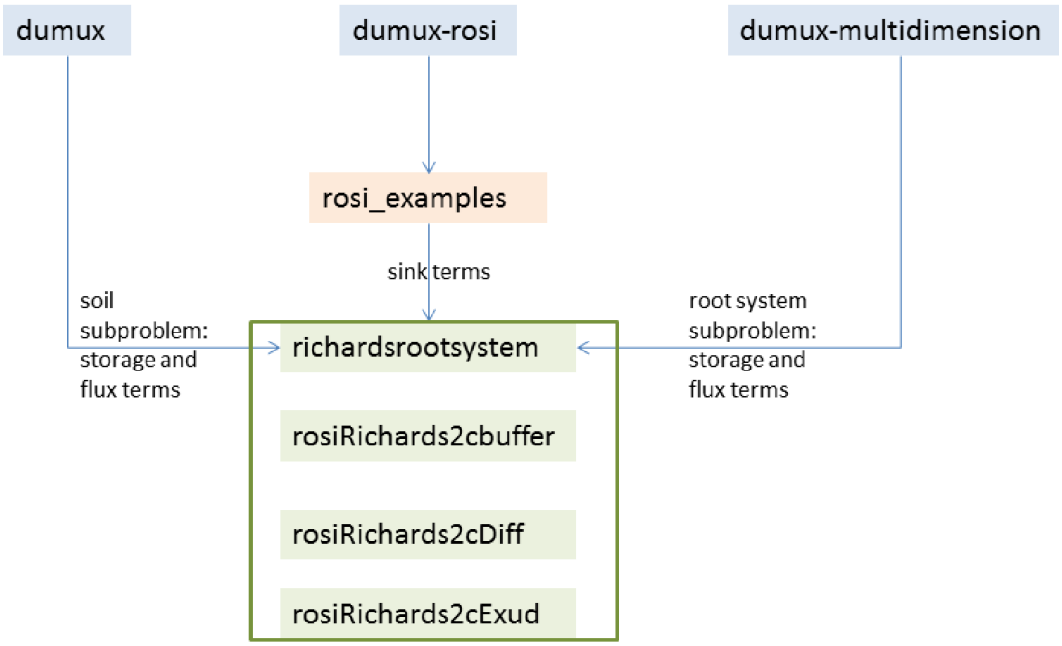
The DuMu^x module "dumux-rosi"

Documentation of examples

A. Schnepf
T. Mai
T. Morandage
C. Sheng

July 4, 2017



dumux-rosi examples and how they are linked to other dumux modules.

Installation

Required compilers and tools

- Install cmake:
`sudo apt-get install cmake`
-Install clang:
`sudo apt-get install clang`
- Install git
`sudo apt-get install git-all`

DuMu^x installation

- Create a DUMUX folder
`mkdir DUMUX`
`cd DUMUX`
- Download DUNE core modules:
`git clone https://gitlab.dune-project.org/core/dune-common.git`
`git clone https://gitlab.dune-project.org/core/dune-geometry.git`
`git clone https://gitlab.dune-project.org/core/dune-grid.git`
`git clone https://gitlab.dune-project.org/core/dune-istl.git`
`git clone https://gitlab.dune-project.org/core/dune-localfunctions.git`
- Download DUNE external modules:
`git clone https://gitlab.dune-project.org/extensions/dune-foamgrid.git`
`git clone https://gitlab.dune-project.org/extensions/dune-grid-glue.git`
-Download Dumux and Dumux-multidimension:
`git clone https://git.iws.uni-stuttgart.de/dumux-repositories/dumux.git`
`git clone https://git.iws.uni-stuttgart.de/dumux-appl/dumux-multidimension.git`
(It requires an account to download dumux-multidimension module. Please go to https://git.iws.uni-stuttgart.de/ to create an account.)

-Download configuration file `optim_cluster.opts` to working folder

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

```
./dune-common/bin/dunecontrol --opts=optim_cluster.opts all -std=c++14
```

Installation done! Good luck!

Running 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_AS_dumux_buffer.pbs`

```
#!/bin/sh
#
#This is an example script example.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_examples/
   RosiRichards2cbuffer
make test_rosiRichards2cbuffernitrate
./test_rosiRichards2cbuffernitrate
```

To start the job, run this file in your working folder with the command
`qsub queue_AS_dumux_buffer.pbs`
Use Filezilla to move the results to your local machine and use Paraview to visualize them.

Numerical grids

All examples in this document simultaneously use two numerical grids: the 3D soil grid and the 1D, branched, root system grid (see Fig. 2).

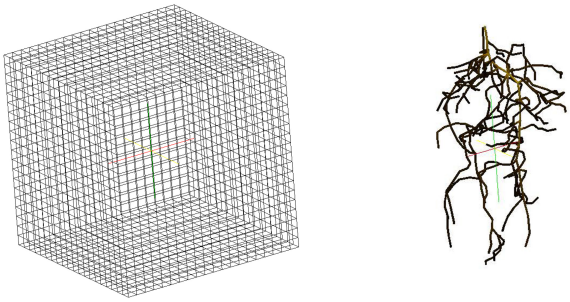


Figure 2: The 3D soil grid and the 1D, branched, grid representing the root architecture

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.

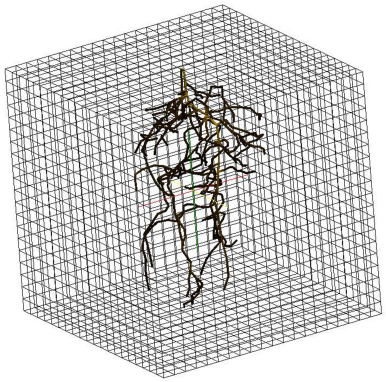


Figure 3: 3D soil grid merged with the 1D, branched, grid representing the root architecture

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 [Grid] and [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
```

The root system grid is usually specified as a file in dgf-format that specifies the coordinates and connection of nodes (verteces).

```
DGF
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 (using the class analysis.cpp).

Example 1: Water flow in the soil-root system

The model

The soil sub-problem

We solve the Richards equation in 3D 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

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

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, q_w sink term for water uptake, $\mathbf{g}$ 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) The simulation domain is a rectangular block of soil, Ω_s, and we prescribe uniform initial conditions and no-flux boundary conditions at the outer faces $\partial\Omega_s$, i.e.,

$$p_w = p_{w,0} \quad \text{at} \quad t = 0 \tag{2}$$

$$-\left[\frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \mathbf{g}\right)\right] \cdot n = 0 \quad \text{at} \quad \partial\Omega_s \tag{3}$$

[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 root system sub-problem

We solve a modified version of the Richards equation on the branched 1D domain that describes the root architecture. We assume that the root is fully saturated with water, i.e., S=1. We further follow the cohesion-tension theory wherein pressure gradients are the driving force for water movement from soil through plants to the atmosphere. Thus, there is negative absolute pressure inside the xylem[2].

$$\overbrace{\frac{\partial}{\partial t}\left(\rho_w \Phi \overbrace{S}^{=1}\right)}^{=0} - \nabla \cdot \left[\rho_w K_x \left(\nabla p_w - \rho \mathbf{g}\right)\right] = \rho_w q_{w,r}, \tag{4}$$

with K_x axial conductance and $q_{w,r}$ the sink term for water uptake by an individual root segment.
We prescribe zero initial conditions and no-flux boundary conditions at the root tips. At the root collar, we prescribe the water flux equal to the potential transpiration rate T_{pot} as long as the pressure at the root collar is above a certain threshold value. When the pressure at the root collar reaches this threshold value, the boundary condition is switched to a dirichlet condition where the pressure is prescribed to be equal to the threshold value.

$$p_w = 0 \qquad \text{at} \qquad t = 0 \tag{5}$$

$$-\left[K_x \left(\nabla p_w - \rho \mathbf{g}\right)\right] \cdot n = 0 \qquad \text{at the root tips} \tag{6}$$

$$-\left[K_x \left(\nabla p_w - \rho \mathbf{g}\right)\right] \cdot n = T_{pot} \qquad \text{at the root collar} \quad \text{if } p_w > p_{w,c} \tag{7}$$

$$p_w = p_{w,c} \qquad \text{at the root collar} \quad \text{if } p_w \leq p_{w,c}, \tag{8}$$

where $Tpot$ is the potential transpiration rate, and $p_{w,c}$ is the critical water pressure (as absolute pressure, permanent wilting point PLUS air pressure!).

Coupling the soil and root system subproblems

The soil and root system subproblems are coupled via the sink term for root water uptake. Water flow across the root membrane is driven by the pressure gradient between each root segment and its surrounding soil.
For each root segment, the radial flux of water $q_{w,r}$ is given by

$$q_{w,r} = \frac{2\pi r l K_r}{V_r}(p_{w,root} - p_{w,soil}), \tag{9}$$

[2]Water can be liquid at negative pressure (metastable) (Caupin et al. 2013). Constitutive relations e.g. between pressure and density are less known in this state (Davitt et al. 2010). However, in our simulations, we assume constant pressure

where K_r is the root hydraulic conductivity, r is the root radius, l is the length of the root segment, $p_{w,root}$ is the absolute pressure inside the root segment, and $p_{w,soil}$ is the local absolute water pressure of the soil at this root segment.

Todo: check where this division by volume is done in the code

Uptake from soil is computed by summing over the root segments that lie inside each soil control volume V_s, i.e.,

$$q_{w,V_s} = \sum_{i=1}^{N} \left(\frac{1}{V_s} (2\pi r_i f l_i K_{r,i} (p_{w,root,i} - p_{w,soil})) \right), \tag{10}$$

where N is the number of root segments that lie inside V_s, f is the fraction of root segment length that lies inside V_s.

The DuMu^x code

In this chapter, 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 soil subproblem

The storage and flux terms are defined in the file `/dumux/dumux/porousmediumflow/Richards/implicit/localresidual.hh`.
The storage term is computed as

Listing 1: storage term

```
void computeStorage(PrimaryVariables &storage, const
   int scvIdx, bool usePrevSol) const
{
    // if flag usePrevSol is set, the solution from the
        previous
    // time step is used, otherwise the current
       solution is
    // used. The secondary variables are used
       accordingly.  This
    // is required to compute the derivative of the
       storage term
    // using the implicit euler method.
    const VolumeVariables &volVars =
        usePrevSol ?
        this->prevVolVars_(scvIdx) :
        this->curVolVars_(scvIdx);
```

```
        // partial time derivative of the wetting phase
           mass
        //pressure head formulation
        storage[contiEqIdx] =
                volVars.saturation(wPhaseIdx)
                * volVars.porosity();
        // pressure formulation
        if (!useHead)
            storage[contiEqIdx] *= volVars.density(
               wPhaseIdx);
    }
```

In the same file, the flux term is computed as

Listing 2: flux term a

```
    void computeFlux(PrimaryVariables &flux, const int fIdx
       , bool onBoundary=false) const
    {
        FluxVariables fluxVars;
        fluxVars.update(this->problem_(),
                        this->element_(),
                        this->fvGeometry_(),
                        fIdx,
                        this->curVolVars_(),
                        onBoundary);

        flux = 0;
        asImp_()->computeAdvectiveFlux(flux, fluxVars);
        asImp_()->computeDiffusiveFlux(flux, fluxVars);
    }

    /*!
     * \brief Evaluates the advective mass flux of all
        components over
     *        a face of a sub-control volume.
     *
     * \param flux The advective flux over the sub-control-
        volume face for each component
     * \param fluxVars The flux variables at the current
        SCV
     */
```

```
void computeAdvectiveFlux(PrimaryVariables &flux, const
    FluxVariables &fluxVars) const
{
    // data attached to upstream and the downstream
       vertices
    // of the current phase
    const VolumeVariables &up = this->curVolVars_(
       fluxVars.upstreamIdx(wPhaseIdx));
    const VolumeVariables &dn = this->curVolVars_(
       fluxVars.downstreamIdx(wPhaseIdx));

    //pressure head formulation
    flux[contiEqIdx] =
        fluxVars.volumeFlux(wPhaseIdx);
    //pressure formulation
    if(!useHead)
        flux[contiEqIdx] *=((    massUpwindWeight_)*up.
           density(wPhaseIdx)
                            +
                            (1 - massUpwindWeight_)*dn.
                               density(wPhaseIdx));
}

/*!
 * \brief Adds the diffusive flux to the flux vector
    over
 *        the face of a sub-control volume.
 *
 * \param flux The diffusive flux over the sub-control-
    volume face for each phase
 * \param fluxVars The flux variables at the current
    SCV
 *
 * This function doesn't do anything but may be used by
     the
 * non-isothermal three-phase models to calculate
    diffusive heat
 * fluxes
 */
void computeDiffusiveFlux(PrimaryVariables &flux, const
    FluxVariables &fluxVars) const
{
```

```
        // diffusive fluxes
        flux += 0.0;
    }
```

The file `fluxvariables.hh` in the same folder defines the volume flux variable for the Richards equation by setting the nonwetting phase flux variable to zero.

Listing 3: flux term b

```
public:
    /*!
     * \brief Return the volumetric flux over a face of a
        given phase.
     *
     *        This is the calculated velocity multiplied by
         the unit normal
     *        and the area of the face.
     *        face().normal
     *        has already the magnitude of the area.
     *        For the Richards model the velocity of the
        non-wetting phase
     *        is set to zero.
     *
     * \param phaseIdx index of the phase
     */
    Scalar volumeFlux(const unsigned int phaseIdx) const
    {
      if(phaseIdx == nPhaseIdx)
          return 0.;
      else
          return ParentType::volumeFlux(phaseIdx);
    }
};
```

The definition for the wetting phase (water) is given in `dumux/dumux/porousmediumflow/implicit/darcyfluxvariables.hh`.

