, for keeping your debug-output organized. In stead of juggling with a bazillion `std::cout <<` statements or keeping some debug-precompiler statements organized, which are generally and strongly discouraged see 4.5 in order not to get flooded away by your output DUNE gives you a nice tool by the so called debug streams.

These are streams like `cout` but they can be switched on and off for the whole project. Maybe if you are really in the dark you want to see all your debug information. Another time you may only want to be warned if something is going seriously wrong during a simulation. This can be achieved by setting the debug streams to desired values. There are five levels:

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

They are used as follows: `Dune::dinfo << "message";` or `Dune::dgrave << "message";` . The debug streams are switched on/off via setting `#define DUNE_MINIMAL_DEBUG_LEVEL 4` in the source your application. If the value is set to e. g. 4 only the output generated after `Dune::dwarn` and `Dune::dgrave` will be printed.

File Name and Line Number by Predefined Macro If you want to know where some output or debug information came from, you can use the predefined macros `__FILE__` and `__LINE__` which are used like

```
dataFile << "# This was written from "<< __FILE__ << ", line "<< __LINE__ << "\n";
```

which translates into a line in the output file reading

```
# This was written from [..]/dumux/dumux/io/outputToFile.hh, line 261
```

This can also be very useful, if you want to have information about where some warning or debug information was issued.

Option Files optim.opts and debug.opts DUNE and DuMu^x are built with the help of `dunecontrol`, as explained on page 6. A lot of options need to be specified for that, which is done in option files. DuMu^x provides two example files `debug.opts` and `optim.opts`. These two files differ in the way DUNE and DuMu^x are compiled: either for debugging or for fast simulation. Switching between these two states can lead to a speedup of factor up to ten! Programs that are compiled with optimization can hardly be debugged because the debugger gets confused. Debugging with the optimization options active will lead to erratic behavior while debugging.

You can modify the files and add third-party dependencies or additional compiler flags.

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 grid. Then you have to configure and build only DUNE-grid and DuMu^x by adding `--only=dune-grid,dumux` to the `dunecontrol` call.

10.3 External Tools

10.3.1 Subversion (svn)

Subversion is a software versioning and revision control system. We use Subversion to manage the source code of DuMu^x, archive changes and central storage.

Basic Commands The basic svn commands are:

- `svn checkout` checkout a repository
- `svn update` updates file/folder
- `svn status` to check which files/folders have been changed `M`odified, `D`eleted, `A`dded, `?` not in repository
- `svn diff` to see the actual changes of a file/folder
- `svn commit` upload changes to the repository (only with meaningful commit messages)

The above shows you the necessary steps if you use the command line. There are also other tools providing a graphical user interface for using svn like kdesvn or eclipse.

Properties/Attributes How to set the SVN attributes:

- *eclipse*: right click on the file/folder → "team" → "add to svn:ignore..."
- *kdesvn*: right click on the file/folder → "ignore/unignore current item"
- *SVN on shell*: `svn propedit svn:ignore .`

10.3.2 Git

Git plays a similar role as Subversion, some see Git as a successor of Subversion. Git is used by DUNE. The basic Git commands are:

- `git clone` clone a repository (similar to svn checkout)
- `git pull` pull changes from the repository (similar to svn update)
- `git status` to check which files/folders have been changed
- `git diff` to see the actual changes of a file/folder

10.3.3 Eclipse

Using the DuMu^x-Eclipse Profile Everybody using the same profile has the advantage of resulting in less conflicts when different developing environments are used:

a) in eclipse open: `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 that now DuMu^x is chosen in `Select a profile`

10.3.4 ParaView

Reload Button: Yes, you read it right. There is script to reload `pvd` files or series of `vtu` files since ParaView 4.2. The scripts are available under the links below. Just save the specific code portion in a file and load it via `Macros` → `Add new macro`.
`vtk`: `http://markmail.org/message/exxynsgishbvtngg#query:+page:1+mid:rxlwxs7uqrfgibyv+state:results`
`pvd`: `http://markmail.org/message/exxynsgishbvtngg#query:+page:1+mid:rxlwxs7uqrfgibyv+state:results`.

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 chaptes. Attention its size is 180 MiB.

11 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 11.1 we list some prerequisites for running DUNE and DuMu^x. Please check in said paragraph whether you can fulfill them. In addition, section 11.4 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 [17]. If you are interested in more details about the build system that is used, they can be found in the DUNE Buildsystem Howto [13].

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.

11.1 Prerequisites

The GNU tool chain of `g++` and the tools of the GNU build system [25], also known as GNU autotools (`autoconf`, `automake`, `autogen`, `libtool`), as well as `make` must be available in a recent version. For a list of prerequisite software packages to install,see [18].

