The new version of DuMu^x including the modules "CRootBox" and "dumux-rosi"

Documentation

Manual written by A. Schnepf

May 14, 2019

Installation

This installation guidelines are for the new version of DuMu^x, version 3, coupled with CRootBox, in Linux systems (e.g. Ubuntu).

Required compilers and tools

If on a recent Ubuntu system, the c++ compiler and python that come with the distribution are recent enough. Otherwise, please make sure you have a recent c++ compiler (e.g. `sudo apt-get install clang`) and python3 (e.g. `sudo apt-get install python3.6`).
- Install git:
`sudo apt-get install git`
- Install cmake:
`sudo apt-get install cmake`
- Install libboost:
`sudo apt-get install libboost-all-dev`
- Install pip:
`sudo apt-get install python3-pip`
- Install the python package numpy:
`pip3 install numpy`
- Install the python package scipy:
`pip3 install scipy`
- Install the python package matplotlib:
`pip3 install matplotlib`
- Install the java runtime environment:
`sudo apt-get install default-jre`
- Install Paraview
`sudo apt-get install paraview`

DuMu^x installation

In all dune modules we stay in version 2.6, the latest stable release version.

- Create a DUMUX folder

mkdir DUMUX

cd DUMUX

- Download DUNE core modules:

```
git clone https://gitlab.dune-project.org/core/dune-common.git
cd dune-common
git checkout releases/2.6
cd ..
git clone https://gitlab.dune-project.org/core/dune-geometry.git
cd dune-geometry
git checkout releases/2.6
cd ..
git clone https://gitlab.dune-project.org/core/dune-grid.git
cd dune-grid
git checkout releases/2.6
cd ..
git clone https://gitlab.dune-project.org/core/dune-istl.git
cd dune-istl
git checkout releases/2.6
cd ..
git clone https://gitlab.dune-project.org/core/dune-localfunctions.git
cd dune-localfunctions
git checkout releases/2.6
cd ..
```

- Download DUNE external modules:

```
git clone https://gitlab.dune-project.org/extensions/dune-foamgrid.git
cd dune-foamgrid
git checkout releases/2.6
cd ..
git clone https://gitlab.dune-project.org/extensions/dune-grid-glue.git
cd dune-grid-glue
git checkout releases/2.6
cd ..
```

-Download dumux and dumux-rosi and alugrid (used for unstructured grids):

```
git clone https://git.iws.uni-stuttgart.de/dumux-repositories/dumux.git
cd dumux
git checkout releases/3.0
cd ..
git clone https://github.com/Plant-Root-Soil-Interactions-Modelling/dumux-rosi.git
cd dumux-rosi
git checkout master
```

```
cd ..
git clone https://gitlab.dune-project.org/extensions/dune-alugrid.git
```

-Download CRootBox (only needed if root growth is used):

```
git clone https://github.com/Plant-Root-Soil-Interactions-Modelling/CRootBox.git
cd CRootBox
git checkout master
cd ..
```

To build CRootBox and its python shared library, move again into the CRootBox folder and type into the console:

cmake .

make

(If building CRootBox on the cluster, two lines in the file CRootBox/CMakeLists.txt need to be outcommented before:

set(CMAKE_C_COMPILER "/usr/bin/gcc")

set(CMAKE_CXX_COMPILER "/usr/bin/g++"))

Now build DuMu^x with the CRootBox module:

-The configuration file optim.opts is stored in the dumux folder. Move a copy of this file to your DuMu^x working folder (one level up)

- To build all downloaded modules and check whether all dependencies and prerequisites are met, run dunecontrol:

./dune-common/bin/dunecontrol --opts=optim.opts all

Installation done! Good luck!

Running an example

```
cd dumux-rosi/build-cmake/rosi_benchmarking/soil
make richards1d      # outcomment if executable is already available
./richards1d benchmarks_1d/b1a.input   # run executable with specific
   input parameter file
```

Installing and running an example on the agrocluster

- Before installing or running DuMu^x on the agrocluster, it is required to type the command module load dumux into the console. This sets the compiler versions and other tools to more recent versions than the standard versions of the agrocluster.

- On the cluster, another onfiguration file optim_cluster.opts is used. Copy this file to the file to your DuMu^x working folder (one level up).

