CPlantBox Tutorial

Daniel Leitner, Andrea Schnepf, et al.

The following tutorial offers scripts to outline the usage of CPlantBox (**?**Zhou et al., 2020) and its Python binding named *plantbox*. CPlantBox was developed from CRootBox (Schnepf et al., 2018), which is based on RootBox (**?**). CPlantBox is largely backward compatible by having the same underlying root-system model. For further documentation please refer to the Doxygen class documentation of the CPlantBox code.

Contents

1 Introduction **3**
1.1 Installation . . . 3
1.2 Basic example . . . 3

2 Topics / Applications **6**
2.1 Development over time . . . 6
2.2 How to virtually mimic experiments . . . 9
2.3 Postprocessing . . . 14
2.4 Visualisation . . . 14
2.5 Playing with parameters . . . 17
2.6 Plant tropism . . . 17
2.7 Sensitivity analysis . . . 17
2.8 Interaction Feedback . . . 17
2.9 RSML . . . 17

3 CPlantbox Structural Model **18**
3.1 General plant organs . . . 18
3.2 Seed . . . 18
3.3 Stem . . . 18
3.4 Leaf . . . 18
3.5 Root . . . 18

4 Plant Structural Functional Models **19**
4.1 Plant hydraulics . . . 19
4.2 Benchmark examples M3.1 and M3.2 . . . 19
4.3 The standard uptake fraction (SUF) and root system conductivity (Krs) . . . 19
4.4 Plant structure mapped to a macrogrid . . . 19
4.5 Model coupling . . . 19
4.6 Stomatal model . . . 19

5 Contributing **20**
5.1 Coding style . . . 20
5.2 Todos . . . 20

Progress compiling the tutorial

- Section 1 finished
- Section 2.1-2.4 mostly finished, 2.5-2.9 in progress
- Section 3 no started
- Section 4 no started
- Section 5 no started

1 Introduction

1.1 Installation

This installation guideline is for CPlantBox on Linux systems (e.g. Ubuntu). If C++ compilers and Python are not recent enough make sure you have a recent C++ compiler (e.g. `sudo apt install build-essential`) and python3 (e.g. `sudo apt-get install python3.6`).

If you wish to be able to follow the examples provided in the Jupyter Notebooks, it is recommended to install the Anaconda Python distribution:

- Install curl if not yet available: `sudo apt-get install curl`
- Download and install Anaconda (adapt the version accordingly):
  ```
  cd /tmp
  curl https://repo.anaconda.com/archive/Anaconda3-2021.11-Linux-x86_64.sh --output anaconda.sh
  bash anaconda.sh
  source ~/.bashrc
  ```

CPlantBox can be installed using either the install script installCPlantBox.py for CPlantBox only, or installDumuxRosi_Ubuntu.py for CPlantBox and dumux-rosi. First make sure Git ist installed, clone the CPlantBox repository, and run the install script:

- Install Git: `sudo apt-get install git`
- Go to your base folder and clone the repository by: `git clone https://github.com/Plant-Root-Soil-Interactions-Modelling/CPlantBox.git`
- Run install script: `python3 CPlantBox/installDumuxRosi_Ubuntu.py`

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

1.2 Basic example

To test the installation first go to the examples `cd tutorial/examples/` and run the following 'Hello World' like example by `python3 intro_basic.py`. It shows a typical CPlantBox simulation and Figure 1 shows the simulation result. The main steps include: open a CPlantBox parameter file (L13), perform the simulation (L20), save the results (L23-L26), and make an interactive plot showing the results (L29):

```
1  """ introctionary example """
2  import sys; sys.path.append("../.."); sys.path.append("../../src/")
3
4  import plantbox as pb
5  import visualisation.vtk_plot as vp
6
7  # Create a new plant
8  plant = pb.Plant()
9
10 # Open plant and root parameter from a file
11 path = "../../modelparameter/structural/plant/"
```

```
12 name = "fspm2023"
13 plant.readParameters(path + name + ".xml")
14
15 # Initialize
16 plant.initialize()
17
18 # Simulate
19 simtime = 40  # days
20 plant.simulate(simtime)
21
22 # Export final result (as vtp)
23 plant.write("results/example_plant.vtp")  # using polylines
24
25 ana = pb.SegmentAnalyser(plant)
26 ana.write("results/example_plant_segs.vtp")  # using segments
27
28 # Interactive plot, using vtk, press x, y, z to change view, r to
       reset view, g to save png
29 vp.plot_plant(plant, "age")  # e.g. organType, subType, age
```

Listing 1: Basic example (intro_basic.py)

Lets revise the above code in more detail:

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

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

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

8 Constructs the plant object.

12 Opens an .xml containing parameters describing the structural properties of the plant by defining the seed (SeedRandomParameters), the stem (StemRandomParameters), the leaf (LeafRandomParameters) and the roots (RootRandomParameters). Alternatively, all parameter can be set or modified directly in Python. A more detailed description is given in Section 3.

16 Initializes the simulation: Creates the initial stem, 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.

20 Performs the simulation. The value 40 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. 40 simulate(1) calls should yield a similar result as simulate(40) (due to stochasticity we cannot expect the exact same result).

23 Saves the resulting plant geometry in the VTK Polygonal Data format (VTP) where each plant organ is repesented as a polylines. Use Paraview to visualize the results, see Section 2.4.

25,26 If we want to visualize simulation results that are given per segment another option is to export the plant geometry segment wise. L25 creates a segment based representation of the plant geometry, and L26 saves it as VTP file.

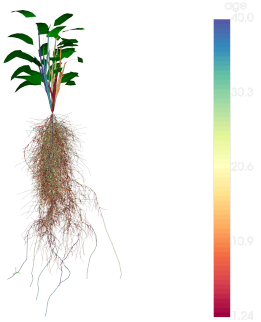


Figure 1: Plant after 40 days simulation time.

```
29 Create an interactive plot (use mouse to rotate or zoom) using VTK. Per
   default plant 'age', 'creationTime', 'radius','organType' or 'subType' can
   be viusalized. The organType is 1 for the seed, 2 for root, 3 for stem,
   and 4 for leafs. The subType is the number of the parameter set within a
   organ class, i.e. the type of root, stem or leaf. You can rotate (left click)
   pan (right click) or zoom (mouse wheel) and save a screenshot as png file
   by pressing 'g', or reset view 'r', or change view by pressing 'x', 'y', 'z',
   and 'v'.
```

2 Topics and Applications

This section describes the usage of CPlantBox by small examples without going into details of the underlying model. A more precise description of the structural model is given in following Section 3.

2.1 Development over time

The first example shows how to run the simulation in a simulation loop. This is important if we want to couple the model with ohter models, or implement feedbacks to the root architecture development, or as in this case, to retrieve information over time. The following script shows how to calculate the organ lengths over time.

```
"""plant organ lengths over time"""
import sys; sys.path.append("../..")
import plantbox as pb
import numpy as np
import matplotlib.pyplot as plt

path = "../../modelparameter/structural/plant/"
name = "hello_world"

plant = pb.Plant()
plant.readParameters(path + name + ".xml")
plant.initialize()

simtime = 60.  # days
dt = 1.
N = round(simtime / dt)  # steps

# calculate organ lengths over time
total, roots, stems, leafs = np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N)
roots_ = [[], [], [], []]

# Simualtion loop
for i in range(0, N):

    plant.simulate(dt)

    t = np.array(plant.getParameter("organType"))
    st = np.array(plant.getParameter("subType"))
    v = np.array(plant.getParameter("length"))  # surface, volume.
    total[i] = np.sum(v)
    roots[i] = np.sum(v[t == 2])  # root
    stems[i] = np.sum(v[t == 3])  # stem
    leafs[i] = np.sum(v[t == 4])  # leaf
    for j in range(0, 4):
        roots_[j].append(np.sum(v[np.logical_and(t == 2, st == j + 1)]))  # roots per sub type

# plot results
t_ = np.linspace(dt, N * dt, N)
plt.plot(t_, total, t_, roots, t_, stems, t_, leafs)
for r in roots_:
    plt.plot(t_, r, '--')
plt.xlabel("time (days)")
plt.ylabel("Organ length (cm)")
```

```
plt.legend(["total", "roots", "stem", "leafs", "tap root", "first
    order", "second order", "basal"])
plt.savefig("results/topics_development.png")
plt.show()
```

Listing 2: Length over time (topics_development.py)

8-13 Sets up the simulation.

