rainymotion Documentation

Release 0.1

Georgy Ayzel

Apr 10, 2019

Contents:

Release 0.1

Date Apr 10, 2019

rainymotion is an open Python library utilizes different models for radar-based precipitation nowcasting based on the optical flow techniques.

You can find the rainymotion source code in the corresponding Github repository.

Note: Please cite rainymotion as *Ayzel, G., Heistermann, M., and Winterrath, T.: Optical flow models as an open benchmark for radar-based precipitation nowcasting (rainymotion v0.1), Geosci. Model Dev. Discuss., https://doi.org/10.5194/gmd-2018-166, in review, 2018.*

rainymotion also provides a bunch of statistical metrics for nowcasting models evaluation (module rainymotion.metrics) and useful utils (module rainymotion.utils) for radar data preprocessing.

CHAPTER 1

Getting Started

1.1 Installation

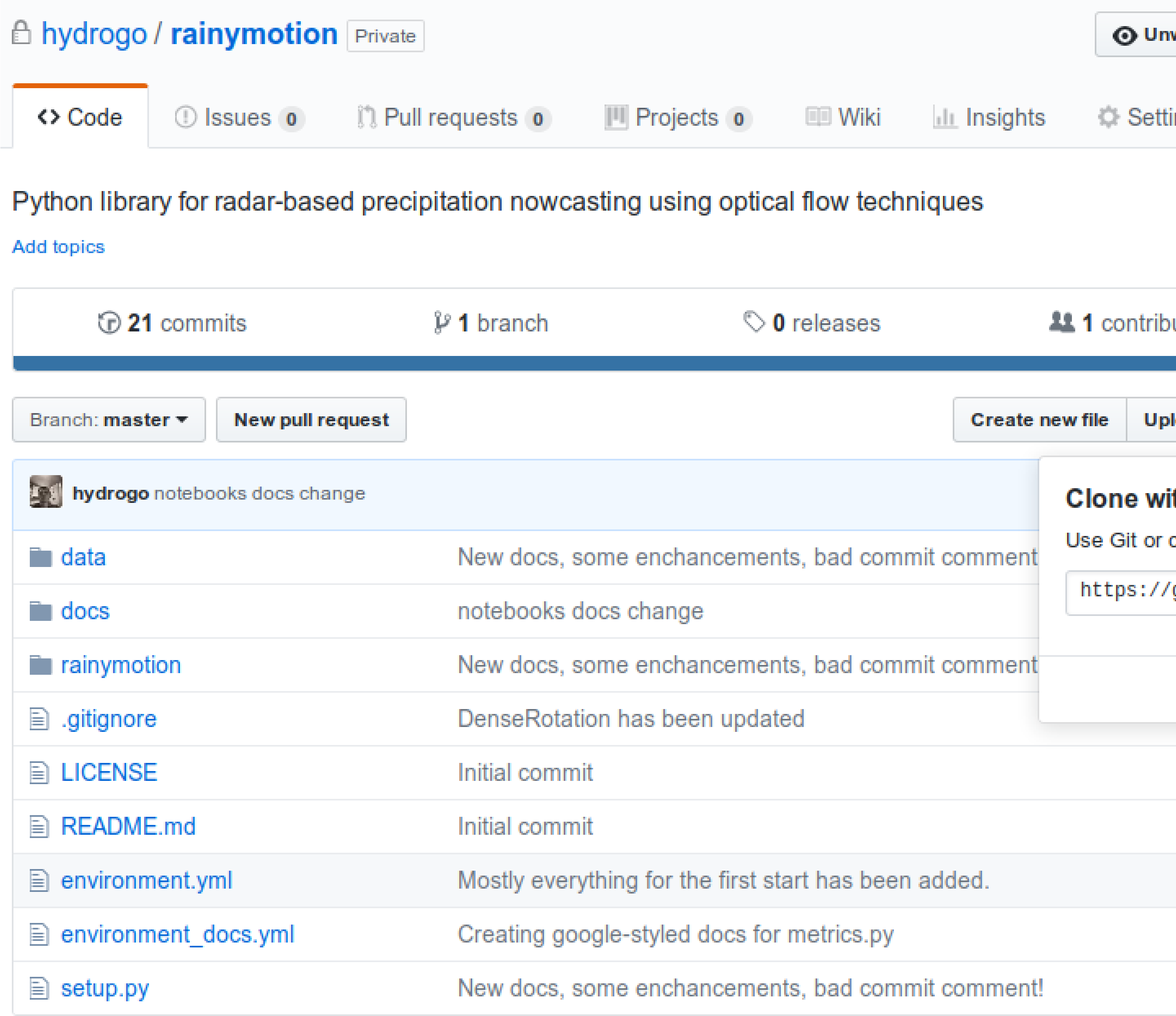
Scientific software installation always a pain. We will try all our best to describe the process clearly, but in case of any difficulties do not hesitate to contact us (see *Contacts*).

