rhizoscan Documentation

Release 1.0.0

Julien Diener

Apr 30, 2018

Contents

CHAPTER 1

Module description

Summary

Version 1.0.0

Release 1.0.0

Date Apr 30, 2018

Author See *Authors* section

ChangeLog See *ChangeLog* section

Overview

This VPlant package provide a set of tool to analyse 2D images of root system. The main goal is to allow extraction of Root System Architecture (RSA) as Multiscale Tree Graph (MTG) from images and image sets. It also provides typical mesurement analysis, such as root axes length (primary, secondary, total) and comparative plots.

CHAPTER 2

Documentation

2.1 Installation

2.1.1 Installation on Ubuntu with Conda

Contents

0. System Install

```
sudo apt-get install git
```

1. Download and install miniconda

See : http://conda.pydata.org/miniconda.html

```
wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh
chmod +x  Miniconda2-latest-Linux-x86_64.sh
./Miniconda2-latest-Linux-x86_64.sh
```

2. Create your own virtual environment

```
conda create --name rhizoscan python


# Activate your virtual environnement each time
source activate rhizoscan
```

3. Install Rhizoscan dependencies

```
conda install sphinx jupyter nose coverage anaconda-client
conda install numpy scipy matplotlib scikit-image opencv pil pillow scikit-learn
conda install -c openalea openalea.mtg openalea.vpltk openalea.visualea openalea.core
```

3.1 Download & install TreeEditor

```
conda install -c openalea -c openalea/label/unstable openalea.treeeditor
```

3.2 Download & install RSML-conversion-tools

```
conda install -c openalea -c openalea/label/unstable rsml
```

4. Install & test Rhizoscan

```
git clone https://github.com/openalea-incubator/rhizoscan
cd rhizoscan
python setup.py develop --prefix=$CONDA_PREFIX
nosetests test
```

2.2 User Manual

Version 1.0.0

Release 1.0.0

Date Apr 30, 2018

This manual explains how to use this package to analyse image of root system architecture. It thus focus on the highest-level functions designed for end-user, and requires very little experience in computer science.

For a complete reference guide of all the package content, see *rhizoscan*.

2.2.1 Root image analysis with python

This tutorial provides the description of how to do analyse root images with rhiziscan using the python programming language. A minimal knowledge of python is recommanded.

Content of this tutorial

To process one root image, you will needs:

- to select an image (or image file name) to be analysed
- to choose the suitable pipeline parameters

In this tutorial, we use an image provided with the package:

```
>>> from rhizoscan import get_data_path
>>>
>>> filename = get_data_path('pipeline/arabido.png')
>>> assert os.path.exists(filename), "could not find test image file:"+filename
```

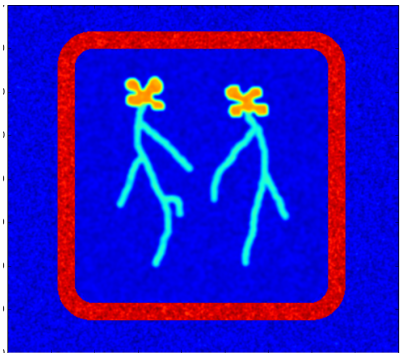
Step by step image analysis

First import the rhizoscan modules we need, and matplotlib to view intermediary output:

```
>>> from rhizoscan.root.pipeline import load_image, detect_petri_plate, compute_graph,
↪ compute_tree
>>> from rhizoscan.root.pipeline.arabidopsis import segment_image, detect_leaves
>>>
>>> from matplotlib import pyplot as plt
```

First step: load the image from file:

```
>>> image = load_image(image_filename)
>>> plt.imshow(image);
```

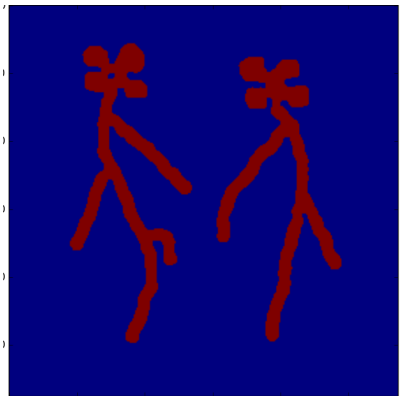


Then find the petri plate in the image, as a image mask

```
>>> pmask, px_scale, hull = detect_petri_plate(image,border_width=25, plate_size=120,␣
↪fg_smooth=1)
>>> plt.imshow(pmask);
```