Listing 4: flux term c

```
/*!
 * \file
 * \brief This file contains the data which is required to
    calculate
```

```
 *        volume fluxes of fluid phases over a face of a
    finite volume by means
 *        of the Darcy approximation.
 *
 */
#ifndef DUMUX_IMPLICIT_DARCY_FLUX_VARIABLES_HH
#define DUMUX_IMPLICIT_DARCY_FLUX_VARIABLES_HH

#include <dune/common/float_cmp.hh>

#include <dumux/common/math.hh>
#include <dumux/common/parameters.hh>

#include <dumux/implicit/properties.hh>


namespace Dumux
{

namespace Properties
{
// forward declaration of properties
NEW_PROP_TAG(ImplicitMobilityUpwindWeight);
NEW_PROP_TAG(SpatialParams);
NEW_PROP_TAG(NumPhases);
NEW_PROP_TAG(ProblemEnableGravity);
}

/*!
 * \ingroup ImplicitFluxVariables
 * \brief Evaluates the normal component of the Darcy
    velocity
 * on a (sub)control volume face.
 */
template <class TypeTag>
class ImplicitDarcyFluxVariables
{
    typedef typename GET_PROP_TYPE(TypeTag, FluxVariables)
       Implementation;
    typedef typename GET_PROP_TYPE(TypeTag, Problem)
       Problem;
```

```
    typedef typename GET_PROP_TYPE(TypeTag, SpatialParams)
      SpatialParams;
    typedef typename GET_PROP_TYPE(TypeTag,
      ElementVolumeVariables) ElementVolumeVariables;
    typedef typename GET_PROP_TYPE(TypeTag, VolumeVariables
      ) VolumeVariables;

    typedef typename GET_PROP_TYPE(TypeTag, GridView)
      GridView;
    typedef typename GridView::template Codim<0>::Entity
      Element;

    enum { dim = GridView::dimension} ;
    enum { dimWorld = GridView::dimensionworld} ;
    enum { numPhases = GET_PROP_VALUE(TypeTag, NumPhases)}
      ;

    typedef typename GET_PROP_TYPE(TypeTag, Scalar) Scalar;
    typedef Dune::FieldMatrix<Scalar, dimWorld, dimWorld>
      DimWorldMatrix;
    typedef Dune::FieldVector<Scalar, dimWorld>
      GlobalPosition;

    typedef typename GET_PROP_TYPE(TypeTag,
      FVElementGeometry) FVElementGeometry;
    typedef typename FVElementGeometry::
      SubControlVolumeFace SCVFace;

public:
    /*!
     * \brief Compute / update the flux variables
     *
     * \param problem The problem
     * \param element The finite element
     * \param fvGeometry The finite-volume geometry
     * \param fIdx The local index of the SCV (sub-control-
        volume) face
     * \param elemVolVars The volume variables of the
        current element
     * \param onBoundary A boolean variable to specify
        whether the flux variables
```

```
     * are calculated for interior SCV faces or boundary
       faces, default=false
     * \todo The fvGeometry should be better initialized,
       passed and stored as an std::shared_ptr
     */
    void update(const Problem &problem,
                const Element &element,
                const FVElementGeometry &fvGeometry,
                const int fIdx,
                const ElementVolumeVariables &elemVolVars,
                const bool onBoundary = false)
    {
        fvGeometryPtr_ = &fvGeometry;
        onBoundary_ = onBoundary;
        faceIdx_ = fIdx;

        mobilityUpwindWeight_ = GET_PARAM_FROM_GROUP(
           TypeTag, Scalar, Implicit, MobilityUpwindWeight)
           ;
        asImp_().calculateGradients_(problem, element,
           elemVolVars);
        asImp_().calculateNormalVelocity_(problem, element,
            elemVolVars);
    }


    /*!
     * \brief Return the volumetric flux over a face of a
       given phase.
     *
     *        This is the calculated velocity multiplied by
        the unit normal
     *        and the area of the face. face().normal has
       already the
     *        magnitude of the area.
     *
     * \param phaseIdx index of the phase
     */
    Scalar volumeFlux(const unsigned int phaseIdx) const
    { return volumeFlux_[phaseIdx]; }

    /*!
```

```
     * \brief Return the velocity of a given phase.
     *
     *        This is the full velocity vector on the
     *        face (without being multiplied with normal).
     *
     * \param phaseIdx index of the phase
     */
    GlobalPosition velocity(const unsigned int phaseIdx)
       const
    { return velocity_[phaseIdx] ; }

    /*!
     * \brief Return intrinsic permeability multiplied with
        potential
     *        gradient multiplied with normal.
     *        I.e. everything that does not need upwind
       decisions.
     *
     * \param phaseIdx index of the phase
     */
    Scalar kGradPNormal(const unsigned int phaseIdx) const
    { return kGradPNormal_[phaseIdx] ; }

    /*!
     * \brief Return the local index of the downstream
       control volume
     *        for a given phase.
     *
     * \param phaseIdx index of the phase
     */
    unsigned int downstreamIdx(const unsigned phaseIdx)
       const
    { return downstreamIdx_[phaseIdx]; }

    /*!
     * \brief Return the local index of the upstream
       control volume
     *        for a given phase.
     *
     * \param phaseIdx index of the phase
     */
    unsigned int upstreamIdx(const unsigned phaseIdx) const
```

```
    { return upstreamIdx_[phaseIdx]; }

    /*!
     * \brief Return the SCV (sub-control-volume) face.
        This may be either
     *        a face within the element or a face on the
        element boundary,
     *        depending on the value of onBoundary_.
     */
    const SCVFace &face() const
    {
        if (onBoundary_)
            return fvGeometry_().boundaryFace[faceIdx_];
        else
            return fvGeometry_().subContVolFace[faceIdx_];
    }

protected:

    //! Returns the implementation of the flux variables (i
       .e. static polymorphism)
    Implementation &asImp_()
    { return *static_cast<Implementation *>(this); }

    //! \copydoc asImp_()
    const Implementation &asImp_() const
    { return *static_cast<const Implementation *>(this); }

    /*!
     * \brief Calculation of the potential gradients
     *
     * \param problem The problem
     * \param element The finite element
     * \param elemVolVars The volume variables of the
        current element
     */
    void calculateGradients_(const Problem &problem,
                             const Element &element,
                             const ElementVolumeVariables &
                                elemVolVars)
    {
        // loop over all phases
```

```
for (int phaseIdx = 0; phaseIdx < numPhases;
   phaseIdx++)
{
    potentialGrad_[phaseIdx]= 0.0;

    for (unsigned int idx = 0;
         idx < face().numFap;
         idx++) // loop over adjacent vertices
    {
        // FE gradient at vertex idx
        const GlobalPosition &feGrad = face().grad[
           idx];

        // index for the element volume variables
        int volVarsIdx = face().fapIndices[idx];

        // the pressure gradient
        GlobalPosition tmp(feGrad);
        tmp *= elemVolVars[volVarsIdx].fluidState()
           .pressure(phaseIdx);
        potentialGrad_[phaseIdx] += tmp;
    }

    // correct the pressure gradient by the
       gravitational acceleration
    if (GET_PARAM_FROM_GROUP(TypeTag, bool, Problem
       , EnableGravity))
    {
        // average the phase density at the
           integration point.
        Scalar SI = elemVolVars[face().i].
           fluidState().saturation(phaseIdx);
        Scalar SJ = elemVolVars[face().j].
           fluidState().saturation(phaseIdx);
        Scalar rhoI = elemVolVars[face().i].
           fluidState().density(phaseIdx);
        Scalar rhoJ = elemVolVars[face().j].
           fluidState().density(phaseIdx);
        // reduce influence if saturation is very
           small
        using std::max;
        using std::min;
```

```
                Scalar fI = max(0.0, min(SI/1e-5, 0.5));
                Scalar fJ = max(0.0, min(SJ/1e-5, 0.5));
                // check whether the phase is not present
                   in both phase
                if (Dune::FloatCmp::eq<Scalar, Dune::
                   FloatCmp::absolute>(fI + fJ, 0.0, 1.0e
                   -30))
                    fI = fJ = 0.5;

                // make gravity acceleration a force
                GlobalPosition f(problem.gravityAtPos(face
                   ().ipGlobal));
                f *= (fI*rhoI + fJ*rhoJ)/(fI + fJ); //
                   gravity times averaged density

                // calculate the final potential gradient
                potentialGrad_[phaseIdx] -= f;
            } // gravity
        } // loop over all phases
    }

    /*!
     * \brief Actual calculation of the normal Darcy
        velocities.
     *
     * \param problem The problem
     * \param element The finite element
     * \param elemVolVars The volume variables of the
        current element
     */
    void calculateNormalVelocity_(const Problem &problem,
                                  const Element &element,
                                  const
                                     ElementVolumeVariables
                                      &elemVolVars)
    {
        // calculate the mean intrinsic permeability
        const SpatialParams &spatialParams = problem.
           spatialParams();
        DimWorldMatrix K;
        if (GET_PROP_VALUE(TypeTag, ImplicitIsBox))
        {
```

```
        spatialParams.meanK(K,
                            spatialParams.
                               intrinsicPermeability(
                               element,
                                                                  f

                                                                  f

                                                                  f

                            spatialParams.
                               intrinsicPermeability(
                               element,
                                                                  f

                                                                  f

    }
    else
    {
        const Element& elementI = fvGeometry_().
           neighbors[face().i];
        FVElementGeometry fvGeometryI;
        fvGeometryI.subContVol[0].global = elementI.
           geometry().center();

        const Element& elementJ = fvGeometry_().
           neighbors[face().j];
        FVElementGeometry fvGeometryJ;
```

```
        fvGeometryJ.subContVol[0].global = elementJ.
            geometry().center();

        spatialParams.meanK(K,
                            spatialParams.
                                intrinsicPermeability(
                                elementI, fvGeometryI,
                                0),
                            spatialParams.
                                intrinsicPermeability(
                                elementJ, fvGeometryJ,
                                0));
    }

    // loop over all phases
    for (int phaseIdx = 0; phaseIdx < numPhases;
        phaseIdx++)
    {
        // calculate the flux in the normal direction
            of the
        // current sub control volume face:
        //
        // v = - (K_f grad phi) * n
        // with K_f = rho g / mu K
        //
        // Mind, that the normal has the length of it's
            area.
        // This means that we are actually calculating
        //  Q = - (K grad phi) dot n /|n| * A


        K.mv(potentialGrad_[phaseIdx], kGradP_[phaseIdx
            ]);
        kGradPNormal_[phaseIdx] = kGradP_[phaseIdx]*
            face().normal;

        // determine the upwind direction
        if (kGradPNormal_[phaseIdx] < 0)
        {
            upstreamIdx_[phaseIdx] = face().i;
            downstreamIdx_[phaseIdx] = face().j;
        }
```

```
            else
            {
                upstreamIdx_[phaseIdx] = face().j;
                downstreamIdx_[phaseIdx] = face().i;
            }

            // obtain the upwind volume variables
            const VolumeVariables& upVolVars = elemVolVars[
                upstreamIdx(phaseIdx) ];
            const VolumeVariables& downVolVars =
               elemVolVars[ downstreamIdx(phaseIdx) ];

            // the minus comes from the Darcy relation
               which states that
            // the flux is from high to low potentials.
            // set the velocity
            velocity_[phaseIdx] = kGradP_[phaseIdx];
            velocity_[phaseIdx] *= - (
               mobilityUpwindWeight_*upVolVars.mobility(
               phaseIdx)
                     + (1.0 - mobilityUpwindWeight_)*
                        downVolVars.mobility(phaseIdx)) ;

            // set the volume flux
            volumeFlux_[phaseIdx] = velocity_[phaseIdx] *
               face().normal;
        }// loop all phases
    }

    // set const reference to the fvGeometry
    void setFVGeometryPtr_(const FVElementGeometry&
       fvGeometry)
    { fvGeometryPtr_ = &fvGeometry; }

    // return const reference to the fvGeometry
    const FVElementGeometry& fvGeometry_() const
    { return *fvGeometryPtr_; }

    unsigned int faceIdx_;                //!< The index of
        the sub control volume face
    bool onBoundary_;                    //!< Specifying
       whether we are currently on the boundary of the
```

```
        simulation domain
    unsigned int     upstreamIdx_[numPhases] ,
        downstreamIdx_[numPhases]; //!< local index of the
        upstream / downstream vertex
    Scalar           volumeFlux_[numPhases] ;     //!<
        Velocity multiplied with normal (magnitude=area)
    GlobalPosition   velocity_[numPhases] ;       //!< The
        velocity as determined by Darcy's law or by the
        Forchheimer relation
    Scalar           kGradPNormal_[numPhases] ;   //!<
        Permeability multiplied with gradient in potential,
        multiplied with normal (magnitude=area)
    GlobalPosition   kGradP_[numPhases] ; //!< Permeability
        multiplied with gradient in potential
    GlobalPosition   potentialGrad_[numPhases] ; //!<
        Gradient of potential, which drives flow
    Scalar           mobilityUpwindWeight_;       //!< Upwind
         weight for mobility. Set to one for full upstream
        weighting

private:
    const FVElementGeometry* fvGeometryPtr_; //!<
        Information about the geometry of discretization

};

} // end namespace

#endif // DUMUX_IMPLICIT_DARCY_FLUX_VARIABLES_HH
```

The Van Genuchten relationships between saturation and capillary pressure and relative permeability, respectively, are given in
`dumux/material/fluidmatrixinteractions/2p/regularizedvangenuchten.hh`.

The sink term is defined in the file
`dumux-rosi/rosi_examples/richardsrootsystem/richardstestproblem.hh`.

Listing 5: sink term

```
    void solDependentPointSource(PointSource& source,
                                 const Element &element,
                                 const FVElementGeometry &
                                    fvGeometry,
```

```
                                    const int scvIdx,
                                    const
                                       ElementVolumeVariables
                                        &elemVolVars) const
{
    // compute source at every integration point
    // needs convertion of  units of 1d pressure if
       pressure head in richards is used
    const Scalar pressure3D = this->couplingManager().
       bulkPriVars(source.id())[hIdx];
    const Scalar pressure1D = this->couplingManager().
       lowDimPriVars(source.id())[hIdx] ;

    const auto& spatialParams = this->couplingManager()
       .lowDimProblem().spatialParams();
    const unsigned int rootEIdx = this->couplingManager
       ().pointSourceData(source.id()).lowDimElementIdx
       ();
    const Scalar Kr = spatialParams.Kr(rootEIdx);
    const Scalar rootRadius = spatialParams.radius(
       rootEIdx);

    // sink defined as radial flow Jr * density [m^2 s
       -1]* [kg m-3]
    const Scalar sourceValue = 2* M_PI *rootRadius * Kr
        *(pressure1D - pressure3D)
                                *elemVolVars[scvIdx].
                                   density(/*phaseIdx=*/
                                   0);
    source = sourceValue*source.quadratureWeight()*
       source.integrationElement();
}
```

In this file, also the initial and boundary conditions are implemented.

Listing 6: initial conditions

```
void initialAtPos(PrimaryVariables &values,
                  const GlobalPosition &globalPos) const
{
    values = GET_RUNTIME_PARAM(TypeTag,
                                Scalar,
                                BoundaryConditions.
                                   InitialSoilPressure);
```

```
    }
```

and

Listing 7: initial conditions

```
    void boundaryTypesAtPos(BoundaryTypes &values,
            const GlobalPosition &globalPos) const
    {
         //if(globalPos[2] > this->bBoxMax()[2]-eps_)
         //{
            values.setAllNeumann();
         //}
         //else
         //{
           // values.setAllDirichlet();
         //}
    }
```

The root system subproblem

The description of the source and flux terms can be found in `dumux/dumux/porousmediumflow/1p/implicit/localresidual.hh` The storage term:

Listing 8: storage term

```
    void computeStorage(PrimaryVariables &storage, const
       int scvIdx, const bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
           solution is
        // used. The secondary variables are used
           accordingly.  This
        // is required to compute the derivative of the
           storage term
        // using the implicit euler method.
        const ElementVolumeVariables &elemVolVars =
           usePrevSol ? this->prevVolVars_() : this->
           curVolVars_();
        const VolumeVariables &volVars = elemVolVars[scvIdx
           ];
```

```
        // partial time derivative of the wetting phase
           mass
        storage[conti0EqIdx] =  volVars.density() * volVars
           .porosity();
    }
```

The flux term is defined in the same file:

Listing 9: flux term

```
    void computeFlux(PrimaryVariables &flux, const int fIdx
       , const bool onBoundary=false) const
    {
        FluxVariables fluxVars;
        fluxVars.update(this->problem_(),
                        this->element_(),
                        this->fvGeometry_(),
                        fIdx,
                        this->curVolVars_(),
                        onBoundary);

        asImp_()->computeAdvectiveFlux(flux, fluxVars);
        asImp_()->computeDiffusiveFlux(flux, fluxVars);
    }

    /*!
     * \brief Evaluate the advective mass flux of all
        components over
     *        a face of a sub-control volume.
     *
     * \param flux The advective flux over the sub-control-
        volume face for each component
     * \param fluxVars The flux variables at the current
        SCV
     */
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        const VolumeVariables &up = this->curVolVars_(
           fluxVars.upstreamIdx(/*phaseIdx=*/0));
        const VolumeVariables &dn = this->curVolVars_(
           fluxVars.downstreamIdx(/*phaseIdx=*/0));
        flux[conti0EqIdx] =
```

```
            ((     upwindWeight_)*up.density()
             +
             (1 - upwindWeight_)*dn.density())
            *
            fluxVars.volumeFlux(/*phaseIdx=*/0);
    }

    /*!
     * \brief Adds the diffusive mass flux of all
        components over
     *        a face of a sub-control volume.
     *
     * \param flux The diffusive flux over the sub-control-
        volume face for each component
     * \param fluxVars The flux variables at the current
        SCV
     */
    void computeDiffusiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        // diffusive fluxes
        flux += 0.0;
    }
```

The corresponding flux variables are defined in
`dumux-multidimension/dumux/porousmediumflow/1d/rootsystem/fluxvariables.hh`

Inside the root system, water flow is driven by gradients of total pressure, i.e., absolute pressure of water plus gravitational potential. Gravity has only recently been included in the code, the related changes are implemented in the file fluxvariables.hh :

Listing 10: gravity

```
        // correct the pressure diffrence by the
           gravitational acceleration
        if (GET_PARAM_FROM_GROUP(TypeTag, bool, Problem,
           EnableGravity))
        {   // calculate the density
            const Element& elementI = fvGeometry_().
               neighbors[face().i];
            const Element& elementJ = fvGeometry_().
               neighbors[face().j];
            auto globalPosI = elementI.geometry().center();
            auto globalPosJ = elementJ.geometry().center();
```

```
            Scalar gravitationalTermI = density_*(problem.
               gravityAtPos(globalPosI)*globalPosI);
            Scalar gravitationalTermJ;
            if (onBoundary_)
            {
                gravitationalTermJ = density_*(problem.
                   gravityAtPos(face().ipGlobal)*face().
                   ipGlobal);
            }
            else
            {
                gravitationalTermJ = density_*(problem.
                   gravityAtPos(globalPosJ)*globalPosJ);
            }
            diffP_ -= (gravitationalTermI -
               gravitationalTermJ);
        }
```

The sink term is defined in the file
dumux-rosi/rosi_examples/richardsrootsystem/rootsystemtestproblem.hh.

Listing 11: sink term

```
    void solDependentPointSource(PointSource& source,
                                 const Element &element,
                                 const FVElementGeometry &
                                    fvGeometry,
                                 const int scvIdx,
                                 const
                                    ElementVolumeVariables
                                     &elemVolVars) const
    {
        // compute source at every integration point
        const SpatialParams &spatialParams = this->
           spatialParams();
        const Scalar Kr = spatialParams.Kr(element,
           fvGeometry, scvIdx);
        const Scalar rootRadius = spatialParams.rootRadius(
           element, fvGeometry, scvIdx);

        // convert units of 3d pressure if pressure head is
            used !!!
```

```
        const Scalar pressure3D = this->couplingManager().
           bulkPriVars(source.id())[pIdx];
        const Scalar pressure1D = this->couplingManager().
           lowDimPriVars(source.id())[pIdx];

        // sink defined as radial flow Jr [m^3 s-1]*density
        const Scalar sourceValue = 2 * M_PI *rootRadius *
           Kr *(pressure3D - pressure1D)
                                    * elemVolVars[scvIdx].
                                       density();
        source = sourceValue*source.quadratureWeight()*
           source.integrationElement();
    }
```

In this file, also the boundary conditions are implemented. Currently, there is a method Neumann in which the flux trough the tips is equal to zero and the flux at the collar is prescribed to be the transpirative flux.

replace Hcrit with Pcrit!

Listing 12: sink term

```
    void boundaryTypesAtPos (BoundaryTypes &values,
                             const GlobalPosition &
                                globalPos ) const
    {
        if (globalPos[2] + eps_ >  this->bBoxMax()[2] )
        {
            Scalar Hcrit = GET_RUNTIME_PARAM(TypeTag,
                                             Scalar,
                                             BoundaryConditions
                                                .
                                             CriticalCollarPressure
                                             );
            //get element index Eid of root segment at root
                colar
            int Eid=-1;
            for (const auto& element : elements(this->
               gridView()))
            {
                Eid ++;
                auto posZ = std::max(element.geometry().
                   corner(0)[2],element.geometry().corner
                   (1)[2]);
```

```
                if (posZ + eps_ > this->bBoxMax()[2])
                    break;
            }
            if (this->timeManager().time()>=0)
            {
                if ((preSol_[Eid] < Hcrit ))
                {
                    std::cout<<"Collar pressure: "<<preSol_
                       [Eid]<<" < " <<Hcrit<<"\n";
                    std::cout<<"WATER STRESS !! SET BC at
                       collar as Dirichlet !!"<<"\n";
                    values.setDirichlet(conti0EqIdx);
                }
                else
                {
                    std::cout<<"Collar pressure: "<<preSol_
                       [Eid]<<" > " <<Hcrit<<"\n";
                    std::cout<<"NO water stress !! SET BC
                       at collar as Neumann !!"<<"\n";
                    values.setNeumann(conti0EqIdx);
                }
            }
            else
            {
                std::cout<<"SET BC at collar as Neumann !!"
                   <<"\n";
                values.setNeumann(conti0EqIdx);
            }

        }
        else
            values.setAllNeumann();
    }
```

Todo: Check where in the code we can see the summation over all segments in one soil control volume.

The input files

replace regularized Van Genuchten with normal Van Genuchten equation!