- To build or run an example on the agrocluster, create a pbs file in your working folder that will put your job in the cluster queue
For example queue_my_job.pbs

```
#!/bin/sh
#
#These commands set up the Grid Environment for your job:
#PBS -N DUMUX
#PBS -l nodes=1:ppn=1,walltime=200:00:00,pvmem=200gb
#PBS -q batch\\
#PBS -M a.schnepf@fz-juelich.de
#PBS -m abe

module load dumux
cd  \$HOME/DUMUX/dumux-rosi/build-cmake/rosi_benchmarking/soil
make richards
./richards benchmarks_1d/b1a.input
```

To start the job, run this file in your working folder with the command
qsub queue_my_job.pbs
Use Filezilla to move the results to your local machine and use Paraview to visualize them.
If you need to install additional python packages (e.g. scipy) on the cluster (without root access), you may do so by using the --user command:
pip3 install --user scipy

Numerical grids

We distinguish two types of numerical grids: the 3D soil grid and the 1D, branched, root system grid (see Fig. 1).

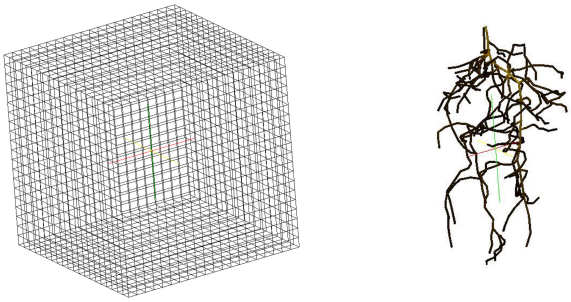


Figure 1: The 3D soil grid and the 1D, branched, grid representing the root architecture

In the example of the coupled problems, both are used simultaneously. In that case, the two grids are merged via source/sink terms in positions where root and soil grids share the same spatial coordinates. This is illustrated in Fig. 2; detailed descriptions can be found in the individual examples.

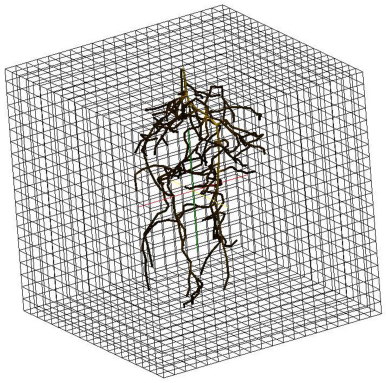


Figure 2: 3D soil grid merged with the 1D, branched, grid representing the root architecture

Grids can be created using different DUNE internal or external grid managers (see documentation of dune-grid). In the input file, the details about the numerical grids are specified in the groups [RootSystem.Grid] or [SoilGrid]. Each folder contains a

folder named “grids” where grids can be provided in dgf format. In the dumux-rosi examples, the soil grid is usually a structured grid created by the default “GridCreator”, where corner points of the domain, spatial resolution and cell type are specified such as in the following example:

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

Unstructured grids can be used, for example, with the grid manager ”Alugrid”. The dune module dune-alugrid can be downloaded from this repository: `https://gitlab.dune-project` It can, for example, read in mesh files generated with the mesh generator GMsh. DuMu^x needs to know the type of grid and grid manager at compile time. Therefore, an entry in the `CMakeLists.txt` file at the ”problem” level tells DuMu^x to use a certain grid manager, and this will then result in a new executable per new grid manager. See below for an example: `dumux-rosi/rosi_benchmarking/soil/CMakeLists.txt`.

```
# create a link to the grid file and the input file in the build
    directory
dune_symlink_to_source_files(FILES "grids" "python" "benchmarks_1d" "
    benchmarks_3d" "benchmarks_ug")

add_executable(richards3d EXCLUDE_FROM_ALL richards.cc)
target_compile_definitions(richards3d PUBLIC GRIDTYPE=Dune::YaspGrid<3>)

add_executable(richards1d EXCLUDE_FROM_ALL richards.cc)
target_compile_definitions(richards1d PUBLIC GRIDTYPE=Dune::FoamGrid
    <1,1>)

add_executable(richardsUG EXCLUDE_FROM_ALL richards.cc)
target_compile_definitions(richardsUG PUBLIC GRIDTYPE=Dune::ALUGrid<3,3,
    Dune::simplex,Dune::conforming>)

# optionally set cmake build type (Release / Debug / RelWithDebInfo)
set(CMAKE_BUILD_TYPE RelWithDebInfo)
```