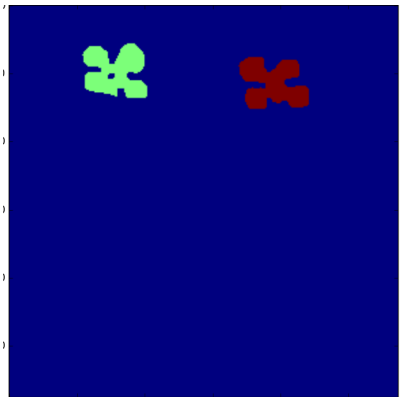
Segment the root (and leaf) pixels:

```
>>> rmask, bbox = segment_image(image,pmask,root_max_radius=5)
>>> plt.imshow(rmask);
```



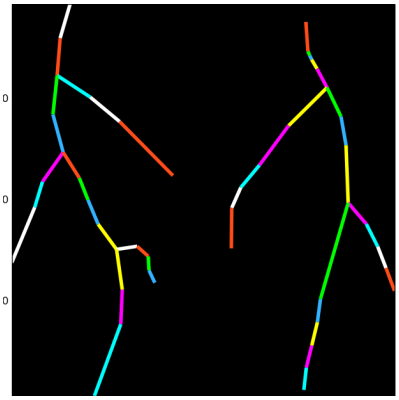
Find the seed of the root system:

```
>>> seed_map = detect_leaves(rmask, image, bbox, plant_number=2, leaf_bbox=[0,0,1,.4])
>>> plt.imshow(seed_map);
>>> #plt.imshow(seed_map+rmask);    # to view the seed map on top of the binary mask
```



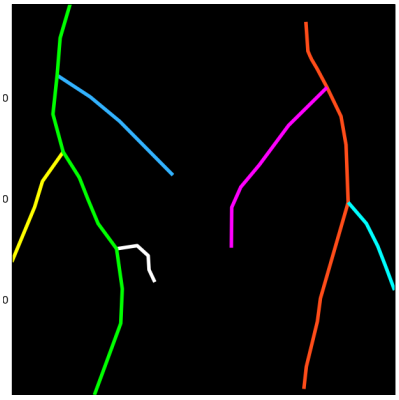
Compute the graph for the root system

```
>>> graph = compute_graph(rmask,seed_map,bbox)
>>> graph.plot()
```



Finally, compute the RSA tree:

```
>>> tree = compute_tree(graph, px_scale=px_scale)
>>> tree.plot()
```



It is probably necessary to convert this RSA tree to MTG format, for interoperability:

```
>>> from rhizoscan.root.graph.mtg import tree_to_mtg
>>> rsa = tree_to_mtg(tree)
```

To save this (root) mtg in a rsml file:

```
>>> from rsml import mtg2rsml
>>> from rsml.continuous import discrete_to_continuous
>>>
>>> rsa_cont = discrete_to_continuous(rsa.copy())
>>> mtg2rsml(rsa_cont,'some_file.rsml')
```

Here is the full code:

```
>>> from rhizoscan.root.pipeline import load_image, detect_petri_plate, compute_graph,
↪ compute_tree
>>> from rhizoscan.root.pipeline.arabidopsis import segment_image, detect_leaves
>>>
>>> from matplotlib import pyplot as plt
>>>
>>> image = load_image(image_filename)
>>> plt.imshow(image);
>>>
>>> rmask, bbox = segment_image(image,pmask,root_max_radius=5)
```

```
>>> plt.imshow(rmask);
>>>
>>>
>>> seed_map = detect_leaves(rmask, image, bbox, plant_number=2, leaf_bbox=[0,0,1,.4])
>>> plt.imshow(seed_map);
>>> #plt.imshow(seed_map+rmask);
>>>
>>> graph = compute_graph(rmask,seed_map,bbox)
>>> graph.plot()
>>>
>>> tree = compute_tree(graph, px_scale=px_scale)
>>> tree.plot()
>>>
>>> from rhizoscan.root.graph.mtg import tree_to_mtg
>>> rsa = tree_to_mtg(tree)
```

