PlantBox: RootSystem Tutorial

Daniel Leitner @ SimWerk
www.simwerk.at

The following tutorial offers scripts to outline the usage of the CPlantBox (Zhou et al., 2020) Python binding named *plantbox* for many different applications. CPlantBox was developed from CRootBox (Schnepf et al., 2018) and is largely backward compatible by having the same underlying rootsystem model. For further documentation please refer to the Doxygen class documentation of the CPlantBox code.

Contents

0.1 Installation

This installation guideline is for CPlantBox on Linux systems (e.g. Ubuntu).

Required compilers and tools

If on a recent Ubuntu system, the c++ compiler and python that come with the distribution are recent enough. Otherwise, please make sure you have a recent c++ compiler (e.g. sudo apt−get install clang), fortran compiler (sudo apt-get install gfortran) and python3 (e.g. sudo apt−get install python3.7).

- Install git:
sudo apt−get install git
- Install cmake:
sudo apt−get install cmake
- Install libboost:
sudo apt−get install libboost−all−dev
- Install pip:
sudo apt−get install python3−pip
- Install the python package numpy:
pip3 install numpy
- Install the python package scipy:
pip3 install scipy
- Install the python package matplotlib:
pip3 install matplotlib[1]
- Install the python package VTK:
alug pip3 install vtk
- Install the java runtime environment:
sudo apt−get install default−jre
- Install Paraview
sudo apt−get install paraview

-Download CPlantBox:

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

To build CPlantBox and its python shared library, move again into the CPlant-Box folder and type into the console:

cd CRootBox
cmake .[2]
make

(If building CPlantBox on the cluster, two lines in the file \textbf{CRootBox /CMakeLists.txt need to be outcommented before:
set(CMAKE_C_COMPILER "/usr/bin/gcc")

[1] Known bug in ubuntu 18.04: needs sudo apt-get install libfreetype6-dev libxft-dev installed before.

[2] It may be necessary on your installation to check the CPlantBox/src/CMakeLists.txt file regarding required python version and out-commenting line 34.

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

After successfully compiling CPlantBox the Python library *plantbox* should be available on your system.

0.2 Small example

The first example shows how to use CPlantBox: open a parameter file (L12), do the simulation (L18), save the results (L21), and make an interactive plot showing the results (L24).

```
1  """small example"""
2  import sys
3  sys.path.append("../../..")
4  import plantbox as pb
5  import vtk_plot as vp
6
7  rs = pb.RootSystem()
8
9  # Open plant and root parameter from a file
10 path = "../../../modelparameter/rootsystem/"
11 name = "Anagallis_femina_Leitner_2010"
12 rs.readParameters(path + name + ".xml")
13
14 # Initialize
15 rs.initialize()
16
17 # Simulate
18 rs.simulate(30, True)
19
20 # Export final result (as vtp)
21 rs.write("results/example_1a.vtp")
22
23 # Plot, using vtk
24 vp.plot_roots(rs, "creationTime")
```

Listing 1: Example 1a

Lets revise the above code in more detail:

2,3 We add the path to find the *plantbox* module.

3 Imports the CPlantBox Python library *plantbox* and name it pb.

4 Imports a auxiliary script for visualization of the rootsystem with VTK and name it vp.

7 Constructs the root system object.

12 Opens an .xml containing parameters describing the types of root (RootRandomParameters), and the type of pant (SeedRandomParameters). Alternatively, all parameter can be set or modified directly in Python (see Section 1.1).

15 Initializes the simulation: Creates the tap root and the base roots (i.e. all basal roots, and shoot borne roots that might emerge). Initializes the tropisms and passing the domain geometry, and creates the elongation functions.

18 Performs the simulation. The value 30 is the simulation time in days. If no simulation time is passed the simulation time is taken from the parameter file. Note that simulation results are independent from the time step, i.e. 30 simulate(1) calls should yield the same result as simulate(30).

21 Saves the resulting root system geometry in the VTK Polygonal Data format (VTP) as polylines, see Figure 1a.

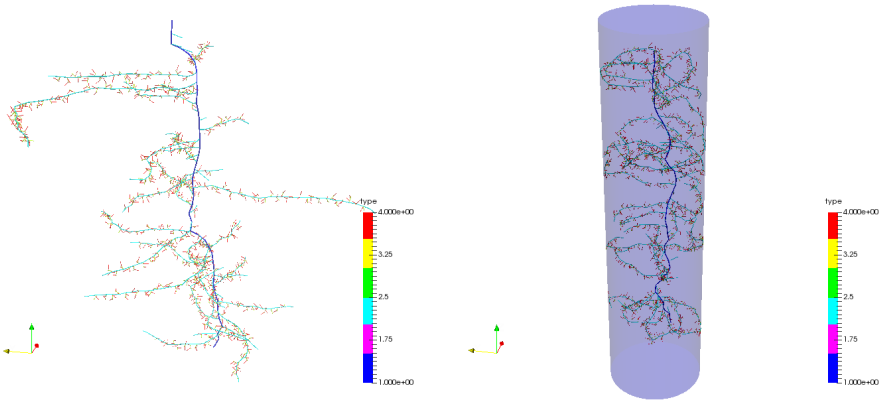
24 Create an interactive plot (use mouse to rotate or zoom) using VTK. Per default 'creationTime', 'radius', or 'type' can be viusalized. You can save a screenshot as png file by pressing 'g', or reset view 'r', or change view by pressing 'x', 'y', 'z', and 'v'.

0.3 Growth in a container

This is an extension of the previous example, where the root system grows in one of two containers (a soil core or rectangular rhizotron). Such geometries are important if we want to mimic experimental settings. In CPlantBox the domain geometry is represented in a mesh free way using signed distance functions (SDF). A SDF returns the distance of a point to its closest boundary, with negative sign if it lies inside of the domain, and a positive if the point is outside.

```
"""small example in a container"""
import sys
sys.path.append("../../..")
import plantbox as pb
import vtk_plot as vp

rs = pb.RootSystem()

# Open plant and root parameter from a file
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

# Create and set geometry

# 1.creates a cylindrical soil core with top radius 5 cm, bot radius 5
    cm, height 50 cm, not square but circular
soilcore = pb.SDF_PlantContainer(5, 5, 40, False)

# 2. creates a square 27*27 cm containter with height 1.4 cm
rhizotron = pb.SDF_PlantBox(1.4, 27, 27)

# Pick 1, or 2
rs.setGeometry(soilcore)  # soilcore, or rhizotron

# Initialize
rs.initialize()

# Simulate
rs.simulate(60)  # days

# Export final result (as vtp)
rs.write("results/example_1b.vtp")

# Export container geometry as Paraview Python script
rs.write("results/example_1b.py")

# Plot, using vtk
vp.plot_roots(rs, "type")
```

Listing 2: Example 1b



(a) Unconfined growth (Example 1a) (b) Confined to a soil core (Example 1b)

Figure 1: Paraview visualizations of results of example 1a and 1b.

The geometry is first created by constructing some specialization of the class SignedDistanceFunction, and is passed to the root system by the method setGeometry:

17 Construct a soil core.

20 Construct a rhizotron.

23 Pick one of the two geometries. Note that it is important to call setGeometry before initialize.

35 Its possible to save the geometry as Paraview Python script for visualization (and debugging) in Paraview, see Figure 1b. Run this script in Paraview by Tools→Python Shell, Run Script.

35 The geometric boundaries can currently not be visualized in the interactive rendering This could be achieved in VTK by creating an iso-surface of the implicit geometry given by the SDF.

38 Interactive VTK plot.

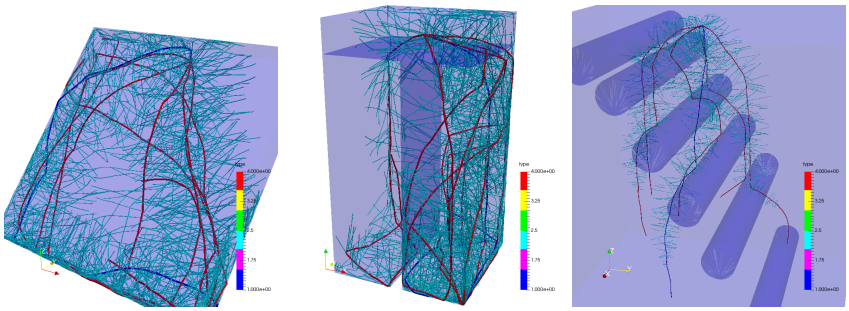
Next, we show how to build more complex container geometries using SDF. Furthermore, we show an example with multiple root systems that is computed in parallel.

0.4 Using SDF with set operations

In the following example we create geometries that we might encounter in experiments. First, we show how to rotate a rhizotron (e.g. to see more roots at the wall due to gravitropism). Second, we create a split box experiment, and furthermore, an example where rhizotubes act as obstacles.

The following examples demonstrates how to build a complex geometry using rotations, translations and set operations on the SDF.

```
"""more complex geometries"""
import sys
sys.path.append("../../..")
import plantbox as pb
import vtk_plot as vp
import math

rs = pb.RootSystem()

# Open plant and root parameter from a file
path = "../../../modelparameter/rootsystem/"
name = "Zea_mays_4_Leitner_2014"
rs.readParameters(path + name + ".xml")

# 1. Creates a square rhizotron r*r, with height h, rotated around the x-axis
r, h, alpha = 20, 4, 45
rhizotron2 = pb.SDF_PlantContainer(r, r, h, True)
posA = pb.Vector3d(0, r, -h / 2)  # origin before rotation
A = pb.Matrix3d.rotX(alpha / 180.*math.pi)
posA = A.times(posA)  # origin after rotation
rotatedRhizotron = pb.SDF_RotateTranslate(rhizotron2, alpha, 0, posA.times(-1))

# 2. A split pot experiment
topBox = pb.SDF_PlantBox(22, 20, 5)
sideBox = pb.SDF_PlantBox(10, 20, 35)
left = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(-6, 0, -5))
right = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(6, 0, -5))
box_ = []
box_.append(topBox)
box_.append(left)
box_.append(right)
splitBox = pb.SDF_Union(box_)

# 3. Rhizotubes as obstacles
box = pb.SDF_PlantBox(96, 126, 130)  # box
rhizotube = pb.SDF_PlantContainer(6.4, 6.4, 96, False)  # a single rhizotube
rhizoX = pb.SDF_RotateTranslate(rhizotube, 90, pb.SDF_Axis.yaxis, pb.Vector3d(96 / 2, 0, 0))

rhizotubes_ = []
y_ = (-30, -18, -6, 6, 18, 30)
z_ = (-10, -20, -40, -60, -80, -120)
tube = []
for i in range(0, len(y_)):
    v = pb.Vector3d(0, y_[i], z_[i])
    tube.append(pb.SDF_RotateTranslate(rhizoX, v))
    rhizotubes_.append(tube[i])

rhizotubes = pb.SDF_Union(rhizotubes_)
rhizoTube = pb.SDF_Difference(box, rhizotubes)

# Set geometry: rotatedRhizotron, splitBox, or rhizoTube
rs.setGeometry(splitBox)

# Simulate
rs.initialize()
rs.simulate(90)  # days

# Export results (as vtp)
```



(a) Rotated rhizotron (b) Split box (c) Rhizotubes

Figure 2: Different geometries described by SDF

```
59 rs.write("results/example_1c.vtp")
60
61 # Export container geometry as Paraview Python script
62 rs.write("results/example_1c.py")
63
64 # Plot, using vtk
65 vp.plot_roots(rs, "type")
```

Listing 3: Example 1c

15-21 Definition of a rotated rhizotron, see Figure 2a: L17 creates the flat container with a small height, this container is then rotated and translated into the desired position. L18 is the position where the origin will lie, and L19 the rotational matrix around the x-axis. In L20 the origin position is rotated. Finally, in L21 the new rotated and translated geometry is created.

23-32 Definition of of a split box, see Figure 2b: The split box is composed of a left box, a right box, and a top box connecting left and right. In L31 the geometry is defined by the set operation union of the three compartments.

34-49 Definition of rhizotubes as obstacles, see Figure 2c: L35 is the surrounding box, L36 a single rhizotube, that is rotated around the y-axis in L37. L39-L46 create a list of rhizotubes at different locations that mimics the experimental setup. L48 and L49 compose the final geometry by to set operation, first a union of all tubes, and then cut them out the surrounding box by taking the difference.

52 Pick one of the three geometries for your simulation.

62 Also more complex geometries can be visualized by the Paraview script, however, set operations are not really performed, only the involved geometries are visualized.

65 We cannot visualize the container geometry in the interactive rendering, but only the roots.

0.5 Multiple root systems

Its possible to simulate multiple root systems. In the following we show a small plot scale simulation.

```
"""multiple root systems"""
import sys
sys.path.append("../../..")
import plantbox as pb
import vtk_plot as vp

path = "../../../modelparameter/rootsystem/"
name = "Zea_mays_4_Leitner_2014"

simtime = 120
N = 3  # number of columns and rows
dist = 40  # distance between the root systems [cm]

# Initializes N*N root systems
allRS = []
for i in range(0, N):
    for j in range(0, N):
        rs = pb.RootSystem()
        rs.readParameters(path + name + ".xml")
        rs.getRootSystemParameter().seedPos = pb.Vector3d(dist * i, dist * j, -3.)  # cm
        rs.initialize(False)  # verbose = False
        allRS.append(rs)

# Simulate
for rs in allRS:
    rs.simulate(simtime, False)  # verbose = False

# Export results as single vtp files (as polylines)
ana = pb.SegmentAnalyser()  # see example 3b
for i, rs in enumerate(allRS):
      vtpname = "results/example_1d_" + str(i) + ".vtp"
      rs.write(vtpname)
      ana.addSegments(rs)  # collect all

# Write all into single file (segments)
ana.write("results/example_1d_all.vtp")

# Plot, using vtk
vp.plot_roots(ana, "radius")
```

Listing 4: Example 1d

11,12 Set the number of columns and rows of the plot, and the distance between the root systems.

15-22 Creates the root systems, and puts them into a list allRS. L20 sets the position of the seed.

25,26 Simulate all root systems

29-36 Saves each root systems, and additionally, saves all root systems into a single file. Therefore, we create an SegmentAnalyser (see Section 2) object in L29 and merge all segments into it L33, and finally export the single file L36. The resulting geometry is shown in Figure 3.

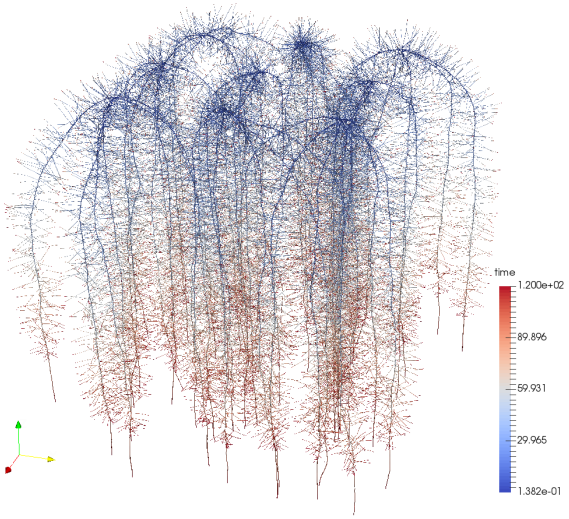


Figure 3: Multiple root systems, colors denote the creation time of the root segments

If we consider only one plant type we often simplify plot scale simulations to a single plant simulation with periodic boundary conditions. We can analyse the root geometry with an underlying periodic grid as described using SegmentAnalyser::mapPeriodic (e.g. as in Section 2.4).

1 The RootSystem class

The class RootSystem is responsible for the model parameters and controls the simulation. Additionally, the class offers utility functions for post processing on a per root level, where each root is represented as a polyline, which is represented by a Python list of nodes.

Post processing functions per segment can be found in the class SegmentAnalyser (see Section 2). A segment is a part of the root and is represented by two nodes that are connected by a straight line.

1.1 Initialize parameters from scratch

In the previous examples we opened the root system parameters from an xml file. In this example we show how to do everything in a Python script without the need of any parameter file. This is especially important if we want to modify parameters in our scripts (e.g. like it is needed for a sensitivity analysis, see Section 1.6).

In order to set up a simulation by hand, we have to define all relevant model parameters. This is done by creating a RootRandomParameter object for each root order or root type, and one SeedRandomParameter for each plant type.