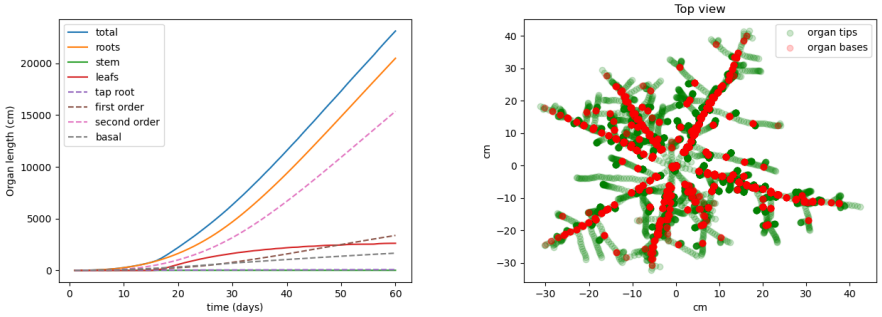
15-17 Defines the simulation time, time step, and the resulting number of simulate(dt) calls.

23-36 The simulation loop executes the simulation for a single time step L26. After each simulation step, we retrieve the organ type, the sub type, and the length of each organ (L28-30). It is possible to access all root random parameters and resulting realisations using plant.getParameter. In C++ the class functions are defined in Plant::getParameter, Organ::getParameter. In L31-36 we sum up specific organ lengths by boolean array indexing.

38-47 Creates Figure 2a.

Next we show how retrieve root tip and root base positions from a simulation. Each organ is represented by a polyline, which is a list of points connected by segments. We use this polyline to plot the organs base (first node of the polyline) and the organs tip (last node of the polyline).

```
"""how to find root tips and bases"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb
import numpy as np
import matplotlib.pyplot as plt

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

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

simtime = 30
dt = 0.5
N = round(simtime / dt)

bases, tips = [], []

for i in range(0, N):

    plant.simulate(dt)

    polylines = plant.getPolylines()  # polyline representation
    for i, r in enumerate(polylines):
        tips.append([r[-1].x, r[-1].y, r[-1].z])  # last index is
    the tip
        bases.append([r[0].x, r[0].y, r[0].z])  # first index is
    the base

bases = np.array(bases)  # convert to numpy arrays
tips = np.array(tips)
```



(a) Total organ lengths versus time (b) Top view of the organ tip and bases

Figure 2: Plant development over time

```
32
33 # Plot results
34 plt.title("Top view")
35 plt.xlabel("cm")
36 plt.ylabel("cm")
37 plt.scatter(tips[:, 0], tips[:, 1], c = "g", label = "organ tips",
       alpha = 0.2)
38 plt.scatter(bases[:, 0], bases[:, 1], c = "r", label = "organ bases
       ", alpha = 0.2)
39 plt.legend()
40 plt.savefig("results/topics_development2.png")
41 plt.show()
```

Listing 3: Organ tips and bases over time (topics_development2.py)

8-17 Sets up the simulation and simulation loop as before.

21-28 L23 performs the simulation for a single time step. First we retrieve all organs as polylines L25, where organ tips are the last nodes of the polylines L27, and organ bases are the first nodes L28. Organs that have not started to emerge have only 1 node, and are not retrieved by getPolylines().

30,31 Convert the lists to numpy array for indexing.

33-41 Creates Figure 2b. In CPlantBox roots are growing with a negative exponential growth rate, i.e. growth becomes slower towards the root reaching its maximal length. Therefore, the node density becomes higher towards the root tips.

Other useful functions are plant.getNodes() to retrieve all node coordinates, andplant.getSegments() which will give pairs of node indices containing the start node index, and end node index of each plant segment. Furthermore, plant.getSegmentOrigin() gives a corresponding reference to the Organ, to which each segment belongs. Creation times can be retrieved by plant.getNodeCTs() for each node, or plant.getSegmentCTs() for each segment.

2.2 How to virtually mimic experiments

In order to mimic experimental settings we can confine root growth by containers, or we can implement obstacles hindering root growth. Furthermore, periodic domains can be used to mimic field conditions. 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 within the domain, and a positive if the point is outside of the domain. CPlantBox has auxiliary functions for creating simple domains, which is shown in the following example.

Growth in a container

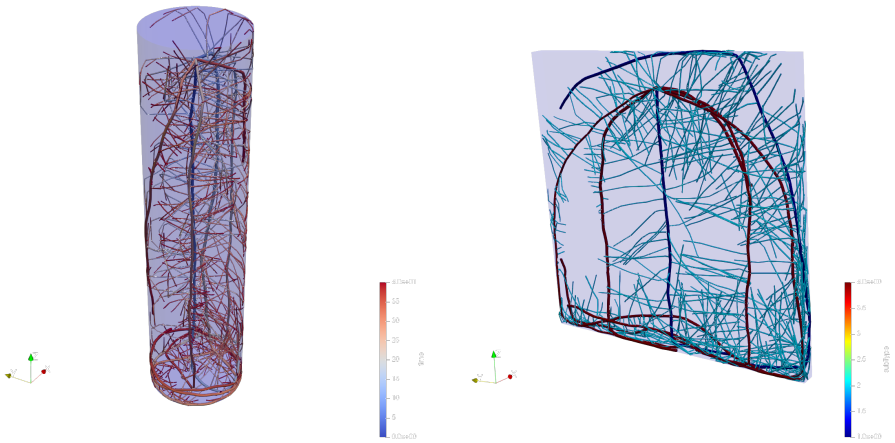
We show two examples where the plants root system grows confined by two types of containers, a cylindrical container or a rectangular rhizotron.

```
"""small example in a cylindrical container or rhizotron"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp

plant = pb.Plant()

path = path = "../../modelparameter/structural/rootsystem/"
name = "Zea_mays_4_Leitner_2014"
plant.readParameters(path + name + ".xml")

# 1.Creates a cylindrical container 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
plant.setGeometry(rhizotron)  # soilcore, or rhizotron

# Initialize & Simulate
plant.initialize()
plant.simulate(40)  # days

# Export final result (as vtp)
plant.write("results/topics_virtual.vtp")
pb.SegmentAnalyser(plant).write("results/topics_virtual_seg.vtp")

# Export containr geometry as Paraview Python script
plant.write("results/topics_virtual.py")

# Plot, using vtk
vp.plot_plant(plant, "subType")
```

Listing 4: Root growth in a container (topics_virtual.py)

The geometry is first created by constructing a specialization of the class SignedDistanceFunction, which is passed to the root system by the method plant.setGeometry():

9-11 Choose the parameter input file



(a) Confined by a cylinder (colour represents root age)

(b) Confined by a rhizotron (colour represents root type)

Figure 3: ParaView visualizations of results.

14 Construct a cylindrical container.

17 Construct a rhizotron.

20 Pick one of the two geometries. Note that it is important to call plant.setGeometry() before plant.initialize().

23,24 Initializes and simulates for 40 days.

27 Exports the plant structure geometry (without the soil domain geometry).

31 Its possible to save the soil domain geometry as Paraview Python script for visualization (and debugging), see Figure **??**. Run this script in Paraview by Tools→Python Shell, Run Script.

34 Interactive VTK plot. 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. visualised.

Next, we show how to build more complex container geometries using SDF.

Complex containers using SDF with set operations

In the following example we create some geometries that we might encounter in actual 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.

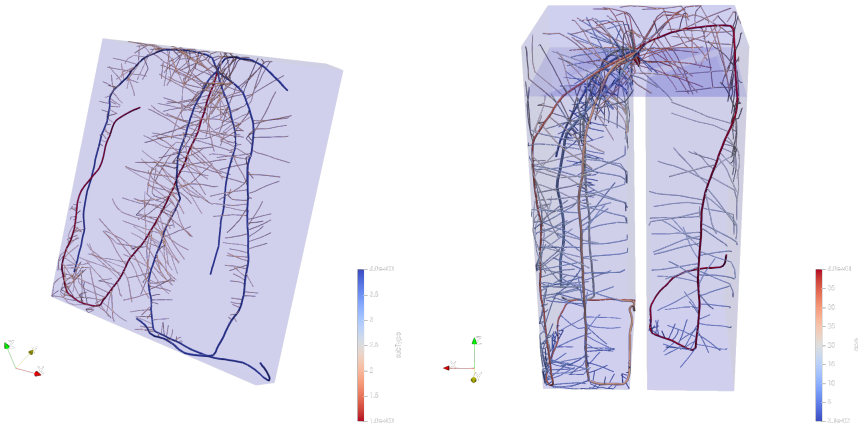
```
"""more complex plant containers"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp
import numpy as np

plant = pb.Plant()
path = path = "../../modelparameter/structural/rootsystem/"
name = "Zea_mays_4_Leitner_2014"
plant.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)  # seed location bevore rotation
A = pb.Matrix3d.rotX(alpha / 180.*np.pi)
posA = A.times(posA)  # seed location 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_)

# Simulate
plant.setGeometry(rotatedRhizotron)  # rotatedRhizotron, splitBox
plant.initialize()
plant.simulate(40)  # days

# Export and plot
plant.write("results/topics_virtual2.vtp")
plant.write("results/topics_virtual2.py")
vp.plot_roots(plant, "subType")
```

Listing 5: Root growth in more complex containers (topics_virtual2.py)

14-19 Definition of a rotated rhizotron, see Figure **??**: L15 creates the flat container with a small height, this container is then rotated and translated into the desired position. L16 is the location of the plant seed within the unrotated rhizotron. L17 defines the rotational matrix rotating around the x-axis. In L18 the seed position is rotated. Finally, in L21 the rhizotron is rotated and translated so that the seed location is moved to the origin.