Using the arabidopsis pipeline

The above steps are all contained in *the arabidopsis pipeline* which is used slike this:

```
>>> from rhizoscan.root.pipeline.arabidopsis import pipeline
>>> from rhizoscan.datastructure import Mapping
>>>
>>> # 1. Create a namespace to execute the pipeline with input image filename and␣
↪parameters
>>> d = Mapping(filename=filename, plant_number=2,
>>>             fg_smooth=1, border_width=.08,leaf_bbox=[0,0,1,.4],root_max_radius=5,␣
↪verbose=1)
>>>
>>> # 2. Run the pipeline
>>> pipeline.run(namespace=d);
>>>
>>> # 3. Access computed data (example)
>>> d.tree.plot()       # plot the estimated RSA (use an internal RSA graph structure)
>>>
>>> d.rsa               # estimated RSA as a MTG
>>> # <openalea.mtg.mtg.MTG at 0x.....>
```

TODO explain the relation between pipeline and namespace

Computed data, final RSA as well as intermediate data, can be store in a given output folder. To do this, one should set the output directory for the namespace, and give the list of data that should be stored:

```
>>> # set the namespace output directory
>>> import tempfile, os
>>> outdir = tempfile.mkdtemp()                  # create a temporary directory
>>> d.set_file(os.path.join(outdir,'test'), storage=True)
>>>
>>> # run the pipeline, setting which data to store
>>> pipeline.run(namespace=d, store=['pmask','rmask','seed_map','tree','rsa'])
```

TODO describe pipeline parameters, or link to pipeline doc

Note: The file name of the storage files will all start by the value of `test` and a suffix made from the data name. E.g. the ‘‘seed_map‘‘ image use the suffix “_seed_map.png”, so in our example a file `[outdir]/test_seed_map.png` will be created.

Once you have finished with the computed data, don't forget to delete it: either manually using your OS file manager, or with python:

```
import shutil
shutil.rmtree(outdir)
```

Dataset analysis

TODO update doc

An *image database* can be process easily. For example, using the testing databse of rhizoscan, this is done using the following:

```
from rhizoscan import get_data_path
from rhizoscan.root.pipeline import database
from rhizoscan.root.pipeline.arabidopsis import pipeline

db = get_data_path('pipeline/arabidopsis/database.ini')
db, invalid, outdir = database.parse_image_db(db)

for elt in db:
    pipeline.run(elt)
```

Todo: To finish

- what are hidden the paremeter => cf *pipeline api*
- how to get output data (ex 'tree')

Finally, if your don't need it anymore, remove the output directory used by the pipeline:

```
import shutil
shutil.rmtree(outdir)
```