Note that during the simulation, the parameters for a specific root (RootSpecificParameter) are generated from the RootRandomParameter class which represents the random distributions of certain parameters.

```
"""everything from scratch (without parameter files)"""
import sys; sys.path.append("../../..")
import plantbox as pb

import matplotlib.pyplot as plt
import numpy as np

rs = pb.RootSystem()
p0 = pb.RootRandomParameter(rs)  # with default values,
p1 = pb.RootRandomParameter(rs)  # all standard deviations are 0

p0.name = "taproot"
p0.a = 0.2  # [cm] radius
p0.subType = 1  # [-] index starts at 1
p0.lb = 5  # [cm] basal zone
p0.la = 10  # [cm] apical zone
p0.lmax = 30  # [cm] maximal root length, number of lateral branching
    nodes = round((lmax-lb-la)/ln) + 1
p0.ln = 1.  # [cm] inter-lateral distance (16 branching nodes)
p0.theta = 0.  # [rad]
p0.r = 1  # [cm/day] initial growth rate
p0.dx = 10  # [cm] axial resolution
p0.successor = [2]  # add successors
p0.successorP = [1]  # probability that successor emerges
p0.tropismT = pb.TropismType.gravi  #
p0.tropismN = 1.8  # [-] strength of tropism
p0.tropismS = 0.2  # [rad/cm] maximal bending

p1.name = "lateral"
p1.a = 0.1  # [cm] radius
p1.subType = 2  # [1] index starts at 1
p1.lmax = 15  # # [cm] apical zone
p1.lmaxs = 0.15  # [cm] standard deviation of the apical zone
p1.theta = 90. / 180.*np.pi  # [rad]
```

```
34 p1.r = 2  # initial growth rate
35 p1.dx = 1  # [cm] axial resolution
36 p1.tropismT = pb.TropismType.gravi  # exo
37 p1.tropismN = 2  # [-] strength of tropism
38 p1.tropismS = 0.1  # [rad/cm] maximal bending
39
40 rs.setOrganRandomParameter(p0)
41 rs.setOrganRandomParameter(p1)
42
43 srp = pb.SeedRandomParameter(rs)  # with default values
44 srp.seedPos = pb.Vector3d(0., 0., -3.)  # [cm] seed position
45 srp.maxB = 0  # [-] number of basal roots (neglecting basal roots and
       shoot borne)
46 srp.firstB = 10.  # [day] first emergence of a basal root
47 srp.delayB = 3.  # [day] delay between the emergence of basal roots
48 rs.setRootSystemParameter(srp)
49
50 rs.initialize()
51
52 fig, axes = plt.subplots(1, 3, figsize = (15, 7))
53 simtimes = [0, 30, 60, 125]  # the last lateral will emerge at
54 for i in range(0, 3):
55
56     rs.simulate(np.diff(simtimes)[i])  #  [day]
57
58     a = axes[i]
59     a.set_xlim([-15, 15.])
60     a.set_ylim([-35., 0.])  # starts at -3 cm, max length 30 cm
61     a.set_title("after {} days".format(rs.getSimTime()))
62
63     roots = rs.getPolylines()
64     for root in roots:
65         for j, n in enumerate(root[:-1]):
66             n2 = root[j + 1]
67             a.plot([n.x, n2.x], [n.z, n2.z], "g")
68
69 fig.tight_layout()
70 plt.show()
71
72 rs.write("../results/example_2a.vtp")
73
74 # Some outputs....
75 print(" length", rs.getParameter("length"))
76 print("    age", rs.getParameter("age"))
77 print("subType", rs.getParameter("subType"))
78 print("     la", rs.getParameter("la"))
79 print("la_mean", rs.getParameter("la_mean"))
80 print(" radius", rs.getParameter("radius"))
```

Listing 5: Example 2a

5 Matplotlib is Python's easy way to create figures like in Matlab.

6 NumPy is Python's scientific computing package.

9,10 Create the root type parameters of type 1 and type 2.

12-38 We set up a simple root system by hand. First we define the tap root L12-L26, then the laterals L28-L38. By default all standard deviations are 0. Most parameters standard deviations can be set with an additional 's' appended to the parameter name, e.g. $lmaxs$ is the standard deviation of $lmax$, see L32

40,41 Set the root type parameters.

43-47 Create an object of class SeedRandomParameter which defines when basal and shoot borne roots emerge. In this example we neglect basal and shoot borne roots, and just define the seed location, and deactivate basal roots by setting their maximal number $maxB$ to 0 ($firstB$ and $delayB$ are ignored in that case).

48 Sets the root system parameters.

53 We choose the simulation times in a way that we can see the temporal development, and that all lateral roots have emerged in the final time step.

54-70 Within the simulation loop we create Figure 4. L58-61 defines the limits and titles. In L63 we retrieve the roots as polylines which are represented by a list of nodes. In L64-67 we plot the x and z coordinates for each segment (n, $n2$) as green line.

75-80 It is not only possible to set all model parameter, but to retrieve the parameters after the simulation with rs.getParameter(), which returns one value per root. For all parameters that are derived from a random distribution the root specific parameter is returned (e.g. la, L78), i.e. the values that were drawn from the normal distribution. The root random parameter can be accessed by adding '_mean', '_dev' to the parameter value (e.g. la_mean, L79).

Note that all parameters can be set and modified within Python. Especially, standard deviations can be set to zero in order to be able to precisely predict the result. For example we can calculate the total root system length analytically, and check if the numerical simulation yield the (exact) same result. This is performed in the tests test_root.py, and test_rootsystem, which is used to test and validate CPlantBox (see folder CPlantBox/test).

With such simple simulations, we can quickly check if the model does, what we expect. For example the maximal number of laterals of above parameters is $16 = round(lmax - la - lb)/l_n + 1$. We can calculate the time when the final lateral emerges as $-(lmax/r) * \ln(1 - (lmax - l_n/2)/lmax) = 122.8$ days. At simulation time 125 the last lateral root that has emerged is 2.2 days old, and therefore approximately 4.4 cm long (initial growth rate $r_1 = 2$), which agrees with Figure 4.

By default the length of the apical zone is fixed, when the root is created. During growth the apical zone stays in the interval $[la - l_n/2, la + l_n/2]$. The first branch emerges at length lb, when the root length reaches $lb + la$.

In the following two subsections we show, how tropism parameters and inter lateral spacing will affect the resulting root system.

1.2 Root tropism parameters N and σ

We show the influence of the parameters N and σ in the case of gravitropism. The parameter N [1] denotes the strength of the tropism, and σ [cm^{-1}] the flexibility of the root, i.e. the expected angular variation per cm root length.

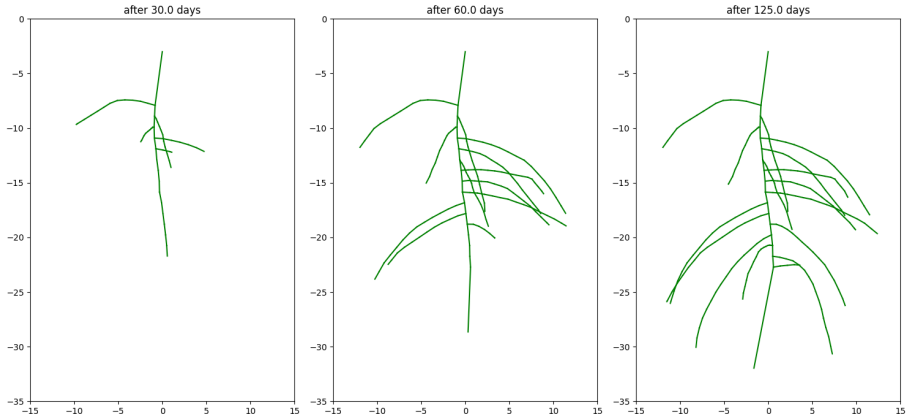


Figure 4: Root development

```
"""shows the influence of tropism paramters"""
import sys
sys.path.append("../../..")
import plantbox as pb
import matplotlib.pyplot as plt
import math

fig, axes = plt.subplots(4, 4, figsize=(15, 10))

N_ = [0, 1, 2, 4]  # strength [1]
sigma_ = [0, 0.1, 0.3, 0.6]  # flexibility [1/m]
dx = 1  # axial resolution
theta = 70 / 180 * math.pi  # insertion angle [1]

rs = pb.RootSystem()
srp = pb.SeedRandomParameter(rs)
srp.firstB, srp.delayB, srp.maxB = 3, 3 , 100
rs.setRootSystemParameter(srp)

p0 = pb.RootRandomParameter(rs)
p0.name, p0.subType, p0.lmax, p0.r, p0.dx, p0.theta = "taproot", 1,
    100, 1, dx, theta
p0.tropismT = 1  # gravitropism
rs.setOrganRandomParameter(p0)

for i, n in enumerate(N_):
    for j, sigma in enumerate(sigma_):
        a = axes[i][j]
        rs.reset()  # does not delete parameters

        p0.tropismN = n
        p0.tropismS = sigma

        rs.initializeLB(1, 1)
        rs.simulate(50, False)

        nodes = rs.getNodes()
        segs = rs.getSegments()
        for s in segs:
            n1, n2 = nodes[s.x], nodes[s.y]
            a.plot([n1.x, n2.x], [n1.z, n2.z], "r")
```

```
41          a.set_title("$N$ = {}, $\sigma$ = {}".format(n, sigma))  #
42          a.axis('equal')
43          a.set_xlim([-30, 30.])
44          a.set_ylim([-40., 0.])
45
46  fig.tight_layout()
47  fig.canvas.set_window_title("Gravitropism parameters")
48  plt.show()
```

Listing 6: Example 2b

10-13 We choose the parameter values N and σ we want to plot. It might be interesting to change the initial insertion *angle*, and the axial resolution dx. Note that a change in axial resolution should not qualitatively change the resulting images.

15-18 The root system is created and SeedRandomParameters are set to produce a basal root every third day.

20-23 The RootRandomParameter for tap and basal roots is defined.

25-28 The loop runs over the parameters we want to modify. We create a subplot for each configuration, and start with a new root system by calling reset in L28.

30-34 We set the parameters (L30,L31) and do the simulation (L33,L34)

36-44 Matplotlib is used for visualization (looping over the segments is rather slow). L36 gives a list of all nodes, and L37 of all segments as two node indices. Therefore, each segment starts at node n1 and ends at node n2, as defined in L39.

L48 Creates Figure 14.

In the first column (σ=0) of Figure 14 nothing happens, because if the root has no flexibility to bend, the strength has no influence on the resulting root system. The first row (N=0) shows the influence of σ only. The growth is undirected, because the strength of the gravitropism is zero. If the roots have small flexibility, they grow downwards because they initially do.

The other subplots show different shapes of the root system. We normally derive the two parameters N and σ by visual comparison. Note that results are independent of the root axial resolution dx and the temporal resolution.

1.3 Inter lateral spacing (ln, lnk)

A single root is divided in basal zone, root branching zone, and root apical zone. Basal and apical zone are given by the parameters la, and lb with standard deviations la_s and lb_s. The branching zone has the size $lmax - la - lb$, where $lmax$ is the maximal root length. The branching zone is divided into inter lateral distances l_n, which are values drawn from a normal distribution with standard deviation l_{ns}. All values are fixed when the root is created This is performed in the method RootRandomParameter::realize(). The chosen parameters reflect the root growth under perfect conditions. Based on this, the root development

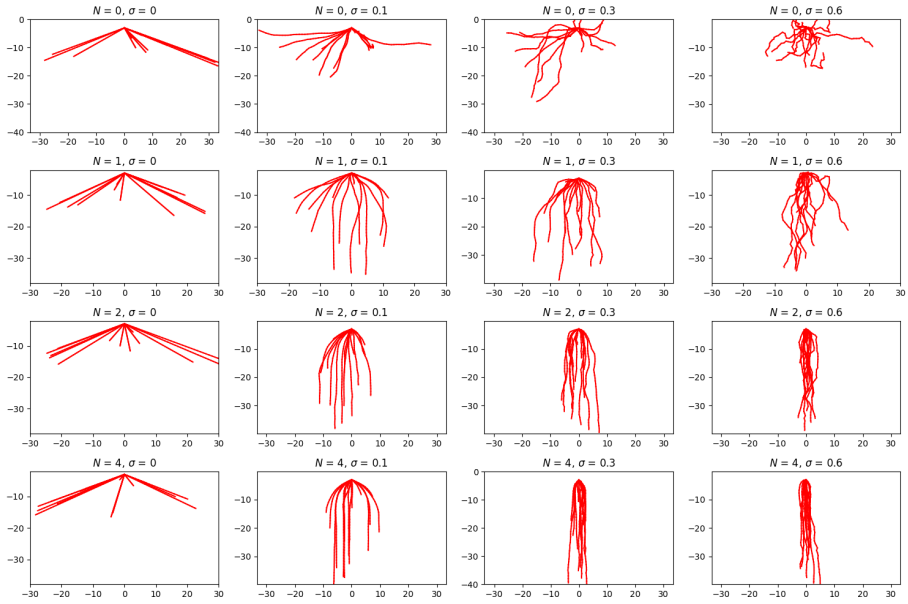


Figure 5: Influence of N and σ in case of gravitropism

can then be influenced by environmental conditions e.g. impeding growth speed, or lateral emergence, see Section 4.

Normally, the setting constant branching distances is sufficient, but sometime experimental data indicate that inter lateral distances are smaller, or larger near the base than near the root tip. The reason for this could be soil root interaction (e.g. root response to dense or nutritious layer), or within the genotype. We added a purely descriptive parameter to mimic such experimental observations. The parameter *lnk*, which is zero per default, defines the slope, at the mid of the branching, altering the inter lateral distance linearly along the root axis. In the following script, we demonstrate the usage of *lnk*.

```
""" shows inter lateral spacing (ln) and how a linear slope (lnk) can
    modify them """
import sys
sys.path.append("../../..")
import plantbox as pb
import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(2, 3, figsize=(15, 7))

rs = pb.RootSystem()
srp = pb.SeedRandomParameter(rs)
srp.firstB, srp.delayB, srp.maxB = 1., 1., 0
rs.setRootSystemParameter(srp)

dx0, dx1 = 100, 1  # very large resolution for taproot
theta = 70 / 180 * np.pi

p0 = pb.RootRandomParameter(rs)
p0.name, p0.subType, p0.lmax, p0.r, p0.dx, p0.theta = "taproot", 1,
    50., 2., dx0, 0.  # parameters as before
```

```
20 p0.tropismT, p0.tropismN, p0.tropismS = pb.TropismType.gravi, 2, 0.1
21 p0.successor, p0.successorP = [2], [1.]  # set up successors
22 p0.lns = 0.  # test with other values...
23 p0.lb, p0.la = 0., 0.
24 rs.setOrganRandomParameter(p0)
25
26 p1 = pb.RootRandomParameter(rs)
27 p1.name, p1.subType, p1.lmax, p1.r, p1.dx, p1.theta = "lateral", 2,
       30., 1., dx1, theta
28 p1.tropismT, p1.tropismN, p1.tropismS = pb.TropismType.exo, 2, 0.2
29 rs.setOrganRandomParameter(p1)
30
31 ln_ = [4., 2.]  # inter lateral distance
32 lnk_ = [-2. / 45., 0, 2. / 45]  # slope
33
34 for i, ln in enumerate(ln_):
35     for j, lnk in enumerate(lnk_):
36         rs.reset()  # does not delete parameters
37
38         p0.ln = ln
39         p0.lnk = ln * lnk  # set up linearly altered spaces
40
41         rs.initializeLB(1, 1)
42         rs.simulate(100, False)
43
44         a = axes[i][j]
45         nodes = rs.getNodes()
46         segs = rs.getSegments()
47         for k, s in enumerate(segs):
48             n1, n2 = nodes[s.x], nodes[s.y]
49             a.plot([n1.x, n2.x], [n1.z, n2.z], 'g')
50
51         pl = rs.getPolylines()
52         for n in pl[0]:
53              axes[i][j].plot([n.x], [n.z], "r*")
54
55         axes[i][j].set_title("$ln$ = {:.2f}, $lnk$ = {:.2f}".format(p0.
       ln, p0.lnk))
56         axes[i][j].axis('equal')
57
58 fig.tight_layout()
59 fig.canvas.set_window_title("Inter lateral distances")
60 plt.show()
```

Listing 7: Example 2c

10-13 A root system with a single tap root is created.

16,16 The axial resolution, and insertion angle is defined. We take a very large axial resolution for the tap root, since we visualize the nodes later on, and we want to see only lateral branching nodes.

18-24 Definition of the tap root. Standard deviations are zero, we do not want any variations. Tropism parameters are chosen in a way, that the tap root grows straight downwards.

26-29 Definition of the first order lateral. Tropism is a strict exotropism (i.e. root follows its initial growth direction).

31,32 The parameter values we want to visualize.

