# The new version of DuMu$^x$ including the modules “CRootBox” and “dumux-rosi”

**Documentation**

Manual written by A. Schnepf

October 27, 2020

# Installation

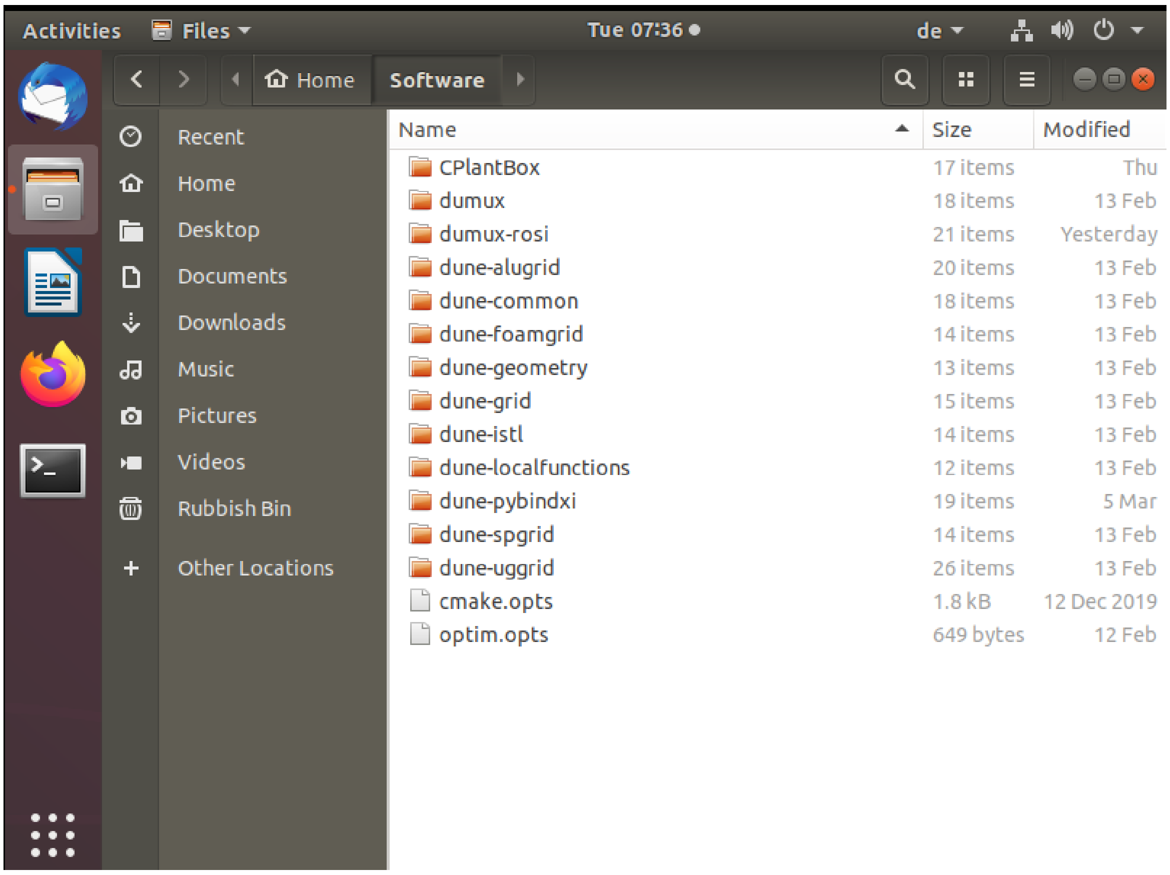
This installation guidelines are for the new version of DuMu$^x$, version 3, coupled with CRootBox, in Linux systems (e.g. Ubuntu).
On Windows or Mac, install a virtual machine with a Linux system, e.g. using `VMWare`. Provide at least 60 GB disk space when setting up the virtual machine.