In this subsection, we summarize the required model parameters. All parameter units

Model input parameters

Parameter	**Units**	**File number**
Water density	kg m^{-3}	5
Dynamic viscosity	kg s^{-1} m^{-1}	5
Soil porosity	m^3 m^{-3}	2
Intrinsic soil permeability	m^2	1
Residual saturation	-	2
Van Genuchten α	Pa	2
Van Genuchten n	-	2
Root porosity	m^3 m^{-3}	3
Root axial conductance	m^5 s kg^{-1}	1
Root radial conductivity	m^2 s kg^{-1}	1

are based on on absolute pressure and DuMu^x standard units (SI). For comparison with R-SWMS or presentations we convert to soil physics standards (matrix potential,) by pre- and post processing. Currently, required model parameters are distributed across several files:

1. `dumux-rosi/rosi_examples/richardsrootsystem/test_rosi.input`.
2. `dumux-rosi/rosi_examples/richardsrootsystem/richardstestspatialparams.hh`.
3. `dumux-rosi/rosi_examples/richardsrootsystem/rootsystemtestspatialparams.hh`.
4. `dumux-rosi/rosi_examples/richardsrootsystem/grid/Anagallis_femina_Leitner_2010`
5. `dumux/dumux/material/components/simpleh2o.hh`

Here is the listing of the .input-file:

Listing 13: input file

```
##############################################################

# Parameter file for test_1p.
# Everything behind a '#' is a comment.
# Type "./test_1p --help" for more information.
##############################################################


##############################################################

# Mandatory arguments
```

```
#####################################################################


[MultiDimension]
UseIterativeSolver = 0

[TimeManager]
DtInitial = 8640 # [s]
DtInitialBulkProblem = 864 # [s]
DtInitialLowDimProblem = 864 # [s]
TEnd = 604800 # [s]
EpisodeTime = 21700 # [s]

[Grid]
#File = ./grids/RootSysMRI_1times.dgf
File = ./grids/Anagallis_femina_Leitner_2010.dgf
Refinement = 0

[SoilGrid]
LowerLeft = -0.25 -0.25 -0.5
UpperRight = 0.25 0.25 0
Cells = 30 30 30
CellType = Cube

[Problem]
Name = rosi

[SpatialParams]
Permeability = 2.57e-12 # [m^2]
### root parameters ###
Kx = 5.0968e-17
Kr = 2.04e-13

[BoundaryConditions]
InitialSoilPressure =  -0.9429e4 # [Pa] -300.0 # [cm]used
   as Dirichlet BC and IC
InitialRootPressure = -1.2e6 # [Pa]
TranspirationRate = 2.15e-8 # [kg / s]
CriticalCollarPressure = -1.5e6 # [Pa]

[IterativeAlgorithm]
MaxIterations = 100
```

```
Tolerance = 1.0e-5
Verbose = 1
IntegrationOrder = 1
```

Technical issues

Solver

The default solver of all dumux-rosi examples is an iterative solver. In `rositestproblem.hh` the solver can be set in line 67:

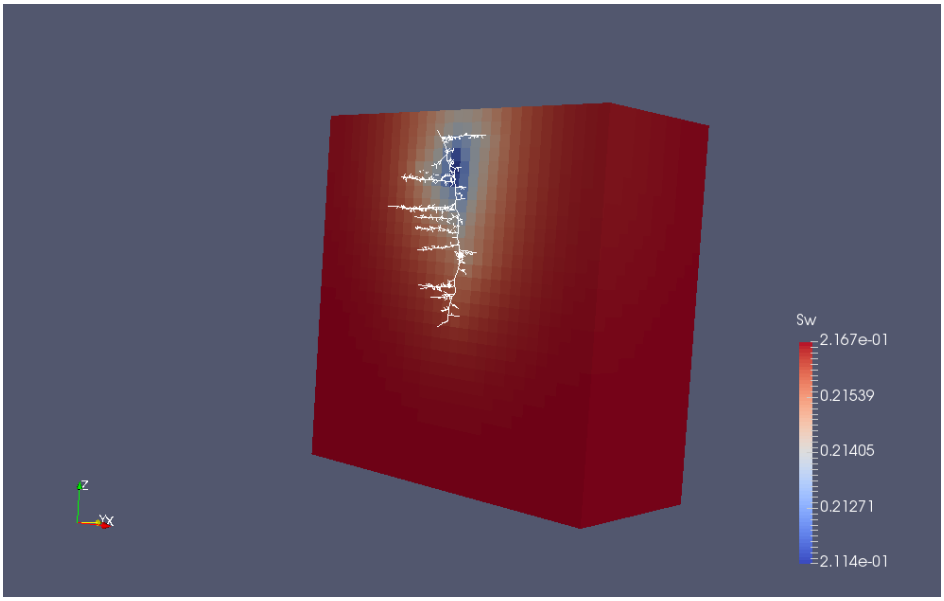
```
SET_TYPE_PROP(RosiTestProblem, LinearSolver,
   ILU0BiCGSTABBackend<TypeTag>);
```

Misc

Attention: dumux-multidimension master branch currently has a bug (23.02.2017). Switch to another branch: fix/1d-fvelementgeometry

Results

The results include vtk output for both the soil and the root system domains and can be found in the folder
`dumux-rosi/build-cmake/rosi_examples/richardsrootsystem`.
The may be visualised using Paraview.



Visualisation of soil saturation as affected by root water uptake.

output transpiration over time!

Example 2: Solute transport in the soil-root system

Model

This example explains the Nitrate (NO_3^-) uptake and transport in the soil-root system implemented in dumux-rosi module.

The soil sub-problem

Solute transport of NO3 in soil is described by two equations: the Richards equation and the transport (convection - diffussion) equations in 3D soil domain. The Richards equation is formulated in multi-phase flow context as

$$\frac{\partial}{\partial t}\left(\rho_w \Phi S\right) - \nabla \cdot \left[\rho_w \frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \boldsymbol{g}\right)\right] = q_w$$

with t - time $[s]$, θ water content, S saturation, Φ porosity, $S\phi = \theta$, ρ_w water density, K intrinsic permeability, μ dynamic viscosity, κ relative permeability, q_w sink term for water uptake, $\boldsymbol{g}$ gravitational acceleration, p_w absolute pressure of wetting phase (water)[3]. θ and h_m are related by the water retention curve: $\theta := \theta(h)$ (e.g. van Genuchten model) and $K_c = \frac{K k_r w \varrho_w g}{\mu_w}$. The simulation domain is a rectangular block of soil, Ω_s, and we prescribe uniform initial conditions and no-flux boundary conditions at the outer faces $\partial\Omega_s$, i.e.,

$$p_w = p_{w,0} \quad \text{at} \quad t = 0 \tag{11}$$

$$\frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \boldsymbol{g}\right) \cdot \mathbf{n} = 0 \quad \text{at} \quad \partial\Omega_s \tag{12}$$

Nitrate transport equation in soil is described by

$$\phi \frac{\partial \rho_w X_c S}{\partial t} - \nabla \cdot \left(D \rho_w \nabla X_c\right) - \nabla \cdot \left(\rho_w X_c \kappa_r \frac{\kappa}{\mu} (\nabla p_w - \rho_w g)\right) = q_c$$

[3] $p_w$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 $1x10^5$ 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., $h_m = \frac{p_m}{\rho_w \boldsymbol{g}}$

with X_c is mass or mole fraction of transported component (in this example is the mass fraction of BaP); D is dispersive - diffusive coefficient of component in soil solution and q_c is sink term for plant-root uptake. The inital conditions and boundary condition as contant mass fraction at the outer soil domain:

$$X_c = X_{c0} \quad \text{at} \quad t = 0 \tag{13}$$

$$X_c = X_{c0} \quad \text{at} \quad \partial\Omega_s \tag{14}$$

The root system sub-problem

We solve a modified version of the Richards equation on the 1D network in 3D space that describes the root architecture. We assume that the root is fully saturated with water, i.e., S=1. We further follow the cohesion-tension theory wherein pressure gradients are the driving force for water movement from soil through plants to the atmosphere. Thus, there is negative absolute pressure inside the xylem[4].

$$\overbrace{\frac{\partial}{\partial t}\left(\rho_w \Phi \overbrace{S}^{=1}\right)}^{=0} - \nabla \cdot \left[\rho_w K_x \left(\nabla p_w - \rho_w \boldsymbol{g}\right)\right] = \rho_w q_{w,r}, \tag{15}$$

with K_x axial conductance and $q_{w,r}$ the sink term for water uptake by an individual root segment.
We prescribe zero initial conditions and no-flux boundary conditions at the root tips. At the root collar, we prescribe the water flux equal to the potential transpiration rate T_{pot} as long as the pressure at the root collar is above a certain threshold value. When the pressure at the root collar reaches this threshold value, the boundary condition is switched to a dirichlet condition where the pressure is prescribed to be equal to the threshold value.

$$p_w = 0 \qquad \text{at} \qquad t = 0 \tag{16}$$

$$K_x \left(\nabla p_w - \rho \boldsymbol{g}\right) \cdot \mathbf{n} = 0 \qquad \text{at the root tips} \tag{17}$$

$$\rho_w K_x \left(\nabla p_w - \rho \boldsymbol{g}\right) \cdot \mathbf{n} = T_{pot} \qquad \text{at the root collar} \quad \text{if } p_w > p_{w,c} \tag{18}$$

$$p_w = p_{w,c} \qquad \text{at the root collar} \quad \text{if } p_w \leq p_{w,c}, \tag{19}$$

where T_{pot} with units kg/s is the potential transpiration rate, and $p_{w,c}$ is the critical water pressure (as absolute pressure, permanent wilting point PLUS air pressure!).
The transport of solutes in the root system is described by the convection-dispersion equation.

[4]Water can be liquid at negative pressure (metastable) (Caupin et al. 2013). Constitutive relations e.g. between pressure and density are less known in this state (Davitt et al. 2010). However, in our simulations, we assume constant pressure

$$\phi\frac{\partial \rho_w X_c S}{\partial t} - \nabla \cdot (D\rho_w \nabla X_c) - \nabla \cdot (\rho_w X_c K_x \nabla(p_w - \rho_w \mathbf{g}) - q_c = 0$$

For the boundary conditions, we describe a free outflow boundary at the root collar and zero flux at root tip

$$X_c = 0 \qquad \text{at} \qquad t = 0 \tag{20}$$

$$(-D\rho_w \nabla X_c + \rho_w X_c K_x \nabla(p_w - \rho_w \mathbf{g})) \cdot \mathbf{n} = 0 \quad \text{at the root tips} \tag{21}$$

$$-(D\rho_w \nabla X_c) \cdot \mathbf{n} = 0 \quad \text{at the root collar} \tag{22}$$

Coupling the soil and the root system subproblems

The soil and root system subproblems are coupled via the sink term for root water uptake. Water flow across the root membrane is driven by the pressure gradient between each root segment and its surrounding soil.
For each root segment, the radial flux of water $q_{w,r}$ is given by

$$q_{w,r} = \frac{2\pi r l K_r}{V_r}(p_{w,root} - p_{w,soil}), \tag{23}$$

where K_r is the root hydraulic conductivity, r is the root radius, l is the length of the root segment, $p_{w,root}$ is the absolute pressure inside the root segment, and $p_{w,soil}$ is the local absolute water pressure of the soil at this root segment.
Uptake from soil is computed by summing over the root segments that lie inside each soil control volume V, i.e.,

$$q_{w,V} = \sum_{i=1}^{N} \left(\frac{1}{V_s} 2\pi r_i f l_i K_{r,i}(p_{w,root,i} - p_{w,soil}) \right), \tag{24}$$

where N is the number of root segments that lie inside V, f is the fraction of root segment length that lies inside V.
The volumetric sink term in a root segment of length l (kg s^{-1}) is described my Michaelis Menten kinetics:

$$q_{c,r} = 2\pi r l \frac{V_{max} \rho_w X_{c,soil}}{K_m + \rho_w X_{c,soil}}$$

Uptake from soil is computed by summing over the root segments that lie inside each soil control volume V, i.e.,

$$q_{c,V} = \sum_{i=1}^{N} (2\pi r_i f D_{membrane} \rho_w (X_{cRoot} - X_{cSoil}))$$

where N is the number of root segments that lie inside V, f is the fraction of root segment length that lies inside V.

The DuMu^x code

In this chapter, 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 soil sub-problem

The storage and flux terms are defined in the file

```
dumux-rosi/dumux/porousmediumflow/richards2cbuffer/
   richards2clocalresidual.hh
```

The storage term for system of two PDE is calculated as:

Listing 14: storage term

```
    void computeStorage(PrimaryVariables &storage, const
       int scvIdx, bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
           solution is
        // used. The secondary variables are used
           accordingly.  This
        // is required to compute the derivative of the
           storage term
        // using the implicit euler method.
        const VolumeVariables &volVars =
            usePrevSol ?
            this->prevVolVars_(scvIdx) :
            this->curVolVars_(scvIdx);

        storage = 0;

        // partial time derivative of the wetting phase
           mass
        // pressure head formulation
        storage[contiEqIdx] =
                volVars.saturation(phaseIdx)
                *volVars.porosity()*volVars.density();;

        if(!useMoles) //mass-fraction formulation
        {
```

```
            //storage term of the transport equation -
               massfractions
            storage[transportEqIdx] +=
                volVars.density() * volVars.massFraction(
                   transportCompIdx) *
                (volVars.saturation(phaseIdx)*volVars.
                   porosity()+volVars.buffer());
        }
        else //mole-fraction formulation
        {
            // storage term of the transport equation -
               molefractions
            storage[transportEqIdx] +=
                volVars.molarDensity()*volVars.moleFraction
                   (transportCompIdx) *
                (volVars.saturation(phaseIdx)*volVars.
                   porosity()+volVars.buffer());
        }

    }
```

The advective flux is calculated in the same file as:

Listing 15: storage term

```
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        // data attached to upstream and the downstream
           vertices
        // of the current phase
        const VolumeVariables &up = this->curVolVars_(
           fluxVars.upstreamIdx(phaseIdx));
        const VolumeVariables &dn = this->curVolVars_(
           fluxVars.downstreamIdx(phaseIdx));

        //pressure head formulation
        flux[contiEqIdx] = fluxVars.volumeFlux(phaseIdx);
        if (!usePH)
            flux[contiEqIdx] *=
                    ((     massUpwindWeight_)*up.density()
                     +
                     ((1 - massUpwindWeight_)*dn.density())
                        );
```

```
        if(!useMoles) //mass-fraction formulation
        {
            // total mass flux - massfraction
            // advective flux of the second component -
               massfraction
            flux[transportEqIdx] +=
                fluxVars.volumeFlux(phaseIdx) *
                ((    massUpwindWeight_)*up.density() * up.
                   massFraction(transportCompIdx)
                 +
                 (1 - massUpwindWeight_)*dn.density()*dn.
                    massFraction(transportCompIdx));
        }
        else //mole-fraction formulation
        {

            // advective flux of the second component -
               molefraction
            flux[transportEqIdx] +=
                fluxVars.volumeFlux(phaseIdx) *
                ((    massUpwindWeight_)*up.molarDensity()
                   * up.moleFraction(transportCompIdx)
                 +
                 (1 - massUpwindWeight_)*dn.molarDensity()
                    * dn.moleFraction(transportCompIdx));
        }

    }
```

The diffusive flux is calculated in the same file as:

Listing 16: storage term

```
    void computeDiffusiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        // diffusive fluxes
        Scalar tmp(0);

        // diffusive flux of second component
        if(!useMoles) //mass-fraction formulation
        {
```

```
            // diffusive flux of the second component -
               massfraction
            tmp = -(fluxVars.massFractionGrad(
               transportCompIdx)*fluxVars.face().normal);
            tmp *= fluxVars.porousDiffCoeff() * fluxVars.
               density();

        //    // dispersive flux of second component -
           massfraction
        //    GlobalPosition normalDisp;
        //    fluxVars.dispersionTensor().mv(fluxVars.face
           ().normal, normalDisp);
        //    tmp -= normalDisp * fluxVars.massFractionGrad
           (transportCompIdx) * fluxVars.density();

            // convert it to a mass flux and add it
            flux[transportEqIdx] += tmp;
        }
    }
```

The dispersion tensor is calculated in

```
dumux-rosi/dumux/porousmediumflow/richards2cbuffer/
   richards2cbufferfluxvariables.hh
```

Listing 17: storage term

```
    void calculateDispersionTensor_(const Problem &problem,
                                    const Element &element,
                                    const
                                       ElementVolumeVariables
                                        &elemVolVars)
    {
        const VolumeVariables &volVarsI = elemVolVars[face
           ().i];
        const VolumeVariables &volVarsJ = elemVolVars[face
           ().j];

        //calculate dispersivity at the interface: [0]:
           alphaL = longitudinal disp. [m], [1] alphaT =
           transverse disp. [m]
        Scalar dispersivity[2];
        dispersivity[0] = 0.5 * (volVarsI.dispersivity()[0]
            +  volVarsJ.dispersivity()[0]);
```

```
        dispersivity[1] = 0.5 * (volVarsI.dispersivity()[1]
            +  volVarsJ.dispersivity()[1]);

        //calculate velocity at interface: v = -1/mu *
           vDarcy = -1/mu * K * grad(p)
        GlobalPosition velocity;
        Valgrind::CheckDefined(potentialGrad());
        Valgrind::CheckDefined(K_);
        K_.mv(potentialGrad(), velocity);
        velocity /= - 0.5 * (volVarsI.viscosity() +
           volVarsJ.viscosity());

        //matrix multiplication of the velocity at the
           interface: vv^T
        dispersionTensor_ = 0;
        for (int i=0; i<dim; i++)
            for (int j = 0; j<dim; j++)
                dispersionTensor_[i][j] = velocity[i]*
                   velocity[j];

        //normalize velocity product --> vv^T/||v||, [m/s]
        Scalar vNorm = velocity.two_norm();

        dispersionTensor_ /= vNorm;
        if (vNorm < 1e-20)
            dispersionTensor_ = 0;

        //multiply with dispersivity difference: vv^T/||v
           ||*(alphaL - alphaT), [m^2/s] --> alphaL =
           longitudinal disp., alphaT = transverse disp.
        dispersionTensor_ *= (dispersivity[0] -
           dispersivity[1]);

        //add ||v||*alphaT to the main diagonal:vv^T/||v
           ||*(alphaL - alphaT) + ||v||*alphaT, [m^2/s]
        for (int i = 0; i<dim; i++)
            dispersionTensor_[i][i] += vNorm*dispersivity
               [1];
    }
```