Installation process consists of multiple (but simple) **steps**:

Set up Python. Please, download and install the latest Anaconda Python3 distribution from https://www.anaconda.com/download/ which will be compatible with your OS. You can refer to conda documentation for a detailed installation instructions.

Download the source code. Unfortunately, on the current stage of development it is impossible to install `rainymotion` using `pip` or `conda` Python package managers. The only way we can offer you is to download the source code from the `rainymotion` repository on the Github on your computer. This way actually has more pros, than cons: you will have not only the code, but sample data, using examples, and documentation in one place.

To download the source code as a .zip archive you have to go to the `rainymotion` repository on the Github and press the green button “Clone or download”. Then you can extract this archive to any folder on your computer.



Python library for radar-based precipitation nowcasting using optical flow techniques

hydrogo notebooks docs change

data	New docs, some enchancements, bad commit comment
docs	notebooks docs change
rainymotion	New docs, some enchancements, bad commit comment
.gitignore	DenseRotation has been updated
LICENSE	Initial commit
README.md	Initial commit
environment.yml	Mostly everything for the first start has been added.
environment_docs.yml	Creating google-styled docs for metrics.py
setup.py	New docs, some enchancements, bad commit comment!

For users familiar with a command line and `git` there is the easier way to download the source code:

```
git clone https://github.com/hydrogo/rainymotion.git
```

Set up Python environment. It is pretty common thing in software installation that something went wrong and not only installation process failed, but other software has beeb corrupted too. To avoid such a headache, it is better to use isolated python environments. To create one, you need to enter the main `rainymotion` directory, run a command line from there, and then run:

```
conda env create -f environment.yml
```

The command above creates the isolated python environment *rainymotion* which can be activated via command:

```
# on Linux and MacOS
source activate rainymotion
```

(continues on next page)

(continued from previous page)

```

# on Windows
activate rainymotion
```

To come back to normal life without Python and radar-based precipitation nowcasting, in your command line run:

```
# on linux and macOS
source deactivate

# on windows
deactivate
```

Installing the rainymotion package. Firstly, be sure that you are in the main `rainymotion` directory, then into your command line run:

```
python setup.py install
```

Well done! You have (probably) installed the `rainymotion` library.

Check the installation. To be sure that installation process had been correctly finalized, in your command line run:

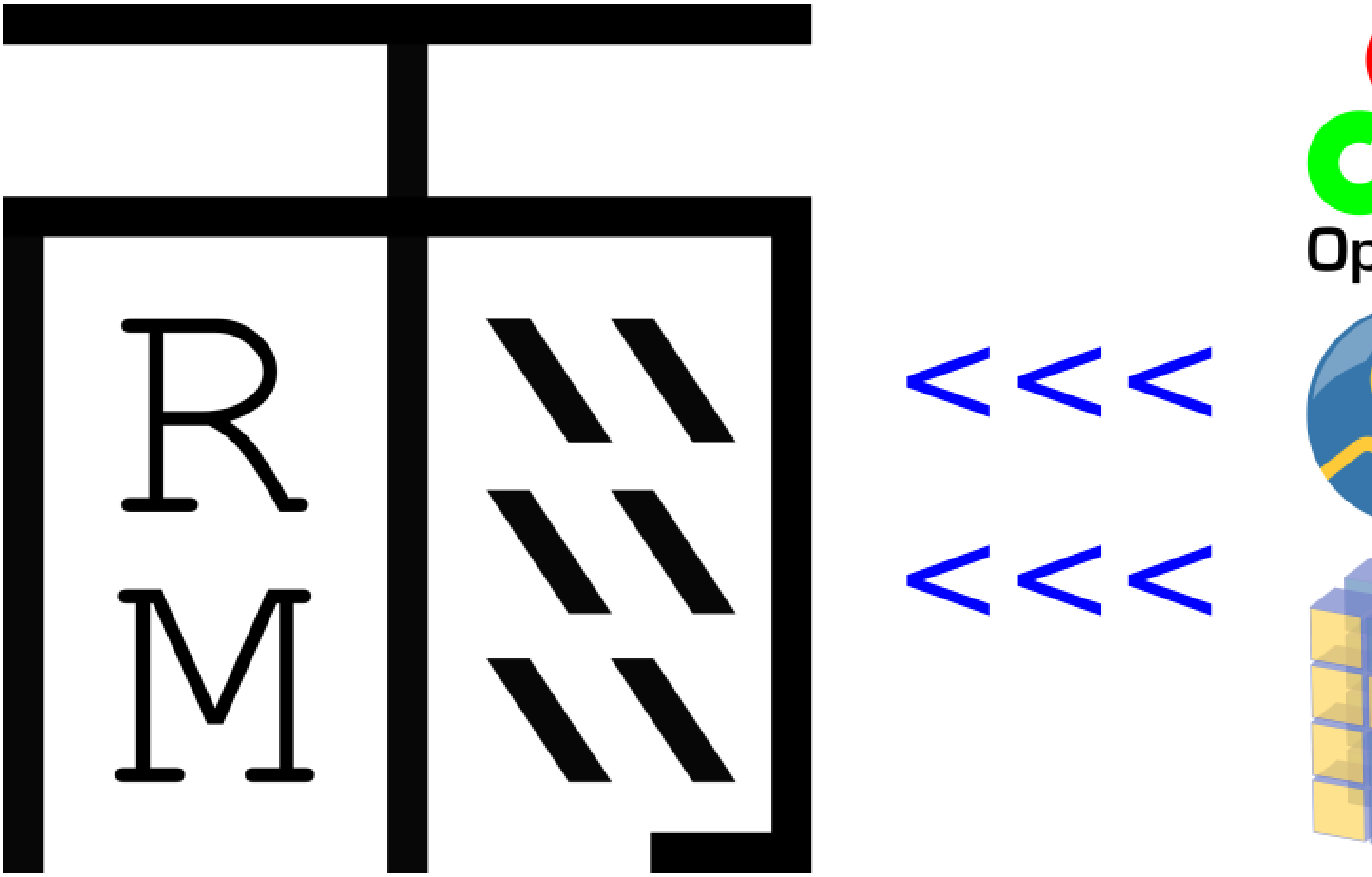
```
# 1. start Python interpreter
python

# 2. import rainymotion
>>>import rainymotion
```

If no error is raised: good job, now you have a working python environment with the `rainymotion` library inside. You are ready to nowcast something.

1.2 Quick overview

The main aim of the `rainymotion` library is to provide open and reliable models for radar-based precipitation nowcasting based on optical flow techniques.

`rainymotion` had not been writing from scratch. We incorporated the best open software solutions and provided a clue to make them work together.



There are two general groups of models we provide for precipitation nowcasting: based on a local optical flow (the Sparse group, and on a global optical flow (the Dense group).

Every model has the same structure, so the default workflow does not vary from model to model:

```
# 0. activate rainymotion environment
source activate rainymotion

# 1. start Python interpreter
python

# 2. import rainymotion models
>>>from rainymotion.models import Dense

# 3. initialize the model instance
>>>model = Dense()

# 4. transfer radar data to .input_data placeholder
>>>model.input_data = np.load("/path/to/data")

# 5. run the model to get nowcast
>>>nowcast = model.run()
```

You can find more examples in the *tutorials and examples* section.

1.3 Contacts

You can contact the team of developers in many ways:

- raise an issue in the `rainymotion` Github repository
- send an email
- send a message in Telegram