## 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. The final folder structure of the different modules should look like in Fig. 1:



| Name | Size | Modified |
|---|---|---|
| CPlantBox | 17 items | Thu |
| dumux | 18 items | 13 Feb |
| dumux-rosi | 21 items | Yesterday |
| dune-alugrid | 20 items | 13 Feb |
| dune-common | 18 items | 13 Feb |
| dune-foamgrid | 14 items | 13 Feb |
| dune-geometry | 13 items | 13 Feb |
| dune-grid | 15 items | 13 Feb |
| dune-istl | 14 items | 13 Feb |
| dune-localfunctions | 12 items | 13 Feb |
| dune-pybindxi | 19 items | 5 Mar |
| dune-spgrid | 14 items | 13 Feb |
| dune-uggrid | 26 items | 13 Feb |
| cmake.opts | 1.8 kB | 12 Dec 2019 |
| optim.opts | 649 bytes | 12 Feb |

Folder structure of DuMu$^x$, Dune and CPlantBox modules

- 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/CPlantBox.git
cd CPlantBox
git checkout master
cd ..
```
To build CPlantBox and its python shared library, move again into the CPlantBox folder and type into the console:
```
cmake .
make
```
(If building CPlantBox on the cluster, two lines in the file CPlantBox/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 CPlantBox 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. 2).

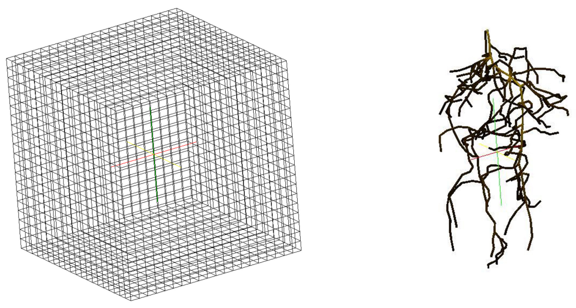


Figure 2: The 3D soil grid and the 1D, branched, grid representing the root architecture

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. 3; detailed descriptions can be found in the individual examples.

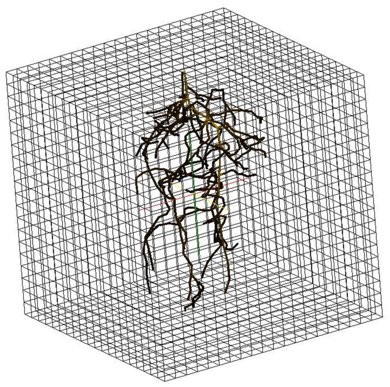


Figure 3: 3D soil grid merged with the 1D, branched, grid representing the root architecture

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 [Soil.Grid]. 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
```

Alternatively, msh-files can be read.

## Choice of grid manager for the soil domain

**YaspGrid**: Structured grids, only non-periodic soil domains
**SpGrid**: Structured grids, also periodic soil domains
**AluGrid**: Unstructured grids, only works for $\texttt{CC}_2 pfa - scheme$
`UGGrid` $: Unstructuredgrids, worksforboth$

To change the grid manager, open the file `dumux-rosi/rosi_benchmarking/coupled_1p_richards` `/CMakeLists.txt`. If not available, add the following lines to make the gridmanager available to build an executable. For the example of UGGrid:

```
add_executable(coupledUG EXCLUDE_FROM_ALL coupled.cc)
target_compile_definitions(coupledUG PUBLIC DGF GRIDTYPE=Dune:: UGrid<3>)
```

## Grids for root systems

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
Vertex
0 0 -0.03
-0.003301 -0.000687124 -0.0394144
-0.00339314 -0.00159054 -0.0473627
-0.00590116 -0.00546716 -0.0538958
-0.0115931 -0.00454388 -0.059441
-0.0105464 -0.00572357 -0.067284
-0.0100044 -0.0069212 -0.0751753
-0.00923561 -0.00814568 -0.0830435
-0.0100507 -0.00678732 -0.0908851
-0.010965 -0.00608665 -0.0962792
-0.00103059 0.000281757 -0.0413509
```