The root sub-problem

The description of the storage and flux terms can be found in

```
dumux-rosi/dumux/porousmediumflow/rootmodel1p2c/
   localresidual1p2c.hh
```

The storage term:

Listing 18: storage term

```
    void computeStorage(PrimaryVariables &storage, const
       int scvIdx, const bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
           solution is
        // used. The secondary variables are used
           accordingly.  This
        // is required to compute the derivative of the
           storage term
        // using the implicit euler method.
        const ElementVolumeVariables &elemVolVars =
           usePrevSol ? this->prevVolVars_() : this->
           curVolVars_();
        const VolumeVariables &volVars = elemVolVars[scvIdx
           ];


        Scalar radius = this->problem_().spatialParams().
           rootRadius(this->element_(),this->fvGeometry_(),
           scvIdx);
        storage[conti0EqIdx] += M_PI*radius*radius*volVars.
           density()*volVars.porosity();
        storage = 0;
        if(!useMoles) //mass-fraction formulation
        {
            //storage term of the transport equation -
               massfractions
            storage[transportEqIdx] += M_PI*radius*radius*
               volVars.density()*volVars.massFraction(
               transportCompIdx)*volVars.porosity();
        }
        else //mole-fraction formulation
```

```
        {
            // storage term of the transport equation -
               molefractions
            storage[transportEqIdx] += M_PI*radius*radius*
               volVars.molarDensity()*volVars.moleFraction(
               transportCompIdx)*volVars.porosity();
        }

    }
```

advective term:

Listing 19: storage term

```
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars, const int faceIdx) const
    {
        ////////////////////////////////////////////////////
        // advective fluxes of all components in all phases
        ////////////////////////////////////////////////////

        const VolumeVariables &up =
            this->curVolVars_(fluxVars.upstreamIdx(phaseIdx
               ));
        const VolumeVariables &dn =
            this->curVolVars_(fluxVars.downstreamIdx(
               phaseIdx));

        // total mass flux
        flux[conti0EqIdx] += fluxVars.volumeFlux(phaseIdx)*
            ((upwindWeight_)*up.density()
            + (1-upwindWeight_)*dn.density());

        if(!useMoles) //mass-fraction formulation
        {
            // advective flux of the second component -
               massfraction
            flux[transportEqIdx] += fluxVars.volumeFlux(
               phaseIdx)*
                ((upwindWeight_)*up.massFraction(
                   transportCompIdx)*up.density()
                + (1-upwindWeight_)*dn.massFraction(
                   transportCompIdx)*dn.density());
            Valgrind::CheckDefined(flux[transportEqIdx]);
```

```
        }
        else //mole-fraction formulation
        {

            // advective flux of the second component -
               molefraction
            flux[transportEqIdx] += fluxVars.volumeFlux(
               phaseIdx) *
                ((upwindWeight_)*up.moleFraction(
                   transportCompIdx)*up.molarDensity()
                + (1-upwindWeight_)*dn.moleFraction(
                   transportCompIdx)*dn.molarDensity());
            Valgrind::CheckDefined(flux[transportEqIdx]);
        }

        Valgrind::CheckDefined(flux[conti0EqIdx]);
    }
```

diffusive terms:

Listing 20: storage term

```
    void computeDiffusiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        Scalar tmp(0);

        // diffusive flux of second component
        if(!useMoles) //mass-fraction formulation
        {
            tmp = fluxVars.diffusiveFlux(transportCompIdx);
            // convert it to a mass flux and add it
            flux[transportEqIdx] += tmp * FluidSystem::
               molarMass(transportCompIdx);
        }
        else //mole-fraction formulation
        {
            tmp = fluxVars.diffusiveFlux(transportCompIdx);
            flux[transportEqIdx] += tmp;
        }
        Valgrind::CheckDefined(flux[transportEqIdx]);
    }
```

Fluidsystem

To model the transport process, it requires to set up a fluid system with 2 components: the main component of the fluid - water and the transport component - a solute (in this case: NO3). All the chemo - physical properties of the component nitrate is set in file

```
/dumux-rosi/dumux/material/components/anions/no3.hh
```

in which the molar mass and diffusive coefficient can be found

Listing 21: component

```
    /*!
     * \brief The mass in [kg] of one mole of NO3.
     */
    static Scalar molarMass()
    { return 62.0049e-3; } // kg/mol

    /*!
     * \brief The diffusion Coefficient of NO3 in water.
     */
    static Scalar liquidDiffCoeff()
    { return 1.7e-9; }
```

The fluidsystem of water and NO3 is set in file

```
/dumux-rosi/dumux/material/fluidsystems/h2ono3.hh
```

and must be included in both soil problem and root problem

```
/dumux-rosi/rosi_examples/RosiRichards2cbuffer/
   soilRichards2cbuffertestproblem.hh
/dumux-rosi/rosi_examples/RosiRichards2cbuffer/
   rootsystem1p2ctestproblem.hh
```

The sink terms are computed in the above mentioned files.

Listing 22: sink term in soil subproblem

```
    void solDependentPointSource( PointSource& source,//
       PrimaryVariables &source, ///
                                  const Element &element,
                                  const FVElementGeometry &
                                     fvGeometry,
                                  const int scvIdx,
                                  const
                                     ElementVolumeVariables
                                      &elemVolVars) const
```

```
{
    const Scalar pressure3D = this->couplingManager().
       bulkPriVars(source.id())[conti0EqIdx];
    const Scalar pressure1D = this->couplingManager().
       lowDimPriVars(source.id())[conti0EqIdx] ;

    const auto& spatialParams = this->couplingManager()
       .lowDimProblem().spatialParams();
    const unsigned int rootEIdx = this->couplingManager
       ().pointSourceData(source.id()).lowDimElementIdx
       ();
    const Scalar Kr = spatialParams.Kr(rootEIdx);
    const Scalar rootRadius = spatialParams.radius(
       rootEIdx);

    PrimaryVariables sourceValues;
    sourceValues=0.0;
    // sink defined as radial flow Jr * density [m] [m
       s-1 Pa-1] * [Pa]* [kg m-3] = [kg s-1 m-1]
    sourceValues[conti0EqIdx] = 2* M_PI *rootRadius *
       Kr *(pressure1D - pressure3D)
                               *elemVolVars[scvIdx].
                                  density();
    // sourceValues positive mean flow from root to
       soil
    // needs mass/mole fraction in soil and root
    Scalar c1D;
    if(useMoles)
        c1D = this->couplingManager().lowDimPriVars(
           source.id())[massOrMoleFracIdx];
    else
        c1D = this->couplingManager().lowDimPriVars(
           source.id())[massOrMoleFracIdx];
    Scalar c3D;
    if(useMoles)
        c3D = this->couplingManager().bulkPriVars(
           source.id())[massOrMoleFracIdx];
    else
        c3D = this->couplingManager().bulkPriVars(
           source.id())[massOrMoleFracIdx];

    //Diffussive flux term of transport
```

```
        const Scalar DiffValue = 0.0;
        //2* M_PI *rootRadius *DiffCoef_*(c1D - c3D)*
           elemVolVars[scvIdx].density(/*phaseIdx=*/0);

        //Advective flux term of transport
        Scalar AdvValue;
        if (sourceValues[conti0EqIdx]>0) // flow from root
           to soil
            AdvValue = 2* M_PI *rootRadius * Kr *(
               pressure1D - pressure3D)
                                    *elemVolVars[scvIdx].
                                       density()*c1D;
        else // flow from soil to root
            AdvValue = 2* M_PI *rootRadius * Kr *(
               pressure1D - pressure3D)
                                    *elemVolVars[scvIdx].
                                       density()*c3D;

        //Active flux - active uptake based on Michaeles
           Menten
        Scalar ActiveValue;
        ActiveValue = 0;
        const Scalar Vmax = spatialParams.Vmax();
        const Scalar Km = spatialParams.Km();
        ActiveValue = -2 * M_PI*rootRadius*Vmax*c3D*
           elemVolVars[scvIdx].density()/(Km+c3D*
           elemVolVars[scvIdx].density());

        Scalar sigma;
        sigma = spatialParams.PartitionCoeff();

        sourceValues[transportEqIdx] = (sigma*(AdvValue +
           DiffValue) + (1-sigma)*ActiveValue)*source.
           quadratureWeight()*source.integrationElement();
        sourceValues[conti0EqIdx] *= source.
           quadratureWeight()*source.integrationElement();
        source =  sourceValues;
    }
```

Listing 23: sink term in root subproblem

```
    void solDependentPointSource(PointSource& source,
                                 const Element &element,
```

```
                                    const FVElementGeometry &
                                       fvGeometry,
                                    const int scvIdx,
                                    const
                                       ElementVolumeVariables
                                        &elemVolVars) const
{
    // compute source at every integration point
    const SpatialParams &spatialParams = this->
       spatialParams();
    const Scalar Kr = spatialParams.Kr(element,
       fvGeometry, scvIdx);
    const Scalar rootRadius = spatialParams.rootRadius(
       element, fvGeometry, scvIdx);

    // convert units of 3d pressure if pressure head is
        used !!!
    const Scalar pressure3D = this->couplingManager().
       bulkPriVars(source.id())[conti0EqIdx];
    const Scalar pressure1D = this->couplingManager().
       lowDimPriVars(source.id())[conti0EqIdx];

    PrimaryVariables sourceValues;
    sourceValues=0.0;
    // sink defined as radial flow Jr [m^3 s-1]*density
    sourceValues[conti0EqIdx] = 2 * M_PI *rootRadius *
       Kr *(pressure3D - pressure1D)
                                * elemVolVars[scvIdx].
                                   density();

    // needs concentrations in soil and root
    Scalar c1D;
    if(useMoles)
        c1D = this->couplingManager().lowDimPriVars(
           source.id())[massOrMoleFracIdx];
    else
        c1D = this->couplingManager().lowDimPriVars(
           source.id())[massOrMoleFracIdx];
    //std::cout << "concentrations c1D " <<c1D<< std::
       endl;
    Scalar c3D;
    if(useMoles)
```

```
        c3D = this->couplingManager().bulkPriVars(
            source.id())[massOrMoleFracIdx];
    else
        c3D = this->couplingManager().bulkPriVars(
            source.id())[massOrMoleFracIdx];

    //Difussive flux term of transport
    const Scalar DiffValue = 0.0;
    //2* M_PI *rootRadius *DiffCoef_*(c1D - c3D)*
       elemVolVars[scvIdx].density(/*phaseIdx=*/0);

    //Advective flux term of transport
    Scalar AdvValue;
    if (sourceValues[conti0EqIdx]>0)
        AdvValue = 2 * M_PI *rootRadius * Kr *(
            pressure3D - pressure1D)
                                * elemVolVars[scvIdx].
                                    density()*c3D;
    else
        AdvValue = 2 * M_PI *rootRadius * Kr *(
            pressure3D - pressure1D)
                                * elemVolVars[scvIdx].
                                    density()*c1D;


    //Active flux - active uptake based on Michaeles
       Menten
    Scalar ActiveValue;
    ActiveValue = 0;
    const Scalar Vmax = spatialParams.Vmax();
    const Scalar Km = spatialParams.Km();
    ActiveValue = 2 * M_PI*rootRadius*Vmax*c3D*
       elemVolVars[scvIdx].density()/(Km+c3D*
       elemVolVars[scvIdx].density());

    Scalar sigma;
    sigma = spatialParams.PartitionCoeff();

    sourceValues[transportEqIdx] = (sigma*(AdvValue +
       DiffValue) + (1-sigma)*ActiveValue)*source.
       quadratureWeight()*source.integrationElement();
```

```
        sourceValues[conti0EqIdx] *= source.
           quadratureWeight()*source.integrationElement();

        //std::cout << "ROOT transportEqIdx " <<
           transportEqIdx <<" "<< sourceValues[
           transportEqIdx]<< std::endl;
        //std::cout << "ROOT conti0EqIdx " << conti0EqIdx
           <<" "<< sourceValues[conti0EqIdx]<< std::endl;

        //sourceValues[transportEqIdx] =1e-9*source.
           quadratureWeight()*source.integrationElement();
        source = sourceValues;
    }
```

The input files

In this subsection, we summarize the required model parameters. All parameter units are based on on absolute pressure and DuMu^x standard units (SI).

Table 2: Model input parameters

Parameter	**Units**	**File number**
Water density	kg m^{-3}	1
Dynamic viscosity	kg s^{-1} m^{-1}	1
Molar mass of BaP	kg mol^{-1}	2
Diffusive coefficient of BaP	m^{2}s^{-1}	2
Soil porosity	m^3 m^{-3}	4
Intrinsic soil permeability	m^2	4
Residual saturation	-	4
Van Genuchten α	Pa	4
Van Genuchten n	-	4
Dispersion coefficient	m^{2}s^{-1}	4
Root porosity	m^3 m^{-3}	3
Root axial conductance	m^5 s kg^{-1}	4
Root radial conductivity	m^2 s kg^{-1}	4

1. `dumux/dumux/material/components/simpleh2o.hh`
2. `dumux-rosi/dumux/material/components/anions/no3.hh`
3. `dumux-rosi/rosi_examples/RosiRichards2cbuffer/rootsystemtestspatialparams.hh`
4. `dumux-rosi/rosi_examples/RosiRichards2cbuffer/test_rosiRichards2cbuffernitrate`

Here is the listing of the .input-file:

Listing 24: component

```
[SpatialParams]
Permeability = 1.e-12 # [m^2] https://en.wikipedia.org/wiki
   /Permeability_%28earth_sciences%29 sd 3e-13
Porosity = 0.4 # [-] sd 0.1
BufferPower = 0 #[-]
### root parameters ###
Kx = 5.0968e-10 # [m/s] 5.0968e-17
Kr = 2.04e-13 # [m/(sPa)]
Vmax = 6.2e-11 #[kg/(m2s)] e-5 mol/(cm2s) -> e-5*e
   -4**62.0049e-3 =6.2e-11
Km = 3.1e-9 #[kg/m3] 0.05 mol/cm3 ->  0.05e-6*62.0049e-3 =
   3.1e-9
PartitionCoeff = 1
```

Simulation

Boundary conditions in soil domain: Neuman condition zero fluxes

```
    void boundaryTypesAtPos(BoundaryTypes &values,
            const GlobalPosition &globalPos) const
    {
                values.setAllDirichlet();
    }
```

Initial condition is homogenous field of water saturation and mass fraction of NO3

```
    void initial_(PrimaryVariables &priVars,
                  const GlobalPosition &globalPos) const
    {          Scalar sw_ = GET_RUNTIME_PARAM(TypeTag,
                            Scalar,
                            BoundaryConditions.
                               InitialSoilSaturation);
        Scalar pc_ = MaterialLaw::pc(this->spatialParams().
           materialLawParams(globalPos),sw_);

        priVars[pressureIdx] = pnRef_ - pc_;

        priVars[massOrMoleFracIdx] = GET_RUNTIME_PARAM(
           TypeTag,
                       Scalar,
```

```
                        BoundaryConditions.
                           InitialSoilFracC20H12);
};
```

The boundary conditions of the root problem are set with zero flux at root tips and free out flow at root colar for tranport equation. The switch boundary condition from Neuman to Dirichlet is implemented in case xylem pressure lower than critical value. This is implemented as the default boundary condition for all root problems in the file

```
dumux-rosi/dumux/porousmediumflow/rootmodel1p2c/problem1p2c
   .hh
```