Listing 1: Boundary conditions

,
There are two options to specify the root system grid. The first option is to specify it as a file in dgf-format that specifies the coordinates and connection of nodes (verteces).

```
DGF\verb+ dumux-rosi/rosi_benchmarking/soil/richardsproblem.hh+.\\
```

```
\lstinputlisting[firstline=188,lastline=224, language=C++, caption=
    Boundary conditions]{dumux-rosi/rosi_benchmarking/soil/richardsproblem
    .hh},
Vertex
0.050000 0.050000 -0.000000
0.050000 0.050000 -0.0250000
0.050000 0.050000 -0.05000
0.050000 0.050000 -0.075000
#
SIMPLEX
parameters 10
0 1 1 0 3.14159e-05 0.01 0.0005 0.00 0.0001 0.00001 21922.1
1 2 1 0 2.51327e-05 0.008 0.0005 0.00 0.0001 0.00001 39940.5
2 3 1 0 2.51327e-05 0.008 0.0005 0.00 0.0001 0.00001 58409.5
3 4 1 0 2.51327e-05 0.008 0.0005 0.00 0.0001 0.00001 77352.1
4 5 1 0 2.51327e-05 0.008 0.0005 0.00 0.0001 0.00001 96793.2
5 6 1 0 2.51327e-05 0.008 0.0005 0.00 0.0001 0.00001 116760
#
BOUNDARYDOMAIN
default 1
#
```

The paragraph "SIMPLEX" specifies 10 parameters for each root segment: node1ID, node2ID, type, branchID, surfaceIdx, length, radiusIdx, massIdx, axialPermIdx, radialPermIdx, creationTimeId in SI units.
Root systems in dgf format can be computed from measured root systems as well as with the root architecture model CRootBox.
The second option is to provide the root architectural parameters in the input file such that the root architecture and related grid is computed by CRootBox while used as a DuMu^x module.

```
[RootSystem.Grid]
File = Triticum_aestivum_a_Bingham_2011
InitialT = 10 # days
```

Important to know: It is currently necessary to build the code either for option 1 or for option 2 (i.e., two executables can be built that need to be provided with the correct input at runtime).

Python for pre- and postprocessing

We created a python layer around CRootBox and dumux-rosi for pre- and postprocessing such that the model can be run without handling the C++ code once an executable is available.
For that, each example folder contains a folder named ”python“ that includes several examples as well a folder that includes the corresponding input files.

The pre-processing

Here, the path to the executable and corresponding input files is provided and the simulation is started, like in this example:

```
# go to the right place
path = os.path.dirname(os.path.realpath(__file__))
os.chdir(path)
os.chdir("../../../build-cmake/rosi_benchmarking/soil")

# run dumux
os.system("./richards1d benchmarks_1d/b1a.input")
```

The input file

Here is an example of an input file,
`/dumux-rosi/rosi_benchmarking/soil/benchmarks_1d/b1a.input`:

```
[Problem]
Name = benchmark1d_1a

[TimeLoop]
TEnd = 3153600 # 0 is steady state
DtInitial =  1 # [s]
MaxTimeStepSize = 864000 # 10 days [s]

[Soil.Grid]
UpperRight = 0
LowerLeft = -2
Cells = 199

[Soil.BC.Top]
Type = 2 # constant flux
```

```
Value = 0.5 # [cm/d]

[Soil.BC.Bot]
Type = 5 # free drainage

[Soil.IC]
P = -200 # cm pressure head (initial guess)

[Soil.VanGenuchten]
# Loam over sand
Qr = 0.08  0.045
Qs = 0.43  0.43
Alpha = 0.04  0.15 # [1/cm]
N = 1.6  3
Ks = 50  1000 # [cm/d]

[Soil.Layer]
Z = -2  -0.5  -0.5  0
Number = 2  2  1  1
```

Listing 2: Example input file

Todo: periodic boundary conditions

The post-processing

3D simulation results are stored in form of vtk files. If not specified otherwise, vtk files are stored for the initial and the final time point of the simulation. Using the key word "CheckTimes" under the category "Time loop" in the input file, additional output times can be specified. Time series, such as transpiration flux or pressure at the root collar over time, are stored as txt files. At the moment, this is specified within the problem file of the C++ code, see for example `/dumux-rosi/rosi_benchmarking/rootsystem/rootsproblem.hh`:

Listing 3: Transpiration output

and

```
    //! sets the current simulation time [s] (within the simulation loop)
    for collar boundary look up
    void setTime(Scalar t) {
```