21-30 Definition of of a split box, see Figure **??**: The split box is composed of a left box, a right box, and a top box connecting left and right. In L30 the geometry is defined by the set operation union of the three compartments.

33 Pick one of the three geometries for your simulation.



(a) Rotated rhizotron (b) Split box experiment

Figure 4: Complex container geometries described by SDF and set operations.

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

40 We cannot visualize the container geometry in the interactive rendering, but only the resulting root system.

Obstacles using SDF

We can also use set operations to create obstacles. The following example shows a rhizotube camera setup, where transparent tubes are used to analyse root growth. We can mimic this setup by defining tubes that act as an obstacle to the growing roots. The following code is similar to before, but using another geometry:

```
"""more complex plant containers with obstacles"""
import sys; sys.path.append("../..");sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp
import numpy as np

plant = pb.Plant()
path = path = "../../modelparameter/structural/rootsystem/"
name = "Moraesetal_2020"
plant.readParameters(path + name + ".xml")

# Rhizotubes as obstacles
box = pb.SDF_PlantBox(96, 126, 130)  # box
rhizotube = pb.SDF_PlantContainer(3., 3., 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)  # cm
z_ = (-10, -20, -40, -60, -80, -120)  # cm
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)

# Simulate
plant.setGeometry(rhizoTube)
plant.initialize()
plant.simulate(90)  # days

# Export results
plant.write("results/topics_virtual3.vtp")
plant.write("results/topics_virtual3.py")
vp.plot_roots(plant, "age")
```

Listing 6: Experimental setup with rhizotubes (topics_virtual3)

Definition of rhizotubes as obstacles, see Figure **??**:

14 Defines the surrounding box

15,16 Definition of a single rhizotube, that is rotated around the y-axis.

21,26 Create a list of rhizotubes at different locations that mimics the experimental setup.

27,28 Composes the final geometry by two set operation: first a union of all tubes, and then cut them out the surrounding box by taking the difference.

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("../.."); sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp

path = path = "../../modelparameter/structural/plant/"
name = "fspm2023"

simtime = 30  # days
N = 3  # number of columns and rows
dist = 40  # distance between the root systems [cm]

# Initializes N*N root systems
all = []
for i in range(0, N):
    for j in range(0, N):
        plant = pb.Plant()
        plant.readParameters(path + name + ".xml")
        seed = plant.getOrganRandomParameter(pb.seed)[0]
        seed.seedPos = pb.Vector3d(dist * i, dist * j, -3.)  # cm
```

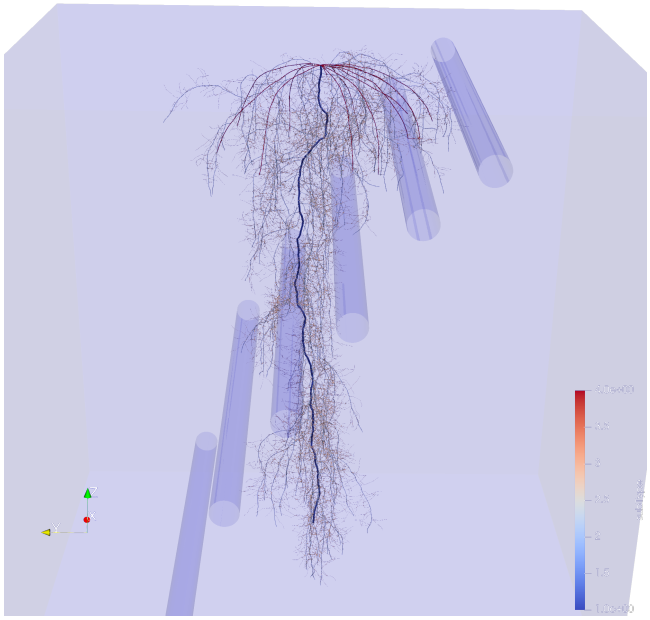


Figure 5: Rhizotubes act as obstacles to the root system

```
22          plant.initialize(verbose = False)
23          all.append(plant)
24
25  # Simulate all plants
26  for plant in all:
27      plant.simulate(simtime, False)  # verbose = False
28
29  # Export results as single vtp files (as polylines)
30  ana = pb.SegmentAnalyser()
31  for i, plant in enumerate(all):
32          filename = "results/topics_virtual4_" + str(i)
33          vp.write_plant(filename, plant)
34          ana.addSegments(plant)  # collect all
35
36  # Write all into single file (as segments)
37  ana.write("results/topics_virtual4_all.vtp")
```

Listing 7: Multiple root systems (topics_virtual4.py)

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

15-24 Creates the root systems, and puts them into a list `all`. L20 retrieves the plant seed, and L21 sets a new seed position.

26,27 Simulate all root systems.

30-37 Saves each individual root systems, and additionally, saves all root systems into a single file. Therefore, we create an SegmentAnalyser object (see Section 2.3) in L30 and merge all organ segments into it (L34). Finally,

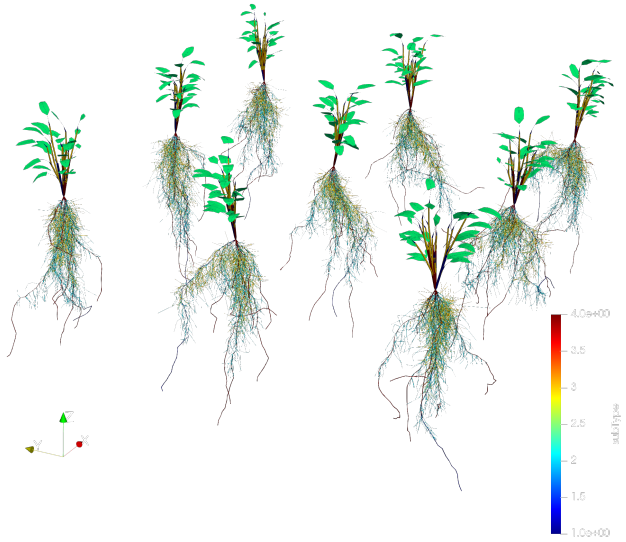


Figure 6: ParaView visualization of multiple root systems.

we export a single VTP file (L37). The resulting geometry is shown in Figure **??**, where ParaView was used for visualisation (see Section 2.4 - ParaView).

Periodic domains

If we consider only one plant type we often simplify field scale simulations to a single plant simulation with a periodic domain where the domain length and width is determined by planting density and inter-row distance. We can analyse the root geometry mapped to a periodic grid using SegmentAnalyser.mapPeriodic(), see Section 2.3. For coupling a root system with a periodic macroscopic soil model an unimpeded single root system is calculated and mapped into the periodic domain by the root to soil mapping function, see Section 4.4.

2.3 Postprocessing

The class SegmentAnalyser offers post-processing methods based on a representation of the plant by segments which connect two nodes with a single line. The segments are derived from the centerlines of the plant organs. We can use this approach to do distributions or calculate densities, and we can analyse the segments within any geometry by cropping the overall geometry to a region of interest. While the class SegmentAnalyser was designed for analysing roots most functionallity works also for the above plant part.

Root surface densities

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