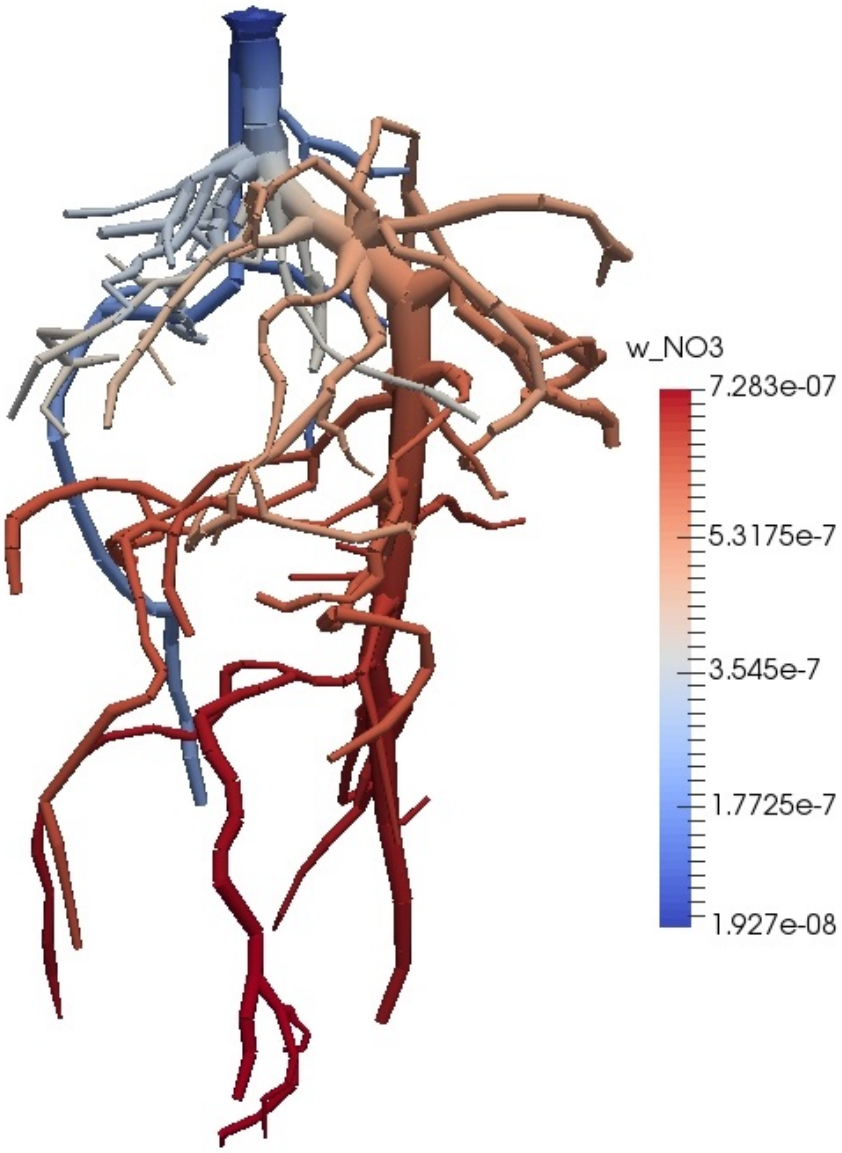
Listing 25: boundary conditions

```
    void boundaryTypesAtPos (BoundaryTypes &values,
                             const GlobalPosition &
                                globalPos ) const
    {
        if (globalPos[2] + eps_ >  this->bBoxMax()[2] )
        {
            values.setOutflow(transportEqIdx);
            Scalar criticalCollarPressure =
               GET_RUNTIME_PARAM(TypeTag,
                                       Scalar,
                                       BoundaryConditions
                                          .
                                          CriticalCollarPressure
                                          );
            //get element index Eid of root segment at root
                colar
            int Eid=-1;
            for (const auto& element : elements(this->
               gridView()))
            {
                Eid ++;
                auto posZ = std::max(element.geometry().
                   corner(0)[2],element.geometry().corner
                   (1)[2]);
                if (posZ + eps_ > this->bBoxMax()[2])
                    break;
            }
            if (this->timeManager().time()>=0)
            {
```

```
                if ((preSol_[Eid][conti0EqIdx] <
                   criticalCollarPressure ))
                {
                    std::cout<<"Collar␣pressure:␣"<<preSol_
                       [Eid][conti0EqIdx]<<"␣<␣" <<
                       criticalCollarPressure<<"\n";
                    std::cout<<"WATER␣STRESS␣!!␣SET␣BC␣at␣
                       collar␣as␣Dirichlet␣!!"<<"\n";
                    values.setDirichlet(conti0EqIdx);
                }
                else
                {
                    //std::cout<<"Collar pressure: "<<
                       preSol_[Eid][conti0EqIdx]<<" > " <<
                       criticalCollarPressure<<"\n";
                    //std::cout<<"NO water stress !! SET BC
                        at collar as Neumann !!"<<"\n";
                    values.setNeumann(conti0EqIdx);
                }
            }
            else
            {
                std::cout<<"SET␣BC␣at␣collar␣as␣Neumann␣!!"
                   <<"\n";
                values.setNeumann(conti0EqIdx);
            }

        }
        else
            values.setAllNeumann();
    }
```

Results: mass fraction of NO3



Technical issues

Solver

The default solver of all dumux-rosi examples is an iterative solver. In rosiRichards2ctest-problem the solver can be set in line 69:

```
SET_TYPE_PROP(RosiRichardsTwoCBufferTestProblem,
   LinearSolver, ILU0BiCGSTABBackend<TypeTag>);
```

At moment, the properties of component tranport is setup and hard coded. It would be better that the definition of component and all its properties (molar density, diffusion coefficient..) be moved to the input file for the ease of use.

Example 3: Contaminated root by benzo[a]pyrene (BaP) transport

Model

This example investigates the tranport of a contaminant in soil and diffusive uptake by root. The contaminant chemical is benzo[a]pyrene (BaP) which has chemical formula of $C_{20}H_{12}$

The soil sub-problem

The transport of BaP in soil is described by two equations: the Richards equation and the transport (convection - diffussion) equations in 3D soil domain. The Richards equation is formulated in multi-phase flow context as

$$\frac{\partial}{\partial t}\left(\rho_w \Phi S\right) - \nabla \cdot \left[\rho_w \frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \boldsymbol{g}\right)\right] = q_w$$

with t - time $[s]$, θ water content, S saturation, Φ porosity, $S\phi = \theta$, ρ_w water density, K intrinsic permeability, μ dynamic viscosity, κ relative permeability, q_w sink term for water uptake, $\boldsymbol{g}$ gravitational acceleration, p_w absolute pressure of wetting phase (water)[5]. θ and h_m are related by the water retention curve: $\theta := \theta(h)$ (e.g. van Genuchten model) and $K_c = \frac{K k_{rw} \varrho_w g}{\mu_w}$. The simulation domain is a rectangular block of soil, Ω_s, and we prescribe uniform initial conditions and no-flux boundary conditions at the outer faces $\partial\Omega_s$, i.e.,

$$p_w = p_{w,0} \quad \text{at} \quad t = 0 \tag{25}$$

$$\frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \boldsymbol{g}\right) \cdot \mathbf{n} = 0 \quad \text{at} \quad \partial\Omega_s \tag{26}$$

The transport equation for contaminant BaP is

$$\phi \frac{\partial \rho_w X_c S}{\partial t} - \nabla \cdot \left(D \rho_w \nabla X_c\right) - \nabla \cdot \left(\rho_w X_c \kappa_r \frac{\kappa}{\mu} (\nabla p_w - \rho_w g)\right) = q_c$$

[5] $p_w$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 $1x10^5$ 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., $h_m = \frac{p_m}{\rho_w \boldsymbol{g}}$

with X_c is mass or mole fraction of transported component (in this example is the mass fraction of BaP); D is dispersive - diffusive coefficient of component in soil solution and q_c is sink term for plant-root uptake. The inital conditions and boundary condition as contant mass fraction at the outer soil domain:

$$X_c = X_{c0} \quad \text{at} \quad t = 0 \tag{27}$$

$$X_c = X_{c0} \quad \text{at} \quad \partial\Omega_s \tag{28}$$

The root system sub-problem

We solve a modified version of the Richards equation on the 1D network in 3D space that describes the root architecture. We assume that the root is fully saturated with water, i.e., S=1. We further follow the cohesion-tension theory wherein pressure gradients are the driving force for water movement from soil through plants to the atmosphere. Thus, there is negative absolute pressure inside the xylem[6].

$$\frac{\partial}{\partial t}\left(\rho_w \Phi \overbrace{S}^{=1}\right) - \nabla \cdot \left[\rho_w K_x \left(\nabla p_w - \rho \boldsymbol{g}\right)\right] = \rho_w q_{w,r}, \tag{29}$$

with K_x axial conductance and $q_{w,r}$ the sink term for water uptake by an individual root segment.
We prescribe zero initial conditions and no-flux boundary conditions at the root tips. At the root collar, we prescribe the water flux equal to the potential transpiration rate T_{pot} as long as the pressure at the root collar is above a certain threshold value. When the pressure at the root collar reaches this threshold value, the boundary condition is switched to a dirichlet condition where the pressure is prescribed to be equal to the threshold value.

$$p_w = 0 \qquad \text{at} \qquad t = 0 \tag{30}$$

$$K_x \left(\nabla p_w - \rho \boldsymbol{g}\right) \cdot \mathbf{n} = 0 \qquad \text{at the root tips} \tag{31}$$

$$\rho_w K_x \left(\nabla p_w - \rho \boldsymbol{g}\right) \cdot \mathbf{n} = T_{pot} \qquad \text{at the root collar} \quad \text{if } p_w > p_{w,c} \tag{32}$$

$$p_w = p_{w,c} \qquad \text{at the root collar} \quad \text{if } p_w \leq p_{w,c}, \tag{33}$$

where T_{pot} is the potential transpiration rate, and $p_{w,c}$ is the critical water pressure (as absolute pressure, permanent wilting point PLUS air pressure!).
The transport of nutrient or contaminant in the root system is described by the convective diffusive equation.

[6]Water can be liquid at negative pressure (metastable) (Caupin et al. 2013). Constitutive relations e.g. between pressure and density are less known in this state (Davitt et al. 2010). However, in our simulations, we assume constant pressure

$$\phi\frac{\partial\rho X_c S}{\partial t} - \nabla\cdot(D\rho\nabla X_c) - \nabla\cdot(\rho X_c K_x(\nabla p_w - \rho_w \boldsymbol{g})) - q_c = 0$$

For the boundary conditions, we describe a free outflow boundary at the root colar and zero flux at root tip

$$X_c = 0 \qquad \text{at} \qquad t = 0 \tag{34}$$

$$(D\rho_w\nabla X_c - \rho_w X_c K_x \nabla p_{root})\cdot\mathbf{n} = 0 \qquad \text{at the root tips} \tag{35}$$

$$(D\rho_w\nabla X_c)\cdot\mathbf{n} \qquad \text{at the root collar} \tag{36}$$

Coupling the soil and the root system subproblems

The soil and root system subproblems are coupled via the sink term for root water uptake. Water flow across the root membrane is driven by the pressure gradient between each root segment and its surrounding soil.
For each root segment, the radial flux of water $q_{w,r}$ is given by

$$q_{w,r} = 2\pi r l K_r(p_{w,root} - p_{w,soil}), \tag{37}$$

where K_r is the root hydraulic conductivity, r is the root radius, l is the length of the root segment, $p_{w,root}$ is the absolute pressure inside the root segment, and $p_{w,soil}$ is the local absolute water pressure of the soil at this root segment.
Uptake from soil is computed by summing over the root segments that lie inside each soil control volume V, i.e.,

$$q_{w,V} = \sum_{i=1}^{N}\left(2\pi r_i f l_i K_{r,i}(p_{w,root,i} - p_{w,soil})\right), \tag{38}$$

where N is the number of root segments that lie inside V, f is the fraction of root segment length that lies inside V.
The passive transport of contaminant q_c $[kg.s^{-1}]$ is based on diffusive flux from soil to root segment through the root membrane

$$q_{c,r} = \frac{2\pi r l}{V_r} D_{membrane}\rho_w(X_{cRoot} - X_{cSoil})$$

Uptake from soil is computed by summing over the root segments that lie inside each soil control volume V_s, i.e.,

$$q_{c,V_s} = \sum_{i=1}^{N}\left(\frac{1}{V_s}2\pi r_i f D_{membrane}\rho_w(X_{c,root} - X_{c,soil})\right)$$

where N is the number of root segments that lie inside V_s, f is the fraction of root segment length that lies inside V_s.

The DuMu^x code

In this chapter, 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 soil sub-problem

The storage and flux terms are defined in the file

```
dumux-rosi/dumux/porousmediumflow/richards2cbuffer/
   richards2cbufferlocalresidual.hh
```

Listing 26: storage term

```
    void computeStorage(PrimaryVariables &storage, const
       int scvIdx, bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
           solution is
        // used. The secondary variables are used
           accordingly.  This
        // is required to compute the derivative of the
           storage term
        // using the implicit euler method.
        const VolumeVariables &volVars =
            usePrevSol ?
            this->prevVolVars_(scvIdx) :
            this->curVolVars_(scvIdx);

        storage = 0;

        // partial time derivative of the wetting phase
           mass
        // pressure head formulation
        storage[contiEqIdx] =
                volVars.saturation(phaseIdx)
                *volVars.porosity()*volVars.density();;

        if(!useMoles) //mass-fraction formulation
        {
            //storage term of the transport equation -
               massfractions
```

```
            storage[transportEqIdx] +=
                volVars.density() * volVars.massFraction(
                    transportCompIdx) *
                (volVars.saturation(phaseIdx)*volVars.
                    porosity()+volVars.buffer());
        }
        else //mole-fraction formulation
        {
            // storage term of the transport equation -
                molefractions
            storage[transportEqIdx] +=
                volVars.molarDensity()*volVars.moleFraction
                    (transportCompIdx) *
                (volVars.saturation(phaseIdx)*volVars.
                    porosity()+volVars.buffer());
        }

    }
```

The advective flux is calculated in the same file as:

Listing 27: advective flux term

```
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        // data attached to upstream and the downstream
            vertices
        // of the current phase
        const VolumeVariables &up = this->curVolVars_(
            fluxVars.upstreamIdx(phaseIdx));
        const VolumeVariables &dn = this->curVolVars_(
            fluxVars.downstreamIdx(phaseIdx));

        //pressure head formulation
        flux[contiEqIdx] = fluxVars.volumeFlux(phaseIdx);
        if (!usePH)
            flux[contiEqIdx] *=
                    ((     massUpwindWeight_)*up.density()
                     +
                     ((1 - massUpwindWeight_)*dn.density())
                        );

        if(!useMoles) //mass-fraction formulation
```

```
        {
            // total mass flux - massfraction
            // advective flux of the second component -
               massfraction
            flux[transportEqIdx] +=
                fluxVars.volumeFlux(phaseIdx) *
                ((    massUpwindWeight_)*up.density() * up.
                   massFraction(transportCompIdx)
                 +
                 (1 - massUpwindWeight_)*dn.density()*dn.
                    massFraction(transportCompIdx));
        }
        else //mole-fraction formulation
        {

            // advective flux of the second component -
               molefraction
            flux[transportEqIdx] +=
                fluxVars.volumeFlux(phaseIdx) *
                ((    massUpwindWeight_)*up.molarDensity()
                   * up.moleFraction(transportCompIdx)
                 +
                 (1 - massUpwindWeight_)*dn.molarDensity()
                    * dn.moleFraction(transportCompIdx));
        }

    }
```

The diffusive flux is calculated in the same file as:

Listing 28: diffusive flux term

```
     * \brief Adds the diffusive flux to the flux vector
        over
     *        the face of a sub-control volume.
     *
     * \param flux The diffusive flux over the sub-control-
        volume face for each phase
     * \param fluxVars The flux variables at the current
        SCV
     *
     * This function doesn't do anything but may be used by
        the
```

```
    * non-isothermal three-phase models to calculate
      diffusive heat
    * fluxes
    */
    void computeDiffusiveFlux(PrimaryVariables &flux, const
      FluxVariables &fluxVars) const
    {
        // diffusive fluxes
        Scalar tmp(0);

        // diffusive flux of second component
        if(!useMoles) //mass-fraction formulation
        {
            // diffusive flux of the second component - 
              massfraction
            tmp = -(fluxVars.massFractionGrad(
              transportCompIdx)*fluxVars.face().normal);
            tmp *= fluxVars.porousDiffCoeff() * fluxVars.
              density();

        // // // dispersive flux of second component - 
          massfraction
        // // GlobalPosition normalDisp;
        // // fluxVars.dispersionTensor().mv(fluxVars.face
          ().normal, normalDisp);
        // // tmp -= normalDisp * fluxVars.massFractionGrad
          (transportCompIdx) * fluxVars.density();

            // convert it to a mass flux and add it
            flux[transportEqIdx] += tmp;
        }
    }
```

the dispersionTensor is calculated in

```
dumux-rosi/dumux/porousmediumflow/richards2c/
  richards2cbufferfluxvariables.hh
```

Listing 29: dispersion tensor

```
    void calculateDispersionTensor_(const Problem &problem,
                                    const Element &element,
                                    const
                                       ElementVolumeVariables
```

```
                                        &elemVolVars)
{
    const VolumeVariables &volVarsI = elemVolVars[face
       ().i];
    const VolumeVariables &volVarsJ = elemVolVars[face
       ().j];

    //calculate dispersivity at the interface: [0]:
       alphaL = longitudinal disp. [m], [1] alphaT =
       transverse disp. [m]
    Scalar dispersivity[2];
    dispersivity[0] = 0.5 * (volVarsI.dispersivity()[0]
        +  volVarsJ.dispersivity()[0]);
    dispersivity[1] = 0.5 * (volVarsI.dispersivity()[1]
        +  volVarsJ.dispersivity()[1]);

    //calculate velocity at interface: v = -1/mu *
       vDarcy = -1/mu * K * grad(p)
    GlobalPosition velocity;
    Valgrind::CheckDefined(potentialGrad());
    Valgrind::CheckDefined(K_);
    K_.mv(potentialGrad(), velocity);
    velocity /= - 0.5 * (volVarsI.viscosity() +
       volVarsJ.viscosity());

    //matrix multiplication of the velocity at the
       interface: vv^T
    dispersionTensor_ = 0;
    for (int i=0; i<dim; i++)
        for (int j = 0; j<dim; j++)
            dispersionTensor_[i][j] = velocity[i]*
               velocity[j];

    //normalize velocity product --> vv^T/||v||, [m/s]
    Scalar vNorm = velocity.two_norm();

    dispersionTensor_ /= vNorm;
    if (vNorm < 1e-20)
        dispersionTensor_ = 0;

    //multiply with dispersivity difference: vv^T/||v
       ||*(alphaL - alphaT), [m^2/s] --> alphaL =
```

```
            longitudinal disp., alphaT = transverse disp.
        dispersionTensor_ *= (dispersivity[0] -
            dispersivity[1]);

        //add ||v||*alphaT to the main diagonal:vv^T/||v
            ||*(alphaL - alphaT) + ||v||*alphaT, [m^2/s]
        for (int i = 0; i<dim; i++)
            dispersionTensor_[i][i] += vNorm*dispersivity
                [1];
    }
```

The root sub-problem

The description of the source and flux terms can be found in

```
dumux-rosi/dumux/porousmediumflow/rootmodel1p2c/
    localresidual1p2c.hh
```

The storage term:

Listing 30: storage term

```
    void computeStorage(PrimaryVariables &storage, const
        int scvIdx, const bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
            solution is
        // used. The secondary variables are used
            accordingly.  This
        // is required to compute the derivative of the
            storage term
        // using the implicit euler method.
        const ElementVolumeVariables &elemVolVars =
            usePrevSol ? this->prevVolVars_() : this->
            curVolVars_();
        const VolumeVariables &volVars = elemVolVars[scvIdx
            ];


        Scalar radius = this->problem_().spatialParams().
            rootRadius(this->element_(),this->fvGeometry_(),
            scvIdx);
```

```
        storage[conti0EqIdx] += M_PI*radius*radius*volVars.
            density()*volVars.porosity();
        storage = 0;
        if(!useMoles) //mass-fraction formulation
        {
            //storage term of the transport equation -
                massfractions
            storage[transportEqIdx] += M_PI*radius*radius*
                volVars.density()*volVars.massFraction(
                transportCompIdx)*volVars.porosity();
        }
        else //mole-fraction formulation
        {
            // storage term of the transport equation -
                molefractions
            storage[transportEqIdx] += M_PI*radius*radius*
                volVars.molarDensity()*volVars.moleFraction(
                transportCompIdx)*volVars.porosity();
        }

    }
```

advective term:

Listing 31: advective flux term

```
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars, const int faceIdx) const
    {
        ////////////////////////////////////////////////////
        // advective fluxes of all components in all phases
        ////////////////////////////////////////////////////

        const VolumeVariables &up =
            this->curVolVars_(fluxVars.upstreamIdx(phaseIdx
                ));
        const VolumeVariables &dn =
            this->curVolVars_(fluxVars.downstreamIdx(
                phaseIdx));

        // total mass flux
        flux[conti0EqIdx] += fluxVars.volumeFlux(phaseIdx)*
            ((upwindWeight_)*up.density()
            + (1-upwindWeight_)*dn.density());
```

```
        if(!useMoles) //mass-fraction formulation
        {
            // advective flux of the second component -
               massfraction
            flux[transportEqIdx] += fluxVars.volumeFlux(
               phaseIdx)*
                ((upwindWeight_)*up.massFraction(
                   transportCompIdx)*up.density()
                + (1-upwindWeight_)*dn.massFraction(
                   transportCompIdx)*dn.density());
            Valgrind::CheckDefined(flux[transportEqIdx]);

        }
        else //mole-fraction formulation
        {

            // advective flux of the second component -
               molefraction
            flux[transportEqIdx] += fluxVars.volumeFlux(
               phaseIdx) *
                ((upwindWeight_)*up.moleFraction(
                   transportCompIdx)*up.molarDensity()
                + (1-upwindWeight_)*dn.moleFraction(
                   transportCompIdx)*dn.molarDensity());
            Valgrind::CheckDefined(flux[transportEqIdx]);
        }

        Valgrind::CheckDefined(flux[conti0EqIdx]);
    }
```

diffusive terms:

Listing 32: diffusive flux term

```
    * \brief Adds the diffusive mass flux of all
       components over
    *        a face of a sub-control volume.
    *
    * \param flux The diffusive flux over the sub-control-
       volume face for each component
    * \param fluxVars The flux variables at the current
       SCV
    */
```

```
    void computeDiffusiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        Scalar tmp(0);

        // diffusive flux of second component
        if(!useMoles) //mass-fraction formulation
        {
            tmp = fluxVars.diffusiveFlux(transportCompIdx);
            // convert it to a mass flux and add it
            flux[transportEqIdx] += tmp * FluidSystem::
               molarMass(transportCompIdx);
        }
        else //mole-fraction formulation
        {
            tmp = fluxVars.diffusiveFlux(transportCompIdx);
            flux[transportEqIdx] += tmp;
        }
        Valgrind::CheckDefined(flux[transportEqIdx]);
    }
```

Fluidsystem

To model the transport process, we require to set up a fluid system with 2 components: the main component of the fluid - water and the transport component - a nutrient or contaminant (in this case: benzo[a]pyrene $C_{20}H_{12}$). All the chemo - physical properties of the component benzo[a]pyrene $C_{20}H_{12}$ is set in file

```
/dumux-rosi/dumux/material/components/benzo[a]pyren.hh
```

in which the molar mass and diffusive coefficient can be found

```
    static Scalar molarMass()
    { return 252.32e-3; } // kg/mol

    static Scalar liquidDiffCoeff()
    { return 4.48e-10 ; }
```

The fluidsystem of water and benzo[a]pyrene is set in file

```
/dumux-rosi/dumux/material/fluidsystems/h2oc20h12.hh
```

and must be included in both soil problem and root problem

```
/dumux-rosi/rosi_examples/RosiRichards2cDiff/
   soilRichards2ctestproblem.hh
/dumux-rosi/rosi_examples/RosiRichards2cDiff/
   rootsystem1p2ctestproblem.hh
```

The sink terms are computed in the above mentioned files.

Listing 33: sink term in soil subproblem

```
    void solDependentPointSource( PointSource& source,//
       PrimaryVariables &source, ///
                                  const Element &element,
                                  const FVElementGeometry &
                                     fvGeometry,
                                  const int scvIdx,
                                  const
                                     ElementVolumeVariables
                                      &elemVolVars) const
    {
        // compute source at every integration point
        // needs convertion of  units of 1d pressure if
           pressure head in richards is used
        const Scalar pressure3D = this->couplingManager().
           bulkPriVars(source.id())[pressureIdx];
        const Scalar pressure1D = this->couplingManager().
           lowDimPriVars(source.id())[pressureIdx] ;

        const auto& spatialParams = this->couplingManager()
           .lowDimProblem().spatialParams();
        const unsigned int rootEIdx = this->couplingManager
           ().pointSourceData(source.id()).lowDimElementIdx
           ();
        const Scalar Kr = spatialParams.Kr(rootEIdx);
        const Scalar rootRadius = spatialParams.radius(
           rootEIdx);

        PrimaryVariables sourceValues;
        sourceValues=0.0;

        // sink defined as radial flow Jr * density [m^2 s
           -1]* [kg m-3]
        sourceValues[conti0EqIdx] = 2* M_PI *rootRadius *
           Kr *(pressure1D - pressure3D)
```

```
                                        *elemVolVars[scvIdx].
                                           density();

        // take mass/mole fraction in soil and root
        Scalar c1D;
        if(useMoles)
            c1D = this->couplingManager().lowDimPriVars(
               source.id())[massOrMoleFracIdx];
        else
            c1D = this->couplingManager().lowDimPriVars(
               source.id())[massOrMoleFracIdx];
        Scalar c3D;
        if(useMoles)
            c3D = this->couplingManager().bulkPriVars(
               source.id())[massOrMoleFracIdx];
        else
            c3D = this->couplingManager().bulkPriVars(
               source.id())[massOrMoleFracIdx];

        //Diffussive flux term of transport
        Scalar DiffValue;
        Scalar DiffCoef_ = GET_RUNTIME_PARAM(TypeTag,
           Scalar, SpatialParams.
           DiffussiveCoefficientMembraneRoot);
        DiffValue = 2* M_PI *rootRadius *DiffCoef_*(c1D -
           c3D)*elemVolVars[scvIdx].density();

        //Advective flux term of transport
        Scalar AdvValue;
        AdvValue = 0;

        sourceValues[transportEqIdx] = (AdvValue +
           DiffValue)*source.quadratureWeight()*source.
           integrationElement();
        sourceValues[conti0EqIdx] *= source.
           quadratureWeight()*source.integrationElement();
        source =  sourceValues;
    }
```

Listing 34: sink term in root subproblem

```
    void solDependentPointSource(PointSource& source,
                                 const Element &element,
```

```
                                    const FVElementGeometry &
                                       fvGeometry,
                                    const int scvIdx,
                                    const
                                       ElementVolumeVariables
                                        &elemVolVars) const
{
    // compute source at every integration point
    const SpatialParams &spatialParams = this->
       spatialParams();
    const Scalar Kr = spatialParams.Kr(element,
       fvGeometry, scvIdx);
    const Scalar rootRadius = spatialParams.rootRadius(
       element, fvGeometry, scvIdx);

    const Scalar pressure3D = this->couplingManager().
       bulkPriVars(source.id())[conti0EqIdx];
    const Scalar pressure1D = this->couplingManager().
       lowDimPriVars(source.id())[conti0EqIdx];

    PrimaryVariables sourceValues;
    sourceValues=0.0;
    // sink defined as radial flow Jr [m^3 s-1]*density
    sourceValues[conti0EqIdx] = 2 * M_PI *rootRadius *
       Kr *(pressure3D - pressure1D)
                               * elemVolVars[scvIdx].
                                  density();

    // needs concentrations in soil and root
    Scalar c1D;
    if(useMoles)
        c1D = this->couplingManager().lowDimPriVars(
           source.id())[massOrMoleFracIdx];//*
           elemVolVars[0].molarDensity();
    else
        c1D = this->couplingManager().lowDimPriVars(
           source.id())[massOrMoleFracIdx];//*1000;//
           elemVolVars[0].density();

    Scalar c3D;
    if(useMoles)
```

```
            c3D = this->couplingManager().bulkPriVars(
                source.id())[massOrMoleFracIdx];//*
                elemVolVars[0].molarDensity();
        else
            c3D = this->couplingManager().bulkPriVars(
                source.id())[massOrMoleFracIdx];//*1000;//
                elemVolVars[0].density();

        //Difussive flux term of transport
        Scalar DiffValue;
        Scalar DiffCoef_ = GET_RUNTIME_PARAM(TypeTag,
            Scalar, SpatialParams.
            DiffussiveCoefficientMembraneRoot);
        DiffValue = 2* M_PI *rootRadius *DiffCoef_*(c3D -
            c1D)*elemVolVars[scvIdx].density();

        //Advective flux term of transport
        Scalar AdvValue;
        AdvValue = 0;
        sourceValues[transportEqIdx] = (AdvValue +
            DiffValue)*source.quadratureWeight()*source.
            integrationElement();
        sourceValues[conti0EqIdx] *= source.
            quadratureWeight()*source.integrationElement();
        source = sourceValues;
    }

    /*!
     * \brief Evaluate the source term for all phases
        within a given
     *        sub-control-volume.
     *
     * This is the method for the case where the source
        term is
     * potentially solution dependent and requires some
        quantities that
     * are specific to the fully-implicit method.
     *
     * \param values The source and sink values for the
        conservation equations in
     units of \f$ [ \textnormal{unit of conserved quantity}
         / (m^3 \cdot s )] \f$
```

```
     * \param element The finite element
     * \param fvGeometry The finite-volume geometry
     * \param scvIdx The local subcontrolvolume index
     * \param elemVolVars All volume variables for the
        element
     *
     * For this method, the \a values parameter stores the
        rate mass
     * generated or annihilate per volume unit. Positive
        values mean
     * that mass is created, negative ones mean that it
        vanishes.
     */
    void solDependentSource(PrimaryVariables &values,
                            const Element &element,
                            const FVElementGeometry &fvGeometry,
                            const int scvIdx,
                            const ElementVolumeVariables &
                               elemVolVars) const
    {
        values = 0.0;
    }

    bool shouldWriteRestartFile() const
    {
        return false;
    }

    // add source terms to the output
    void addOutputVtkFields()
```

The input files

In this subsection, we summarize the required model parameters. All parameter units are based on on absolute pressure and DuMu^x standard units (SI).

1. `dumux/dumux/material/components/simpleh2o.hh`
2. `dumux-rosi/dumux/material/components/benzo[a]pyren.hh`
3. `dumux-rosi/rosi_examples/RosiRichards2cDiff/rootsystemtestspatialparams.hh`
4. `dumux-rosi/rosi_examples/RosiRichards2cDiff/test_rosiRichards2cDiff.input`

Here is the listing of the .input-file:

Table 3: Model input parameters

Parameter	**Units**	**File number**
Water density	kg m^{-3}	1
Dynamic viscosity	kg s^{-1} m^{-1}	1
Molar mass of BaP	kg mol^{-1}	2
Diffusive coefficient of BaP	m^2s^{-1}	2
Soil porosity	m^3 m^{-3}	4
Intrinsic soil permeability	m^2	4
Residual saturation	-	4
Van Genuchten α	Pa	4
Van Genuchten n	-	4
Dispersion coefficient	m^2s^{-1}	4
Root porosity	m^3 m^{-3}	3
Root axial conductance	m^5 s kg^{-1}	4
Root radial conductivity	m^2 s kg^{-1}	4

Listing 35: input file

```
#####################################################################

# Mandatory arguments
#####################################################################


[MultiDimension]
UseIterativeSolver = 0

[TimeManager]
DtInitial = 259.2 # [s]
TEnd = 5184 # [s]
EpisodeTime = 259.2 # [s]

[Grid]
File = ./grids/RootSysMRI_1times.dgf
Refinement = 0

[SoilGrid]
LowerLeft = -0.05 -0.05 -0.1
UpperRight = 0.05 0.05 0
Cells = 20 20 20
CellType = Cube
```

```
[Problem]
Name = rosi

[materialParams]
VgAlpha = 0.03e-2
Vgn = 2
Swr = 0.03
Snr = 0

[SpatialParams]
Permeability = 1.e-12 # [m^2]
Porosity = 0.4 #
Dispersivity = 0 #
### root parameters ###
Kx = 5.0968e-10 #
Kr = 2.04e-13
BufferPower = 0
DiffussiveCoefficientMembraneRoot = 1e-8

[BoundaryConditions]
InitialSoilSaturation = 0.6
InitialRootPressure = 9.4e4
TranspirationRate = 1.65e-09 # [kg / s]
CriticalCollarPressure = -14e5 #
InitialSoilFracC20H12 = 1e-8 # http://healthycanadians.gc.
   ca/publications/healthy-living-vie-saine/water-benzo-a-
   pyrene-eau/alt/water-benzo-a-pyrene-eau-eng.pdf
InitialRootFracC20H12 = 0 #
SoilTemperature = 283
RootTemperature = 283

[IterativeAlgorithm]
MaxIterations = 100
Tolerance = 1.0e-4
Verbose = 1
IntegrationOrder = 1

[Newton]
MaxRelativeShift = 1e-4
```

Simulation

The soil boundary conditions are set in the file

```
dumux-rosi/rosi_examples/RosiRichards2cDiff/
   soilRichards2ctestproblem.hh
```

Listing 36: boundary conditions

```
    void boundaryTypesAtPos(BoundaryTypes &values,
            const GlobalPosition &globalPos) const
    {
        values.setAllNeumann();
    }
```

Initial condition is homogenous field of water saturation and mass fraction of BaP are set in the same file:

Listing 37: initial conditions

```
    void initial_(PrimaryVariables &priVars,
                  const GlobalPosition &globalPos) const
    {
        Scalar sw_ = GET_RUNTIME_PARAM(TypeTag,
                                       Scalar,
                                       BoundaryConditions.
                                          InitialSoilSaturation
                                          );
        Scalar pc_ = MaterialLaw::pc(this->spatialParams().
           materialLawParams(globalPos),sw_);

        priVars[pressureIdx] = pnRef_ - pc_;

        priVars[massOrMoleFracIdx] = GET_RUNTIME_PARAM(
           TypeTag,
                                    Scalar,
                                    BoundaryConditions.
                                       InitialSoilFracC20H12
                                       );
};
```

The initial condition of the root subproblem are set in the file

```
dumux-rosi/rosi_examples/RosiRichards2cDiff/
   rootsystem1p2ctestproblem.hh
```

Listing 38: initial conditions

```
    void initial(PrimaryVariables &priVars,
                 const Element &element,
                 const FVElementGeometry &fvGeometry,
                 const int scvIdx) const
    {
        priVars[pressureIdx] =  GET_RUNTIME_PARAM(TypeTag,
                                                  Scalar,
                                                  BoundaryConditions
                                                     .
                                                     InitialRootPressure
                                                     );
        priVars[massOrMoleFracIdx] = 0.0;
    }
```

The boundary conditions of the root problem are set with zero flux at root tips and free out flow at root colar for tranport equation. The switch boundary condition from Neuman to Dirichlet is implemented in case xylem pressure lower than critical value. This is implemented as the default boundary condition for all root problems in the file

```
dumux-rosi/dumux/porousmediumflow/rootmodel1p2c/problem1p2c
   .hh
```