Visualisation and measurements

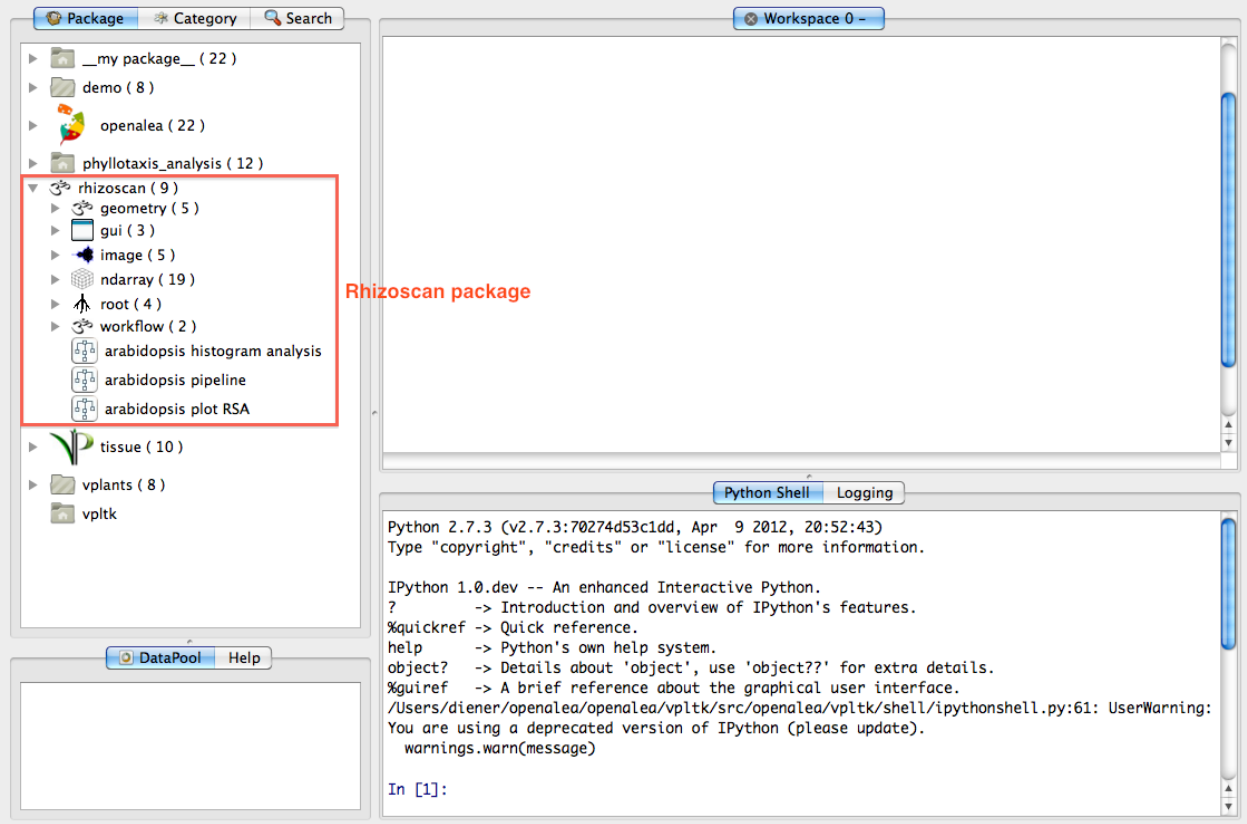
Note: Most of the following requires a matplotlib

Todo: split in the 2 previous parts?

plotting graph & tree exemple of getting some measurement from a tree: root.measurement

2.2.2 Root image analysis with visualea

Visualea provide a graphical iterface to use openalea components which does not require any programming. If you have installed the *rhizoscan package*, it should appear in the package manager (click on the triangle on the left of **rhizoscan** in the package manager to open it and display the package content):



Three tutorials have been made to show how to use this package functionalities. The rhizoscan package comes with a couple of image data which the following tutorials are made to process by default: so you can just go and try.

Analysis of one root image with visualea

This allows to extract root system architecture from one image file. It is also usefull for testing the image pipeline an a couple of images before processing a whole image data set.

Analysis of an image database with visualea

Once the pipeline parameters are choosen, a large image set can be analysed automatically using a *database* system. This tutorial show how this works.

Visualisation of the extracted root system with visualea

Here this tutorial show simple ways to plot extracted root system, show comparative measurement on processed database, and export analysis to table files.