Listing 4: Transpiration output

Here, results are read and plotted or further analysed, like in the following example. Using the vtk_tools is particularly helpful for creating 1D plots such as depth profiles or time series in Python rather than using Paraview (Paraview of course is helpful for 3D visualisation).

```
# Figure 2a
s_, p_, z1_ = read1D_vtp_data("benchmark1d_1a-00001.vtp", False)
h1_ = vg.pa2head(p_)
ax1.plot(h1_, z1_ * 100, "r+")

np.savetxt("dumux1d_b1", np.vstack((z1_, h1_, z2_, h2_, z3_, h3_)),
    delimiter = ",")
```

Benchmarking example 1: Water flow in soil

Currently, benchmarks are developed to test dumux-rosi against analytical solutions and results of other numerical models. They are all described in Jupyter Notebooks at https://github.com/RSA−benchmarks/collaborative−comparison. Here, we describe the DuMu^x-implementation of the 1D benchmarks of Vanderborght et al. (2005) for water flow in soil.

The model

We solve the Richards equation for water flow in soil. Since DuMu^x is developed for multi-phase flow in porous media, it uses units of absolute pressure of wetting and non-wetting phases. In the Richards equation, we assume that the non-wetting phase (air) does not change over time and has a constant pressure of 1.0×10^5 Pa. Thus, we need to solve only the equation for the wetting phase (water). We stick to the standard DuMu^x units for pressure, although in soil physics, head units are more common, in order to avoid mistakes of e.g. unconsidered hard coded constants, etc. The Richards equation thus can be written as

$$\frac{\partial}{\partial t}\left(\rho_w \Phi S\right) - \nabla \cdot \left[\rho_w \frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \mathbf{g}\right)\right] = 0, \tag{1}$$

with t time, θ water content, S saturation, Φ porosity, $S\phi = \theta$, ρ_w water density, K intrinsic permeability, μ dynamic viscosity, κ relative permeability, $\mathbf{g}$ gravitational acceleration, p_w absolute pressure of wetting phase (water)[1]. θ and h_m are related by the water retention curve: $\theta := \theta(h)$ (e.g. van Genuchten model).
Different initial and boundary conditions can be prescribed via the input file. Boundary conditions have number codes following (previous versions of) Hydrus:
constantPressure = 1,
constantFlux = 2,
atmospheric = 4,
freeDrainage = 5.

[1] p_w is the absolute pressure. The matric pressure p_m is defined as $p_m = p_w - p_a$, where p_a is the air pressure, assumed to be constant and equal to 1.0×10^5 Pa in this Richards equation model. In order to have head units, we need to convert the water potential from energy per unit volume of water (pressure) to energy per unit weight, i.e., $h_m = \frac{p_m}{\rho_w \mathbf{g}}$

The input files

Model parameters, initial and boundary conditions can be specified via the input file such that no re-building of the code is required. Here is the listing of the input file dumux−rosi/rosi_benchmarking/soil/benchmarks_1d/b1a.input:

```
[Problem]
Name = benchmark1d_1a

[TimeLoop]
TEnd = 3153600 # 0 is steady state
DtInitial = 1 # [s]
MaxTimeStepSize = 864000 # 10 days [s]

[Soil.Grid]
UpperRight = 0
LowerLeft = -2
Cells = 199

[Soil.BC.Top]
Type = 2 # constant flux
Value = 0.5 # [cm/d]

[Soil.BC.Bot]
Type = 5 # free drainage

[Soil.IC]
P = -200 # cm pressure head (initial guess)

[Soil.VanGenuchten]
# Loam over sand
Qr = 0.08  0.045
Qs = 0.43  0.43
Alpha = 0.04  0.15 # [1/cm]
N = 1.6  3
Ks = 50  1000 # [cm/d]

[Soil.Layer]
Z = -2  -0.5  -0.5  0
Number = 2  2  1  1
```

Listing 5: input file

The DuMu^x code representation of model equations

In this section, we explain where the different terms of the model equations can be found in the DuMu^x code, i.e., the storage, flux and sink terms. The storage term is defined in the file

/dumux/dumux/porousmediumflow/Richards/localresidual.hh, and is computed as

```
        storage[conti0EqIdx] = volVars.porosity()
                             * volVars.density(liquidPhaseIdx)
                             * volVars.saturation(liquidPhaseIdx);
```