```
0.00284233 0.00172545 -0.0449409
0.00619859 0.00411514 -0.0466909
-6.51815e-05 0.00081926 -0.041154
0.000989275 0.00146981 -0.0404847
-0.00013962 0.00162442 -0.0411274
-0.00376417 0.00152097 -0.0477906
-0.00509243 0.00809897 -0.0472907
-0.00667036 0.0130109 -0.0453872
-0.00784569 0.0163412 -0.0446723
-0.00343402 0.00160078 -0.048867
-0.00272936 0.00153492 -0.0511603
-0.00257298 0.00150318 -0.0517729
-0.00605319 0.00128567 -0.051235
-0.00509341 0.00874088 -0.0479026
-0.0051018 0.00890818 -0.0480774
-0.0105648 -0.00490006 -0.0536398
-0.0155357 -0.00427108 -0.0535009
-0.0187161 -0.00393935 -0.0540514
-0.0129246 -0.00794987 -0.0599204
-0.0158011 -0.0137093 -0.0604436
-0.0159531 -0.0140359 -0.0604843
-0.0103269 -0.00178528 -0.069703
-0.0097869 0.000348939 -0.071026
-0.0101116 -0.00387942 -0.0769314
-0.00866863 -0.0079157 -0.0834016
#
SIMPLEX
parameters 10 # id0, id1, order, branchId, surf[cm2], length[cm], radius[
    cm], kz[cm4 hPa-1 d-1], kr[cm hPa-1 d-1], emergence time [d], subType,
     organType
0 1 0 1 0.314159 1 0.05 0 0 0.253641 1 2
1 2 0 1 0.251327 0.8 0.05 0 0 0.461984 1 2
2 3 0 1 0.251327 0.8 0.05 0 0 0.675409 1 2
3 4 0 1 0.251327 0.8 0.05 0 0 0.894171 1 2
4 5 0 1 0.251327 0.8 0.05 0 0 1.11854 1 2
5 6 0 1 0.251327 0.8 0.05 0 0 1.34882 1 2
6 7 0 1 0.251327 0.8 0.05 0 0 1.58532 1 2
7 8 0 1 0.251327 0.8 0.05 0 0 1.82839 1 2
8 9 0 1 0.17328 0.551569 0.05 0 0 2 1 2
1 10 1 2 0.0591391 0.313743 0.03 0 0 0.81929 2 2
10 11 1 2 0.103194 0.547463 0.03 0 0 1.36938 2 2
11 12 1 2 0.084377 0.447634 0.03 0 0 2 2 2
10 13 2 3 0.0141039 0.112236 0.02 0 0 1.88447 3 2
13 14 2 3 0.0176961 0.140821 0.02 0 0 2 3 2
13 15 3 4 0.0101666 0.080903 0.02 0 0 2 4 2
2 16 1 8 0.0596143 0.316264 0.03 0 0 0.976223 2 2
16 17 1 8 0.126845 0.672936 0.03 0 0 1.39819 2 2
17 18 1 8 0.103656 0.549912 0.03 0 0 1.75726 2 2
18 19 1 8 0.0679195 0.360324 0.03 0 0 2 2 2
16 20 2 9 0.0141835 0.112869 0.02 0 0 1.55953 3 2
```