CHAPTER 2

Tutorials and Examples

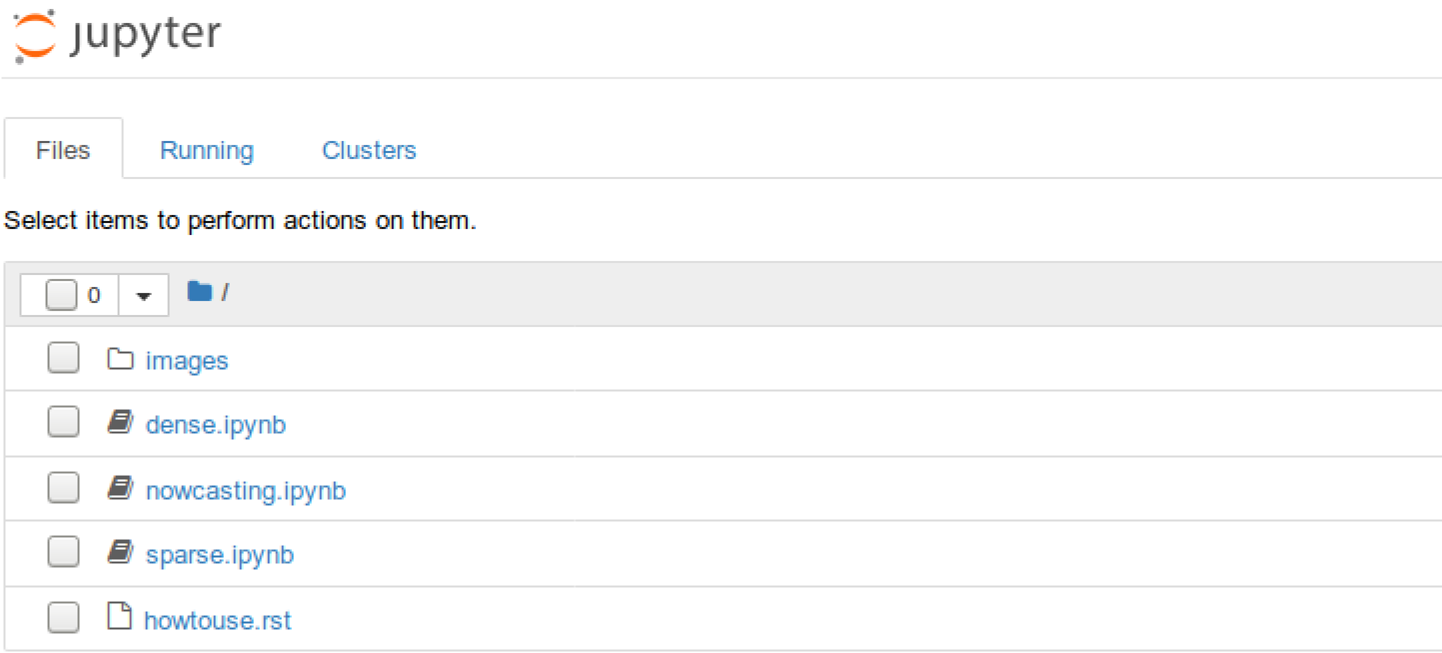
2.1 How to use?

The following tutorials and examples are stored in Jupyter notebooks. If you want to run them on your local computer, you need to:

1. Properly install rainymotion library. Follow instructions provided in the *Getting Started* section.
2. Move to the main rainymotion directory and run the command line there.
3. In the command line enter:

```
# 0. activate rainymotion environment
source activate rainymotion

# 1. change present directory to notebooks directory
cd docs/source/notebooks/

# 2. start Jupyter notebook server
jupyter notebook
```

4. This will start the Jupyter notebook server and open a new tab in your default browser.



Select items to perform actions on them.