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 `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 11.3.1.

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 11.4.

Subversion (SVN) and a Git clients must be installed to download modules from Subversion and Git repositories.

11.2 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 2.3, 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 2.3) 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-2.7` (or whatever version you downloaded).

```
$ mkdir dune
$ cd dune
$ tar xzvf path_to_tarball_of/dune-common-2.3.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-geometry-2.3.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-grid-2.3.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-istl-2.3.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-localfunctions-2.3.1.tar.gz
$ tar xzvf path_to_tarball_of/dune-pdelab-2.0.0.tar.gz
$ tar xzvf path_to_tarball_of/dune-typetree-2.3.1.tar.gz
$ tar xzvf path_to_tarball_of/dumux-2.7.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 for the user later on. It can be easier for him to keep up with code changes and to receive important bug fixes using the update or pull command of the revision control system. DUNE uses Git and DuMu^x uses Apache Subversion for their software repositories. To access them a certain programs are needed which is referred to here shortly as Subversion client or Git client. In our description, we use the Subversion client `svn` of the Apache Subversion software itself.

In the technical language of Apache Subversion *checking out 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*.

The installation procedure is done as follows: Create a DUNE root directory, named `dune` 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 2.3 release branch are checked out as described on the DUNE website [14]:

```
$ mkdir DUMUX
$ cd DUMUX
$ git clone http://git.dune-project.org/repositories/dune-common
$ cd dune-common
$ git checkout releases/2.3
$ cd ..
$ git clone http://git.dune-project.org/repositories/dune-geometry
$ cd dune-geometry
$ git checkout releases/2.3
$ cd ..
$ git clone http://git.dune-project.org/repositories/dune-grid
$ cd dune-grid
$ git checkout releases/2.3
$ cd ..
$ git clone http://git.dune-project.org/repositories/dune-istl
$ cd dune-istl
$ git checkout releases/2.3
$ cd ..
$ git clone http://git.dune-project.org/repositories/dune-localfunctions
$ cd dune-localfunctions
$ git checkout releases/2.3
$ cd ..
$ git clone http://git.dune-project.org/repositories/dune-pdelab
$ cd dune-pdelab
$ git checkout releases/2.0
$ cd ..
$ git clone http://git.dune-project.org/repositories/dune-typetree
$ cd dune-typetree
$ git checkout releases/2.3
$ cd ..
```

The newest and maybe unstable developments are also provided in these repositories and is called *master*. Please check the DUNE website [14] for further information. However, the current DuMu^x release is based on the stable 2.3 release and it might not compile without further adaptations using the 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 [10]). 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
$ svn checkout --username=anonymous --password='' svn://svn.iws.uni-stuttgart
   .de/DUMUX/dumux/trunk dumux
```

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. A developer usually checks out non-anonymously the modules `dumux` and

`dumux-devel`. `Dumux-devel` itself makes use of the stable part `dumux`. Hence, the two parts have to be checked out together. This is done using the commands below. But `joeuser` needs to be replaced by the actual user name of the developer for accessing the software repository. One can omit the `--username` option in the commands above if the user name for the repository access is identical to the one for the system account.

```
$ svn co --username=joeuser svn://svn.iws.uni-stuttgart.de/DUMUX/dumux/trunk
   dumux
$ svn co --username=joeuser svn://svn.iws.uni-stuttgart.de/DUMUX/dune-mux/
   trunk dumux-devel
```

Please choose either not to store the password by subversion in an insecure way or choose to store it by subversion in a secure way, e. g. together with KDE's KWallet or GNOME Keyring. Check the documentation of Subversion for info on how this is done. A leaked out password can be used by evil persons to abuse a software repository.

11.3 Building Documentation

11.3.1 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 [9].

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.

11.3.2 Handbook

To build the DuMu^x handbook go into the `build-`directory and run `make doc` or `make dumux-handbook_-pdf`. The pdf can then be found in `MODULE_BUILD_DIRECTORY/doc/handbook/dumux-handbook.pdf`.

11.4 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 [15]. If you are going to use an external DUNE module the website on external modules [16] 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.