```
20 21 2 9 0.0301593 0.24 0.02 0 0 1.81594 3 2
21 22 2 9 0.00795504 0.0633042 0.02 0 0 2 3 2
20 23 3 10 0.0445473 0.354496 0.02 0 0 2 4 2
17 24 2 12 0.0111449 0.088688 0.02 0 0 1.98626 3 2
24 25 2 12 0.00304236 0.0242104 0.02 0 0 2 3 2
3 26 1 16 0.088687 0.470499 0.03 0 0 1.35817 2 2
26 27 1 16 0.0944815 0.50124 0.03 0 0 1.74414 2 2
27 28 1 16 0.0611608 0.324468 0.03 0 0 2 2 2
4 29 1 19 0.0695225 0.368828 0.03 0 0 1.49515 2 2
29 30 1 19 0.121751 0.645908 0.03 0 0 1.97249 2 2
30 31 1 19 0.00683211 0.0362455 0.03 0 0 2 2 2
5 32 1 22 0.0872183 0.462707 0.03 0 0 1.79818 2 2
32 33 1 22 0.0484141 0.256845 0.03 0 0 2 2 2
6 34 1 24 0.0662369 0.351397 0.03 0 0 2 2 2
7 35 1 25 0.0133627 0.0708913 0.03 0 0 2 2 2
#
BOUNDARYDOMAIN
default 1
#
```

The paragraph ``SIMPLEX" specifies 10 parameters for each root segment: node1ID, node2ID, order, branchID, surfaceIdx in cm$^2$, length in cm , radiusIdx in cm, axialCondIdx cm$^4$ hPa$^{-1}$ d$^{-1}$, radialCondIdx cm hPa$^{-1}$ d$^{-1}$, emergenceTimeId[1]
Order numbering starts with 0, i.e., primary roots have order 0. Potential artificial shoot segements have order -1.
Root systems in dgf format can be computed from measured root systems as well as with the root architecture module of CPlantBox.
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).

[1]Note: in the code we often use the term ´´creation time”, however, we always mean ´´emergence time”. Branch nodes exist twice, once in the mother branch, once as the starting node of the daughter branch. They have different emergence times but the same nodeID.

# Numerical schemes

## Numercical schemes available in DuMu$^x$

### Box Method

### CC$_2pfaMethod$

## How to switch numerical scheme in a DuMu$^x$ simulation

Open the main file of your application. For coupled root-soil problems, it is for example the file `coupled.cc`.
Change the line that specifies the numerical scheme for the soil subproblem:
`using SoilTypeTag = Properties::TTag::RichardsBox;`
or `using SoilTypeTag = Properties::TTag::RichardsCC;`

Change the line that specifies the numerical scheme for the root subproblem:
`using RootTypeTag = Properties::TTag::RootsCCTpfa;`
or `using RootTypeTag = Properties::TTag::RootsBox;`

Make sure that the relevant properties file is given (line 62 in coupled.cc).

# Python for pre- and postprocessing

We created a python layer around CRootBox and dumux-rosi for pre- and postprocessin
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 soil_richards/input/b1a_1d.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 1: Example input file

## 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/roots_1p/rootsproblem.hh:

```
    //! calculates transpiraton, as the sum of radial fluxes (slow but
   accurate) [cm^3/day]
```

Listing 2: Transpiration output

and

```

    //! if true, sets bc to Dirichlet at criticalCollarPressure (false
   per default)
```

Listing 3: 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).

```python
# 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 $1.0 \times 10^5$ 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, $\theta$ water content, $S$ saturation, $\Phi$ porosity, $S\phi = \theta$, $\rho_w$ water density, $K$ intrinsic permeability, $\mu$ dynamic viscosity, $\kappa$ relative permeability, g gravitational acceleration, $p_w$ absolute pressure of wetting phase (water)[2].
$\theta$ 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:

[2] $p_w$ 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 $1.0 \times 10^5$ 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., $h_m = \frac{p_m}{\rho_w \mathbf{g}}$

constantPressure = 1,
constantFlux = 2,
atmospheric = 4,
freeDrainage = 5.

## 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_richards/input/b1a_1d .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

[Vtk]
```

```
AddProcessRank = "false"
AddVelocity = "false"
```

Listing 4: 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 5: 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/porousmediumflow/richards/.... 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_richards/richardsproblem.hh.