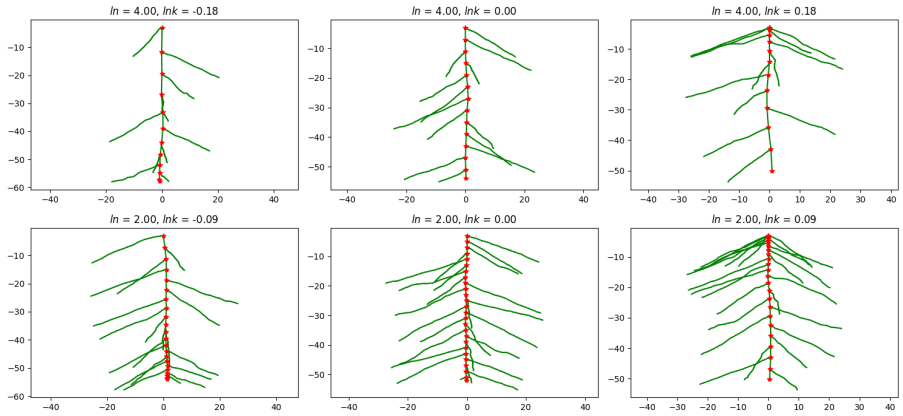


Figure 6: Inter-lateral spacing, larger near base (left column), constant (mid column), and smaller near base (right column)

36 Resets the root system (to $simTime = 0$). Root parameters are not changed.

38,39 Sets the values for this subplot.

41,42 Runs the simulation

44-56 Creates the subplots. First (L47-49) we plot all segments in green. And second (L51-53) we plot all nodes of the tap root as red asterisks.

58-60 Creates Figure 6

The mid column of Figure 6 shows to different inter lateral distances, 4 (top) and 2 (cm) bot. The left column demonstrate the use of negative values for lnk which results in larger distances near the base. The right column has positive values for lnk, which will result in smaller distances near the base. The i-th inter distance is calculated as $ln_i = ln + lnk(x_i - mid_x)$, where x_i is the position within the branching zone, and mid_x is the mid of the branching zone. This is done in RootRandomParameter::realize(). Note that lnk is dimensionless and the slope in the linear equation. At mid of the branching zone the inter-lateral distance equals ln.

In the following we show, how to analyse model results on a per root basis using the RootSystem class. To create density distributions the resulting root segments are analysed using the SegmentAnalyser class, described in Section 2.

1.4 Root system length over time

The following script shows how to analyse root system length versus time.

```
"""root system length over time"""
import sys
sys.path.append("../../..")
import plantbox as pb

import numpy as np
import matplotlib.pyplot as plt
```

```
8
9  path = "../../../modelparameter/rootsystem/"
10 name = "Brassica_napus_a_Leitner_2010"
11
12 rs = pb.RootSystem()
13 rs.readParameters(path + name + ".xml")
14 rs.initialize()
15
16 simtime = 60.  # days
17 dt = 1.
18 N = round(simtime / dt)  # steps
19
20 # Plot some scalar value over time
21 stype = "length"
22 v_, v1_, v2_, v3_ = np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N)
23 for i in range(0, N):
24     rs.simulate(dt)
25     t = np.array(rs.getParameter("type"))
26     v = np.array(rs.getParameter(stype))
27     v_[i] = np.sum(v)
28     v1_[i] = np.sum(v[t == 1])
29     v2_[i] = np.sum(v[t == 2])
30     v3_[i] = np.sum(v[t == 3])
31
32 t_ = np.linspace(dt, N * dt, N)
33 plt.plot(t_, v_, t_, v1_, t_, v2_, t_, v3_)
34 plt.xlabel("time (days)")
35 plt.ylabel(stype + " (cm)")
36 plt.legend(["total", "tap root", "lateral", "2. order lateral"])
37 plt.savefig("results/example_2d.png")
38 plt.show()
```

Listing 8: Example 2d

9-14 Sets up the simulation.

16-18 Defines the simulation time, time step, and the resulting number of simulate(dt) calls.

21 First we state which scalar type we want to analyse ('volume', 'surface', or 'one' would also make sense).

22 Pre-definition of the numpy arrays storing the lengths over time.

23-30 The simulation loop executes the simulation for a single time step L24. L25 calculates the type of each root (the organ subType), L26 the length (or any other parameter) of the root. L27-L30 calculates the total root length at the current time step for all roots, and for specific root types. The method rs.getParameter collects this data from the RootSystem organs. It is possible to access all root random parameters and resulting realisations using rs.getParameter. In C++ the class functions are defined in Root::getParameter, Organ::getParameter.

32-38 Creates Figure 7a.

1.5 Root tips and root bases

Next we show two options how to retrieve root tip and root base positions from a simulation:

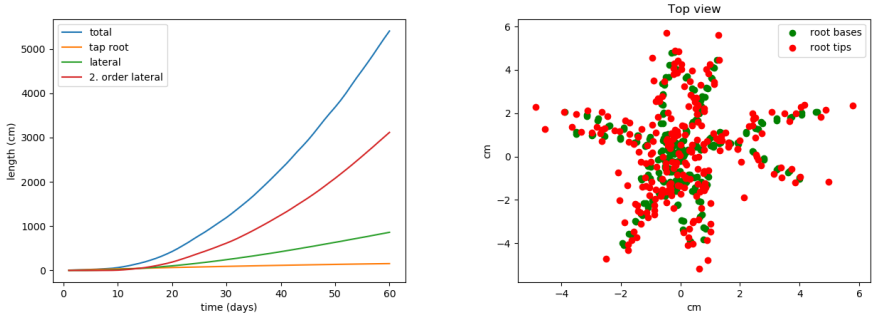
```
"""find root tips and bases (two approaches)"""
import sys
sys.path.append("../../..")
import plantbox as pb

import numpy as np
import matplotlib.pyplot as plt

path = "../../../modelparameter/rootsystem/"
name = "Brassica_napus_a_Leitner_2010"

rs = pb.RootSystem()
rs.readParameters(path + name + ".xml")
rs.initialize()
rs.simulate(21, True)

print(rs.getNumberOfNodes(), "nodes")
print(rs.getNumberOfSegments(), "segments")

# Use polyline representation of the roots
polylines = rs.getPolylines()
bases = np.zeros((len(polylines), 3))
tips = np.zeros((len(polylines), 3))
for i, r in enumerate(polylines):
    bases[i, :] = [r[0].x, r[0].y, r[0].z]  # first index is the base
    tips[i, :] = [r[-1].x, r[-1].y, r[-1].z]  # last index is the tip

# Or, use node indices to find tip or base nodes
nodes = np.array((list(map(np.array, rs.getNodes()))))
tipI = rs.getRootTips()
baseI = rs.getRootBases()

# Plot results (1st approach)
plt.title("Top view")
plt.xlabel("cm")
plt.ylabel("cm")
plt.scatter(nodes[baseI, 0], nodes[baseI, 1], c = "g", label = "root
    bases")
plt.scatter(nodes[tipI, 0], nodes[tipI, 1], c = "r", label = "root tips
    ")
plt.legend()
plt.savefig("results/example_2e.png")
plt.show()

 # check if the two approaches yield the same result
uneq = np.sum(nodes[baseI, :] != bases) + np.sum(nodes[tipI, :] != tips
    )
print("Unequal tips and basals:", uneq)
```

Listing 9: Example 2e

14,15 Reset the simulation and simulate for only 7 days (otherwise there are so many root tips).

17-18 Outputs the number of nodes and segments to get an idea how big the resulting root system is. Note that number of segments equals the number of nodes minus the number of base roots that will emerge. Base roots are tap roots, basal roots and shootborne roots.

20-26 The first approach retrieves all roots as polylines L21. Root tips are the last nodes of the polylines L26, root bases the first nodes L25. Roots



(a) Total root length versus time (b) Top view of the root tip and root bases

Figure 7: Root system analysis: Example 2d (a), Example 2e (b)

that have not started to grow have only 1 node, and are not retrieved by getPolylines().

28-31 Second approach: L29 rs.getNodes() returns all nodes of the root system as a list of nodes, i.e. Vector3d objects. Each Vector3d object can be converted into a numpy array automatically, but is necessary to do that for each element of the list. The methods L30, L31 return the indices of the tips and bases.

33-41 Creates Figure 7b using the second approach.

44,45 Verifies that both approaches yield the same result.

1.6 Sensitivity analysis

In the next part we vary given parameters in order to make a sensitivity analysis. This takes a lot of simulation runs, and we demonstrate the of use parallel computing to speed up execution.

In this exemplary sensitivity study we will vary the insertion angle of the tap and basal root, and look at the resulting change in mean root tip depth and root tip radial distance.

```
"""sensitivity analysis: impact of insertion angle on root tip
    distribution"""
import sys
sys.path.append("../../..")
import plantbox as pb

import math
from multiprocessing import Pool
import numpy as np
import matplotlib.pyplot as plt


# sets all standard deviation to a percantage, i.e. value*s
def set_all_sd(rs, s):
    for p in rs.getRootRandomParameter():
        p.lmaxs = p.lmaxs * s
        p.lbs = p.lb * s
```

```
        p.las = p.la * s
        p.lns = p.ln * s
        p.rs = p.r * s
        p.a_s = p.a * s


# Parameters
path = "../../../modelparameter/rootsystem/"
name = "Zea_mays_1_Leitner_2010"
simtime = 25
N = 25  # resolution of paramter
runs = 25  # iterations
theta0_ = np.linspace(0, math.pi / 2, N)


# One simulation
def simulate(i):
    rs = pb.RootSystem()
    rs.readParameters(path + name + ".xml")
    set_all_sd(rs, 0.)  # set all sd to zero
    p1 = rs.getRootRandomParameter(1)  # tap and basal root type
    # 1. vary parameter
    p1.theta = theta0_[i]
    # 2. simulate
    rs.initializeLB(1, 1, False)
    rs.simulate(simtime, False)
    # 3. calculate target
    depth = 0.  # mean depth
    rad_dist = 0.  # mean raidal distance
    roots = rs.getPolylines()
    for r in roots:
        depth += r[-1].z
        rad_dist += math.hypot(r[-1].x, r[-1].y)
    depth /= len(roots)
    rad_dist /= len(roots)
    return depth, rad_dist


depth_ = np.zeros(N)
rad_dist_ = np.zeros(N)

for r in range(0, runs):

    print("run", r + 1)

    # Parallel execution
    param = []  # param is a list of tuples
    for i in range(0, N):
        param.append((i,))
    pool = Pool()
    output = pool.starmap(simulate, param)
    pool.close()

    # Copy results
    for i, o in enumerate(output):
        depth_[i] += (o[0] / runs)
        rad_dist_[i] += (o[1] / runs)

# Figure
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (10, 8))
axes[0].set_xlabel('Insertion angle theta (-)')
axes[1].set_xlabel('Insertion angle theta (-)')
```

```
79 axes[0].set_ylabel('Mean tip depth (cm)')
80 axes[1].set_ylabel('Mean tip radial distance (cm)')
81 axes[0].plot(theta0_, depth_)
82 axes[1].plot(theta0_, rad_dist_)
83 fig.subplots_adjust()
84 plt.savefig("results/example_2f.png")
85 plt.show()
```

Listing 10: Example 2f

12-20 Defines a function to set all standard deviations proportional to the parameter values. We use this function in the following to set the standard deviation to zero everywhere.

23-29 Parameters of the analysis. N denotes the resolution of the parameter we vary, and $runs$ the number of iterations, i.e. a total of $N \cdot runs$ simulations are performed. In L29 we define the insertion angle to be varied linearly between 0 and $\pi/2$.

33-52 Definition of a function that performs the simulation and returns mean root tip depth and radial distance. First we create a root system and set the standard deviation to zero L34-L36. L37, amd L39 sets the insertion angle. L41 initializes the root system and states that basal root are of root type 1 (same as tap root). The False value turns of verbosity to avoid any outputs to the console. L42 performs the simulation. L44-L51 calculates the mean root tip depth and radial distance.

55-73 This section performs the computation of all simulation runs. L55-56 preallocates the resulting arrays. L62-L68 performs the parallel computations, index i is the index of the insertion angle. L71-73 calculates the mean values per simulation run.

74-83 Creates the resulting Figure 8. Note that the resulting curves become smoother, if the number of runs is increased (L28).

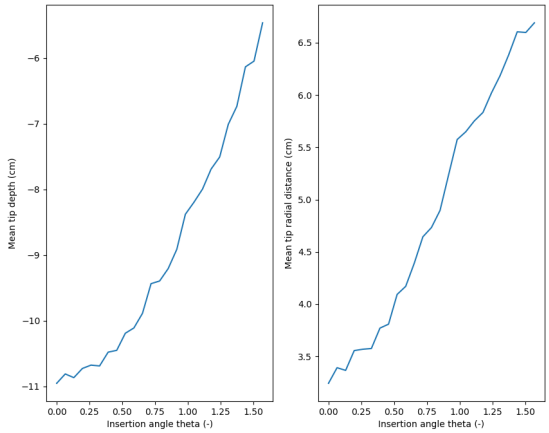


Figure 8: Sensitivity of mean root tip depth (left) and radial distance (right) to the insertion angle theta (Example 2f)

2 The SegmentAnalyser class

The class SegmentAnalyser offers post-processing methods per root segment. The advantage is that we can do distributions or densities, and that we can analyse the segments within any geometry.

We start with a small example plotting the root surface densities of a root system versus root depth.

2.1 Root surface densities

```
"""root system surface density"""
import sys
sys.path.append("../../..")
import plantbox as pb
import numpy as np
import matplotlib.pyplot as plt

path = "../../../modelparameter/rootsystem/"
name = "Brassica_napus_a_Leitner_2010"  # "Crypsis_aculeata_Clausnitzer_1994"

rs = pb.RootSystem()
rs.readParameters(path + name + ".xml")

depth = 220
layers = 50
runs = 10

rl_ = []
for i in range(0, runs):
    rs.initialize(False)
    rs.simulate(120, False)
    ana = pb.SegmentAnalyser(rs)
    rl_.append(ana.distribution("length", 0., -depth, layers, True))
```

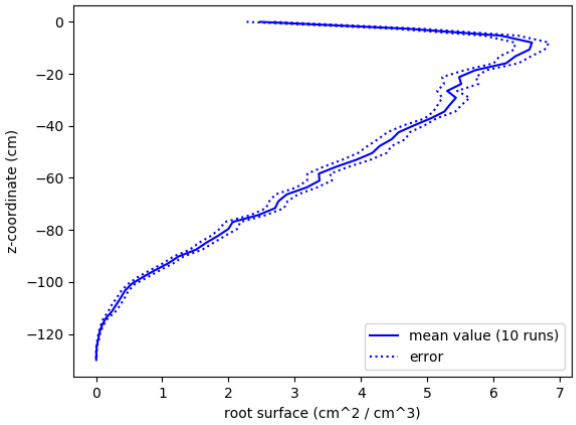


Figure 9: Root surface densitiy versus depth(Example 3a)

```
24
25 soilvolume = (depth / layers) * 10 * 10
26 rl_ = np.array(rl_) / soilvolume  # convert to density
27 rl_mean = np.mean(rl_, axis=0)
28 rl_err = np.std(rl_, axis=0) / np.sqrt(runs)
29
30 z_ = np.linspace(0, -depth, layers)  # z - axis
31 plt.plot(rl_mean, z_, "b")
32 plt.plot(rl_mean + rl_err, z_, "b:")
33 plt.plot(rl_mean - rl_err, z_, "b:")
34
35 plt.xlabel("root surface (cm^2 / cm^3)")
36 plt.ylabel("z-coordinate (cm)")
37 plt.legend(["mean value (" + str(runs) + " runs)", "error"])
38 plt.savefig("results/example_3a.png")
39 plt.show()
```

Listing 11: Example 3a

8-12 Pick a root system.

14-16 Depth describes the y-axis of the graph, layers the number of vertical soil layers, where the root surface is accumulated, and runs is the number of simulation runs.

18-23 Performs the simulations. L23 creates a distribution of a parameter (name) over a vertical range (bot, top). The data are accumulated in each layer, segments are either cut (exact = True) or accumulated by their mid point (exact = False).

25 In order to calculate a root surface density from the summed up surface, we need to define a soil volume. The vertical height is the layer length, length and width (here 10 cm), can be determined by planting width, or by a confining geometry.

26-28 Calculates the densities mean and the standard error.

30-39 Prepares the plot (see Figure 9).