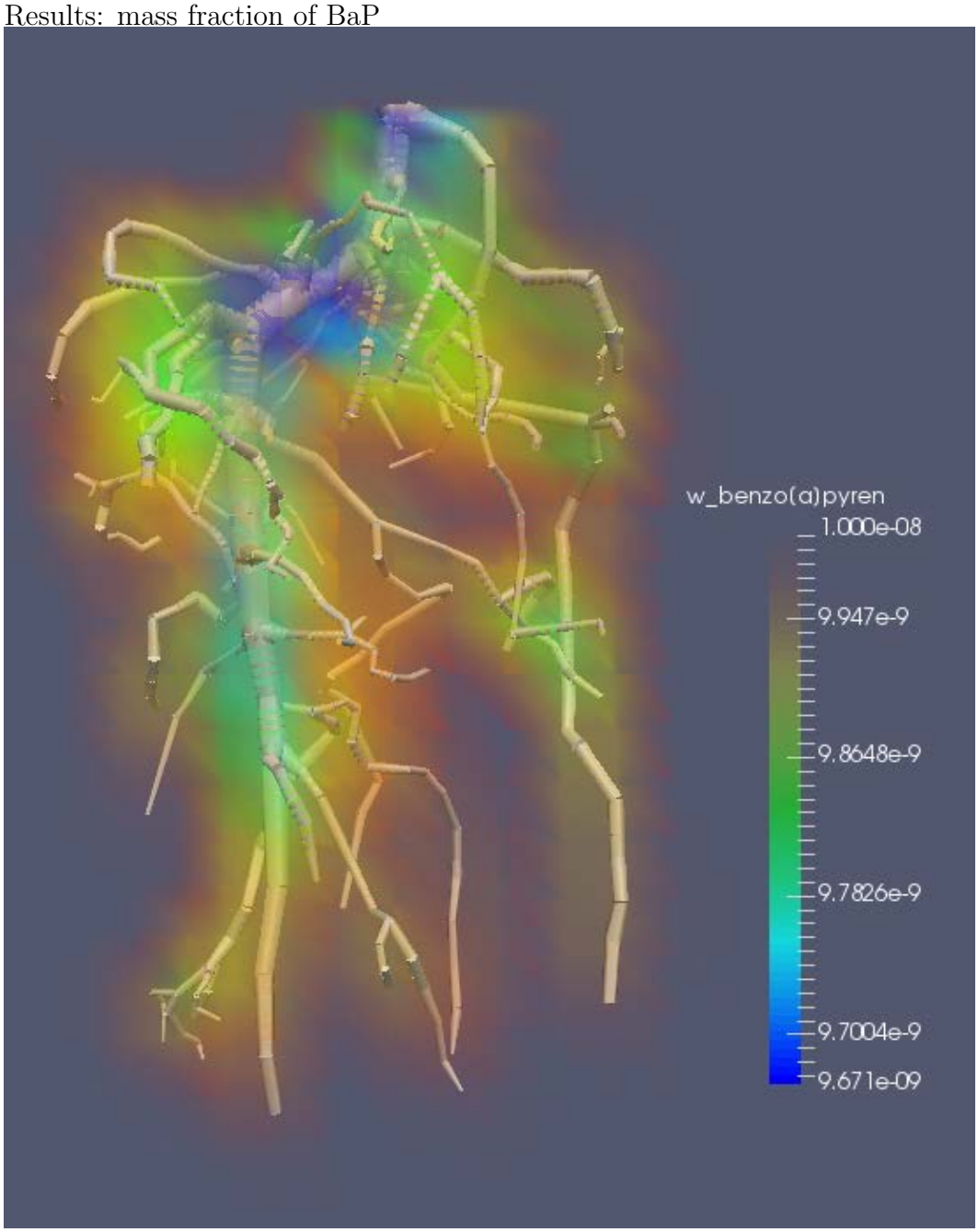
Listing 39: boundary conditions

```
    void boundaryTypesAtPos (BoundaryTypes &values,
                             const GlobalPosition &
                                globalPos ) const
    {
        if (globalPos[2] + eps_ >  this->bBoxMax()[2] )
        {
            values.setOutflow(transportEqIdx);
            Scalar criticalCollarPressure =
               GET_RUNTIME_PARAM(TypeTag,
                                 Scalar,
                                 BoundaryConditions
                                    .
                                    CriticalCollarPressure
                                    );
            //get element index Eid of root segment at root
                colar
            int Eid=-1;
            for (const auto& element : elements(this->
               gridView()))
```

```
            {
                Eid ++;
                auto posZ = std::max(element.geometry().
                    corner(0)[2],element.geometry().corner
                    (1)[2]);
                if (posZ + eps_ > this->bBoxMax()[2])
                    break;
            }
            if (this->timeManager().time()>=0)
            {
                if ((preSol_[Eid][conti0EqIdx] <
                    criticalCollarPressure ))
                {
                    std::cout<<"Collar pressure: "<<preSol_
                        [Eid][conti0EqIdx]<<" < " <<
                        criticalCollarPressure<<"\n";
                    std::cout<<"WATER STRESS !! SET BC at
                        collar as Dirichlet !!"<<"\n";
                    values.setDirichlet(conti0EqIdx);
                }
                else
                {
                    //std::cout<<"Collar pressure: "<<
                        preSol_[Eid][conti0EqIdx]<<" > " <<
                        criticalCollarPressure<<"\n";
                    //std::cout<<"NO water stress !! SET BC
                        at collar as Neumann !!"<<"\n";
                    values.setNeumann(conti0EqIdx);
                }
            }
            else
            {
                std::cout<<"SET BC at collar as Neumann !!"
                    <<"\n";
                values.setNeumann(conti0EqIdx);
            }

        }
        else
            values.setAllNeumann();
    }
```

Results: mass fraction of BaP



Technical issues

Solver

The default solver of all dumux-rosi examples is an iterative solver. In rosiRichards2ctest-problem the solver can be set in line 69:

```
SET_TYPE_PROP(RosiRichardsTwoCBufferTestProblem,
   LinearSolver, ILU0BiCGSTABBackend<TypeTag>);
```

At moment, the properties of component tranport is setup and hard coded. It would be better that the definition of component and all its properties (molar density, diffusion coefficient..) be moved to the input file for the ease of use.

Example 4: Root exudation in the soil-root system

The model

This example investigates the transportation (convection - diffusion) of root exudation in soil and root exudation decomposition. Initially, citrate and mucilage are considered in the modelling.

The soil sub-problem

The transport of root exudation in soil is described by two equations: the Richards equation and the transport (convection - diffussion) equations in 3D soil domain. The Richards equation is formulated in multi-phase flow context as

$$\frac{\partial}{\partial t}\left(\rho_w \Phi S\right) - \nabla \cdot \left[\rho_w \frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \boldsymbol{g}\right)\right] = q_w$$

with t - time $[s]$, θ water content, S saturation, Φ porosity, $S\phi = \theta$, ρ_w water density, K intrinsic permeability, μ dynamic viscosity, κ relative permeability, q_w sink term for water uptake, $\boldsymbol{g}$ gravitational acceleration, p_w absolute pressure of wetting phase (water)[7]. θ and h_m are related by the water retention curve: $\theta := \theta(h)$ (e.g. van Genuchten model) and $K_c = \frac{K k_r w \varrho_w g}{\mu_w}$. The simulation domain is a rectangular block of soil, Ω_s, and we prescribe uniform initial conditions and no-flux boundary conditions at the outer faces $\partial\Omega_s$, i.e.,

$$p_w = p_{w,0} \quad \text{at} \quad t = 0 \tag{39}$$

$$\rho_w \frac{\kappa}{\mu} K \left(\nabla p_w - \rho_w \boldsymbol{g}\right) \cdot \mathbf{n} = 0 \quad \text{at} \quad \partial\Omega_s \tag{40}$$

The transport equation for root exudation is

$$\phi \frac{\partial \rho_w X_c S}{\partial t} - \nabla \cdot \left(D \rho_w \nabla X_c\right) - \nabla \cdot \left(\rho_w X_c \kappa_r \frac{\kappa}{\mu} (\nabla p_w - \rho_w g)\right) = q_c$$

[7] $p_w$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 $1x10^5$ 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., $h_m = \frac{p_m}{\rho_w \boldsymbol{g}}$

with X_c is mass or mole fraction of transported component (in this example is the mass fraction of root exudation); D is dispersion - diffusion coefficient of component in soil solution and q_c is sink term for plant-root uptake. The inital conditions and boundary condition as contant mass fraction at the outer soil domain:

$$\left(-D\rho_w\nabla X_c - \rho_w X_c \kappa_r \frac{\kappa}{\mu}(\nabla p_w - \rho_w \mathbf{g})\right)\cdot \mathbf{n} = 0 \quad \text{at the geometry boundary,} \tag{41}$$

The root system sub-problem

We solve a modified version of the Richards equation on the 1D network in 3D space that describes the root architecture. We assume that the root is fully saturated with water, i.e., S=1. We further follow the cohesion-tension theory wherein pressure gradients are the driving force for water movement from soil through plants to the atmosphere. Thus, there is negative absolute pressure inside the xylem[8].

$$\frac{\partial}{\partial t}\left(\rho_w \Phi \overbrace{S}^{=1}\right) - \nabla\cdot\left[\rho_w K_x\left(\nabla p_w - \rho \boldsymbol{g}\right)\right] = \rho_w q_{w,r}, \tag{42}$$

with K_x axial conductance and $q_{w,r}$ the sink term for water uptake by an individual root segment.
We prescribe zero initial conditions and no-flux boundary conditions at the root tips. At the root collar, we prescribe the water flux equal to the potential transpiration rate T_{pot} as long as the pressure at the root collar is above a certain threshold value. When the pressure at the root collar reaches this threshold value, the boundary condition is switched to a dirichlet condition where the pressure is prescribed to be equal to the threshold value.

$$p_w = 0 \qquad \text{at} \qquad t = 0 \tag{43}$$

$$\rho_w K_x\left(\nabla p_w - \rho \boldsymbol{g}\right)\cdot \mathbf{n} = 0 \qquad \text{at the root tips} \tag{44}$$

$$\rho_w K_x\left(\nabla p_w - \rho \boldsymbol{g}\right)\cdot \mathbf{n} = T_{pot} \qquad \text{at the root collar} \quad \text{if } p_w > p_{w,c} \tag{45}$$

$$p_w = p_{w,c} \qquad \text{at the root collar} \quad \text{if } p_w \leq p_{w,c}, \tag{46}$$

where T_{pot} is the potential transpiration rate, and $p_{w,c}$ is the critical water pressure (as absolute pressure, permanent wilting point PLUS air pressure!).
The transport of root exudation in the root system is described by the convective diffusive equation.

$$\phi\frac{\partial \rho X_c S}{\partial t} - \nabla\cdot(D\rho\nabla X_c) - \nabla\cdot(\rho X_c K_x \nabla p_w - \rho \boldsymbol{g}) - q_c = 0$$

[8]Water can be liquid at negative pressure (metastable) (Caupin et al. 2013). Constitutive relations e.g. between pressure and density are less known in this state (Davitt et al. 2010). However, in our simulations, we assume constant pressure

For the boundary conditions, we describe zero flux at the root tips and free outflow boundary at the root collar.

$$X_c = 0 \qquad \text{at} \qquad t = 0 \tag{47}$$

$$(-D\rho_w \nabla X_c - \rho_w X_c \left(K_x \nabla_w p_w - \rho_w \boldsymbol{g}\right) \cdot \mathbf{n} = 0 \qquad \text{at the root tips} \tag{48}$$

$$(-D\rho_w \nabla X_c) \cdot \mathbf{n} = 0 \qquad \text{at the root collar} \tag{49}$$

Coupling the soil and the root system subproblems

In this example, water uptake depends on both the pressures inside the soil and inside the root system. The exudates, however, are exuded into the soil; there is no uptake of exudates by the roots. Thus, the interaction is only in one direction, i.e., the sink term of exudates in the root system is equal to zero.
For each root segment, the radial flux of water $q_{w,r}$ is given by

$$q_{w,r} = \frac{2\pi r l}{V_r} K_r (p_{w,root} - p_{w,soil}), \tag{50}$$

where V_r is the volume of the root segment, K_r is the root hydraulic conductivity, r is the root radius, l is the length of the root segment, $p_{w,root}$ is the absolute pressure inside the root segment, and $p_{w,soil}$ is the local absolute water pressure of the soil at this root segment.
Water uptake from soil is computed by summing over the root segments that lie inside each soil control volume V_s, i.e.,

$$q_{w,V_s} = \sum_{i=1}^{N} \left(\frac{1}{V_s} 2\pi r_i f l_i K_{r,i} (p_{w,root,i} - p_{w,soil}) \right), \tag{51}$$

where N is the number of root segments that lie inside V_s, f is the fraction of root segment length that lies inside V_s.
Exudation by a growing root system is defined via the age of the root segment, only the youngest parts of the root system (i.e.the parts near the root tip) can contribute to exudation. Root exudation into a each soil control volume V is given by summing over the exudation of each root segment that lies insice V_s,

$$q_{c,V} = \sum_{i=1}^{N} \left(\frac{1}{V_s} 2\pi r_i f F_{ex} e^{-\text{rootAge}\tau} \right)$$

)
where N is the number of root segments that lie inside V_s, f is the fraction of root segment length that lies inside V_s, F_{ex} is the maximal exudation rate, rootAge is the age of the root segment, τ is the rate of decrease of exudation with age.

The DuMu^x code

In this chapter, 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 soil sub-problem

The storage and flux terms are defined in the file

```
dumux-rosi/dumux/porousmediumflow/richards2cbuffer/
   richards2clocalresidual.hh
```

Listing 40: storage terms

```
    void computeStorage(PrimaryVariables &storage, const
       int scvIdx, bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
           solution is
        // used. The secondary variables are used
           accordingly.  This
        // is required to compute the derivative of the
           storage term
        // using the implicit euler method.
        const VolumeVariables &volVars =
            usePrevSol ?
            this->prevVolVars_(scvIdx) :
            this->curVolVars_(scvIdx);

        storage = 0;

        // partial time derivative of the wetting phase
           mass
        // pressure head formulation
        storage[contiEqIdx] =
                volVars.saturation(phaseIdx)
                *volVars.porosity()*volVars.density();;

        if(!useMoles) //mass-fraction formulation
        {
            //storage term of the transport equation -
               massfractions
```

```
            storage[transportEqIdx] +=
                volVars.density() * volVars.massFraction(
                    transportCompIdx) *
                (volVars.saturation(phaseIdx)*volVars.
                    porosity()+volVars.buffer());
        }
        else //mole-fraction formulation
        {
            // storage term of the transport equation -
                molefractions
            storage[transportEqIdx] +=
                volVars.molarDensity()*volVars.moleFraction
                    (transportCompIdx) *
                (volVars.saturation(phaseIdx)*volVars.
                    porosity()+volVars.buffer());
        }

    }
```

The advective flux is calculated in the same file as:

Listing 41: advective flux terms

```
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        // data attached to upstream and the downstream
            vertices
        // of the current phase
        const VolumeVariables &up = this->curVolVars_(
            fluxVars.upstreamIdx(phaseIdx));
        const VolumeVariables &dn = this->curVolVars_(
            fluxVars.downstreamIdx(phaseIdx));

        //pressure head formulation
        flux[contiEqIdx] = fluxVars.volumeFlux(phaseIdx);
        if (!usePH)
            flux[contiEqIdx] *=
                    ((     massUpwindWeight_)*up.density()
                     +
                     ((1 - massUpwindWeight_)*dn.density())
                        );

        if(!useMoles) //mass-fraction formulation
```

```
        {
            // total mass flux - massfraction
            // advective flux of the second component -
               massfraction
            flux[transportEqIdx] +=
                fluxVars.volumeFlux(phaseIdx) *
                ((    massUpwindWeight_)*up.density() * up.
                   massFraction(transportCompIdx)
                 +
                 (1 - massUpwindWeight_)*dn.density()*dn.
                    massFraction(transportCompIdx));
        }
        else //mole-fraction formulation
        {

            // advective flux of the second component -
               molefraction
            flux[transportEqIdx] +=
                fluxVars.volumeFlux(phaseIdx) *
                ((    massUpwindWeight_)*up.molarDensity()
                   * up.moleFraction(transportCompIdx)
                 +
                 (1 - massUpwindWeight_)*dn.molarDensity()
                    * dn.moleFraction(transportCompIdx));
        }

    }
```

The diffusive flux is calculated in the same file as:

Listing 42: diffusive flux terms

```
    void computeDiffusiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        // diffusive fluxes
        Scalar tmp(0);

        // diffusive flux of second component
        if(!useMoles) //mass-fraction formulation
        {
            // diffusive flux of the second component -
               massfraction
```

```
            tmp = -(fluxVars.massFractionGrad(
               transportCompIdx)*fluxVars.face().normal);
            tmp *= fluxVars.porousDiffCoeff() * fluxVars.
               density();

        //    // dispersive flux of second component -
           massfraction
        //    GlobalPosition normalDisp;
        //    fluxVars.dispersionTensor().mv(fluxVars.face
           ().normal, normalDisp);
        //    tmp -= normalDisp * fluxVars.massFractionGrad
           (transportCompIdx) * fluxVars.density();

            // convert it to a mass flux and add it
            flux[transportEqIdx] += tmp;
        }
    }
```

the dispersionTensor is calculated in

```
dumux-rosi/dumux/porousmediumflow/richards2cbuffer/
   richards2cbufferfluxvariables.hh
```

Listing 43: dispersion tensor

```
    void calculateDispersionTensor_(const Problem &problem,
                                    const Element &element,
                                    const
                                       ElementVolumeVariables
                                        &elemVolVars)
    {
        const VolumeVariables &volVarsI = elemVolVars[face
           ().i];
        const VolumeVariables &volVarsJ = elemVolVars[face
           ().j];

        //calculate dispersivity at the interface: [0]:
           alphaL = longitudinal disp. [m], [1] alphaT =
           transverse disp. [m]
        Scalar dispersivity[2];
        dispersivity[0] = 0.5 * (volVarsI.dispersivity()[0]
            +  volVarsJ.dispersivity()[0]);
        dispersivity[1] = 0.5 * (volVarsI.dispersivity()[1]
            +  volVarsJ.dispersivity()[1]);
```

```
        //calculate velocity at interface: v = -1/mu *
           vDarcy = -1/mu * K * grad(p)
        GlobalPosition velocity;
        Valgrind::CheckDefined(potentialGrad());
        Valgrind::CheckDefined(K_);
        K_.mv(potentialGrad(), velocity);
        velocity /= - 0.5 * (volVarsI.viscosity() +
           volVarsJ.viscosity());

        //matrix multiplication of the velocity at the
           interface: vv^T
        dispersionTensor_ = 0;
        for (int i=0; i<dim; i++)
            for (int j = 0; j<dim; j++)
                dispersionTensor_[i][j] = velocity[i]*
                   velocity[j];

        //normalize velocity product --> vv^T/||v||, [m/s]
        Scalar vNorm = velocity.two_norm();

        dispersionTensor_ /= vNorm;
        if (vNorm < 1e-20)
            dispersionTensor_ = 0;

        //multiply with dispersivity difference: vv^T/||v
           ||*(alphaL - alphaT), [m^2/s] --> alphaL =
           longitudinal disp., alphaT = transverse disp.
        dispersionTensor_ *= (dispersivity[0] -
           dispersivity[1]);

        //add ||v||*alphaT to the main diagonal:vv^T/||v
           ||*(alphaL - alphaT) + ||v||*alphaT, [m^2/s]
        for (int i = 0; i<dim; i++)
            dispersionTensor_[i][i] += vNorm*dispersivity
               [1];
    }
```

The root sub-problem

The description of the source and flux terms can be found in

```
dumux-rosi/dumux/porousmediumflow/rootmodel1p2c/
   localresidual1p2c.hh
```

The storage term:

Listing 44: storage term

```
    void computeStorage(PrimaryVariables &storage, const
       int scvIdx, const bool usePrevSol) const
    {
        // if flag usePrevSol is set, the solution from the
            previous
        // time step is used, otherwise the current
           solution is
        // used. The secondary variables are used
           accordingly.  This
        // is required to compute the derivative of the
           storage term
        // using the implicit euler method.
        const ElementVolumeVariables &elemVolVars =
           usePrevSol ? this->prevVolVars_() : this->
           curVolVars_();
        const VolumeVariables &volVars = elemVolVars[scvIdx
           ];


        Scalar radius = this->problem_().spatialParams().
           rootRadius(this->element_(),this->fvGeometry_(),
           scvIdx);
        storage[conti0EqIdx] += M_PI*radius*radius*volVars.
           density()*volVars.porosity();
        storage = 0;
        if(!useMoles) //mass-fraction formulation
        {
            //storage term of the transport equation -
               massfractions
            storage[transportEqIdx] += M_PI*radius*radius*
               volVars.density()*volVars.massFraction(
               transportCompIdx)*volVars.porosity();
        }
        else //mole-fraction formulation
        {
            // storage term of the transport equation -
               molefractions
```

```
            storage[transportEqIdx] += M_PI*radius*radius*
               volVars.molarDensity()*volVars.moleFraction(
               transportCompIdx)*volVars.porosity();
        }

    }
```

The advective terms are computed in the same file:

Listing 45: advective flux term

```
    void computeAdvectiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars, const int faceIdx) const
```

diffusive terms:

Listing 46: advective flux term

```
    void computeDiffusiveFlux(PrimaryVariables &flux, const
        FluxVariables &fluxVars) const
    {
        Scalar tmp(0);

        // diffusive flux of second component
        if(!useMoles) //mass-fraction formulation
        {
            tmp = fluxVars.diffusiveFlux(transportCompIdx);
            // convert it to a mass flux and add it
            flux[transportEqIdx] += tmp * FluidSystem::
               molarMass(transportCompIdx);
        }
        else //mole-fraction formulation
        {
            tmp = fluxVars.diffusiveFlux(transportCompIdx);
            flux[transportEqIdx] += tmp;
        }
        Valgrind::CheckDefined(flux[transportEqIdx]);
    }
```

Fluidsystem

To model the transport process, it requires to set up a fluid system with 2 components: the main component of the fluid - water and the transport component - a solute (in this case citrate). All the chemo - physical properties of the component citrate are set in the file

```
/dumux-rosi/dumux/material/components/anions/citrate.hh
```

in which the molar mass and diffusive coefficient can be found

```
    static Scalar molarMass()
    { return 189.101e-3; } // kg/mol
```

```
    static Scalar molarMass()
    { return 189.101e-3; } // kg/mol
```

```
    /*!
     * \brief The diffusion Coefficient of C6H5O7 in water.
     http://www.unige.ch/cabe/dynamic/ESTDynamicPartISI.pdf
     */
    static Scalar liquidDiffCoeff()
    { return 6.23e-10; }
```

Replace with diffusion coefficient of citrate

The fluidsystem of water and citrate is set in file

```
/dumux-rosi/dumux/material/fluidsystems/h2ocitrate.hh
```

and must be included in both soil problem and root problem

```
/dumux-rosi/rosi_examples/RosiRichards2cExud/
   soilRichards2ctestproblem.hh
/dumux-rosi/rosi_examples/RosiRichards2cExud/
   rootsystem1p2ctestproblem.hh
```

The sink terms are computed in the above mentioned files.