11.4.1 List of External Libraries and Modules

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

- **ALBERTA**: External grid library. Adaptive multi-level grid manager using bisectioning refinement and error control by residual techniques for scientific Applications. Requires a Fortran compiler like `gfortran`. Download: `http://www.alberta-fem.de` or for version 3.0 `http://www.mathematik.uni-stuttgart.de/fak8/ians/lehrstuhl/nmh/downloads/alberta/`.
- **ALUGrid**: External grid library. ALUGrid is built by a C++ compiler like `g++`. If you want to build a parallel version, you will need `MPI`. It was successfully run with `openmpi`. The parallel version needs also a graph partitioner, such as `ParMETIS`. Download: `http://aam.mathematik.uni-freiburg.de/IAM/Research/alugrid`
- **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: `http://www.dune-project.org/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 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`
- **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: `http://www.iwr.uni-heidelberg.de/frame/iwrwikiequipment/software/ug`

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**: Alberta and SuperLU make 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. A good choice is the GNU compiler suite consisting of `gcc`, `g++` and `gfortran`.

11.4.2 Hints for Users from IWS

We provide some features to make life a little bit easier for users from the Institute for Modelling Hydraulic and Environmental Systems, University of Stuttgart. There exists internally a Subversion repository made for several external libraries. If you are allowed to access it, go to the DUNE-Root, then type the following.

Prepare external directory:

```
$ # Make sure you are in DUNE-Root
$ svn checkout svn://svn.iws.uni-stuttgart.de/DUMUX/external/trunk external
```

This directory `external` contains a script to install external libraries. type `help` to see which modules are currently available:

```
$ cd external
$ ./installExternal.sh help

Usage: ./external/installExternal.sh [OPTIONS] PACKAGES

Where PACKAGES is one or more of the following
  all              Install everything and the kitchen sink.
  alberta          Install the alberta grid library.
  alu              Download dune-alugrid.
  metis            Install the METIS graph partitioner.
  multidomain      Download dune-multidomain.
  multidomaingrid  Download dune-multidomaingrid.
  ug               Install the UG grid library.
  gstat            Install the Gstat library.

The following options are recoginzed:
    --parallel       Enable parallelization if available.
    --debug          Compile with debugging symbols and without optimization.
    --clean          Delete all files for the given packages.
```

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`.

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.

[2] The ALBERTA website: `http://www.alberta-fem.de/`.

[3] The ALUGrid website: `http://www.mathematik.uni-freiburg.de/IAM/Research/alugrid/`.

[4] The apache subversion website: `http://subversion.apache.org/`.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part ii: implementation and tests in DUNE. *Computing*, 82(2):121–138, 2008.

[6] A. Bielinski. *Numerical Simulation of CO_2 Sequestration in Geological Formations*. PhD thesis, Institut für Wasserbau, Universität Stuttgart, 2006.

[7] 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.

[8] 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.

[9] Doxygen homepage: `http://www.stack.nl/~dimitri/doxygen/`.

[10] DuMuX homepage: `http://www.dumux.org/`.

[11] DuMuX download folder: `http://www.dumux.org/download/`.

[12] The DUNE project: `http://www.dune-project.org/`.

[13] DUNE build system howto: `http://www.dune-project.org/doc/buildsystem/buildsystem.pdf`.

[14] Download of DUNE via git: `http://www.dune-project.org/downloadgit.html`.

[15] Use of external libraries in dune `http://www.dune-project.org/external_libraries/index.html`.

[16] Use of external modules in dune `http://www.dune-project.org/downloadext.html`.

[17] Installation notes to DUNE: `http://www.dune-project.org/doc/installation-notes.html`.

[18] DUNE user wiki, prerequisite software: `http://users.dune-project.org/projects/main-wiki/wiki/Installation_prerequisite_software`.

[19] IAPWS (The International Association for the Properties of Water and Steam). Revised release on the iapws industrial formulation 1997 for the thermodynamic properties of water and steam. `http://www.iapws.org/IF97-Rev.pdf`, 1997.

[20] R. Helmig. *Multiphase Flow and Transport Processes in the Subsurface — A Contribution to the Modeling of Hydrosystems*. Springer Verlag, 1997.

[21] J. E. Killough and C. A. Kossack. Fifth comparative solution project: Evaluation of miscible flood simulators. *Society of Petroleum Engineers*, SPE 16000, 1987.

[22] A. Lauser, C. Hager, R. Helmig, and B. Wohlmuth. A new approach for phase transitions in miscible multi-phase flow in porous media. *Advances in Water Resources*, 34(8):957–966, 2011.

[23] R.C. Reid, J.M. Prausnitz, and B.E. Poling. *The Properties of Gases and Liquids*. McGraw-Hill Inc., 1987.

[24] The UG homepage: `http://www.iwr.uni-heidelberg.de/frame/iwrwikiequipment/software/ug`.

[25] Wikipedia about gnu build system: `http://en.wikipedia.org/wiki/GNU_build_system`.

[26] Wikipedia about aliasing data location in memory: `http://en.wikipedia.org/wiki/Aliasing_(computing)`.