```
   *
   * 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);
    return values;
  }

  /*!
   * \copydoc FVProblem::neumann // [kg/(m²*s)]
   *
   * called by BoxLocalResidual::evalFlux,
   * mass flux in \f$ [ kg / (m^2 \cdot s)] \f$
   * Negative values mean influx.
   */
  NumEqVector neumann(const Element& element,
      const FVElementGeometry& fvGeometry,
      const ElementVolumeVariables& elemVolVars,
      const SubControlVolumeFace& scvf) const {

    NumEqVector flux;
    double f = 0.; // return value
    GlobalPosition pos = scvf.center();

    if ( onUpperBoundary_(pos) || onLowerBoundary_(pos) ) {
```

Listing 6: Boundary conditions

,

```
      Scalar p = MaterialLaw::pc(params, s) + pRef_; // [Pa]
      Scalar h = -toHead_(p); // cm
      GlobalPosition ePos = element.geometry().center();
      Scalar dz = 100 * std::fabs(ePos[dimWorld - 1] - pos[dimWorld - 1]); // m-> cm (*2 ?)
      Scalar krw = MaterialLaw::krw(params, s);

      if (onUpperBoundary_(pos)) { // top bc
        switch (bcTopType_) {
        case constantFlux: { // with switch for maximum in- or outflow
          f = -bcTopValue_*rho_/(24.*60.*60.)/100; // cm/day -> kg/(m²*s)
          if (f < 0.) { // inflow
            Scalar imax = rho_ * kc * ((h - 0.) / dz - gravityOn_); // maximal inflow
            // std::cout << "in:" << f <<", " << imax <<"\n";
            f = std::max(f, imax);
          } else { // outflow
            Scalar omax = rho_ *  kc * krw * ((h - criticalPressure_) / dz - gravityOn_); // maximal outflow (evaporation)
            // std::cout << "outflow " << f*1.e6 << ", " << omax*1.e6 << " krw " << krw*1.e6 << "\n";
            f = std::min(f, omax);
          }
          break;
        }
        case constantFluxCyl: { // upper = outer, with switch for maximum in- or outflow
          f = -bcTopValue_*rho_/(24.*60.*60.)/100 * pos[0];  // [cm /day] -> [kg/(m²*s)] (Eqns are multiplied by cylindrical radius)
          if (f < 0.) { // inflow
```

Listing 7: Boundary conditions

and