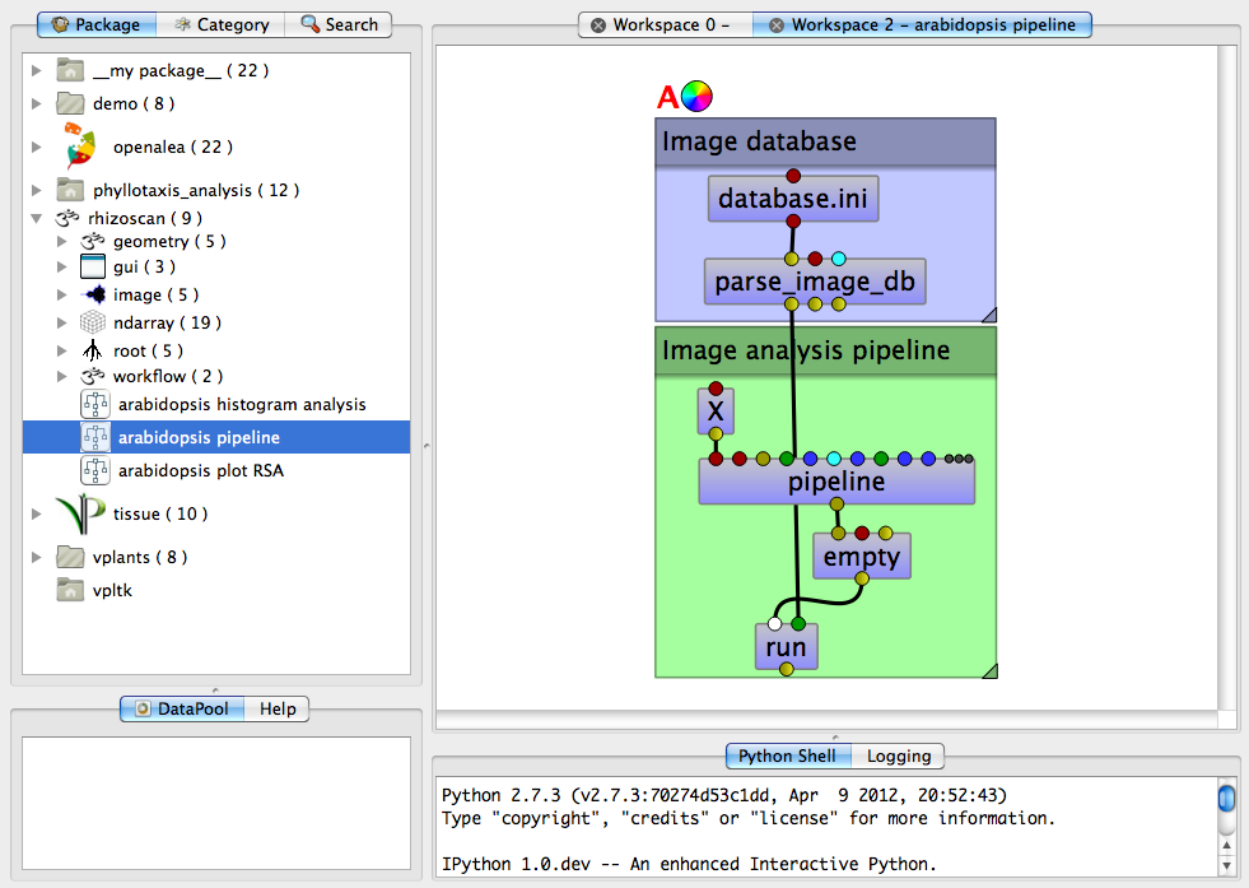
Related documentation

Analysis of one root image with visualea

Todo: all

Analysis of an image database with visualea

Automatical analysis of a set of root images can be done using an *image database*. The rhizoscan package provide a visualea dataflow for this task. To open it, doulbe click on **arabidopsis pipeline** at the bottom of the rhizoscan package:



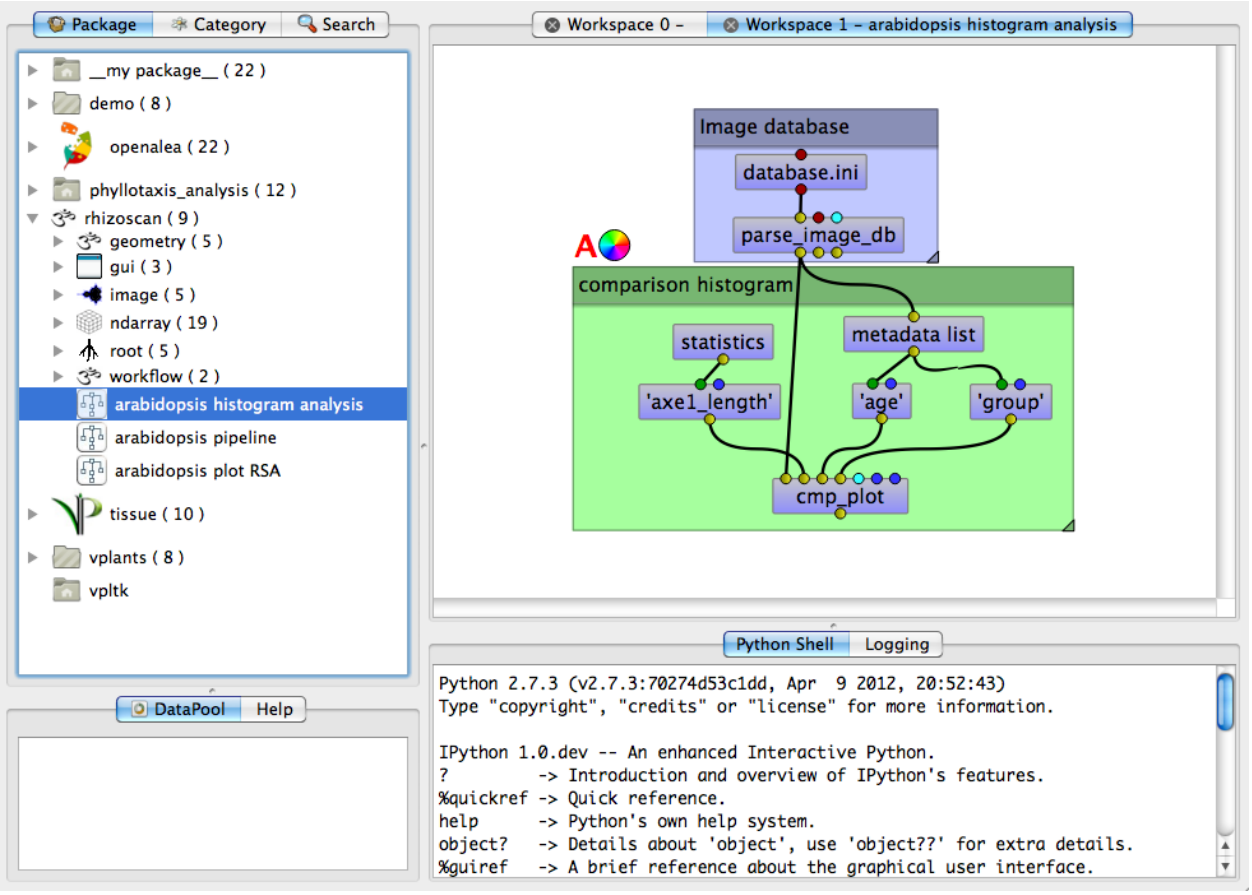
This dataflow is made of two parts:

1. **The top one loads an image database. It contains 2 modules:**
    - The first is to indicates the database file to load (see *image database* for details). By default it points to a little example database contained in the rhizoscan package. If you want toselect another file, double click on the top modules. It opens a file selection user interface where you can browse for the database file you want to load. You will need to have a valid database file in ini file format: see the page on *image database* for a description.
    - The second is the module that load all images from the database. It does not require any configuration.
2. **The bottom one extracts the root systems from all images. It has two main modules:**
    - The **pipeline** module is the image *arabidopsis image pipeline* which analysis root images.
    - The lower module named **run** is the *"start button"*: to apply the image pipeline analysis to the whole database, right click on the **run** module then select run in the menu.

Visualisation of the extracted root system with visualea

Ploting comparative histogram

The rhizoscan package provide a simple tool to compare measured root system visually: open the dataflow **arabidopsis histogram analysis** at the bottom of the rhizoscan package:



This dataflow as two parts:

- The upper part is the database loader: select the database file using the top modules (named *databse.ini*)
- The second part provide the plotting tools: - the module **cmp_plot** do the plot. To start it right click on it and select **run**. The plot is done with respect to three main inputs - the one on the left (*axe1_length* in the example) is the measurement to plot. Double click on the module to open the selection interface. - The center one (*age*) indicates the metadata name that contains the

2.2.3 The image pipeline

The analysis of root system from images is done using on of a set of *image pipelines*. Each pipeline comes as a all-in-one function that iteratively run the *processing modules* of the pipeline.

These modules typically do each of the following tasks:

1. frame detection
2. image segmentation

3. seeds detection
4. convertion to graph
5. extraction of the Root System Architecture

The arabidopsis pipeline

This pipeline has been developed to analysis image of arabidopsis root system grown and imaged using a specific *experimental protocol*. It contains the following steps:

1. **Petri dish detection** The root system have been growned and imaged in a squared Petri dish which is marked by four hand drawn curves, one on each corner, using a black pen.
2. **Image segmentation** It follows the standard algorithm which first estimate and remove the lighting background using an overestimate of the maximum root radius, in pixels. It then separate root area from background using a simple expectation maximization (EM) algorithm.
3. **Leaves detections** The analysis pipeline uses the detected leaves to determine automatically the starts of the root systems. This is done by image segmentation based on the leaves opacity being higher than the root axes.
4. **Convertion to graph** This is the standard algorithm and doesn't need any parametrization
5. **Extraction of the Root System Architecture** The Extraction of the RSA is done using a apriori model suitable for arabidopsis: it detect only one main root axes (the longest), all other being at least secondary.