Listing 6: Storage term

In this example, there is no source or sink term.
The flux term is hidden in deeper layers of the code as part of the numerical scheme. In order to see it, we have to find out the flux type of the problem in the file /dumux/dumux/porousmediu
In this example, the flux type is "darcyslaw". Its implementation can then be found in the folder /dumux/dumux/flux, and is then different for the different numerical schemes (e.g. cell-centered finite volume scheme with two-point flux approximation (TPFA)).
The implementation of the boundary conditions specified in the input file are implemented in the file
dumux-rosi/rosi_benchmarking/soil/richardsproblem.hh.

```
     */
    BoundaryTypes boundaryTypesAtPos(const GlobalPosition &globalPos)
   const
    {
        BoundaryTypes bcTypes;
        if (onUpperBoundary_(globalPos)) { // top bc
            switch (bcTopType_) {
            case constantPressure:
                bcTypes.setAllDirichlet();
                break;
            case constantFlux:
                bcTypes.setAllNeumann();
                break;
            case atmospheric:
                bcTypes.setAllNeumann();
                break;
            default:
                DUNE_THROW(Dune::InvalidStateException,"Top boundary type
     not implemented");
            }
        } else if (onLowerBoundary_(globalPos)) { // bot bc
            switch (bcBotType_) {
            case constantPressure:
                bcTypes.setAllDirichlet();
                break;
            case constantFlux:
                bcTypes.setAllNeumann();
                break;
            case freeDrainage:
                bcTypes.setAllNeumann();
```

```
                break;
            default:
                DUNE_THROW(Dune::InvalidStateException,"Bottom boundary
    type not implemented");
            }
        } else {
            bcTypes.setAllNeumann(); // no top not bottom is no flux
        }
        return bcTypes;
    }
```

Listing 7: Boundary conditions

,

```
     * dirchlet(...) is called by the local assembler, e.g.
    BoxLocalAssembler::evalDirichletBoundaries
     */
    PrimaryVariables dirichletAtPos(const GlobalPosition &globalPos)
    const {
        PrimaryVariables values;
        if (onUpperBoundary_(globalPos)) { // top bc
            switch (bcTopType_) {
            case constantPressure:
                values[Indices::pressureIdx] = toPa_(bcTopValue_);
                break;
            default:
                DUNE_THROW(Dune::InvalidStateException,
                    "Top boundary type Dirichlet: unknown boundary type")
    ;
            }
        } else if (onLowerBoundary_(globalPos)) { // bot bc
            switch (bcBotType_) {
            case constantPressure:
                values[Indices::pressureIdx] = toPa_(bcBotValue_);
                break;
            default:
                DUNE_THROW(Dune::InvalidStateException,
                    "Bottom boundary type Dirichlet: unknown boundary
    type");
            }
        }
        values.setState(Indices::bothPhases);
```

Listing 8: Boundary conditions

and

```
     * \copydoc FVProblem::neumann // [kg/(m²*s)]
     *
     * called by BoxLocalResidual::evalFlux
     */
```

```
5     NumEqVector neumann(const Element& element,
6         const FVElementGeometry& fvGeometry,
7         const ElementVolumeVariables& elemVolVars,
8         const SubControlVolumeFace& scvf) const {
9
10        NumEqVector values;
11        GlobalPosition pos = scvf.center();
12        if (onUpperBoundary_(pos)) { // top bc
13            switch (bcTopType_) {
14            case constantFlux: {
15                values[conti0EqIdx] = -bcTopValue_*rho_/(24.*60.*60.)
    /100; // cm/day -> kg/(m²*s)
16                break;
17            }
18            case atmospheric: { // atmospheric boundary condition (with
    surface run-off) // TODO needs testing & improvement
19                Scalar s = elemVolVars[scvf.insideScvIdx()].saturation(0)
    ;
20                Scalar Kc = this->spatialParams().hydraulicConductivity(
    element); //  [m/s]
21                MaterialLawParams params = this->spatialParams().
    materialLawParams(element);
22                Scalar p = MaterialLaw::pc(params, s) + pRef_;
23                Scalar h = -toHead_(p); // todo why minus -pc?
24                GlobalPosition ePos = element.geometry().center();
25                Scalar dz = 100 * 2 * std::abs(ePos[dimWorld - 1] - pos[
    dimWorld - 1]); // cm
26                Scalar t = time_/(24.*60.*60.); // s -> day
27                Scalar prec = -precipitation_.f(t)*rho_/(24.*60.*60.)
    /100; // cm/day -> kg/(m²*s)
28
29                if (prec < 0) { // precipitation
30                    Scalar imax = rho_ * Kc * ((h - 0.) / dz - 1.); //
    maximal infiltration
31                    Scalar v = std::max(prec, imax);
32                    values[conti0EqIdx] = v;
33                } else { // evaporation
34                    Scalar krw = MaterialLaw::krw(params, s);
35                    Scalar emax = rho_ * krw * Kc * ((h -
    criticalPressure_) / dz - 1.); // maximal evaporation
36                    Scalar v = std::min(prec, emax);
37                    // std::cout << prec << ", " << emax << ", " << h <<
    "\n";
38                    values[conti0EqIdx] = v;
39                }
40                // hack for benchmark 4 TODO some better concept for
    output
41                if (time_ > last_time_) { // once per time step
42                    myfile_ << time_ << ", "; //
43                    myfile_ << values[conti0EqIdx] << "\n";
```