Listing 47: sink term in soil subproblem

```
    void solDependentPointSource( PointSource& source,//
       PrimaryVariables &source, ///
                                  const Element &element,
                                  const FVElementGeometry &
                                     fvGeometry,
                                  const int scvIdx,
                                  const
                                     ElementVolumeVariables
                                      &elemVolVars) const
    {
        // compute source at every integration point
        // needs convertion of  units of 1d pressure if
           pressure head in richards is used
```

```
const Scalar pressure3D = this->couplingManager().
   bulkPriVars(source.id())[conti0EqIdx];
const Scalar pressure1D = this->couplingManager().
   lowDimPriVars(source.id())[conti0EqIdx];

const auto& spatialParams = this->couplingManager()
   .lowDimProblem().spatialParams();
const unsigned int rootEIdx = this->couplingManager
   ().pointSourceData(source.id()).lowDimElementIdx
   ();
const Scalar Kr = spatialParams.Kr(rootEIdx);
const Scalar rootRadius = spatialParams.radius(
   rootEIdx);

PrimaryVariables sourceValues;
sourceValues = 0.0;

// sink defined as radial flow Jr * density [m^2 s
   -1]* [kg m-3]
sourceValues[conti0EqIdx] = 2* M_PI *rootRadius *
   Kr *(pressure1D - pressure3D)
                           *elemVolVars[scvIdx].
                              density();

sourceValues[conti0EqIdx] *= source.
   quadratureWeight()*source.integrationElement();
Scalar rootAge = this->timeManager().time()-
   spatialParams.rootcreationTime(rootEIdx);
Scalar MaxValue = 1e-6;
Scalar Exdudation_value;
if (rootAge >= 0)
    Exdudation_value = 2* M_PI *rootRadius *
       MaxValue * exp(-rootAge*5e-5); //random
       function
else
    Exdudation_value = 0;
sourceValues[transportEqIdx] = Exdudation_value*
   source.quadratureWeight()*source.
   integrationElement();
                //-c3D*5e-5*elemVolVars[scvIdx].
                   density();
source =  sourceValues;
```

```
    }
```

Listing 48: sink term in root subproblem

```
    void solDependentPointSource(PointSource& source,
                                 const Element &element,
                                 const FVElementGeometry &
                                    fvGeometry,
                                 const int scvIdx,
                                 const
                                    ElementVolumeVariables
                                     &elemVolVars) const
    {
        // compute source at every integration point
        const SpatialParams &spatialParams = this->
           spatialParams();
        const Scalar Kr = spatialParams.Kr(element,
           fvGeometry, scvIdx);
        const Scalar rootRadius = spatialParams.rootRadius(
           element, fvGeometry, scvIdx);

        // convert units of 3d pressure if pressure head is
            used !!!
        const Scalar pressure3D = this->couplingManager().
           bulkPriVars(source.id())[pressureIdx];
        //std::cout << "pressure 3D " <<pressure3D<< std::
           endl;
        const Scalar pressure1D = this->couplingManager().
           lowDimPriVars(source.id())[pressureIdx];
        //std::cout << "pressure 1D " <<pressure1D<< std::
           endl;
        const Scalar density3D = this->couplingManager().
           bulkVolVars(source.id()).density();

        PrimaryVariables sourceValues;
        sourceValues=0.0;
        // sink defined as radial flow Jr [m^3 s-1]*density
        sourceValues[conti0EqIdx] = 2 * M_PI *rootRadius *
           Kr *(pressure3D - pressure1D)
                                    *density3D;//*
                                       elemVolVars[scvIdx].
                                       density();
```

```
//         // needs concentrations in soil and root
//         Scalar c1D;
//         if(useMoles)
//             c1D = this->couplingManager().lowDimPriVars(
   source.id())[massOrMoleFracIdx];//*elemVolVars[0].
   molarDensity();
//         else
//             c1D = this->couplingManager().lowDimPriVars(
   source.id())[massOrMoleFracIdx];//*1000;//elemVolVars
   [0].density();
//         //std::cout << "mass fraction c1D " <<c1D<< std::
   endl;
//         Scalar c3D;
//         if(useMoles)
//             c3D = this->couplingManager().bulkPriVars(
   source.id())[massOrMoleFracIdx];//*elemVolVars[0].
   molarDensity();
//         else
//             c3D = this->couplingManager().bulkPriVars(
   source.id())[massOrMoleFracIdx];//*1000;//elemVolVars
   [0].density();
//         //std::cout << "      mass fraction c3D " <<c3D<<
    std::endl;
//
//         //Difussive flux term of transport
//         Scalar DiffValue;
//         Scalar DiffCoef_ = GET_RUNTIME_PARAM(TypeTag,
   Scalar, SpatialParams.DiffCoeffRootSurface);
//         DiffValue = 2* M_PI *rootRadius *DiffCoef_*(c3D -
    c1D)*elemVolVars[scvIdx].density();
//
//         //Advective flux term of transport
//         Scalar AdvValue;
//         //if (sourceValues[conti0EqIdx]>0)
//         //    AdvValue = sourceValues[conti0EqIdx]*c3D;
//         //else
//         //    AdvValue = sourceValues[conti0EqIdx]*c1D;
//
//         AdvValue = 0;
//         sourceValues[transportEqIdx] = (AdvValue +
   DiffValue)*source.quadratureWeight()*source.
   integrationElement();
```

```
        sourceValues[conti0EqIdx] *= source.
           quadratureWeight()*source.integrationElement();

        //std::cout << "ROOT transportEqIdx " <<
           transportEqIdx <<" "<< sourceValues[
           transportEqIdx]<< std::endl;
        //std::cout << "ROOT conti0EqIdx " << conti0EqIdx
           <<" "<< sourceValues[conti0EqIdx]<< std::endl;

        //sourceValues[transportEqIdx] =1e-9*source.
           quadratureWeight()*source.integrationElement();
        source = sourceValues;
    }
```

The input files

In this subsection, we summarize the required model parameters. All parameter units are based on on absolute pressure and DuMu^x standard units (SI).

1. dumux/dumux/material/components/simpleh2o.hh
2. dumux-rosi/dumux/material/components/anions/citrate.hh
3. dumux-rosi/rosi_examples/RosiRichards2cExud/rootsystemtestspatialparams.hh
4. dumux-rosi/rosi_examples/RosiRichards2cExud/test_rosiRichards2cExud.input

Here is the listing of the .input-file:

Listing 49: input file

```
#####################################################################

# Mandatory arguments
#####################################################################


[MultiDimension]
UseIterativeSolver = 0

[TimeManager]
DtInitial = 21600 # 864 # [s]
#DtInitialBulkProblem = 8640 # [s]
#DtInitialLowDimProblem = 8640 # [s]
TEnd = 2268000# 2160000 #25920 # 86400 # [s]
EpisodeTime = 21600 # 864 # [s]
```

```
[Grid]
#File = ./grids/Maize.dgf
File = ./grids/myGrid.dgf
Refinement = 0

[SoilGrid]
LowerLeft = -0.15 -0.15 -0.4
UpperRight = 0.15 0.15 0
Cells = 20 20 20
CellType = Cube

[Problem]
#Name = rosi4
Name = myGrid

[materialParams]
VgAlpha = 2.956e-4
Vgn = 1.5
Swr = 0.1

[SpatialParams]
Permeability = 1.e-12 # 1e-12 #  [m^2] https://en.wikipedia
   .org/wiki/Permeability_%28earth_sciences%29 sd 3e-13
Porosity = 0.4 #sd 0.1
BufferPower= 0
### root parameters ###
Kx = 5.0968e-10 # 5.0968e-17
Kr = 2.04e-13
DiffCoeffRootSurface = 1e-9

[BoundaryConditions]
InitialSoilSaturation = 0.6
InitialSoilPressure = 9.5e4 #-3e4 #-0.3e6 #-0.9429e4 # [Pa]
    -300.0 # [cm]used as Dirichlet BC and IC  sd -0.2e3
InitialRootPressure = 9.5e4 #-5e5 #-0.6e6 #-1.2e6 # [Pa]

TranspirationRate = 2.15e-5 # [kg / s]
CriticalCollarPressure = 7e4#-14e5 # -1.5e6 # [Pa] sd -1.5
   e5
InitialSoilFracExud = 0
InitialRootFracExud = 0
```

```
SoilTemperature = 283
RootTemperature = 283

[IterativeAlgorithm]
MaxIterations = 100
Tolerance = 1.0e-4
Verbose = 1
IntegrationOrder = 1

[Newton]
MaxRelativeShift = 1e-4
```

Simulation

Boundary conditions in soil domain are defined in the file

```
    dumux-rosi/rosi_examples/RosiRichards2cExud/
       soilRichards2ctestproblem.hh}
```

Listing 50: boundary conditions

```
     */
    void boundaryTypesAtPos(BoundaryTypes &values,
            const GlobalPosition &globalPos) const
    {
        //values.setAllNeumann();

        //values.setAllDirichlet();
        if(globalPos[2] > this->bBoxMax()[2] - eps_)
        {
            values.setAllNeumann();
        }
        else
        {
            values.setAllDirichlet();
            //values.setOutflow(transportEqIdx);
        }
```

Initial condition is homogenous field of water saturation and mass fraction of exudate

Listing 51: initial conditions

```
     */
    void initialAtPos(PrimaryVariables &values,
```

```
                const GlobalPosition &globalPos) const
    {
      initial_(values, globalPos);
    }

    bool shouldWriteRestartFile() const
    {
        return false;
```

The boundary conditions in root problem are set with zero flux at root tips and free out flow at root colar for tranport equation. The switch boundary condition from Neuman to Dirichlet is implemented in case xylem pressure lower than critical value.
The boundary conditions of the root problem are set with zero flux at root tips and free out flow at root colar for tranport equation. The switch boundary condition from Neuman to Dirichlet is implemented in case xylem pressure lower than critical value. This is implemented as the default boundary condition for all root problems in the file

```
dumux-rosi/dumux/porousmediumflow/rootmodel1p2c/problem1p2c
   .hh
```

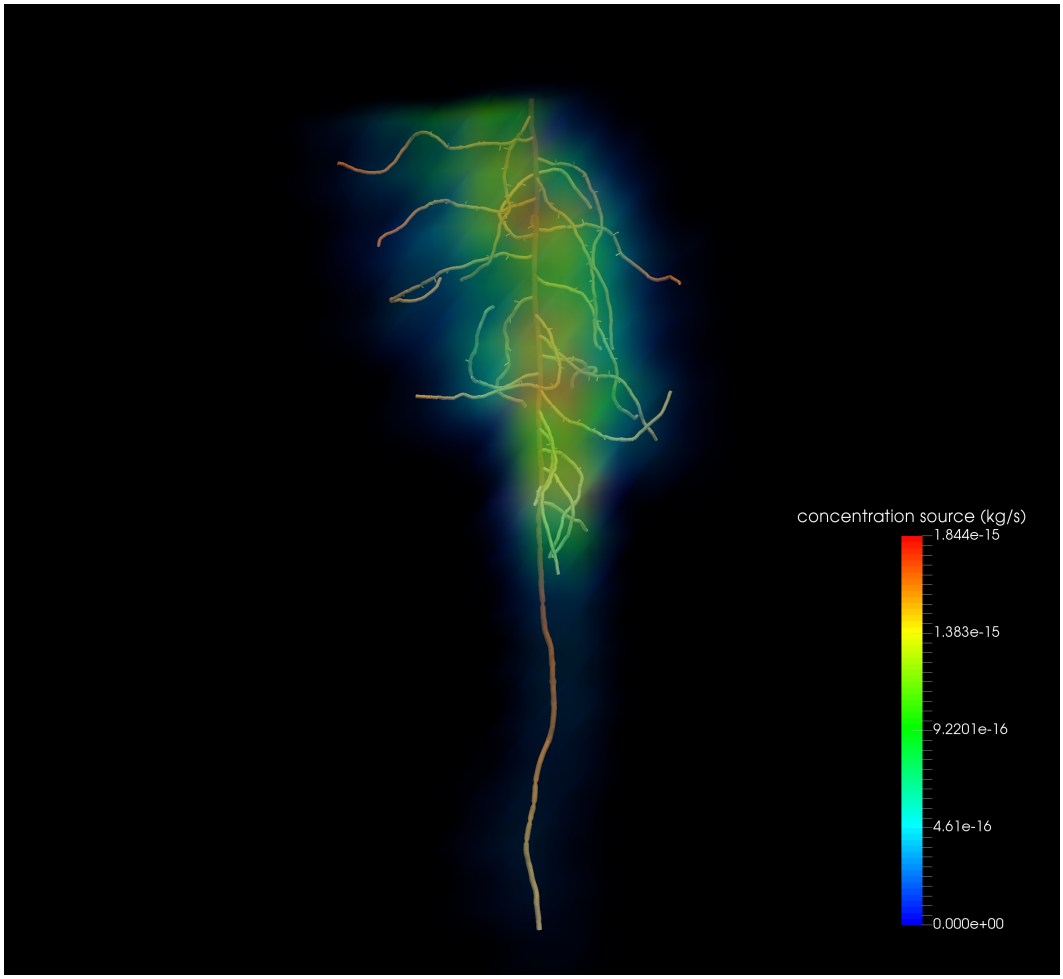
Listing 52: boundary conditions

```
    void boundaryTypesAtPos (BoundaryTypes &values,
                             const GlobalPosition &
                                globalPos ) const
    {
        if (globalPos[2] + eps_ >  this->bBoxMax()[2] )
        {
            values.setOutflow(transportEqIdx);
            Scalar criticalCollarPressure =
               GET_RUNTIME_PARAM(TypeTag,
                                 Scalar,
                                 BoundaryConditions
                                    .
                                    CriticalCollarPressure
                                    );
            //get element index Eid of root segment at root
                colar
            int Eid=-1;
            for (const auto& element : elements(this->
               gridView()))
            {
                Eid ++;
```

```
                auto posZ = std::max(element.geometry().
                    corner(0)[2],element.geometry().corner
                    (1)[2]);
                if (posZ + eps_ > this->bBoxMax()[2])
                    break;
            }
            if (this->timeManager().time()>=0)
            {
                if ((preSol_[Eid][conti0EqIdx] <
                    criticalCollarPressure ))
                {
                    std::cout<<"Collar pressure: "<<preSol_
                        [Eid][conti0EqIdx]<<" < " <<
                        criticalCollarPressure<<"\n";
                    std::cout<<"WATER STRESS !! SET BC at
                        collar as Dirichlet !!"<<"\n";
                    values.setDirichlet(conti0EqIdx);
                }
                else
                {
                    //std::cout<<"Collar pressure: "<<
                        preSol_[Eid][conti0EqIdx]<<" > " <<
                        criticalCollarPressure<<"\n";
                    //std::cout<<"NO water stress !! SET BC
                        at collar as Neumann !!"<<"\n";
                    values.setNeumann(conti0EqIdx);
                }
            }
            else
            {
                std::cout<<"SET BC at collar as Neumann !!"
                    <<"\n";
                values.setNeumann(conti0EqIdx);
            }

        }
        else
            values.setAllNeumann();
    }
```

Results: mass fraction of exudate



Technical issues

Solver

The default solver of all dumux-rosi examples is an iterative solver. In rosiRichards2ctest-problem the solver can be set in line 69:

```
SET_TYPE_PROP(RosiRichardsTwoCBufferTestProblem,
   LinearSolver, ILU0BiCGSTABBackend<TypeTag>);
```

At moment, the properties of component tranport is setup and hard coded. It would be better that the definition of component and all its properties (molar density, diffusion coefficient..) be moved to the input file for the ease of use.