```
        f = std::min(f, omax);
      }
      break;
    }
    case atmospheric: { // atmospheric boundary condition (with surface run-off)
      Scalar prec = -precipitation_.f(time_);
      if (prec < 0.) { // precipitation
        // std::cout << "in" << "\n";
        Scalar imax = rho_ * kc * ((h - 0.) / dz - gravityOn_); // maximal infiltration
        f = std::max(prec, imax);
      } else { // evaporation
        // std::cout << "out" << ", at " << h << " cm \n";
          Scalar p2 = toPa_(-10000);
          Scalar h3 = 0.5*(h + criticalPressure_);
          Scalar p3 = toPa_(h);
          Scalar s2 = MaterialLaw::sw(params, -(p2- pRef_));
                    Scalar s3 = MaterialLaw::sw(params, -(p3- pRef_));
          // std::cout << s2 << "\n";
          Scalar krw2 = MaterialLaw::krw(params, s2);
          Scalar krw3 = MaterialLaw::krw(params, s3);
                    Scalar arithmetic = 0.5*(krw2+krw); // arithmetic currently best
          Scalar harmonic = 2*krw2*krw/(krw2+krw);
        Scalar emax = rho_ * kc * arithmetic *((h - criticalPressure_) / dz + gravityOn_); // maximal evaporation KRW???
        f = std::min(prec, emax);
      }
      break;
    }
    default: DUNE_THROW(Dune::InvalidStateException, "Top boundary type Neumann: unknown error");
```

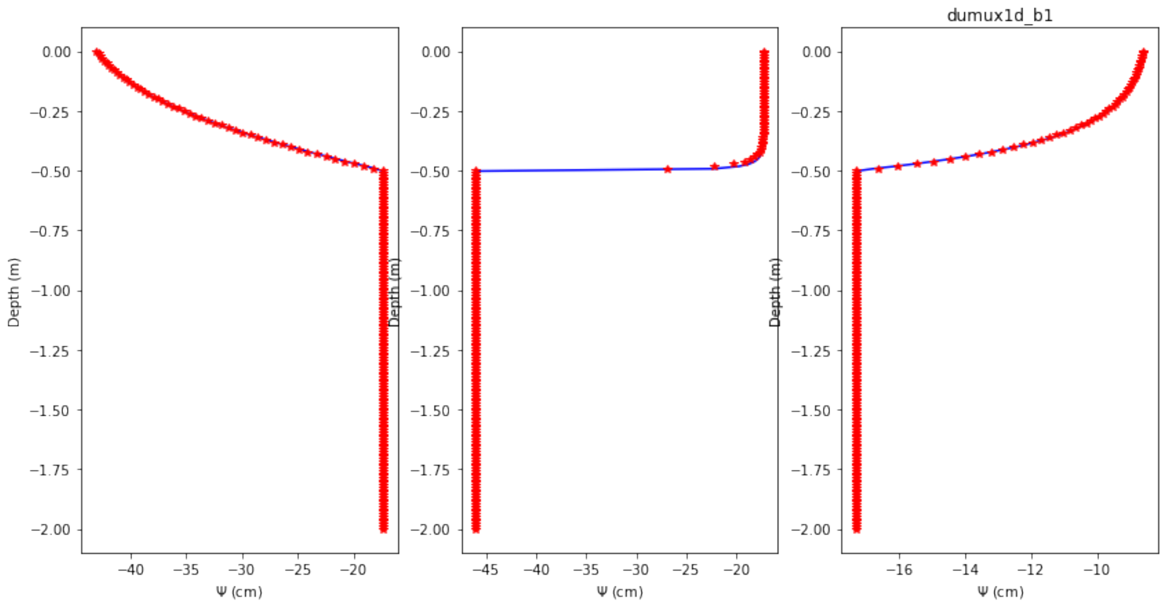
```
        }
      } else if (onLowerBoundary_(pos)) { // bot bc
        switch (bcBotType_) {
        case constantFlux: { // with switch for maximum in- or outflow
          f = -bcBotValue_*rho_/(24.*60.*60.)/100.; // [cm /day] -> [kg/(m²*s)]
          if (f < 0.) { // inflow
            Scalar imax = rho_ * kc * ((h - 0.) / dz - gravityOn_); // maximal inflow
            imax = std::min(imax, 0.); // must stay negative
            f = std::max(f, imax);
          } else { // outflow
            Scalar omax = rho_ * kc * krw *((h - criticalPressure_) / dz - gravityOn_); // maximal outflow (evaporation)
            // std::cout << "outflow " << f << ", " << omax << "\n";
            omax = std::max(omax, 0.); // must stay positive
            f = std::min(f, omax);
          }
          break;
        }
        case constantFluxCyl: { // lower = inner, with switch for maximum in- or outflow
          f = -bcBotValue_*rho_/(24.*60.*60.)/100. * pos[0]; // [cm /day] -> [kg/(m²*s)]  (Eqns are multiplied by cylindrical radius)
          if (f < 0.) { // inflow
            Scalar imax = rho_ * kc * ((h - (-10.)) / dz - gravityOn_)* pos[0]; // maximal inflow
            imax = std::min(imax, 0.); // must stay negative
            f = std::max(f, imax);
          } else { // outflow
            Scalar omax = rho_ * kc * krw *((h - criticalPressure_) / dz - gravityOn_)* pos[0]; // maximal outflow (evaporation)
//            std::cout << " f " << f*1.e9 << ", omax "<< omax*1.e9  << ", value " << bcBotValue_
//                << ", crit "  << criticalPressure_ << ", " << pos[0] << ", krw " << krw <<"\n";
            omax = std::max(omax, 0.); // must stay positive
            f = std::min(f, omax);
          }
          break;
        }
        case freeDrainage: {
          f = krw * kc * rho_; // * 1 [m]
          break;
        }
        default: DUNE_THROW(Dune::InvalidStateException, "Bottom boundary type Neumann: unknown error");
        }
      }
    }

    flux[conti0EqIdx] = f;
    return flux;
  }

  /*!
   * \copydoc FVProblem::source
   *
   * called by FVLocalResidual:computeSource(...)
   *
```

Listing 8: 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$^x$