2.2 Analysis per segment within a geometry

The following script demonstrates some of the post processing possibilities by setting up a virtual soil core experiment (see Figure 10), where we analyse the content of two soil cores located at different positions.

```
"""analysis of results using signed distance functions"""
import sys; sys.path.append("../../..")
import numpy as np
import matplotlib.pyplot as plt

import plantbox as pb

path = "../../../modelparameter/rootsystem/"
name = "Zea_mays_1_Leitner_2010"  # Zea_mays_1_Leitner_2010, Brassica_napus_a_Leitner_2010

rs = pb.RootSystem()
rs.readParameters(path + name + ".xml")
rs.initialize()
rs.simulate(120)
rs.write("results/example_3b.vtp")

r, depth, layers = 5, 100., 100  # Soil core analysis
soilcolumn = pb.SDF_PlantContainer(r, r, depth, False)  # in the center of the root
soilcolumn2 = pb.SDF_RotateTranslate(soilcolumn, 0, 0, pb.Vector3d(10, 0, 0))  # shift 10 cm

# pick one geometry for further analysis
geom = soilcolumn

z_ = np.linspace(0, -1 * depth, layers)
fig, axes = plt.subplots(nrows = 1, ncols = 3, figsize = (16, 8))
for a in axes:
    a.set_xlabel('RLD (cm/cm^3)')  # layer size is 1 cm
    a.set_ylabel('Depth (cm)')

# Make a root length distribution
layerVolume = depth / layers * 20 * 20
times = [120, 60, 30, 10]
ana = pb.SegmentAnalyser(rs)
ana.cropDomain(20, 20, depth)  # ana.mapPeriodic(20, 20)
rl_ = []
axes[0].set_title('All roots in 20*20*100')
for t in times :
    ana.filter("creationTime", 0, t)
    rl_.append(ana.distribution("length", 0., -depth, layers, True))
    axes[0].plot(np.array(rl_[-1]) / layerVolume, z_)
axes[0].legend(["10 days", "30 days", "60 days", "120 days"])

# Make a root length distribution along the soil core
layerVolume = depth / layers * r * r * np.pi
ana = pb.SegmentAnalyser(rs)
ana.crop(geom)
ana.pack()
rl_ = []
axes[1].set_title('Soil core')
for t in times :
    ana.filter("creationTime", 0, t)
    rl_.append(ana.distribution("length", 0., -depth, layers, True))
    axes[1].plot(np.array(rl_[-1]) / layerVolume, z_)
axes[1].legend(["10 days", "30 days", "60 days", "120 days"])
```

```
55
56 # distributions per root type
57 ana = pb.SegmentAnalyser(rs)
58 ana.crop(geom)
59 ana.pack()
60 rl_ = []
61 for i in range(1, 5):
62     a = pb.SegmentAnalyser(ana)  # copy
63     a.filter("subType", i)
64     rl_.append(a.distribution("length", 0., -depth, layers, True))
65 axes[2].set_title('Soil core')
66 axes[2].plot((np.array(rl_[0]) + np.array(rl_[3])) / layerVolume, z_)
67 axes[2].plot(np.array(rl_[1]) / layerVolume, z_)
68 axes[2].plot(np.array(rl_[2]) / layerVolume, z_)
69 axes[2].legend(["basal roots", "first order roots", "second order roots
     "])
70
71 fig.subplots_adjust()
72 plt.savefig("results/example_3b.png")
73 plt.show()
```

Listing 12: Example 3b

11-15 Performs the simulation.

17-22 We define two soil cores, one in the center of the root and one 10 cm translated. In L22 we pick which one we use for the further analysis. Figure 10 shows the resulting geometry, with a soil core radius of 10 cm.

24-28 Prepares three sub-figures.

31-41 Creates a root length distribution versus depth at different ages. L33 creates the SegmentAnalyser object, and L34 crops it to a fixed domain or maps it into a periodic domain. In L38 the filter function keeps only the segments, where the parameter (first argument) is in the range between second and third argument. L39 creates the distribution.

44-54 We repeat the procedure, but we crop to the soil core selected in L22.

57-89 In the third sub plot we make densities of specific root types like basal roots, first order roots, and second order roots. In L58 we crop the segments to the soil core geometry. In L63 we filter for the selected sub type, and in L64 we create the density distribution.

71-73 Show and save resulting Figure 11 and 12 for the two soil cores (chosen in L22).

The example shows differences between the central core and shifted core (see Figure 11 and 12) because the central core captures all roots emerging from the seed. The basic idea is that such analysis can help to increase the understanding of variations in experimental observations.

2.3 SegmentAnalyser for DGF or VTP export

If we want to export our root system as dune grid file (dgf) we need to introduce an artificial shoot. By default the tap root and basal root starts at the seed

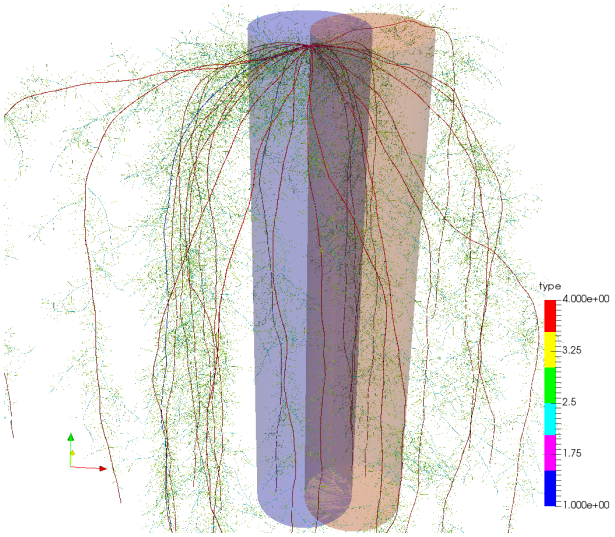


Figure 10: Virtual soil cores experiment (Example 3b): Central core (blue), shifted core (red)

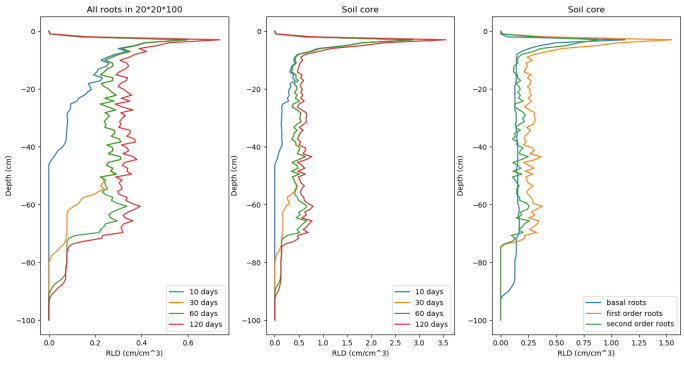


Figure 11: Central core (Example 3b)

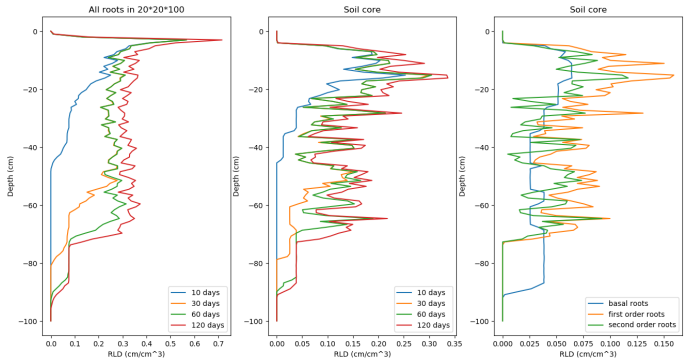


Figure 12: Shifted core (Example 3b)

node (i.e. multiple segments start at the same node), and its difficult to define a boundary condition for such a situation (e.g. in DuMux). Furthermore, if there are shoot borne roots, they emerge out of nothing above the seed node. Therefore, we introduce artificial segments eventually connecting shoot borne roots (if there are any) and connecting the seed node that is normally located at (0,0,-3) to the origin at (0,0,0). The following code snippet shows how to export a root system as dgf file:

```
"""dgf and vtp export example"""
import sys; sys.path.append("../../..")
import plantbox as pb

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")
rs.initialize()
rs.simulate(15, True)

ana = pb.SegmentAnalyser(rs)

aseg = rs.getShootSegments()  # if there are no shoot borne roots, it is only one segment
for s in aseg:
    print("Shoot segment", s)
    ana.addSegment(s, 0., 0.1, True)  # ct, radius, insert first

ana.write("results/example_3c.vtp", ["radius", "surface"])
ana.write("results/example_3c.dgf")
```

Listing 13: Example 3c

6-11 Defines a root sytem

13 Create the analyser object

15-18 Get the artificial shoot segments from the root system (L15) and manually add them to the SegmentAnalyser (L18), first argument is the segment s, second is creation time, third is radius, and fourth arguments states that

the segment is inserted at the top of the list (True), while (False) would append it to segment list.

19 Write the VTP file. For VTP files its possible to add a list of parameters that will be exported.

20 Write the DGF file. The parameters are fixed (see documentation of SegmentAnalyser::writeDGF)

It is also possible to make use of the SegmentAnalyser class without any other CPlantBox classes (e.g. for writing vtp from measurements). The following example shows how to construct the class with arbitrary nodes and segments (e.g. from measurements).

```
""" nodes and segments from measurements """
import sys; sys.path.append("../../..")
import plantbox as pb

# Data from any source, as Python types
nodes = [ [0, 1, 0], [0, 1, -1], [0, 1, -2], [0, 1, -3], ]
segs = [ [0, 1], [1, 2], [2, 3] ]
cts = [0., 0., 0.]
radii = [ 0.1, 0.1, 0.1 ]

# convert from Python to C++ binding types
nodes = [pb.Vector3d(n[0], n[1], n[2]) for n in nodes]
segs = [pb.Vector2i(s[0], s[1]) for s in segs]

# create the SegmentAnalyser without underlying RootSystem
ana = pb.SegmentAnalyser(nodes, segs, cts, radii)

print("length", ana.getSummed("length"))
ana.write("results/example_3d.vtp", ["creationTime", "radius"])
```

Listing 14: Example 3d

6-9 Define some segments with data

12,13 We convert the Python list to lists of C++ types

16 We create the SegmentAnalyser object without an underlying Organism

18,19 Use the Analyser object, by printing information, or writing a vtp.

Next, we describe how to make an animation from a CPlantBox simulation.

2.4 How to make an animation

In order to create an animation in Paraview we have to consider some details. The main idea is to export the result file as segments using the class SegmentAnalyser. A specific frame is then obtained by thresholding within Paraview using the segments creation times. In this way we have to only export one vtp file.

We modify example1b.py to demonstrate how to create an animation.

```
"""increase axial resolution (e.g. for animation)"""
import sys; sys.path.append("../../..")
import plantbox as pb
import vtk_plot as vp

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

# Modify axial resolution
for p in rs.getRootRandomParameter():
    p.dx = 0.1  # adjust resolution

# Simulate
rs.initialize()
rs.simulate(60)  # days

# Export results as segments
ana = pb.SegmentAnalyser(rs)
ana.write("results/example_3e.vtp")

ana.mapPeriodic(15, 10)
ana.write("results/example_3e_periodic.vtp")

# Export geometry as Paraview Python script
box = pb.SDF_PlantBox(15, 10, 35)
rs.setGeometry(box)
rs.write("results/example_3e_periodic.py")

# Plot final (periodic) image, using vtk
vp.plot_roots(ana, "creationTime", True, 'oblique')
```

Listing 15: Example 4c (modified from Example 1b)

11,12 Its important to use a small resolution in order to obtain a smooth animation. L18 set the axial resolution to 0.1 cm.

19,29 Instead of saving the root system as polylines, we use the SegmentAnalyser to save the root system as segments.

22,23 It is also possible to make the root system periodic in the visualization in x and y direction to mimic field conditions.

26-28 We save the geometry as Python script for the visualization in ParaView.

After running the script we perform the following operations Paraview to create the animation:

1. Open the .vtp file in ParaView (File→Open...), and open tutorial/examples/python/results/example_3e.vtp.
2. Optionally, create a tube plot with the help of the script tutorial/pyscript/rsTubePlot.py (Tools→Python Shell, press 'Run script').
3. Run the script tutorial/pyscript/rsAnimate.py (Tools→Python Shell, press 'Run script'). The script creates the threshold filter and the animation.

4. Optionally, visualize the domain boundaries by running the script tutorial/examples/python/results/example_4e.py (Tools→Python Shell, press 'Run script'). Run after the animation script (otherwise it does not work).

5. Use File→Save Animation... to render and save the animation. Pick quality (<100 %), and the frame rate in order to achieve an appropriate video length, e.g. 300 frames with 50 fps equals 6 seconds. The resulting files might be uncompressed and are very big. The file needs compression, for Linux e.g. ffmpeg -i in.avi -vcodec libx264 -b 4000k -an out.avi, produces high quality and tiny files, and it plays with VLC.

3 Tropisms

The change in root growth direction is described by tropisms. In the following we show how to implement directed growth toward higher water content or nutrient concentration, and demonstrate how to simply make new user defined tropism rules.

3.1 Hydro- and chemotropism

Root growth direction is influenced by soil conditions such as water content, soil strength, or nutrient concentration. In the following example we model the influence of a nutrient rich layer to root system development

```
"""hydrotropism in a thin layer"""
import sys; sys.path.append("../../..")
import plantbox as pb
import vtk_plot as vp

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

# Manually set tropism to hydrotropism for the first ten root types
sigma = [0.4, 1., 1., 1., 1. ] * 2
for p in rs.getRootRandomParameter():
        p.dx = 0.25  # adjust resolution
        p.tropismT = pb.TropismType.hydro
        p.tropismN = 2  # strength of tropism
        p.tropismS = sigma[p.subType - 1]

# Static soil property in a thin layer
maxS = 0.7  # maximal
minS = 0.1  # minimal
slope = 5  # linear gradient between min and max (cm)
box = pb.SDF_PlantBox(30, 30, 2)  # cm
layer = pb.SDF_RotateTranslate(box, pb.Vector3d(0, 0, -16))
soil_prop = pb.SoilLookUpSDF(layer, maxS, minS, slope)

# Set the soil properties before calling initialize
rs.setSoil(soil_prop)

# Initialize
rs.initialize()

# Simulate
simtime = 100  # e.g. 30 or 60 days
dt = 1
N = round(simtime / dt)
for _ in range(0, N):
    # in a dynamic soil setting you would need to update the soil
     properties (soil_prop)
     rs.simulate(dt)

# Export results (as vtp)
rs.write("results/example_4a.vtp")

# Export geometry of static soil
rs.setGeometry(layer)  # just for vizualisation
rs.write("results/example_4a.py")
```

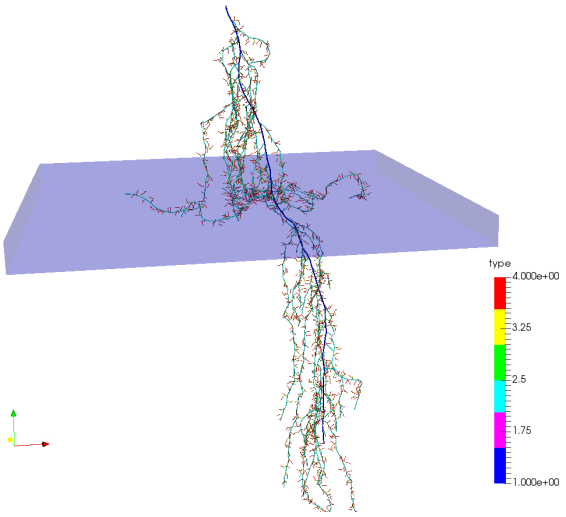


Figure 13: Chemotropism in a nutrient rich layer (Example 4a)

```
47
48 # Plot, using vtk
49 vp.plot_roots(rs, "type")
```

Listing 16: Example 4a

6-9 Creates the root system and opens the parameter file

12-17 Change the tropism for all root types: L13 modifies the axial resolution, L14 set the tropism to hydrotropism, and L15-16 sets the two tropism parameters. The parameter σ is set to 0.4 for the tap root ($subType = 1$), and to 1. for the rest of the root types.

19-25 Definition of a static soil property using SDF. We first define the geometry (L20-L21), and then create a static soil (L22) that obtains the maximal value $maxS$ inside the geometry, $minS$ outside the geometry, and linear slope with length $slope$. At the boundary the soil has the value $(maxS + minS)/2$.

28 Sets the soil. Must be called before RootSystem::initialize()

41 Initializes the root system, and among others sets up the hydrotropism.

33-39 Simulation loop

42 Exports the root system geometry

45-46 We actually do not wish to set this geometry, but we abuse the writer of the class RootSytem to export a Python script showing the layer geometry. The resulting ParaView visualization is presented in Figure 13.