```
"""root system surface density"""
import sys; sys.path.append("../..");sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp
import numpy as np
import matplotlib.pyplot as plt

plant = pb.Plant()
path = "../../modelparameter/structural/rootsystem/"
name = "Brassica_napus_a_Leitner_2010"
plant.readParameters(path + name + ".xml")

depth = 220
layers = 50
runs = 10

box = pb.SDF_PlantContainer(5, 5, depth, True)  # [cm3]

rs_ = []
for i in range(0, runs):
    plant.setGeometry(box)
    plant.initialize(False)
    plant.simulate(90, False)
    ana = pb.SegmentAnalyser(plant)
    rs_.append(ana.distribution("surface", 0., -depth, layers, True
    ))

soilvolume = (depth / layers) * 10 * 10  # [cm3]
rs_ = np.array(rs_) / soilvolume  # convert to density [cm2/cm3]
rs_mean = np.mean(rs_, axis = 0)
rs_err = np.std(rs_, axis = 0)

dx2 = 0.5 * (depth / layers)  # half layer width
z_ = np.linspace(-dx2, -depth + dx2, layers)  # layer mid points
plt.plot(rs_mean, z_, "b*")
plt.plot(rs_mean + rs_err, z_, "b:")
plt.plot(rs_mean - rs_err, z_, "b:")
plt.xlabel("root surface (cm^2 / cm^3)")
plt.ylabel("z-coordinate (cm)")
plt.legend(["mean value (" + str(runs) + " runs)", "std"])
plt.savefig("results/topics_postprocessing.png")
plt.show()

print(ana.getMinBounds(), ana.getMaxBounds())
vp.plot_roots(ana, "subType")
```

Listing 8: Calculating root surface densities in a soil column

9-12 Pick a plant or root system.

14-16 Depth is the length of the soil column (into z-direction), layers the number of vertical soil layers, where the root surfaces are accumulated, and runs is the number of simulation runs.

18 Creates a confing geometry.

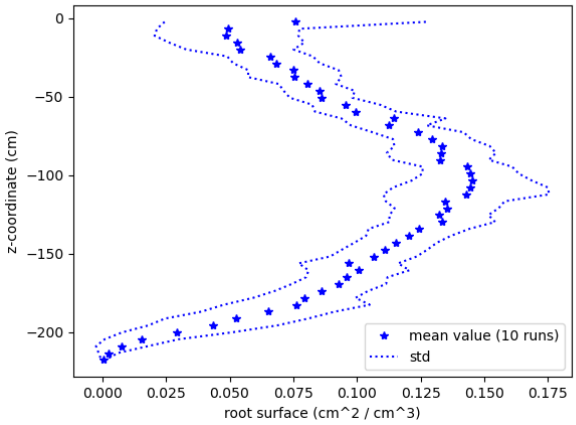


Figure 7: Root surface densitiy versus depth including standard deviation (based on 10 simulation runs)

20-26 Performs the simulations for *runs* times. L26 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).

28,29 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.

30, 31 Calculates the densities mean and the standard error.

33-42 Prepares the plot (see Figure **??**).

Analysis of segment within a cropping geometry

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

```
1  """analysis of results using signed distance functions to crop the
       segments to a certain geometry"""
2  import sys; sys.path.append("../..");sys.path.append("../../src/")
3
4  import plantbox as pb
5  import visualisation.vtk_plot as vp
6  import numpy as np
7  import matplotlib.pyplot as plt
8
9  path = "../../modelparameter/structural/rootsystem/"
10 name = "Zea_mays_1_Leitner_2010"
11
12 plant = pb.Plant()
13 plant.readParameters(path + name + ".xml")
```

```
# plant.setGeometry(pb.SDF_PlantContainer(10, 10, 200, True))
plant.initialize()
plant.simulate(120)
plant.write("results/topics_postprocessing2.vtp")

ana = pb.SegmentAnalyser(plant)
print(ana.getMinBounds(), ana.getMaxBounds())

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 for 10 cm

geom = soilcolumn  # pick one geometry for further analysis

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(plant)
ana.cropDomain(20, 20, depth)  # ana.mapPeriodic(20, 20)
axes[0].set_title('All roots in 20*20*100')
rl_ = []
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_, label = "{:g}
     days".format(t))
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(plant)
ana.crop(geom)
ana.pack()
ana_vis = pb.SegmentAnalyser(ana)
axes[1].set_title('Soil core (over time)')
rl_ = []
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_, label = "{:g}
     days".format(t))
axes[1].legend()

# distributions per root type
ana = pb.SegmentAnalyser(plant)
ana.crop(geom)
ana.pack()
axes[2].set_title('Soil core (per subType)')
rl_ = []
for i in range(1, 5):
    a = pb.SegmentAnalyser(ana)  # copy
    a.filter("subType", i)
```

```
70      rl_.append(a.distribution("length", 0., -depth, layers, True))
71
72  axes[2].plot((np.array(rl_[0]) + np.array(rl_[3])) / layerVolume,
        z_)
73  axes[2].plot(np.array(rl_[1]) / layerVolume, z_)
74  axes[2].plot(np.array(rl_[2]) / layerVolume, z_)
75  axes[2].legend(["basal roots", "first order roots", "second order
        roots"])
76
77  fig.subplots_adjust()
78  plt.savefig("results/topics_postprocessing2.png")
79  plt.show()
80
81  vp.plot_roots(ana_vis, "subType")
```

Listing 9: A virtual soil core experiment (topics_postprocessing2)

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 **??** 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 **??** and **??** for the two soil cores (chosen in L22).

The example shows differences between the central core and shifted core (see Figure **??** and **??**) 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.

SegmentAnalyser for measurements

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).

```
1  """ nodes and segments from measurements """
2  import sys; sys.path.append("../.."); sys.path.append("../../src/")
3
4  import plantbox as pb
```

```
5  import visualisation.vtk_plot as vp
6
7  # Data from any source, as Python types
8  nodes = [ [0, 1, 0], [0.2, 1.8, -1], [0, 1.3, -2], [0, 1, -3] ]
9  segs = [ [0, 1], [1, 2], [2, 3] ]
10 cts = [0., 0.1, 0.2]
11 radii = [ 0.1, 0.2, 0.1 ]
12
13 # convert from Python to C++ binding types
14 nodes = [pb.Vector3d(n[0], n[1], n[2]) for n in nodes]
15 segs = [pb.Vector2i(s[0], s[1]) for s in segs]
16
17 # create the SegmentAnalyser without underlying RootSystem
18 ana = pb.SegmentAnalyser(nodes, segs, cts, radii)
19 print("length", ana.getSummed("length"))
20 ana.write("results/topics_postprocessing3.vtp", ["creationTime", "
       radius"])
21 vp.plot_roots(ana, "creationTime")
```

Listing 10: SegmentAnalyser can be used on measured data (topics_postprocessing3.py)

7-11 Define some segments with data

14,15 We convert the Python list to lists of C++ types

18 We create the SegmentAnalyser object without an underlying plant

19,20 Use the Analyser object, by printing information, or writing a vtp.

21 Visualize your results

2.4 Visualisation

Quick visualizations from Python

It is possible to quickly visualize resulting geometry from Python using the module *vtk_plot*, which offers auxiliary functions using VTK (VTK User's Guide). Use

```
1 import visualisation.vtk_plot as vp
```

to import the module and to name it *vp*.

After creation and simulation of the plant, there are different ways to start an interactive plot:

```
1 vp.plot_segments(plant, "subType") # Option 1
2
3 ana = pb.SegmentAnalyser(plant)
4 vp.plot_segments(ana, "subType") # Option 2
5
6 vp.plot_plant(plant, "subType") # Option 3
```

The interactive plot can be to rotated (left mouse button), panned (mid mouse button, joystick like from plot center) or zoomed (right mouse button). Per default *creationTime*, *radius*, *subType* or *organType* can be viusalized. But generally all model parameters that will be presented in Section 3 can be visualized. 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'.

Option 1 and 2 are identical and can be used on Plant or SegmentAnalyser class. It visualizes the centerlines or polylines of all plant organs. A tube geometry representing the organs radius is created around each segment. In this way it is possible to map a parameter to a colour at segment level. Option 3 does the plot the same way as 1 and 2, but additionally plots the leafs a polygons. The first argument is the plant or SegmentAnalyser object, the second the parameter that shall be visualized.

It is possible to add arbitrary parameters using the SegmentAnalyser class:

```
ana.addData(name, data)
```

will add the parameter *name*, with *data* given per segment or nodes. The data array must have the right length, i.e. the number of segments or nodes within the SegmentAnalyser object.

Using a functional structural model the class *SegmentAnalyser* offers auxiliary functions to add additional model parameters:

```
ana.addAge(simtime)
ana.addConductivities(xylem, simtime, kr_max = 1.e6, kx_max = 1.e6)
ana.addFluxes(xylem, rx, sx, simTime)
ana.addCellIds(plant)
```

Line 1 creates the segment age from the segment parameter *creationTime* for the final simulation time *simtime*. Line 2 uses a a $XylemFlux$ object (named *xylem*) to evaluate the age dependent hydraulic conductivities, and optionally offers maximal values (for visualisation). It will create two parameters named kr and kx for the radial and axial conductivities. Line 3 adds radial and axial volumetric water fluxes named $axial\_flux$, $radial\_flux$ based on the xylem potentials rx, and soil potentials sx (both either matric potentials, or total potentials). Line 4 uses a $MappedSegments$ object (named *plant*) to create cell indices named *cell_id* per root segment. After adding the parameters, they can be visualized by creating an interactive plot, or exported to a file format.

How to export plant geometry to various file formats

There are various ways to export plant geometry for later analysis, visualization, or to use the geometry as mesh for other simulation software. The Plant class provides outputs in RSML (Lobet et al., 2015), VTP (VTK User's Guide), and for ParaView python scripts describing the soil container geometry. For RSML and VTP each plant organ is represented as a polyline:

```
plant.write("myfile.rsml")
plant.write("myfile.vtp")
plant.write("myfile.py")
```

The SegmentAnalyser class provides outputs in VTP (ParaView documentation), DGF (Dune Grid Format), and text files. The plant organs are represented by their segments.