Note: This pipeline can thus analyse root images if the following apply:

- the Petri dish respect the frame detection protocol
- leaves are more opaque than roots
- there is one main axes (order 1), and it is the longuest

Arabidopsis pipeline API

To use the arabidopsis pipeline from python, do:

```
from rhizoscan.root.pipeline.arabidopsis import pipeline
data = pipeline.run(**inputs_arguments)
```

With the following `inputs_arguments`:

image `(R)` The image filename or a numpy array-like to analyse

output `(R)` The commun base of output file. Each module of the pipeline will save a file with path like *output_suffix*

update An optional list of the module name to recompute even if previously computed data is accessible

metadata Optional dictionary-like structure with arbitrary field names (keys). The metadata is appended to the *'tree'* output data. Moreover all fields are added to the pipeline *namespace* and the metadata can be provide values for the pipeline inputs arguments.

plant_number `(D)` Number of roots systems. default is `1`

plate_width `(D)` The *real* size of the Petri plate side in the desired unit for output measurements (default `120`).

leaf_height `(D)` A list of 2 numbers between 0 and 1 that reduces the search area for leaf with respect to the detected Petri plate. The default is `[0.,0.2]`, meaning that the leaves appear in the 20% superior part of the plate.

root_max_radius `(D)` Overestimate of the maximum root radius **in pixels**. Anything between 1 and 3 times the real value is suitable.

root_min_radius Estimate lower bound of root radius used by the leaf detection algorithm. It is not a sensitive parameter. Increasing it tend to increase the leaf area.

min_dimension Minimum size of root system in pixels (default `50`). Anything less than this size is not analysed.

smooth Initial smoothing of the input image before processing. This value is the sigma parameter (in pixels) of a gaussian kernel (default `1`).

verbose If positive, print some intermediate computing state.

`(R)` Are required arguments, and `(D)` are arguments that depend on the image data and should be asserted and probably changed if default values are not suitable. For the others, the default values are generic, and should not need to be changed. .. `image` and `output` ar required parameters. Moreover, `plant_number`, `plate width`, `leaf_height` and `root_max_radius` should be provided if the respective default values are not suitable.

The pipeline.run returns a dictionary of the pipeline *namespace* (`data` in the above example). It is the set of variables used thro

pmask (numpy arrays) The mask of the detected Petri plate

rmask (numpy arrays) The binary mask of the root axes

seed_map (numpy arrays) The detected leaf area map

graph (RootGraph) The graph representing the root axes in `rmask`

tree (RootAxialTree) The extracted axial tree representing the root systems

bbox (tupel of slices) The bounding box of the detected Petri plate in the original image. `rmask` and `seed_map` are for to this region.

px_ratio (float) The size of 1 pixel in the designed measurement unit. It is computed based on the `plate_width` and the detected plate area

See also

- **tutorials:**
  - *Root image analysis with python*
  - *Root image analysis with visualea*
- back to the *User Manual*

2.2.4 Image Database

Database file system

The *rhizoscan* package provide a simple method to manage and process sets of images through a *database* mechanism. Here, a d

filename the image file name

metadata a (hierarchical) structure of descriptive parameters

output the file name base for all data computed from the orignal image

---

Note: this is not a real database, but it provides similar behaviors

---

An image database is made of:

- a set of image files stored such that they can all be listed using a shell globbing pattern process by the python glob tool. For example the pattern `*/*.jpg` lists all `.jpg` images found in any subfolder for the current directory.
- a `.ini` file that describes the database. In particular, it contains the `globbing pattern` mentioned above, and the metadata related to the images.