3.2 Root age dependent tropism

Normally, the simulation is created from a set of parameters. For tropisms these are the type of tropism T, number of trials N , and tortuosity σ. There are only a few predefined tropisms called 'gravi', 'plagio', 'exo', or 'hydro', but it is simple to add user defined tropisms. The following example demonstrates how to define a root age dependent tropism, where roots first grow according to exotropism (following the inital growth direction), and after a certain age change to gravitropic growth.

The new tropism class must be derived from the class Tropism. In CPlantBox tropism is realised with a random optimization process, where the 'best' direction is chosen from N possible direction, according to an objective function that is minimized. Normally, it is sufficient to overwrite this function, called Tropism ::tropismObjective, to change the tropism behaviour. This can be done in Python or in C++. The classes Hydrotropism, Gravitropism, and Plagiotropism (in tropisms.h) are examples for this procedure.

If the whole concept of random optimization is altered, Tropism ::getUCHeading must be overwritten, which is only possible in C++. If the geometry model is also changed Tropism::getHeading must be overwritten.

The following example shows how to implement a new tropism in Python. Two new tropism are introduced: The first does nothing but to output the incoming arguments of the method Tropism::tropismObjective to the command line (e.g. for debugging). The second one describes a root age dependent tropism that starts with exotropism and changes with root age to gravitropism.

```
"""user defined tropism in python"""
import sys; sys.path.append("../../..")
import numpy as np
import plantbox as pb
import vtk_plot as vp


class My_Info_Tropism(pb.Tropism):
    """ User tropism 1: print input arguments to command line """

    def tropismObjective(self, pos, old, a, b, dx, root):
        print("Postion \t", pos)
        print("Heading \t", old.column(0))
        print("Test for angle alpha = \t", a)
        print("Test for angle beta = \t", b)
        print("Length of next segment \t", dx)
        print("Root id \t", root.getId(), "\n")
        print("Root age \t", root.getAge(), "\n")
        return 0.


class My_Age_Tropism(pb.Tropism):
    """ User tropism 2: depending on root age use plagio- or
    gravitropism """

    def __init__(self, rs, n, sigma, age):
        super(My_Age_Tropism, self).__init__(rs)
        self.exo = pb.Exotropism(rs, 0., 0.)
        self.gravi = pb.Gravitropism(rs, 0., 0.)
        self.setTropismParameter(n, sigma)
        self.age = age

    def tropismObjective(self, pos, old, a, b, dx, root):
```

```
33          age = root.getAge()
34          if age < self.age:
35              return self.exo.tropismObjective(pos, old, a, b, dx, root)
36          else:
37              return self.gravi.tropismObjective(pos, old, a, b, dx, root
        )
38
39
40  # set up the root system
41  rs = pb.RootSystem()
42  path = "../../../modelparameter/rootsystem/"
43  name = "Zea_mays_1_Leitner_2010"
44  rs.readParameters(path + name + ".xml")
45  rs.initialize()
46
47  # Set useer defined after initialize
48  mytropism1 = My_Info_Tropism(rs)
49  mytropism1.setTropismParameter(2., 0.2)
50  mytropism2 = My_Age_Tropism(rs, 1.5, 0.5, 5)  # after 5 days switch
        from plagio- to gravitropism
51  rs.setTropism(mytropism2, 4)  # 4 for base roots, -1 for all root types
52
53  # Simulate
54  simtime = 100  # e.g. 30 or 60 days
55  dt = 1
56  N = round(simtime / dt)
57  for _ in range(0, N):
58      rs.simulate(dt)
59
60  # Export results (as vtp)
61  rs.write("results/example_4b.vtp")
62
63  # Plot, using vtk
64  vp.plot_roots(rs, "age", True, 'oblique')
```

Listing 17: Example 4b

8-19 Creates a new tropism that just writes incoming arguments of Tropism ::tropismObjective to the command line. This can be used for debugging. The new class is extended from rb.Tropism, and the method Tropism ::tropismObjective is overwritten with the right number of arguments.

22-37 Again, we extend the new class from rb.Tropism. In L25-30 we define our own constructor. Doing this two things are important: (a) the constructor of the super class must be called (L26), and (b) the tropism parameters n, and σ must be set (L29). Furthermore, the constructor defines two tropisms: exo- and gravitropism, that are used in Tropism::tropismObjective at a later point, and a root age that dermines when to switch betwee exo- and gravitropism.
In L32-L37 the method Tropism::tropismObjective is defined. We choose the predefined objective function depending on the root age.

41-45 Sets up the simulation.

48-51 L48,L49 creates the first user defined tropsim. Since we did not define a constructor Tropism::setTropismParameter must be called. L50 creates the second user defined tropism. In L51 the tropism is chosen, using the method Tropism::setTropism. The second argument states for which root

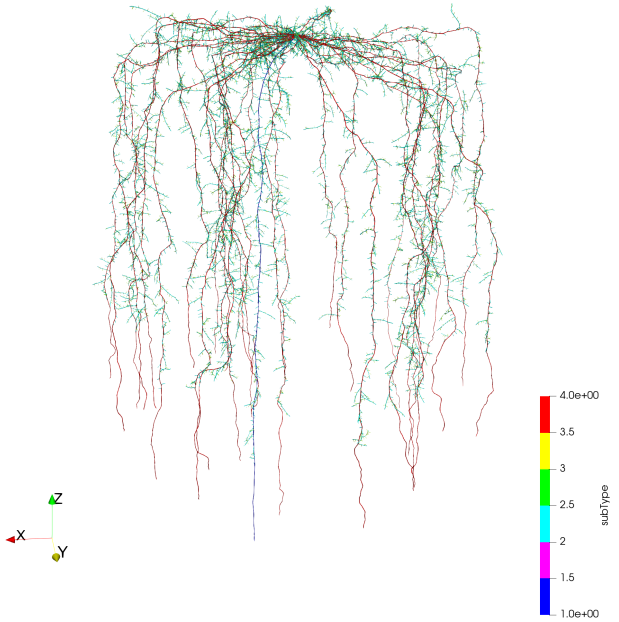


Figure 14: Depending on root age the laterals follow plagio- or gravitropism (Example 4b)

type it applies. Number 4 is the (default) root type for basal roots, -1 states that the tropism applies for all root types (default = -1).

54-58 The simulation loop.

61 Exports the result producing Figure (14).

64 VTK plot.

4 Root functional modelling

Root growth is strongly influenced by pedo-climatic conditions, and plant internal state. CPlantBox offers 'build in' ways to develop such models, see Schnepf et al. (2018).

CPlantBox is a bottom up model were root growth is first defined under perfect conditions. Adding mechanisms to take environmental conditions into account will alter the root system development by impeding root growth, and by changing soil allocation by roots due to root tropism and due to altered branching patterns.

In this section we assume static soil conditions, and demonstrate the predefined ways how the soil can affect root growth. Dynamic soil conditions are described in the following Section 5.

Implemented root responses are the change in direction of the growing root tip, as described in previous Section 3. Further root responses are

- scaling of the elongation rate
- change of insertion angle
- change of lateral emergence probability

4.1 Scaling the elongation rate

Root elongation rate is influenced by soil properties such as water content, temperature, soil density, solutes, and many more. Regarding the processes that are investigated, various models can be applied. In CPlantBox the elongation rate is scaled with no predefined interpretation, i.e. we have to define a elongation rate scaling, which is dependent on such soil properties. The following example defines two compartments (one left, one right), where we change this scaling, and then analyse the results. The same procedure will be used in Example 4.2 and 4.3.

```
"""scales the root elongation rate"""
import sys; sys.path.append("../../..")
import math
import plantbox as pb
import vtk_plot as vp

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

# box with a left and a right compartment for analysis
sideBox = pb.SDF_PlantBox(10, 20, 50)
left = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(-4.99, 0, 0))
right = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(4.99, 0, 0))
leftright = pb.SDF_Union(left, right)
rs.setGeometry(leftright)

# left compartment has a minimum of 0.01, 1 elsewhere
maxS = 1.  # maximal
minS = 0.05  # minimal
slope = 1.  # [cm] linear gradient between min and max
leftC = pb.SDF_Complement(left)
soilprop = pb.SoilLookUpSDF(leftC, maxS, minS, slope)
```

```
25
26 # Manually set scaling function and tropism parameters
27 sigma = [0.4, 1., 1., 1., 1. ] * 2
28 for p in rs.getRootRandomParameter():
29     p.dx = 0.25  # adjust resolutionx
30     p.tropismS = sigma[p.subType - 1]
31     p.f_se = soilprop  # 1. Scale elongation
32
33 # simulation
34 rs.initialize()
35 simtime = 60.
36 dt = 1.
37 for i in range(0, round(simtime / dt)):
38     # in a dynamic setting change soilprop here
39     rs.simulate(dt, False)
40
41 # analysis
42 l = rs.getSummed("length")
43 al = pb.SegmentAnalyser(rs)
44 al.crop(left)
45 ll = al.getSummed("length")
46 ar = pb.SegmentAnalyser(rs)
47 ar.crop(right)
48 lr = ar.getSummed("length")
49 print('\nLeft  compartment total root length {:g} cm, {:g}%'.format(ll,
       100 * ll / l))
50 print('\nRight compartment total root length {:g} cm, {:g}% \n'.format(
       lr, 100 * lr / l))
51
52 # write results
53 rs.write("results/example_5a.py")  # compartment geometry
54 rs.write("results/example_5a.vtp")  # root system
55
56 # plot, using vtk
57 vp.plot_roots(rs, "rootLength")  # press 'y'
```

Listing 18: Example 5a

7-10 Creates the root system and opens the parameter file.

13-17 We create a confining box with two overlapping boxes called *left* and *right*. These geometries are used for later analysis.

20-24 We define static soil properties using SDF (L23, L24) as we did in Section 3.1. The left compartment has the value *minS*, the right *maxS*, between them is a linear gradient of length *slope*.

27-31 Sets the scaling functions. L29 adjusts axial resolution and L30 tortuosity *sigma*. L31 sets the scale elongation function f_{se} to the soil property (i.e. scales to *minS* in the left, *maxS* in the right compartment).

34-39 Initialization and simulation loop. In a dynamic setting, *soilprop* needs to be updated in each time step (comment L38).

42-50 Analysis the root length in the left and right compartment. With parameters *minS* and *slope* only approximately 21% are located in the left compartment.

53, 54 Writes the results for Paraview visualization (see Figure 15).

57 A vtk simulation of root lengths. Press 'y' to obtain a x-z view of the root system to better see the effect.

Next, we give a short layout, how the code would look like, if we take measured data (e.g. density versus depth), and include it in the simulation.

```
"""Scale root elongation based on EquidistantGrid1D"""
import sys; sys.path.append("../../..")
import plantbox as pb
import numpy as np
import vtk_plot as vp

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

scale_elongation = pb.EquidistantGrid1D(0, -50, 100)
soil_strength = np.ones((99,))
soil_strength[20:30] = 5  # data, with a very dense layer at -10 to -15 cm
scales = np.exp(-0.4 * soil_strength)  # equation (TODO)
scale_elongation.data = scales  # set proportionality factors
print("-3 cm ", scale_elongation.getValue(pb.Vector3d(0, 0, -3)))
print("-25 cm", scale_elongation.getValue(pb.Vector3d(0, 0, -25)))

for p in rs.getRootRandomParameter():
    p.f_se = scale_elongation  # set scale elongation function

rs.initialize()

ana = pb.SegmentAnalyser(rs)
anim = vp.AnimateRoots(ana)
anim.root_name = "creationTime"
anim.file = "example5b"
anim.min = np.array([-10, -10, -50])
anim.max = np.array([10, 10, 0.])
anim.res = np.array([1, 1, 1])
anim.start()

simtime = 60.
dt = 0.1  # small, for animation
for i in range(0, round(simtime / dt)):  # Simulation

    # update soil model (e.g. soil_strength)

    # update scales (e.g. from water content, soil_strength)
    scales = np.exp(-0.4 * soil_strength)  # (TODO)

    # copy scales into scaling funciton
    scale_elongation.data = scales

    rs.simulate(dt, False)

    ana = pb.SegmentAnalyser(rs)
    anim.rootsystem = ana
    anim.update()

rs.write("../results/example_5b.vtp")
vp.plot_roots(rs, "age")
```

Listing 19: Example 5b

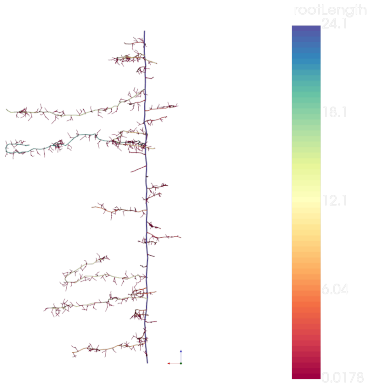


Figure 15: Root elongation rate

12-16 In L12 an EquidistantGrid1D is created, which is a specialisation from SoilLookUp (in soil.hh). It represents a 1D grid from 0 cm to -100 cm as soil with 100 layers. Additionally scalar data are attached to the grid, L13, L14 we create example soil strength data. From this data we calculate the elongation scales (L15), and set it as grid data (L16).

L17, L18 Retrieve the elongation scale data at two points. The data is given per layer, no data interpolation is performed.

20, 21 Sets the elongation scaling function to all root types.

25-32 Subsection 2.4 explains how to make an animation in Paraview. Here we use another (slower) approach to create the animation (and a preview) directly in vtk. For this we create an AnimateRoots object (L26), choose the domain size, and start the rendering (L32).

34-50 Simulation loop. If soil strength changes, the elongation scales must be calculated again (L32), and attached to the grid (L35). The frames of the animations are written in L50. While the approach is convenient, it is rather slow since for each frame a SegmentAnalyser object is created and rendered from scratch which is not feasible for large root systems.

52,53 Exports results as vtp, and creates a vtk plot. The effect of the dense layer can hardy be seen in the results (between -10 and -15 cm depth, laterals will be shorter). But the created animation reveals the effect (see exported video example5b.ogv).

4.2 Change of insertion angle

Nutrient concentration influences the angle between parent roots and laterals. Analog to Example 5a two compartments are created, and the insertion angle is scaled accordingly.

```
"""scales insertion angle"""
import sys; sys.path.append("../../..")
import numpy as np
import plantbox as pb
import vtk_plot as vp

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

# box with a left and a right compartment for analysis
sideBox = pb.SDF_PlantBox(10, 20, 50)
left = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(-4.99, 0, 0))
right = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(4.99, 0, 0))
leftright = pb.SDF_Union(left, right)
rs.setGeometry(leftright)

# left compartment has a minimum of 0.01, 1 elsewhere
maxS = 1.  # maximal
minS = 0.1  # minimal
slope = 1.  # [cm] linear gradient between min and max
leftC = pb.SDF_Complement(left)
soilprop = pb.SoilLookUpSDF(leftC, maxS, minS, slope)

# Manually set scaling function and tropism parameters
sigma = [0.4, 1., 1., 1., 1. ] * 2
for p in rs.getRootRandomParameter():
    if p.subType > 2:
        p.dx = 0.25  # adjust resolution
        p.f_sa = soilprop  # Scale insertion angle
        p.lmax = 2 * p.lmax  # make second order laterals longer

# simulation
rs.initialize()
simtime = 60.
dt = 1.
for i in range(0, round(simtime / dt)):
    rs.simulate(dt, False)

# analysis
al = pb.SegmentAnalyser(rs)
al.crop(left)
lm_theta = np.mean(al.getParameter("theta"))
ar = pb.SegmentAnalyser(rs)
ar.crop(right)
rm_theta = np.mean(ar.getParameter("theta"))
print('\nLeft  compartment mean insertion angle is {:g} degrees'.format
    (lm_theta))
print('\nRight compartment mean insertion angle is {:g} degrees\n'.
    format(rm_theta))

# write results
rs.write("results/example_5c.py")  # compartment geometry
rs.write("results/example_5c.vtp")  # root system

# plot, using vtk
vp.plot_roots(rs, "theta")  # press 'y'
```

Listing 20: Example 5c

27-32 Sets the insertion angle scaling functions to second order laterals only

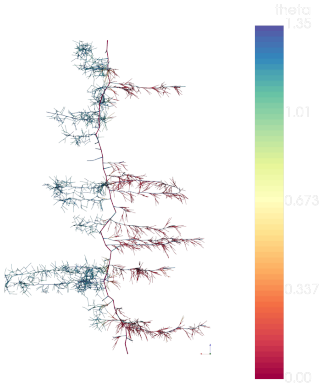


Figure 16: Insertion angle

(L29). L20 adjusts axial resolution and, L31 tortuosity *sigma*, and L32 sets the scale insertion angle function f_{sa} to the soil property. Additionally, the maximal length of second order roots is redoubled.

34-39 Initialization and simulation loop.

42-50 Analysis the root insertion angle in the left and right compartment.