```
ana = pb.SegmentAnalyser(plant)
ana.write("myfile.vtp", add_params)
ana.write("myfile.dgf")
ana.write("myfile.txt")
```

Writing VTPs we can additional add a list of parameter names (named add_params) which we want to export. Per default 'radius', 'subType', 'creationTime', and 'organType' are exported.

The interactive plotting tools are based on VTK and it is easy to write the resulting geometries as VTP in a binary format resulting in smaller file size. The organs are represented as polylines names 'filename.vtp'. Leafs are stored as polygons in a second file named 'filename_leafs.vtp':

```
vp.write_plant(filename, plant, add_params)
```

In the following paragraph we shortly present how to visualise VTP files using ParaView.

Paraview

After installing and running ParaView, open a VTP file. The structure is represented by lines. To get a better visualisation it is necessary to create a tube plot around these lines. This can be done by running the ParaView macro rsTubePlot, which is located in CPlantBox/src/visualisation/paraview_macros/. Start the macro by first adding the macro: Macros→Add new macro..., and then starting the macro by clicking the rsTubePlot button. In the properties menu, choose the parameter for viusalisation in the Coloring menu.

In CPlantBox soil container geometry is given in an implicit way

Additionally, if vp.write_plant() was used, a VTP for leafs can be openend, where a solid colour can be chosen from the menu. Multiple root systems as in Figure **??** can be visualized by using the SegmentAnalyser class to merge the root system (as in the example) and to visualze the VTP using rsTubePlot. Leafs can be added in a second step by opening the single leaf files (not as a group, but by opening the single files). The leaf polygons can be joined together using the Append Geometry filter.

How to make an animation

In order to create an animation in Paraview we have to consider some details. There are two approaches: One is to make one vtp file per animation frame (whih will need a lot of disc space). The second approach 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. The advantage of the first approach is, that we can also visualize plant leafs, if the visualization of leafs is not necessary the second approach is recommended.

The first approach uses a single VTP file per animation frame:

```
"""increase axial resolution (e.g. for animation)"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp

plant = pb.Plant()
path = "../../modelparameter/structural/plant/"
name = "fspm2023"  # "hello_world"
```

```
10 plant.readParameters(path + name + ".xml")
11
12 # Parameters for animation
13 sim_time = 30  # days
14 fps = 30  # frames per second
15 anim_time = 5  # seconds
16 N = fps * anim_time  # [1]
17 dt = sim_time / N  # days
18 filename = "animate"
19
20 # Modify axial resolution of all relevant organs
21 for organ_type in [pb.root, pb.stem, pb.leaf]:
22     for p in plant.getOrganRandomParameter(organ_type):
23         p.dxMin = 0.05
24         p.dx = 0.1  # adjust resolution
25
26 # Simulate
27 plant.initialize()
28 for i in range(0, N):
29     plant.simulate(dt)
30     vp.write_plant("results/" + filename + "{:04d}".format(i),
       plant)
31     print(i, '/', N)
```

Listing 11: Produces one VTP filel per animation frame (topics_visualisation)

12-18 The parameters from the animation.

21-24 Decrease the spatial axial resolution of all organs to obtain a smooth represenation of plant growth.

27-31 For each animation frame two files are written, one containing the organ centerlines, one containing the leaf geometries.

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

1. The VTP files can be opened as a group in ParaView, first open the files containing the plant organ centerlines
2. Create a tube plot with the help of the script tutorial src/visualisation/-paraview_macros/rsTubePlot.py (add the macro, then run it).
3. Next open the file for the leaf geometry and pick a leaf colour.
4. 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.
5. The resulting file might be uncompressed and very large. If the file needs compression, for Linux us e.g. ffmpeg -i in.avi -vcodec libx264 -b 4000k -an out.avi, which produces high quality and tiny files, and it plays with VLC.

In the second approach we create an animation with a single file and thresholding:

```
1  """increase axial resolution (e.g. for animation)"""
2  import sys; sys.path.append("../.."); sys.path.append("../../src/")
3
4  import plantbox as pb
5  import visualisation.vtk_plot as vp
6
7  plant = pb.Plant()
8  path = "../../modelparameter/structural/rootsystem/"
9  name = "Anagallis_femina_Leitner_2010"
10 plant.readParameters(path + name + ".xml")
11
12 # Modify axial resolution
13 for p in plant.getOrganRandomParameter(pb.root):
14     p.dx = 0.1  # adjust resolution
15
16 # Simulate
17 plant.initialize()
18 plant.simulate(60)  # days
19
20 # Export results as segments
21 ana = pb.SegmentAnalyser(plant)
22 ana.write("results/animation.vtp")
23
24 ana.mapPeriodic(15, 10)
25 ana.write("results/animation_periodic.vtp")
26
27 # Export geometry as Paraview Python script
28 box = pb.SDF_PlantBox(15, 10, 50)
29 plant.setGeometry(box)
30 plant.write("results/animation_periodic.py")
31
32 # Plot final (periodic) image, using vtk
33 vp.plot_roots(ana, "creationTime")
```

Listing 12: Produces a single file for later animation (topics_visualisation2)

13,14 Its important to use a small spatial resolution in order to obtain a smooth animation. L14 sets the axial resolution of roots to 0.1 cm.

21,22 Instead of saving the root system as polylines, we use the SegmentAnalyser to save the root system as segments.

24,25 It is also possible to make the root system periodic in the visualization in x and y direction to mimic field conditions.

28-30 We save the geometry as Python script for the visualization in ParaView.

After running the above script we perform the following operations Paraview to create the animation based on thresholding:

1. Open the .vtp file in ParaView (File→Open...), and open animation.vtp or animation_periodic.vtp.
2. Optionally, create a tube plot with the help of the script tutorial src/visualisation/paraview_macros/rsTubePlot.py (add the macro, then run it).
3. Run the script tutorial src/visualisation/paraview_macros/rsAnimate.py (add it as a macro, and run it). The script creates the threshold filter and the animation.

4. Optionally, visualize the domain boundaries by running the script animation_periodic.py. This step must be performed after the animation script rsAnimate.

5. Use File→Save Animation... as before.

2.5 Playing with parameters

Initialize plant parameters from scratch

In the previous examples we opened the plant parameters from an XML file. In the following example we demonstrate how to construct a plant with 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 2.7).

In order to set up a simulation by hand, we have to define all relevant model parameters. This is done by creating a *RootRandomParameter*, a *StemRandomParameter*, and a *LeafRandomParameter* object for each organ order or organ sub-type, and a single *SeedRandomParameter* for each plant type. In this section we will show how to create parameter sets, modify parameters and write XML plant parameter files. In Section 3 we present a detailed model description including all organ parameters.

Note that during the simulation, the parameters for a specific organ (i.e. OrganSpecificParamter with the specialisations RootSpecificParameter, StemSpecificParameter, and LeafSpecificParameter) are generated from the OrganRandomParameter class (with the specialisations RootRandomParameter, StemRandomParameter, and LeafRandomParameter) which represents the random distributions of certain parameters. First, we show how to construct instances of these classes, and show their parameters:

```
""" something basic"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp
import numpy as np

plant = pb.Plant()
root = pb.RootRandomParameter(plant)
stem = pb.StemRandomParameter(plant)
leaf = pb.LeafRandomParameter(plant)
seed = pb.SeedRandomParameter(plant)

print(stem)
stem.subType = 1
stem.lmax = 10
stem.theta = 0.
leaf.subType = 1
root.subType = 1
root.lmax = 100

plant.setOrganRandomParameter(stem)
plant.setOrganRandomParameter(leaf)
plant.setOrganRandomParameter(seed)
```