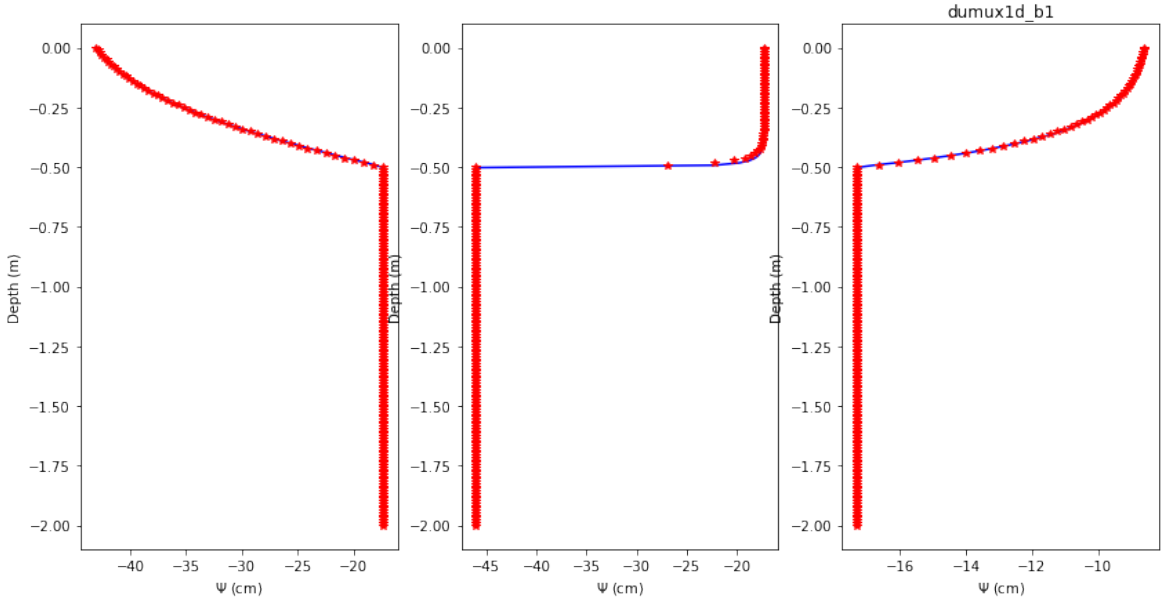
```
                    last_time_ = time_;
                }
                break;
            }
            default:
                DUNE_THROW(Dune::InvalidStateException,
                    "Top boundary type Neumann: unknown error");
            }
        } else if (onLowerBoundary_(pos)) { // bot bc
            switch (bcBotType_) {
            case constantFlux: {
                values[conti0EqIdx] = -bcBotValue_*rho_/(24.*60.*60.)
    /100; // cm/day -> kg/(m²*s)
                break;
            }
            case freeDrainage: {
                Scalar Kc = this->spatialParams().hydraulicConductivity(
    element);
                Scalar s = elemVolVars[scvf.insideScvIdx()].saturation(0)
    ;
                MaterialLawParams params = this->spatialParams().
    materialLawParams(element);
                Scalar krw = MaterialLaw::krw(params, s);
                values[conti0EqIdx] = krw * Kc * rho_; // * 1 [m]
                break;
            }
            default:
                DUNE_THROW(Dune::InvalidStateException,
                    "Bottom boundary type Neumann: unknown error");
            }
        } else {
            values[conti0EqIdx] = 0.;
        }
        return values;
    }

    /*!
     * \copydoc FVProblem::source
     *
```

Listing 9: Boundary conditions

.

Results

The vtk output of 3D simulations may be visualised using Paraview. In this case, we only have a 1D simulation, therefore, we do the visualisation after postprocessing in Python.



Results of benchmark problem 1. Blue: analytical solution, Red: numerical solution by DuMu^x