53, 54 Writes the results for Paraview visualization.

57 A vtk simulation of root lengths. Press 'y' to obtain a x-z view of the root system to better see the effect (see Figure 16).

4.3 Change of lateral emergence probability

Soil properties can affect branching patterns. In the following example two compartments are created (analog to Example 5a), and the branching probability is modified in each of them.

```
"""scales branching probability"""
import sys; sys.path.append("../../..")
import math
import plantbox as pb
import vtk_plot as vp

rs = pb.RootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"
rs.readParameters(path + name + ".xml")

# box with a left and a right compartment for analysis
sideBox = pb.SDF_PlantBox(10, 20, 50)
left = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(-4.99, 0, 0))
right = pb.SDF_RotateTranslate(sideBox, pb.Vector3d(4.99, 0, 0))
leftright = pb.SDF_Union(left, right)
rs.setGeometry(leftright)

```

```
19 # left compartment has a minimum of 0.01, 1 elsewhere
20 maxS = 1.  # maximal
21 minS = 0.002  # minimal
22 slope = 1.  # [cm] linear gradient between min and max
23 leftC = pb.SDF_Complement(left)
24 soilprop = pb.SoilLookUpSDF(left, maxS, minS, slope)
25
26 # Manually set scaling function and tropism parameters
27 sigma = [0.4, 1., 1., 1., 1. ] * 2
28 for p in rs.getRootRandomParameter():
29     p.dx = 0.25  # adjust resolution
30     p.tropismS = sigma[p.subType - 1]
31
32 # 3. Scale branching probability
33 p = rs.getRootRandomParameter(2)
34 p.ln = p.ln / 5
35 p = rs.getRootRandomParameter(3)
36 p.f_sbp = soilprop
37
38 # simulation
39 rs.initialize()
40 simtime = 60.
41 dt = 1.
42 for i in range(0, round(simtime / dt)):
43     rs.simulate(dt, True)
44
45 # analysis
46 l = rs.getSummed("length")
47 al = pb.SegmentAnalyser(rs)
48 al.crop(left)
49 ll = al.getSummed("length")
50 ar = pb.SegmentAnalyser(rs)
51 ar.crop(right)
52 lr = ar.getSummed("length")
53 print('\nLeft  compartment total root length {:g} cm, {:g}%'.format(ll,
       100 * ll / l))
54 print('\nRight compartment total root length {:g} cm, {:g}% \n'.format(
       lr, 100 * lr / l))
55
56 # write results
57 rs.write("results/example_5d.py")  # compartment geometry
58 rs.write("results/example_5d.vtp")  # root system
59
60 # Plot, using vtk
61 vp.plot_roots(rs, "type")
```

Listing 21: Example 5d

27-30 Adjusts axial resolution and tortuosity.

33, 34 We adjust the inter lateral distances by making it smaller for a factor five.

35, 36 We set the branching probability scaling for the second order laterals. The scaling value means the probability that the branch occurs per day, i.e. 1 means the laterals always emerge (left compartment), or that they emerge with a chance of 0.2 % per day (right compartment).

46-54 Analysis the root insertion angle in the left and right compartment.

61 A vtk simulation of root lengths. Press 'y' to obtain a x-z view of the root system to better see the effect (see Figure 17).

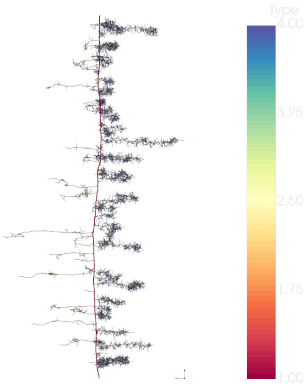


Figure 17: Branching density (probabilistic model)

Note that a scaling of zero means, that the laterals do never emerge, one means they always do. While the branching probability model is limited, it is easy to modify it to implement plant systemic responses. For this a suitable SoilLookUP must be defined and the method getValue(x, organ) must be overwritten, implementing the plant control of the branching probabilities.

5 Coupling to numerical models

CPlantBox is a bottom up model where no water movement or solute transport is calculated per default. The idea is to make it easy to link the root growth model to any numerical models or solvers. This is achieved by member functions that make it easy to (a) obtain a numerical grid from the root system, and (b) to offer utility functions to map root segments to an underlying soil grid.

In this section we consider a static root system. The additional steps that are needed for a developing root system (i.e. with a changing root geometry) are explained in the next section.

5.1 Mapping between root segments and an underlying soil

The classes MappedSegments and MappedRootSystem, defined in MappedOrganism.h, manage the coupling between the root system and external models or solvers.

The RootSystem class stores the nodes of each root in an object of class Root. This is convenient for simulating the root development, but not suitable for coupling to numerical models. Commonly, numerical models need the nodes as sequential list (obtained by RootSystem::getNodes) and the information on how they are connected (e.g. RootSystem::getSegments). Additionally, some information on the segments like radius or root type is needed. Furthermore, when coupling to a soil model, we need information in which soil cell the segment is located, and vice versa, which segments are located within a specific soil cell.

The class MappedSegments helps to manage all that, and offers the data structure for sequential nodes, connections between two nodes as segments, and the most commonly needed data such as creation time of the node, and radius or type per segment. Most important, it contains two hash maps seg2cell and cell2seg, linking segments to cells, and cells to list of segments.

MappedRootSystem is derived from MappedSegments and RootSystem, i.e. it can be used in the exact same way as the RootSystem class, but keeps track of the lists, and (optionally) the mapping between soil grid and root system.

To demonstrate basic functionality we will map a root system to a soil rectangular soil grid.

```
""" map root segments to a soil grid """
import sys; sys.path.append("../../..")
import numpy as np
import plantbox as pb
import vtk_plot as vp

""" root system """
rs = pb.MappedRootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"  # Zea_mays_1_Leitner_2010
rs.readParameters(path + name + ".xml")
rs.initialize()
rs.simulate(10., False)

""" soil """
min_ = np.array([-5, -5, -15])
max_ = np.array([9, 4, -5])
res_ = np.array([3, 1, 5])
```

```
19 rs.setRectangularGrid(pb.Vector3d(min_), pb.Vector3d(max_), pb.Vector3d
       (res_), True)  # cut and map segments
20
21 """ add segment indices """
22 segs = rs.segments
23 x = np.zeros(len(segs))
24 for i, s in enumerate(segs):
25     try:
26         x[i] = rs.seg2cell[i]
27     except:  # in case the segment is not within the domain
28         x[i] = -1
29
30 """ infos on a specific cell"""
31 ci = rs.soil_index(0, 0, -7)
32 print("Cell at [0,0,-7] has index", ci)
33 try:
34     print(len(rs.cell2seg[ci]), "segments in this cell:")
35     print(rs.cell2seg[ci])
36 except:
37     print("There are no segments in this cell")
38
39 """ vizualise roots """
40 # ana = pb.SegmentAnalyser(rs)  #<---- wrong!
41 ana = pb.SegmentAnalyser(rs.mappedSegments())
42 ana.addData("linear_index", x)
43 pd = vp.segs_to_polydata(ana, 1., ["radius", "subType", "creationTime",
       "linear_index"])
44 rootActor, rootCBar = vp.plot_roots(pd, "linear_index", "segment index
       plot", False)
45
46 """  vizualise soil  """
47 grid = vp.uniform_grid(min_, max_, res_)  # for visualization
48 meshActor, meshCBar = vp.plot_mesh(grid, "", "", False)
49 vp.render_window([meshActor, rootActor], "Test mapping", rootCBar, grid
       .GetBounds()).Start()
```

Listing 22: Example 6a

8-12 Creation of a small root system. Instead of the class RootSystem, MappedRootSystem is used (L8). MappedRootSystem is a specialisation of the 'normal' RootSystem class and can be used in the exactly same way.

16-19 We choose a small soil domain, where some roots are not inside. Calling MappedRootSystem::setRectangularGrid first cuts segments at the cell faces, and then maps the resulting segments to a soil index (creates the maps MappedRootSystem::seg2cell, and MappedRootSystem::cell2seg). The value True indicates that cutting is performed, False just maps the segment mid points without cutting.

22-28 Next we create an array x containing the soil indices for each segment, that will be later used for vizualisation. We use the hash map (a Python dictionary) MappedRootSystem::seg2cell to obtain the linear index of the cell, where the segment's mid point is located.

31-37 To demonstrate how to retrieve all segments that lie within a given cell, we output the segments at the cell located around the position [0,0,-7]. In L31 we retrieve the cell index for that position. L34 and L35 print out the segment indices. Note that the map will have no entry for a given cell, if no segments are located in the cell.

43-48 To visualize the soil cell indices, we first create a SegmentAnalyser class. The class MappedRootSystem is derived from a RootSystem and MappedSegments. If we create a SegmentAnalyser class directly from rs (L40) MappedRootSystem will be considered as RootSystem, and the resulting class will contain the original segments before cutting at the cell boundaries. The function MappedRootSystem::mappedSegments (L41) will return a reference to rs but with the type MappedSegments. In this way the SegmentAnalyser class will be created including the cut segments. Next, in L42 we add the indices as data to the SegmentAnalyser class. L43, L44 produces the VTK plot, but rendering is started at a later point, since we also want to visualize the underlying soil grid.

51-53 L47 creates the mesh for vizualisation, L48 creates the VTK plot, and L49 starts the rendering window, rendering both, root system and underlying soil. Press 'y', 'x', and 'z' to obtain axis aligned views of the root system and soil grid.

More, generally, when coupling to an external solver like DuMux (Koch et al., 2020), we need to set the soil_index function, that returns the cell index for a certain position. Additionally, periodic soil domains are already implemented. Both coupling to DuMux and periodicity will be demonstrated in Section 6.2.

Coupling to an unstructured grid is as simple as to a structured one. But automatic cutting of segments at the cell faces is currently not implemented. That means that axial resolution should be small in order to keep the introduced error small.

5.2 Water movement within the roots

Since water movement within the roots is often needed, it is implemented directly in CPlantBox (in the class XylemFlux) following Meunier et al. (2017). However, usage is optional, and any other transport code can be used (e.g. if solutes are considered). XylemFlux sets up the linear system, and the sparse linear system is then solved in Python using scipy (using the class XylemFluxPython defined in xylem_flux.py).

The following example is based on benchmark M3.1 (Schnepf et al., 2019) with constant conductivities, but not with a given root system, but a simulated one.

```
""" water movement within the root (static soil) """
import sys; sys.path.append("../../..")
from xylem_flux import XylemFluxPython  # Python hybrid solver
import plantbox as pb
import vtk_plot as vp

import numpy as np
import matplotlib.pyplot as plt

""" Parameters """
kz = 4.32e-2  # axial conductivity [cm^3/day]
kr = 1.728e-4  # radial conductivity [1/day]
p_s = -200  # static soil pressure [cm]
p0 = -500  # dircichlet bc at top
simtime = 14  # [day] for task b

```

```
17 """ root system """
18 rs = pb.MappedRootSystem()
19 path = "../../../modelparameter/rootsystem/"
20 name = "Anagallis_femina_Leitner_2010"  # Zea_mays_1_Leitner_2010
21 rs.readParameters(path + name + ".xml")
22 rs.initialize()
23 rs.simulate(simtime, False)
24
25 """ root problem """
26 r = XylemFluxPython(rs)
27 r.setKr([kr])
28 r.setKx([kz])
29 nodes = r.get_nodes()
30 soil_index = lambda x, y, z : 0
31 r.rs.setSoilGrid(soil_index)
32
33 """ Numerical solution """
34 rx = r.solve_dirichlet(0., p0, p_s, [p_s], True)
35 fluxes = r.segFluxes(simtime, rx, -200 * np.ones(rx.shape), False)  #
       cm3/day
36 print("Transpiration", r.collar_flux(simtime, rx, [p_s]), "cm3/day")
37
38 """ plot results """
39 plt.plot(rx, nodes[:, 2] , "r*")
40 plt.xlabel("Xylem pressure (cm)")
41 plt.ylabel("Depth (m)")
42 plt.title("Xylem matric potential (cm)")
43 plt.show()
44
45 """ Additional vtk plot """
46 ana = pb.SegmentAnalyser(r.rs)
47 ana.addData("rx", rx)
48 ana.addData("fluxes", np.maximum(fluxes, -1.e-3))  # cut off for
       vizualisation
49 vp.plot_roots(ana, "rx", "Xylem matric potential (cm)")  # "fluxes"
```

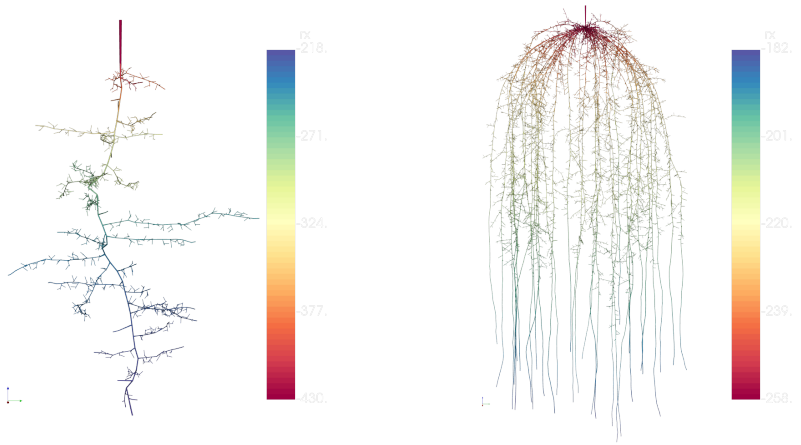
Listing 23: Example 6b

11-15 All parameters that will be used later on.

18-23 The root system (similar to last section).

26-31 The MappedRootSystem is wrapped with the XylemFluxPython class, which extends the XylemFlux class, which computes the xylem matric flux potential. L27, L28 sets the radial and axial conductivity. L29 retrieves the root system nodes for later visualisation. The pressure surrounding the the root system is either defined as pressure surrounding each root segment, or as soil cells, in which the root segments are located. L30, L31 sets a soil containing of one single cell with index 0.

34-36 XylemFluxPython defines solvers like solve_dirichlet for Dirichlet boundary condition (predefined collar matric potential), solve_neumann for Neumann boundary condition (predefined collar flux), and solve which switches between Dirichlet and Neumann at some critical pressure (the plant wilting point). The arguments of solve_dirichlet are the simulation time (to calculate age dependent conductivities), the root collar pressure head, the pressure around the root collar, the soil matric potential around the root segments or per soil cell, and a boolean value that decides if the



(a) Anagallis (b) Maize

Figure 18: Calculated xylem matric potential (cm)

potentials are given per soil cell (True) or per segments (False). The return value rx contains the xylem matric potentials per segment. L35 calculates the fluxes into the soil (negative values mean into the root). The bool argument determines if we approximate the flux, or use the exact solution by Meunier et al. (2017). L36 determines the root collar flux.

39-43 Plots the results.

46-49 Creates the VTK plot, adding the soil matric potentials and fluxes. L49 picks either rx or $fluxes$ for vizualisation, see Figure 18a and 18b.

It is possible to set conductivties per root type (see XylemFlux::setKr, and setKx), and root age dependent conductivities per root type using linear lookup tables (see XylemFlux::setKrTables, and setKxTables).

5.3 Coupling a static root system to DuMux

Putting the last two sections together, we can easily set up an example, using the classic sink term in the soil model, similar as in (Leitner et al., 2014) based on (Doussan et al., 1998). For solving the Richards equation we use DuMux (Koch et al., 2020).