```
plant.setOrganRandomParameter(root)

plant.initialize()
plant.simulate(20)
vp.plot_plant(plant, "organType")
```

Listing 13: How to create random parameter sets (specialisations of OrganRandomParameter) (topics_parameters.py)

8-12 ljlkjlkjlkjlkj

14-20

22-25

```
"""simple root system from scratch (without parameter files)"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb

import matplotlib.pyplot as plt
import numpy as np

plant = pb.Plant()
p0 = pb.RootRandomParameter(plant)  # with default values,
p1 = pb.RootRandomParameter(plant)  # 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]
p1.r = 2  # initial growth rate
p1.dx = 1  # [cm] axial resolution
p1.tropismT = pb.TropismType.gravi  # exo
p1.tropismN = 2  # [-] strength of tropism
p1.tropismS = 0.1  # [rad/cm] maximal bending

plant.setOrganRandomParameter(p0)
plant.setOrganRandomParameter(p1)

srp = pb.SeedRandomParameter(plant)  # with default values
srp.seedPos = pb.Vector3d(0., 0., -3.)  # [cm] seed position
```

```
srp.maxB = 0  # [-] number of basal roots (neglecting basal roots and shoot borne)
srp.firstB = 10.  # [day] first emergence of a basal root
srp.delayB = 3.  # [day] delay between the emergence of basal roots
plant.setOrganRandomParameter(srp)

plant.initialize()

fig, axes = plt.subplots(1, 3, figsize = (15, 7))
simtimes = [0, 30, 60, 125]  # the last lateral will emerge at
for i in range(0, 3):

    plant.simulate(np.diff(simtimes)[i])  #  [day]
    a = axes[i]
    a.set_xlim([-15, 15.])
    a.set_ylim([-35., 0.])  # starts at -3 cm, max lengthsimple root system from scratch (without parameter files) 30 cm
    a.set_title("after {} days".format(plant.getSimTime()))
    roots = plant.getPolylines()
    for root in roots:
        for j, n in enumerate(root[:-1]):
            n2 = root[j + 1]
            a.plot([n.x, n2.x], [n.z, n2.z], "g")

fig.tight_layout()
plt.show()

plant.write("/results/topics_parameters2.vtp")

# Some outputs....
print(" length", plant.getParameter("length"))
print("    age", plant.getParameter("age"))
print("subType", plant.getParameter("subType"))
print("     la", plant.getParameter("la"))
print("la_mean", plant.getParameter("la_mean"))
print(" radius", plant.getParameter("radius"))
```

Listing 14: A root system from scratch (topics_parameters2.py)

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 **??**. 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 **??**.

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.

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

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*.

```
"""simple root system from scratch (without parameter files)"""
import sys; sys.path.append("../..");sys.path.append("../../src/")

import plantbox as pb

import matplotlib.pyplot as plt
import numpy as np

plant = pb.Plant()
p0 = pb.RootRandomParameter(plant)  # with default values,
p1 = pb.RootRandomParameter(plant)  # 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]
p1.r = 2  # initial growth rate
p1.dx = 1  # [cm] axial resolution
p1.tropismT = pb.TropismType.gravi  # exo
p1.tropismN = 2  # [-] strength of tropism
p1.tropismS = 0.1  # [rad/cm] maximal bending

plant.setOrganRandomParameter(p0)
plant.setOrganRandomParameter(p1)

srp = pb.SeedRandomParameter(plant)  # with default values
srp.seedPos = pb.Vector3d(0., 0., -3.)  # [cm] seed position
srp.maxB = 0  # [-] number of basal roots (neglecting basal roots
    and shoot borne)
srp.firstB = 10.  # [day] first emergence of a basal root
srp.delayB = 3.  # [day] delay between the emergence of basal roots
plant.setOrganRandomParameter(srp)

plant.initialize()

fig, axes = plt.subplots(1, 3, figsize = (15, 7))
simtimes = [0, 30, 60, 125]  # the last lateral will emerge at
for i in range(0, 3):
```

```
56
57      plant.simulate(np.diff(simtimes)[i])  #  [day]
58      a = axes[i]
59      a.set_xlim([-15, 15.])
60      a.set_ylim([-35., 0.])  # starts at -3 cm, max lengthsimple
        root system from scratch (without parameter files) 30 cm
61      a.set_title("after {} days".format(plant.getSimTime()))
62      roots = plant.getPolylines()
63      for root in roots:
64          for j, n in enumerate(root[:-1]):
65              n2 = root[j + 1]
66              a.plot([n.x, n2.x], [n.z, n2.z], "g")
67
68  fig.tight_layout()
69  plt.show()
70
71  plant.write("/results/topics_parameters2.vtp")
72
73  # Some outputs....
74  print(" length", plant.getParameter("length"))
75  print("    age", plant.getParameter("age"))
76  print("subType", plant.getParameter("subType"))
77  print("     la", plant.getParameter("la"))
78  print("la_mean", plant.getParameter("la_mean"))
79  print(" radius", plant.getParameter("radius"))
```

Listing 15: Example 2c (topics_parameters2.py)

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.

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 **??**

The mid column of Figure **??** 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 **??**.

```
"""summarize modify paramters..."""
import sys; sys.path.append("../..");sys.path.append("../../src/")

import plantbox as pb

import matplotlib.pyplot as plt
import numpy as np

plant = pb.Plant()
p0 = pb.RootRandomParameter(plant)  # with default values,
p1 = pb.RootRandomParameter(plant)  # 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]
p1.r = 2  # initial growth rate
p1.dx = 1  # [cm] axial resolution
p1.tropismT = pb.TropismType.gravi  # exo
p1.tropismN = 2  # [-] strength of tropism
p1.tropismS = 0.1  # [rad/cm] maximal bending

plant.setOrganRandomParameter(p0)
plant.setOrganRandomParameter(p1)

srp = pb.SeedRandomParameter(plant)  # with default values
srp.seedPos = pb.Vector3d(0., 0., -3.)  # [cm] seed position
srp.maxB = 0  # [-] number of basal roots (neglecting basal roots
    and shoot borne)
srp.firstB = 10.  # [day] first emergence of a basal root
srp.delayB = 3.  # [day] delay between the emergence of basal roots
```

```
plant.setOrganRandomParameter(srp)

plant.initialize()

fig, axes = plt.subplots(1, 3, figsize = (15, 7))
simtimes = [0, 30, 60, 125]  # the last lateral will emerge at
for i in range(0, 3):

    plant.simulate(np.diff(simtimes)[i])  #  [day]
    a = axes[i]
    a.set_xlim([-15, 15.])
    a.set_ylim([-35., 0.])  # starts at -3 cm, max lengthsimple
    root system from scratch (without parameter files) 30 cm
    a.set_title("after {} days".format(plant.getSimTime()))
    roots = plant.getPolylines()
    for root in roots:
        for j, n in enumerate(root[:-1]):
            n2 = root[j + 1]
            a.plot([n.x, n2.x], [n.z, n2.z], "g")

fig.tight_layout()
plt.show()

plant.write("/results/topics_parameters2.vtp")

# Some outputs....
print(" length", plant.getParameter("length"))
print("    age", plant.getParameter("age"))
print("subType", plant.getParameter("subType"))
print("     la", plant.getParameter("la"))
print("la_mean", plant.getParameter("la_mean"))
print(" radius", plant.getParameter("radius"))
```

Listing 16: Example 2c (topics_parameters3.py)

2.6 Plant tropism

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

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.