To load such as databse with python, do:

```
>>> from rhizoscan.root.pipeline import database
>>> db, invalid, output = database.parse_image_db('openalea/rhizoscan/test/data/
↪pipeline/arabidopsis/database.ini')
```

This function returns:

db The database

invalid the list of files that correspond to the globbing pattern but for which filename included metadata were not recognized

output the relative path for storing computed data

Database descriptor:

To understand how to make a database *ini* file, let's look at the exemple in `[rhisoscan-dir]/test/data/pipeline/arabidopsis/database.ini`:

```
[PARSING]
pattern=[age:str]/Photo_[id:int].jpg
out_dir=outputs
group={0:'A',1:'B'}

[metadata]
plant=arabidopsis thaliana
xp=test RhizoScan
plant_number=5

[A]
group=A
descriptif=.5gln5
glutamin=0.5

[B]
group=B
descriptif=1mM5
nitrate=1
```

With folder content:

```
J10/
  + Photo_001.jpg
  + Photo_011.jpg
  + Photo_invalid.jpg
J11/
  + Photo_001.jpg
  + Photo_011.jpg
database.ini
```

The `ini` file contains four parts:

PARSING Describe which files to process and which metadata to attach to them.

The `pattern` field indicates what files should be processed, by replacing brackets by `*`, it gives the file globbing patt

- the 1st is a string (str) and should be stored as the `age` metadata
- the 2nd is an integer (int) and is stored as the `id` metadata

The brackets is always a pair `metadata_name:parameter_type` where `parameter_type` should be either:

- a python type (`int`, `float`, `str`, ...)
- `$`: which means to use the content the fields in the ini file that has the respective name (see *adding metadata to labeled file name*)
- `date`: identified as a date, it requires `PARSING` to have a `date` field (see *storing date metadata in file name* for details)

If a detected file has a parameter in its file name that does not respect the given data type, it is not added to the database but returned in the `invalid` output.

The `ini` file of this exemple also provide a `group` field to attach metadata to group of images with respect to there position in their respective folder. See *grouping database files* section for details

metadata default parameters-values to attach to **all detected files**.

A & B metadata set that can be attach to the group of files with respective label. Here these are metadata to attach to the images of group `A` and `B` respectively.

grouping database files

A simple way to attach specific metadata to a group of files is to group them by their position (sorted by file name) in the folder they are stored in. This is usefull if images are given in folder such that images sharing metadata appears in a specific order.

The grouping mechanism is obtained using the `PARSING` keyword `group`. In the example above, it indicates that:

- from the image `0` (i.e. the 1st), files are of the group `A`
- from the image `1` (i.e. the 2nd), files are of the group `B`

In this exemple there are four images in the database (and one invalid files), which stored by pairs in 2 folders. The first group `A` contains the first image of each folder, and the group `B` the second.

adding metadata to labeled file name

If image file or folder name contains specific labels (i.e. word), it an be used to attach related metadata to the respective database elements using the `$` parameter type. With the following file structure:

```
2012_03_21/
  + GEN01/
      + nitrate_001.jpg
      + nitrate_010.jpg
  + GEN02/
      + nitrate_001.jpg
      + nitrate_010.jpg

2012_03_22/
  + GEN01/
      + nitrate_001.jpg
      + nitrate_010.jpg
  + GEN02/
      + nitrate_001.jpg
      + nitrate_010.jpg
database.ini
```

Using the `ini` file:

```
[PARSING]
pattern=[date:date]/[genotype:$]/nitrate_[nitrate:int].jpg
date=%Y_%m_%d

[metadata]
xp=example

[GEN01]
name=genotype number 1
gene=GEN01

[GEN02]
name=genotype number 2
gene=GEN02
```

Then, each database element will have a `genotype` metadata with the content of field `GEN01` or `GEN02` respectively.

Note: If `[$:$]` is used then all the fields of the relative metadata group (`GEN01` or `GEN02`) are appended directly at the metadata base of the db element. Here, each element will have the respective `name` and `gene` metadata.

---

storing date metadata in file name

As in *the example above*, a date type of metadata can be given in the file or folder name. In this case, a `date` field should be present in `PARSING` that describe how the date is written, with one of the time format (see the listing of the function `strftime` for details)

Database operations

Todo: database operation

```
>>> from rhizoscan.root.pipeline import database
>>> database.filter(db, key=None, value=None, metadata=True)
>>> database.cluster(db, key, metadata=True)
>>> database.get_metadata(db)
```

2.3 Reference Guide

CHAPTER 3

Authors

Julien Diener

CHAPTER 4

ChangeLog

Use RST format for the change log or put a link to the wiki

- 08/07 : add this package layout

CHAPTER 5

License

VPlants.RhizoScan is released under a Cecill-C License.

Note: Cecill-C license is a LGPL compatible license.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

R