The next example mimics benchmark C12 (Schnepf et al., 2019), but with a static simulated root system. The example must be run out of dumux-rosi (located in dumux-rosi/rosi_benchmarking/python_solver/coupled), otherwise the DuMux Python coupling is not available.

```
""" coupling with DuMux as solver for the soil, run in dumux-rosi """
import sys
sys.path.append("../../../build-cmake/rosi_benchmarking/python_solver/")
sys.path.append("../solvers/")  # for pure python solvers
from rosi_richards import RichardsSP  # C++ part (Dumux binding)
from richards import RichardsWrapper  # Python part
from xylem_flux import XylemFluxPython  # Python hybrid solver
```

```
from root_conductivities import *  # hard coded conductivities
import plantbox as pb
import vtk_plot as vp

import numpy as np
import matplotlib.pyplot as plt
import timeit


def sinusoidal(t):
    return np.sin(2. * np.pi * np.array(t) - 0.5 * np.pi) + 1.


""" Parameters """
min_b = [-4., -4., -25.]
max_b = [4., 4., 0.]
cell_number = [8, 8, 25]  # [16, 16, 30]  # [32, 32, 60]
periodic = True

path = "../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"  # Zea_mays_1_Leitner_2010
loam = [0.08, 0.43, 0.04, 1.6, 50]
initial = -659.8 + 12.5  # -659.8

trans = 6.4  # cm3 /day (sinusoidal)
wilting_point = -15000  # cm

sim_time = 7  # [day] for task b
rs_age = 10  # root system initial age
age_dependent = False  # conductivities
dt = 120. / (24 * 3600)  # [days] Time step must be very small

""" Initialize macroscopic soil model """
s = RichardsWrapper(RichardsSP())
s.initialize()
s.createGrid(min_b, max_b, cell_number, periodic)  # [cm]
s.setHomogeneousIC(initial, True)  # cm pressure head, equilibrium
s.setTopBC("noFlux")
s.setBotBC("noFlux")
s.setVGParameters([loam])
s.initializeProblem()
s.setCriticalPressure(wilting_point)

""" Initialize xylem model """
rs = pb.MappedRootSystem()
rs.readParameters(path + name + ".xml")
if not periodic:
    sdf = pb.SDF_PlantBox(0.99 * (max_b[0] - min_b[0]), 0.99 * (max_b
    [1] - min_b[1]), max_b[2] - min_b[2])
else :
    sdf = pb.SDF_PlantBox(np.Inf, np.Inf, max_b[2] - min_b[2])
rs.setGeometry(sdf)
rs.initialize()
rs.simulate(rs_age, False)
r = XylemFluxPython(rs)
init_conductivities(r, age_dependent)

""" Coupling (map indices) """
picker = lambda x, y, z : s.pick([x, y, z])
r.rs.setSoilGrid(picker)  # maps segments
r.rs.setRectangularGrid(pb.Vector3d(min_b), pb.Vector3d(max_b), pb.
    Vector3d(cell_number), True)
```

```
68 r.test()  # sanity checks
69 nodes = r.get_nodes()
70 cci = picker(nodes[0, 0], nodes[0, 1], nodes[0, 2])  # collar cell
       index
71
72 """ Numerical solution """
73 start_time = timeit.default_timer()
74 x_, y_ = [], []
75 sx = s.getSolutionHead()  # inital condition, solverbase.py
76 N = round(sim_time / dt)
77 t = 0.
78
79 for i in range(0, N):
80
81     rx = r.solve(rs_age + t, -trans * sinusoidal(t), sx[cci], sx, True,
         wilting_point)  # xylem_flux.py
82     x_.append(t)
83     y_.append(float(r.collar_flux(rs_age + t, rx, sx)))
84
85     fluxes = r.soilFluxes(rs_age + t, rx, sx, False)
86     s.setSource(fluxes)  # richards.py
87     s.solve(dt)
88     sx = s.getSolutionHead()  # richards.py
89
90     min_sx, min_rx, max_sx, max_rx = np.min(sx), np.min(rx), np.max(sx)
       , np.max(rx)
91     n = round(float(i) / float(N) * 100.)
92     print("[" + ''.join(["*"]) * n + ''.join([" "]) * (100 - n) + "],
       [{:g}, {:g}] cm soil [{:g}, {:g}] cm root at {:g} days {:g}"
93           .format(min_sx, max_sx, min_rx, max_rx, s.simTime, rx[0]))
94     t += dt
95
96 print ("Coupled benchmark solved in ", timeit.default_timer() -
       start_time, " s")
97
98 """ VTK visualisation """
99 vp.plot_roots_and_soil(r.rs, "pressure head", rx, s, periodic, np.array
       (min_b), np.array(max_b), cell_number, name)
100
101 """ transpiration over time """
102 fig, ax1 = plt.subplots()
103 ax1.plot(x_, trans * sinusoidal(x_), 'k')  # potential
104 ax1.plot(x_, -np.array(y_), 'g')  # actual
105 ax2 = ax1.twinx()
106 ax2.plot(x_, np.cumsum(-np.array(y_) * dt), 'c--')  # cumulative
107 ax1.set_xlabel("Time [d]")
108 ax1.set_ylabel("Transpiration $[cm^3 d^{-1}]$")
109 ax1.legend(['Potential', 'Actual', 'Cumulative'], loc = 'upper left')
110 np.savetxt(name, np.vstack((x_, -np.array(y_))), delimiter = ';')
111 plt.show()
```

Listing 24: Example 6c

3,4 Add paths for DuMux Python coupling (L3) and Python solvers (L4).

5,6 The direct C++ part of the DuMux binding, and the Python wrapper class.

17,18 Defines a sinusoidal function for the collar boundary condition.

21-37 All parameters that are needed for this simulation. L25 decides if periodic boundary conditions are used, or not. L35 states the simulation time,

L36 the initial root system age. L37 defines if age dependent root conductivities are used. The rooot conductivities are hard coded in the file root_conductivities.py, that is imported in L8. Age dependent conductivities can be used to mimic root growth in a predefined way, i.e. the root system is already fully grown, but the radial conductivities are turned on during the simulation.

41-49 Sets up the soil solver (DuMux Python binding from dumux-rosi).

52-62 Sets up the Xylem model as in Subsection 5.2. If the root system is not periodic, a confining geometry is set L54-L58. L62 passes the axial and radial conductivities to the XylemFluxPython object (the function is defined in root_conductivities.py).

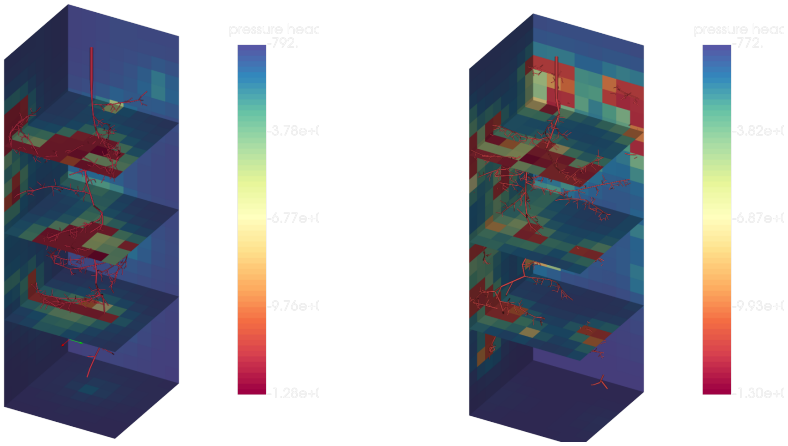
65-70 Coupling between the soil soil and root part is performed by setting the picking function that assigns a cell index to each spatial position L65, and L66. In L67 root segments are cut to the rectangular grid, and in L70 the cell index of the root collar is determined.

73-77 Initializes the simulation, initially the soil values are the same as the initial conditions (L75).

79-94 First, we calculate the xylem matric potential rx for a given soil matric potential sx (L81) and save the actual collar flux for later analysis (L83). Next, we calculate teh sink (L85) and apply it to the soil model (L86). The soil model is simulated (L87) and the resulting matric potential sx is updated (L88). The simulation takes some time (around 15 minutes), and L90-93 print debugging information and a progress bar. L94 increases the current simulation time. This is needed if age dependent conductivities are used.

99-111 L99 creates Figure 19 and, L102-111 plots uptake and cumulative uptake over time, see Figure 20.

With above code we can compare total root system water uptake in a container, with the uptake if periodic boundary conditions are used. While the first scenario reflects the situation in a plant pot, periodic boundary conditions reflect the situation when the plant grows in the field, where the planting distance is approximately the domain size. Figure 20 shows both situations, showing that the boundary conditions will strongly affect total water uptake, because roots are less evenly distributed in the pot scenario and water redistribution is impeded by the pot boundaries.



(a) Confined (b) Periodic boundary condtions

Figure 19: Water depletion due to root uptake after one week.

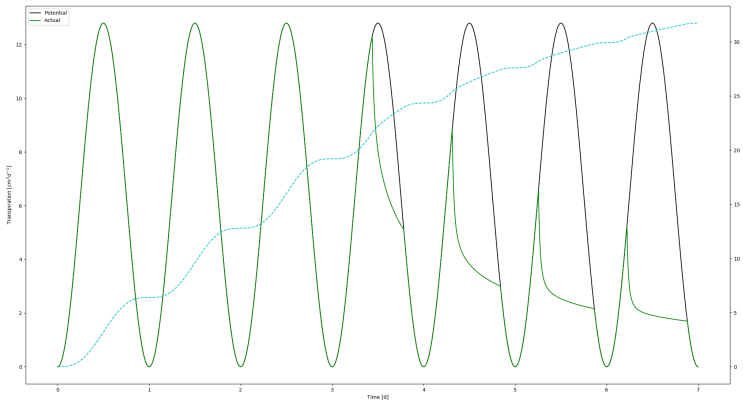
6 Coupling growing rootsystems to numerical models

After coupling a static root system to a soil model, it is straight forward to couple a growing root system. All there is to do is to update the geometry and the mapping between the grids in every time step. All the work is done by MappedRootSystem.

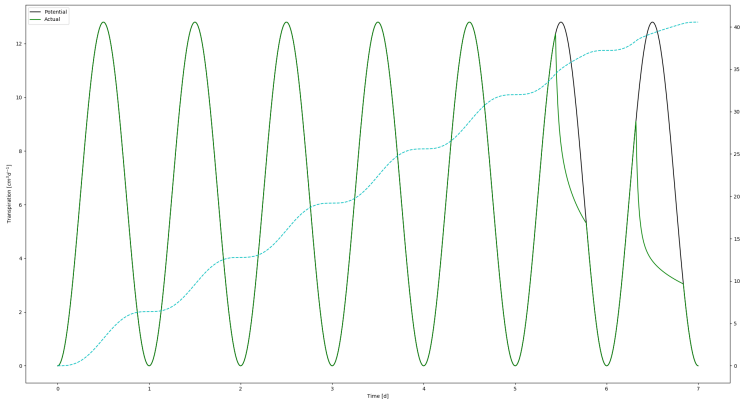
6.1 Mapping of growing roots and underlying soil

In the following we show how the mappings between root system grid and soil grids are updated (see Section 5.1 for the static case). For demonstration we create an animation, where we can see the growth and the dynamic mapping.

```
""" map root segments to a soil grid """
import sys; sys.path.append("../../..")
import numpy as np
import plantbox as pb
import vtk_plot as vp

""" parameters """
sim_time = 14  # [day]
rs_age = 1  # initial age
age_dependent = False  # conductivities
dt = 0.1  # [days] Time step must be very small
periodic = False

""" root system """
rs = pb.MappedRootSystem()
path = "../../../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"  # Zea_mays_1_Leitner_2010
rs.readParameters(path + name + ".xml")

""" soil """
min_ = np.array([-5, -5, -15])
```



(a) Confined



(b) Periodic boundary condtions

Figure 20: Water uptake and cumulated uptake

```
22 max_ = np.array([5, 5, 0.])
23 res_ = np.array([1, 3, 5])
24 if not periodic:
25     sdf = pb.SDF_PlantBox(0.99 * (max_[0] - min_[0]), 0.99 * (max_[1] -
        min_[1]), 0.99 * (max_[2] - min_[2]))
26     rs.setGeometry(sdf)
27 rs.setRectangularGrid(pb.Vector3d(min_), pb.Vector3d(max_), pb.Vector3d
       (res_), False)  # cut and map segments
28
29 rs.initialize()
30 rs.simulate(rs_age, False)
31 N = round(sim_time / dt)
32
33 ana = pb.SegmentAnalyser(rs.mappedSegments())
34 anim = vp.AnimateRoots(ana)
35 anim.min = min_
36 anim.max = max_
37 anim.res = res_
38 anim.file = "results/example7a"
39 anim.start()
40
41 for i in range(0, N):
42
43     rs.simulate(dt, False)
44
45     """ add segment indices """
46     segs = rs.segments
47     x = np.zeros(len(segs))
48     for i, s in enumerate(segs):
49         try:
50             x[i] = rs.seg2cell[i]
51         except:  # in case the segment is not within the domain
52             x[i] = -10
53
54     ana = pb.SegmentAnalyser(rs.mappedSegments())
55     ana.addData("linear_index", x)
56
57     anim.rootsystem = ana
58     anim.root_name = "linear_index"
59     anim.update()
```

Listing 25: Example 7a

8-12 Parameters we might want to modify.

15-18 Initializes the model.

21-27 Defines a coarse soil grid. If we do not use periodicity we set the domain as confining geometry to the root growth. L27 sets the underlying soil grid.

29-31 Initializes the simulation with an initial simulation run for rs_age.

33-39 Initializes an VTK animation using the class vp.AnimateRoots, which is work in progress.

41-59 The simulation loop: L43 performs the simulation, and updates the mappers (no additional steps are needed, everything is updated by pb.MappedRootSystem). L46-52 determines the cell index for each segment for visualization. L54,L L55 makes a SegmentAnalyser object and adds the

soil cell indices. L57-L58 updates the animation figure. This is convenient for debugging, and the object vp.AnimateRoots will create an ogg vorbis movie file (which is small and high quality), but for bigger root systems this will be very slow, since a SegmentAnalyser object is created and plotted for each frame (see Section 2.4 for a faster method).

6.2 Coupling a dynamic root system to DuMux with soil feedback

We modify Example 6.2. Using the class MappedRootSystem (as before) it is sufficient to call MappedRootSystem::simulate() to implement real root growth, i.e. to add rs.simulate(dt) in L80. The modified segments and the mapping of new segments is automatically managed (see example7b_coupling.py). Figure 21 shows the root actual uptake over time, with a slightly altered shape compared to 19. Note that the jumps in actual uptake are due to emerging root segments.

In this example soil state does not affect root system growth in any way. We need to add the processes we are interested in (see Section 3 and 4). For demonstration how to do that, we demonstrate the implementation steps using hydrotropim. Note that the other interactions from Section 4 could be implemented in the same way.

In order to use hydrotropism, we need to define a SoilLookUp that accesses the dynamic soil data. We present two approaches: the first uses nearest neighbour interpolation, which is fast but a coarse approximation. The second uses linear interpolation which is more exact, but slower.

```
""" coupling with DuMux as solver for the soil part """
import sys
from builtins import isinstance
sys.path.append("../../../build-cmake/rosi_benchmarking/python_solver/")
sys.path.append("../solvers/")  # for pure python solvers
from rosi_richards import RichardsSP  # C++ part (Dumux binding)
from richards import RichardsWrapper  # Python part
from xylem_flux import XylemFluxPython  # Python hybrid solver
from root_conductivities import *  # hard coded conductivities
import plantbox as pb
import vtk_plot as vp

import numpy as np
import matplotlib.pyplot as plt
import timeit


class SoilNN(pb.SoilLookUp):

    def __init__(self, soil):
        super(SoilNN, self).__init__()
        self.soil = soil

    def getValue(self, pos, organ):
        return self.soil.getSolutionAt(self.soil.pick([pos.x, pos.y, pos.z]))


class SoilLinear(pb.SoilLookUp):