```
"""shows the influence of tropism paramters"""
import sys; sys.path.append("../.."); sys.path.append("../../src/")

import plantbox as pb
import matplotlib.pyplot as plt
import numpy as np

N_ = [0, 0.5, 1, 2]  # strength [1]
sigma_ = [0.1, 0.2, 0.4, 0.8]  # flexibility [1/m]
dx = 0.5  # axial resolution
theta = 70 / 180 * np.pi  # insertion angle [1]
simtime = 65

```

```
14 plant = pb.Plant()
15 srp = pb.SeedRandomParameter(plant)
16 srp.firstB, srp.delayB, srp.maxB = 3, 3 , 100
17 plant.setOrganRandomParameter(srp)
18
19 p0 = pb.RootRandomParameter(plant)
20 p0.name, p0.subType, p0.lmax, p0.r, p0.dx, p0.theta = "taproot", 1,
       100, 1, dx, theta
21 p0.tropismT = 1  # gravitropism
22 plant.setOrganRandomParameter(p0)
23
24 fig, axes = plt.subplots(4, 4, figsize = (15, 10))
25 for i, n in enumerate(N_):
26     for j, sigma in enumerate(sigma_):
27         a = axes[i][j]
28         plant.reset()  # does not delete parameters
29
30         p0.tropismN = n
31         p0.tropismS = sigma
32
33         plant.initializeLB(True)
34         plant.simulate(65, True)
35
36         nodes = plant.getNodes()
37         segs = plant.getSegments()
38         for s in segs:
39             n1, n2 = nodes[s.x], nodes[s.y]
40             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 plt.savefig("results/topics_tropism.png")
48 plt.show()
```

Listing 17: Example 2b (topics_tropism.py)

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 **??**.

In the first column (σ=0) of Figure **??** 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.

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("../..");sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp

rs = pb.RootSystem()
path = "../../modelparameter/structural/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)
```

```
43 rs.write("results/example_4a.vtp")
44
45 # Export geometry of static soil
46 rs.setGeometry(layer)  # just for vizualisation
47 rs.write("results/example_4a.py")
48
49 # Plot, using vtk
50 vp.plot_roots(rs, "type")
```

Listing 18: 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 **??**.

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("../..");sys.path.append("../../src/")

import plantbox as pb
import visualisation.vtk_plot as vp

import numpy as np


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):
        age = root.getAge()
        if age < self.age:
            return self.exo.tropismObjective(pos, old, a, b, dx,
    root)
        else:
            return self.gravi.tropismObjective(pos, old, a, b, dx,
    root)


# set up the root system
rs = pb.RootSystem()
path = "../../modelparameter/structural/rootsystem/"
name = "Zea_mays_1_Leitner_2010"
rs.readParameters(path + name + ".xml")
rs.initialize()

# Set useer defined after initialize
mytropism1 = My_Info_Tropism(rs)
mytropism1.setTropismParameter(2., 0.2)
mytropism2 = My_Age_Tropism(rs, 1.5, 0.5, 5)  # after 5 days switch
     from plagio- to gravitropism
```

```
53 rs.setTropism(mytropism2, 4)  # 4 for base roots, -1 for all root
       types
54
55 # Simulate
56 simtime = 100  # e.g. 30 or 60 days
57 dt = 1
58 N = round(simtime / dt)
59 for _ in range(0, N):
60     rs.simulate(dt)
61
62 # Export results (as vtp)
63 rs.write("results/example_4b.vtp")
64
65 # Plot, using vtk
66 vp.plot_roots(rs, "age")
```

Listing 19: 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 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 (**??**).

64 VTK plot.

2.7 Sensitivity analysis

2.8 Interaction Feedback

2.9 RSML

3 CPlantbox Structural Model

In this section we describe the underlying strutural model of CPlantBox including all model parameters.

3.1 General plant organs

3.2 Seed

3.3 Stem

3.4 Leaf

3.5 Root

3.6 Topology

In this section, we present how to create any type of topology from the organ types defined above, using the example example1f_branchingDescription.py. In the text bellow, the define as linking node all plant nodes which can carry one or more latterals. The number and position of the linking nodes are defined by lmax, la, lb and ln.

We start by loading the necessary libraries:

```
import sys; sys.path.append("../..");
path = "../../modelparameter/structural/plant/"
sys.path.append( path)
import plantbox as pb
import visualisation.vtk_plot as vp
from example1f import template_text
```

Here, we also load the incomplete parameter string "template_text" from the file "example1f.py". This string is shaped like a normal rsml parameter file (see section 2.9), except for the definition of the laterals of the stem and basal root, wich are missing. In this example, adapt an rsml file using python string which are then written in example1f.rsml, but the rsml files can be adapted directly. We create a dictionnary of strings, which defines the branching of the stem and basal roots:

```
successors= {
        "root":"""<parameter name="successor" ruleId="0" organType
    ="2" subType="2" probability="1"/>""",
        "stem":"""<parameter name="successor" ruleId="0" organType
    ="3" subType="2" probability="1"/>"""
        }
```

As can be seen in the code, the name of the branching parameter is "successor". The other arguments of interests are:

- ruleId, (numbered per organ subtype) an integer giving the index of the branching rule (in case rules are defined over several lines) [int]
- organType: one or several organ type(s) of the successor. [int]
- subType: one or several subtype(s) of the successor. [int]. In some older parameter file, this argument is still writen as "type"

In the example above, we define one rule for the root and one rule for the shoot. An organ of type 2 (root) and subtype 2 will grow out of the root, while an organ of type 3 (stem) and subtype 2 will grow out of the stem.

We then fill out the parameter string with the dictionnary, write the resulting string in a rsml parameter file.

```
filledText = template_text.format(**successors)
f = open(path+"example1f.rsml", "w")
f.write(filledText)
f.close()
```

We can then simulate and look at the resulting plant

```
p = pb.Plant(2)
p.readParameters(path+"example1f.rsml")
p.initialize(False)
time = 100
p.simulate(time, True)
ana = pb.SegmentAnalyser()  # see example 3b

vp.plot_plant(p, "organType")
```

We obtain as output Fig.**??**.

We can also define more complex branching patterns:

```
successors= {
        "root":
"""<parameter name="successor" ruleId="0" numLat="4" where
    ="-1,-3,-5,-7" organType="2" subtype="2" probability="1"/>
    <parameter name="successor" ruleId="1" numLat="1" organType="4"
     subtype="1" probability="1"/>
    <parameter name="successor" ruleId="2" numLat="6" organType="3"
     subtype="2" probability="1"/>""",
        "stem":"""<parameter name="successor" ruleId="0" where
    ="4,6,8" organType="2" subtype="1" probability="1"/>
        <parameter name="successor" ruleId="1" numLat="4" where
    ="-3" organType="4" subtype="1" probability="1"/>"""
        }
```

The following arguments become of interest:

- numLat, the number of lateral to create at each branching point [int]
- where, linking node index at which the rule applyed [int]

As seen above, the integers given for the "where" argument can be either positive or negative. A positive list of integers means that the rule will only be applied on those points. A negative list means that the rule will be applied everywhere except on those points.

As before, we overwrite the parameter file and look at the results

```
filledText = template_text.format(**successors)
f = open(path+"example1f.rsml", "w")
f.write(filledText)
f.close()
p = pb.Plant(2)
p.readParameters(path+"example1f.rsml")
p.initialize(False)
time = 100
p.simulate(time, True)
ana = pb.SegmentAnalyser()  # see example 3b