```

```
    def __init__(self, soil):
        super(SoilLinear, self).__init__()
        self.soil = soil
        self.points = self.soil.getDofCoordinates() / 100.
        self.update()

    def update(self):
        self.sol = self.soil.getSolution()

    def getValue(self, pos, organ):
        p = np.expand_dims(np.array(pos), axis = 0)  # make 1x3
        return self.soil.interpolate_(p, self.points, self.sol)


def sinusoidal(t):
    return np.sin(2. * np.pi * np.array(t) - 0.5 * np.pi) + 1.


""" Parameters """
min_b = [-4., -4., -25.]
max_b = [4., 4., 0.]
cell_number = [8, 8, 25]  # [16, 16, 30]  # [32, 32, 60]
periodic = False

path = "../modelparameter/rootsystem/"
name = "Anagallis_femina_Leitner_2010"  # Zea_mays_1_Leitner_2010
loam = [0.08, 0.43, 0.04, 1.6, 50]
initial = -659.8 + 12.5  # -659.8

trans = 6.4  # cm3 /day (sinusoidal)
wilting_point = -15000  # cm

sim_time = 7  # [day] for task b
rs_age = 3  # root system initial age
age_dependent = False  # conductivities
dt = 120. / (24 * 3600)  # [days] Time step must be very small

""" Initialize macroscopic soil model """
s = RichardsWrapper(RichardsSP())
s.initialize()
s.createGrid(min_b, max_b, cell_number, periodic)  # [cm]
s.setHomogeneousIC(initial, True)  # cm pressure head, equilibrium
s.setTopBC("noFlux")
s.setBotBC("noFlux")
s.setVGParameters([loam])
s.initializeProblem()
s.setCriticalPressure(wilting_point)

""" Initialize xylem model """
rs = pb.MappedRootSystem()
rs.readParameters(path + name + ".xml")
if not periodic:
    sdf = pb.SDF_PlantBox(0.99 * (max_b[0] - min_b[0]), 0.99 * (max_b
    [1] - min_b[1]), max_b[2] - min_b[2])
else :
    sdf = pb.SDF_PlantBox(np.Inf, np.Inf, max_b[2] - min_b[2])
rs.setGeometry(sdf)
r = XylemFluxPython(rs)
init_conductivities(r, age_dependent)

""" Coupling (map indices) """
picker = lambda x, y, z : s.pick([x, y, z])
```

```
r.rs.setSoilGrid(picker)  # maps segments
r.rs.setRectangularGrid(pb.Vector3d(min_b), pb.Vector3d(max_b), pb.Vector3d(cell_number), True)

# Manually set tropism to hydrotropism for the first ten root types
sigma = [0.4, 1., 1., 1., 1. ] * 2
for p in rs.getRootRandomParameter():
        p.dx = 0.25  # adjust resolution
        p.tropismT = pb.TropismType.hydro
        p.tropismN = 2  # strength of tropism
        p.tropismS = sigma[p.subType - 1]

soil = SoilNN(s)  # SoilLinear(s)
rs.setSoil(soil)

rs.initialize()
rs.simulate(rs_age, False)
r.test()  # sanity checks
nodes = r.get_nodes()
cci = picker(nodes[0, 0], nodes[0, 1], nodes[0, 2])  # cell index

""" Numerical solution """
start_time = timeit.default_timer()
x_, y_ = [], []
sx = s.getSolutionHead()  # inital condition, solverbase.py
N = round(sim_time / dt)
t = 0.

for i in range(0, N):

    if isinstance(soil, SoilLinear):
        soil.update()  # for hydrotropism look up
    rs.simulate(dt)

    rx = r.solve(rs_age + t, -trans * sinusoidal(t), sx[cci], sx, True, wilting_point)  # xylem_flux.py
    x_.append(t)
    y_.append(float(r.collar_flux(rs_age + t, rx, sx)))

    fluxes = r.soilFluxes(rs_age + t, rx, sx, False)
    s.setSource(fluxes)  # richards.py
    s.solve(dt)
    sx = s.getSolutionHead()  # richards.py

    min_sx, min_rx, max_sx, max_rx = np.min(sx), np.min(rx), np.max(sx), np.max(rx)
    n = round(float(i) / float(N) * 100.)
    print("[" + ''.join(["*"]) * n + ''.join([" "]) * (100 - n) + "], [{:g}, {:g}] cm soil [{:g}, {:g}] cm root at {:g} days {:g}"
          .format(min_sx, max_sx, min_rx, max_rx, s.simTime, rx[0]))
    t += dt

print ("Coupled benchmark solved in ", timeit.default_timer() - start_time, " s")

""" VTK visualisation """
vp.plot_roots_and_soil(r.rs, "pressure head", rx, s, periodic, np.array(min_b), np.array(max_b), cell_number, name)

""" transpiration over time """
fig, ax1 = plt.subplots()
ax1.plot(x_, trans * sinusoidal(x_), 'k')  # potential transpiration
```

```
147 ax1.plot(x_, -np.array(y_), 'g')  # actual transpiration (neumann)
148 ax2 = ax1.twinx()
149 ax2.plot(x_, np.cumsum(-np.array(y_) * dt), 'c--')  # cumulative
        transpiration (neumann)
150 ax1.set_xlabel("Time [d]")
151 ax1.set_ylabel("Transpiration $[cm^3 d^{-1}]$")
152 ax1.legend(['Potential', 'Actual', 'Cumulative'], loc = 'upper left')
153 np.savetxt(name, np.vstack((x_, -np.array(y_))), delimiter = ';')
154 plt.show()
```

Listing 26: Example 7c

We only describe the changes to Section :

18-25 We first create a class for nearest neighbour interpolation that extends pb.SoilLookUp. L22 stores a reference to the DuMux soil model. For specialisation of pb.SoilLookUp we need to overwrite the getValue function (L24-25). First we retrieve the cell index by calling *pick*. And, then we return the solution at this cell (in Pascal). Note that the solution is the same for each point within a cell.

28-41 Secondly, we want to use linear interpolation, which is slower, but more exact. The constructor stores a reference to the soil model (L30-34) the coordinates of the degrees of freedom (i.e. where the points where numerical solution is defined) and calls update(), which fetches the current model solution. Update will be called in after each solving step in the simulation loop. L39-L41 performs the linear interpolation. This is rather slow because the method assumes an unstructured grid, and is not optimized for a structured rectangular grid.

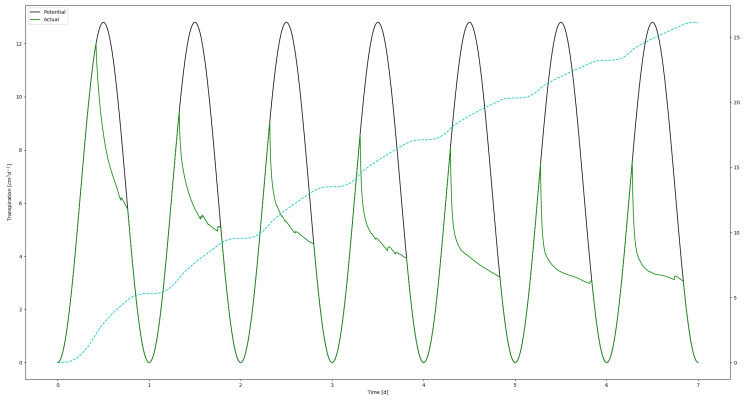
94-100 We set tropism to hydrotropism (L98) and modify the tropism parameters (L99,100)

102,103 We set the soil, using nearest neighbour or linear interpolation.

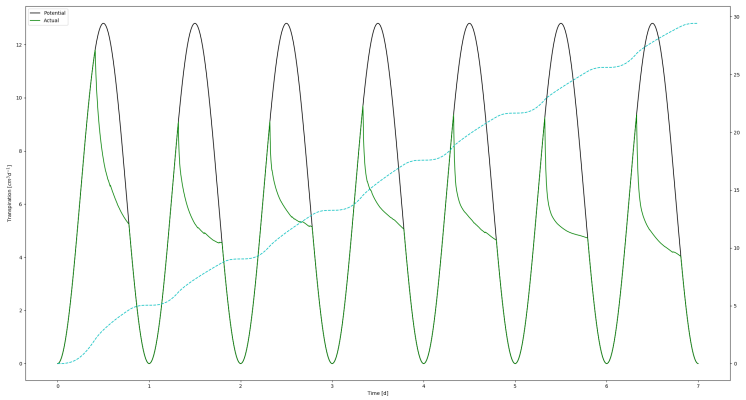
120,121 If we use linear interpolation we need to call update to copy the current solution.

122 Root system simulation, updates geometry and mappers.

Figure 21 shows that hydrotropism will lead to increased cumulative uptake, because the roots are more evenly distributed. Note that there is a big variation in the results, and for a more profound analysis we would have to make many simulation runs.



(a) Gravi- and Plagiotropism



(b) Hydrotropism

Figure 21: Water uptake a in plant pot

7 Estimating the hydraulic conductivity drop in the rhizosphere

The resolution of the soil grid is often too coarse to catch small-scale rhizosphere gradients in hydraulic conductivity near the roots. The consequence is that root water uptake is overestimated. One solution would be (adaptive) grid refinement, however at the cost of computation time and the validity of the line-source assumption that we take for the roots in our setup. An alternative solution is to keep the coarse soil grid resolution and estimate the soil water potential at the root-soil interface. We implemented two ways of doing so within dumux-rosi. Firstly, we split the volume of each soil control element between the root segments that lie inside it, and assign a soil cylinder around each root segment. In each soil cylinder, the 1D radially symmetric Richards equation is solved and coupled to the macroscopic scale in a mass conservative way via the outer boundary conditions of the soil cylinders.

7.1 The steady-rate approximation

The second approach is described in this example and follows the approach described in Schröder et al. (2008). Each root segment inside a soil control element of volume V_s has an assigned "rhizosphere soil volume" equal to

$$V_{rhizo} = kV_s,$$

where the weighting factor k is the fraction of the root segment volume V_{rs} relative to the overall volume of roots V_{rst} inside V_s, $k = \frac{V_{rs}}{Vrst}$. From this, the outer radius of the rhizosphere cylinder is computed by

$$r_{out} = \sqrt{\frac{V_{rhizo} + V_{rs}}{\pi l}},$$

where l is the root segment length.

Then the flux at the root-soil interface is estimated based on the analytical solutions of the 1D radially symmetric Richards equation. Based on the steady-rate assumption and using the matric flux potential $\Phi(h_c) = \int_{-\infty}^{h_c} K(h)dh$ that linearizes the Richards equation, the radial matric flux potential profiles for non-stressed and stressed conditions are given by

$$\Phi(r) = \Phi_{r_{out}} + (q_{root}r_{root} - q_{out}r_{out}) \left[\frac{r^2/r_{root}^2}{2(1-\rho^2)} + \frac{\rho^2}{1-\rho^2} \left(\ln \frac{r_{out}}{r} - \frac{1}{2} \right) \right] + q_{out}r_{out} \ln \frac{r}{r_{out}} \quad (1)$$

and

$$\Phi(r) = \left(\Phi_{r_{out}} - \Phi_{r_{root}} + q_{out}r_{out} \ln \frac{1}{\rho} \right) \frac{r^2/r_{root}^2 - 1 + 2\rho^2 \ln r_{root}/r}{\rho^2 - 1 + 2\rho^2 ln 1/\rho} + q_{out}r_{out} \ln \frac{r}{r_{root}} + \Phi_{root}, \quad (2)$$

where $\rho = \frac{r_{out}}{r_{root}}$, r_{root} is the root radius, r_{out} is the outer radius, q_{root} is the flux prescribed at the root-soil interface, q_{out} is the flux at the outer boundary.

Given the soil matric potential at the outer boundary, and either the flux (non-stressed conditions) or soil matric potential (stressed) at the root-soil interface, the solution computes the radial matrix flux potential profile, and from the matric flux potential, the soil matric potential can be inferred. The soil matric potential at the outer boundary is taken to be the macroscopic soil matric potential value of the given soil control element at the given time step. The water flux at the outer boundary, q_{out}, is computed such that the sum of the fluxes at the outer boundaries of the rhizosphere cylinders is equal to the net flux into or out of the given soil control element, q_{V_s}, at a given time step, and

$$q_{out} = kq_{V_s}$$

, where k is the same weighting factor as described above. However, setting the outer flux to zero is sufficient in most cases, as there is redistribution after every time step.

We start the simulation by computing the root water uptake according to the classical (see section 5). Then, for every root segment and at every time step, we check whether the matric flux potential at the root-soil interface is positive or negative. In case the matric flux potential is positive, root water uptake follows non-stressed conditions, and the sink term is left according to the classical approach. In case the matric flux potential is negative, root water uptake follows stressed conditions and the soil matric potential at the root-soil interface is equal to the wilting point. The water flux from soil into the root is computed by numerically approximating the soil matric potential gradient at the root-soil interface and multiplying this by the hydraulic conductivity.

Finally, the flux at the root-soil interface is the maximum of the classical and steady-rate approximation value. This makes sure that hydraulic redistribution (i.e. flow from roots to soil) is still possible.

Listing X shows the code of file "coupled$_c$12$_s$*chroeder.py*".

106-110 Root xylem potentials are computed based on the current soil (at first call) or rhizosphere (from second call onwards) water potentials.

112 Rhizosphere soil water potentials are updated from the current soil and xylem water potentials using the steady-rate assumption.

113 For each root segment, the water fluxes are computed from the current xylem and rhizosphere water potentials.

115 For the soil control element, sum of root water uptake over all root segments inside a given soil control element is computed.

127 The fluxes are set as sources in the soil problem (Richards equation)

131 Soil problem is solved for one time step.

Listing Y shows the C++ code for the computation of matric potential at the root-soil interface according to Schröder. The fluxes are then computed according to the pressure differences...

234 Computation of the matrix flux potential according to the non-stressed equation.

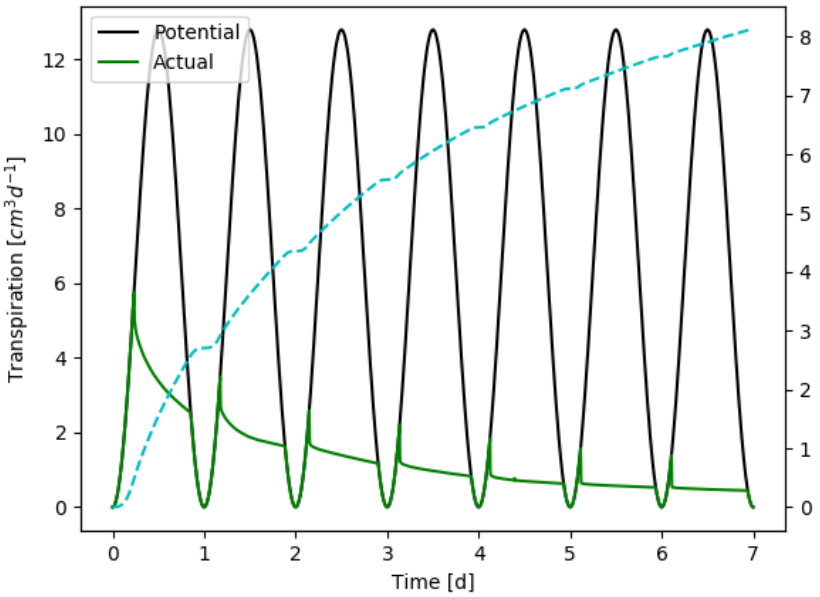


Figure 22: Transpiration rate and cumulative transpiration computed based on the steady-rate assumption.

237 For positive mfp (no stress), the soil matric potential at the root-soil interface is set to the corresponding value.

240 For negative mfp (stress), the soil matric potential at the root-soil interface is set equal to the wilting point.

245 If the xylem water potential is larger than the root-soil interface potential, root-soil interface potential is set equal to the xylem water potential (no flux).

248 On the macroscopic scale, flow of water from root to soil is enabled.

Figure 22 the transpiration rate and cumulative transpiration computed based on the steady-rate assumption for the case of a clay soil.

8 Todos

Topics that are not covered yet or should be improved

- The DuMux binding and MPI
- Elongation rate according to Moacir et al.
- Delay based, versus length based lateral emeragnce.
- Better branching probability example.
- Periodicity without DuMux coupling, but with a soil 3d grid (untested, and an example is missing)
- Carbon limited root system growth
- Direct vtk animation with vp.AnimateRoots (under development...)

References

Doussan, C., Pagès, L., and Vercambre, G. (1998). Modelling of the hydraulic architecture of root systems: an integrated approach to water absorption—model description. *Annals of botany*, 81(2):213–223.

Koch, T., Gläser, D., Weishaupt, K., Ackermann, S., Beck, M., Becker, B., Burbulla, S., Class, H., Coltman, E., Emmert, S., et al. (2020). Dumux 3–an open-source simulator for solving flow and transport problems in porous media with a focus on model coupling. *Computers & Mathematics with Applications*.

Leitner, D., Meunier, F., Bodner, G., Javaux, M., and Schnepf, A. (2014). Impact of contrasted maize root traits at flowering on water stress tolerance–a simulation study. *Field crops research*, 165:125–137.

Meunier, F., Draye, X., Vanderborght, J., Javaux, M., and Couvreur, V. (2017). A hybrid analytical-numerical method for solving water flow equations in root hydraulic architectures. *Applied Mathematical Modelling*, 52:648–663.

Schnepf, A., Black, C. K., Couvreur, V., Delory, B. M., Doussan, C., Koch, A., Koch, T., Javaux, M., Landl, M., Leitner, D., et al. (2019). Call for participation: Collaborative benchmarking of functional-structural root architecture models. the case of root water uptake. *BioRxiv*, page 808972.

Schnepf, A., Leitner, D., Landl, M., Lobet, G., Mai, T., Morandage, S., Sheng, V., Zörner, M., Vanderborght, J., and Vereecken, H. (2018). CRootBox: a structural–functional modelling framework for root systems. *Annals of botany*, 121(5):1033–1053.

Schröder, T., Javaux, M., Vanderborght, J., Körfgen, B., and Vereecken, H. (2008). Effect of local soil hydraulic conductivity drop using a three-dimensional root water uptake model. *Vadose Zone Journal*, 7(3):1089–1098.

Zhou, X., Schnepf, A., Vanderborght, J., Leitner, D., Lacointe, A., Vereecken, H., and Lobet, G. (2020). CPlantBox, a whole-plant modelling framework for the simulation of water-and carbon-related processes. *in silico Plants*, 2(1).