vp.plot_plant(p, "organType")
```

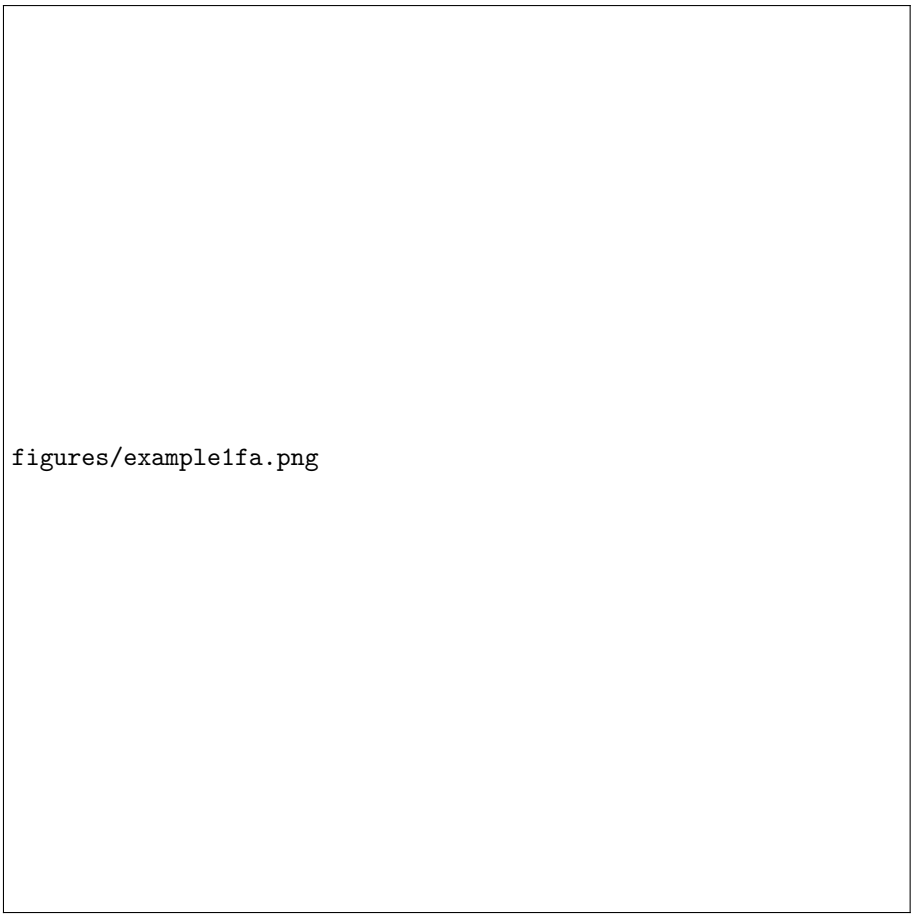


Figure 8: Visualizations of results of example 1fA

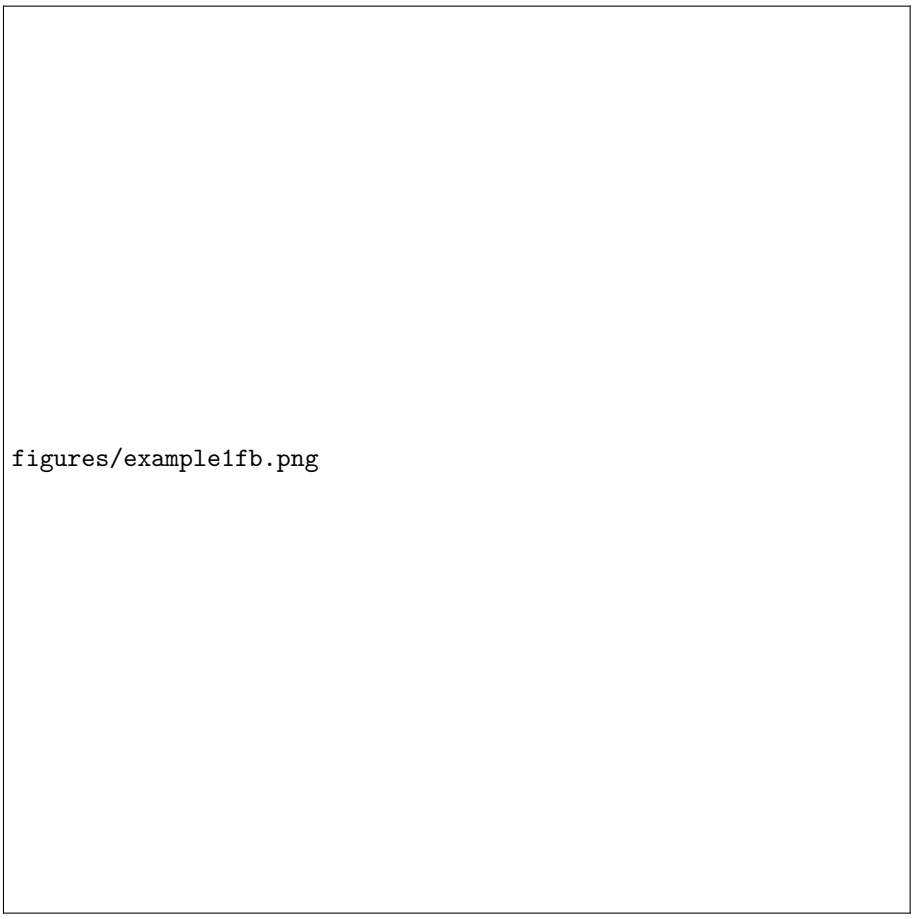


Figure 9: Visualizations of results of example 1fB

We obtain as output Fig.**??**. We can observe that shoot organs are growing out of the roots and vice versa. Also, because of the 'where' arguments, we do not have roots growing out of the upper linking nodes of the stem.

We can also setup a probabilistic branching pattern

```
successors= {
        "root":"""<parameter name="successor" ruleId="0" where="3,5" organType="2" subtype="2" probability="1"/>""",
        "stem":"""<parameter name="successor" ruleId="0" where="-3" numLat="4" organType="3,4,2" subtype="2,1,3" probability="0.2,0.3,0.3"/>
        <parameter name="successor" ruleId="1" where="3" numLat="1" organType="3" subtype="2" probability="1"/>"""
        }
```

We then use the "probability" parameter (see 1st "stem" successor). The sum of probabilities for one successor rule must be ¡ or = 1. As can be seen for the 1st stem successor, we can define a list of organ types, suptypes and growth probabilities. In the example above, according to ruleId 0 for the stem, up to 4 laterals will be created at each linking node (numLat ="4"). 20% of those laterals will be of type 3, subtype 2, 30% of type 4, subtype 1, 30% of type 2,

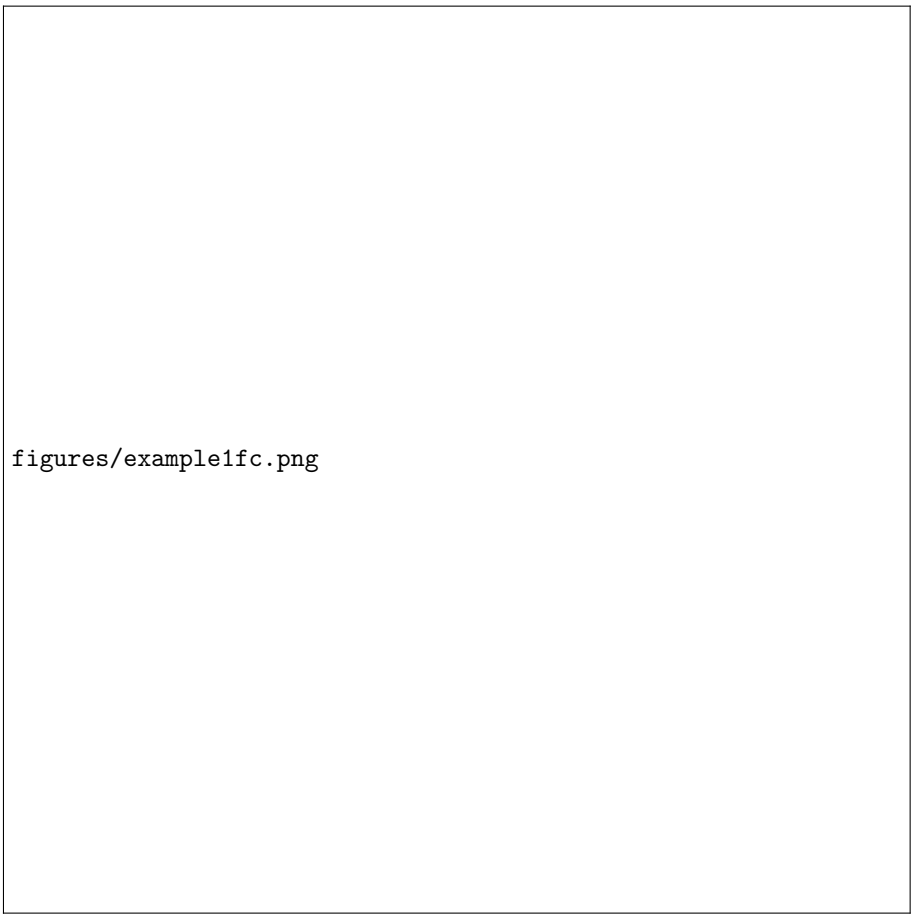


Figure 10: Visualizations of results of example 1fB

subtype 3. 20% of the time, no lateral will grow.

As before, we fill the parameter file and look at the results

```
filledText = template_text.format(**successors)
f = open(path+"example1f.rsml", "w")
f.write(filledText)
f.close()
p = pb.Plant(2)
p.readParameters(path+"example1f.rsml")
p.initialize(False)
time = 100
p.simulate(time, True)
ana = pb.SegmentAnalyser()  # see example 3b

vp.plot_plant(p, "organType")
```

We obtain as output Fig.**??**.

4 Plant Structural Functional Models

4.1 Plant hydraulics

4.2 Benchmark examples M3.1 and M3.2

4.3 The standard uptake fraction (SUF) and root system conductivity (Krs)

4.4 Plant structure mapped to a macrogrid

4.5 Model coupling

4.6 Stomatal model

5 Contributing

5.1 Coding style

5.2 Todos

Topics that are not covered yet or should be improved

- Elongation rate according to Moacir et al.
- Delay based, versus length based lateral emergance.
- 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...)
- The DuMux binding and MPI (probably belongs to dumux-rosi)

References

Lobet, G., Pound, M. P., Diener, J., Pradal, C., Draye, X., Godin, C., Javaux, M., Leitner, D., Meunier, F., Nacry, P., et al. (2015). Root system markup language: toward a unified root architecture description language. *Plant physiology*, 167(3):617–627.

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.

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).