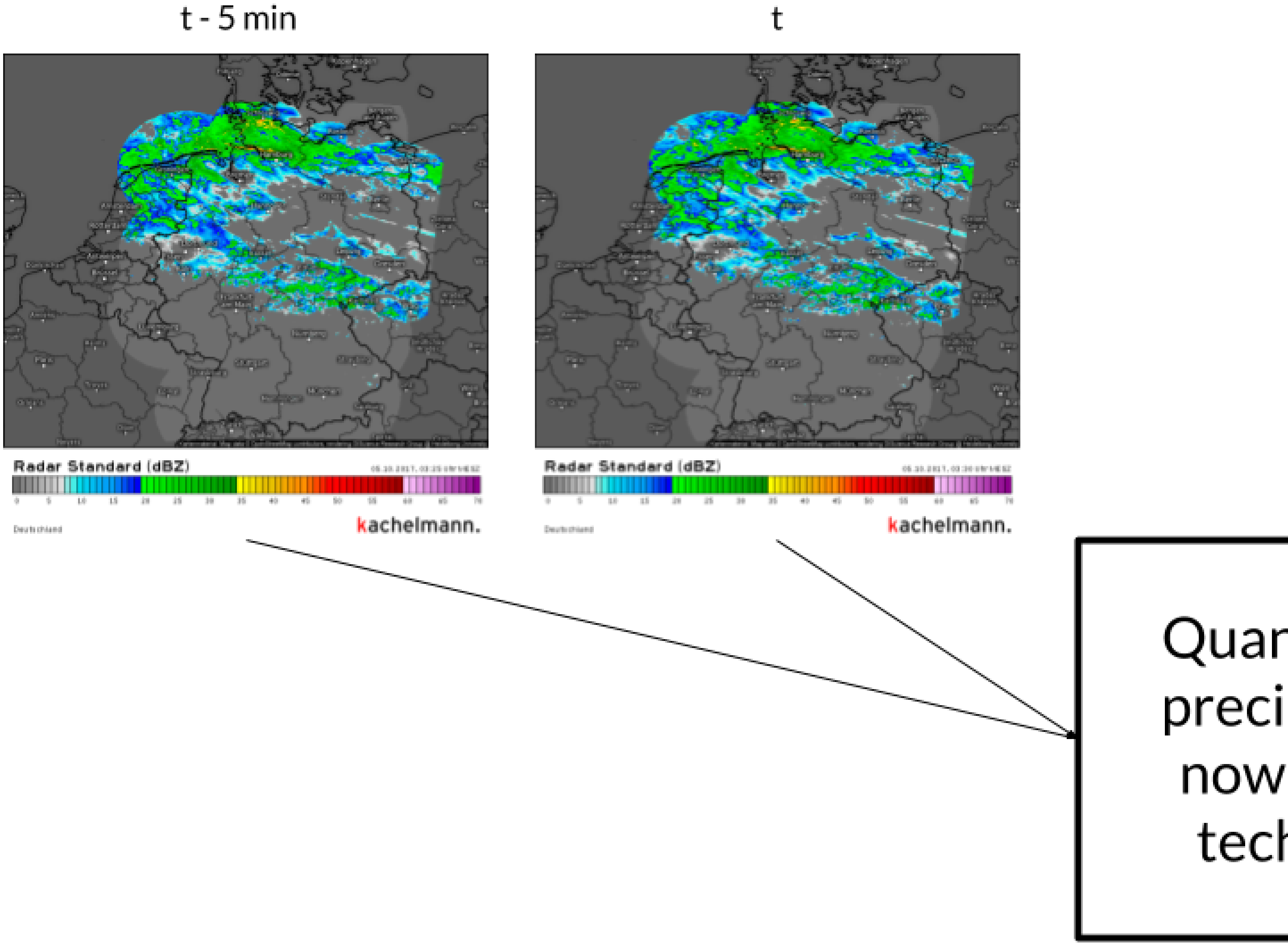
□ images
□ dense.ipynb
□ nowcasting.ipynb
□ sparse.ipynb
□ howtouse.rst

If you are not familiar with Jupyter notebooks, please, read the official documentation.

2.2 Sparse optical flow models

Severe weather conditions evolve fast, so it might be not enough to use NWP forecasts only to predict (especially local) rainfall rates correctly for an hour (or less) in advance.

General idea of nowcasting is to use series of consecutive radar images for predicting the next ones with lead times up to 2 hours and time resolution from 5 to 20 minutes.



You can refer to the best review of nowcasting techniques and its classification by J. Wilson et. al. (1998)

Here we will briefly describe and implement the models from the Sparse group of the rainymotion library:

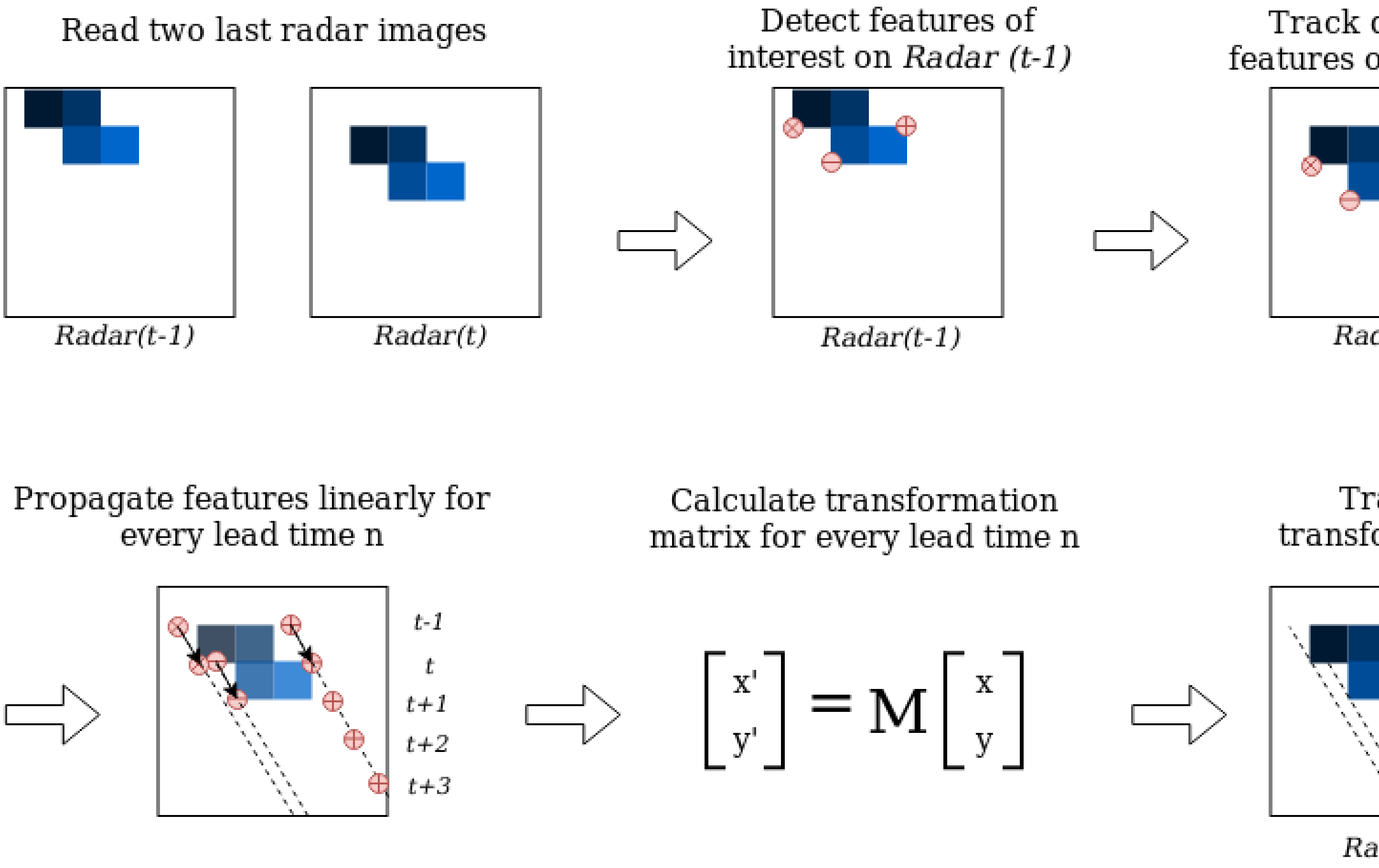
1. SparseSD model
2. Sparse model

2.2.1 The SparseSD model

The implementation of the SparseSD model can be summarized as follows:

1. Identify features in a radar image at time t-1 using the Shi-Tomasi (Shi and Tomasi, 1994) corner detector;
2. Track these features at time t using the local Lukas-Kanade (Lukas and Kanade, 1981) optical flow algorithm;
3. Linearly extrapolate the features' motion in order to predict the features' locations at each lead time n;
4. Calculate the affine transformation matrix (Schneider and Eberly, 2003) for each lead time n based on the features' locations at time t and t+n;

5. Warp the radar image at time t for each lead time n using the corresponding affine matrix, and linearly interpolate remaining discontinuities (Wolberg, 1990).


Read two last radar images
Radar(t-1)
Radar(t)
Detect features of interest on Radar (t-1)
Radar(t-1)
Track c
features o
Rad
Propagate features linearly for every lead time n
t-1
t
t+1
t+2
t+3
Calculate transformation matrix for every lead time n
x'
y'
= M
x
y
Tr
transfo
Ra

The SparseSD model usage example:

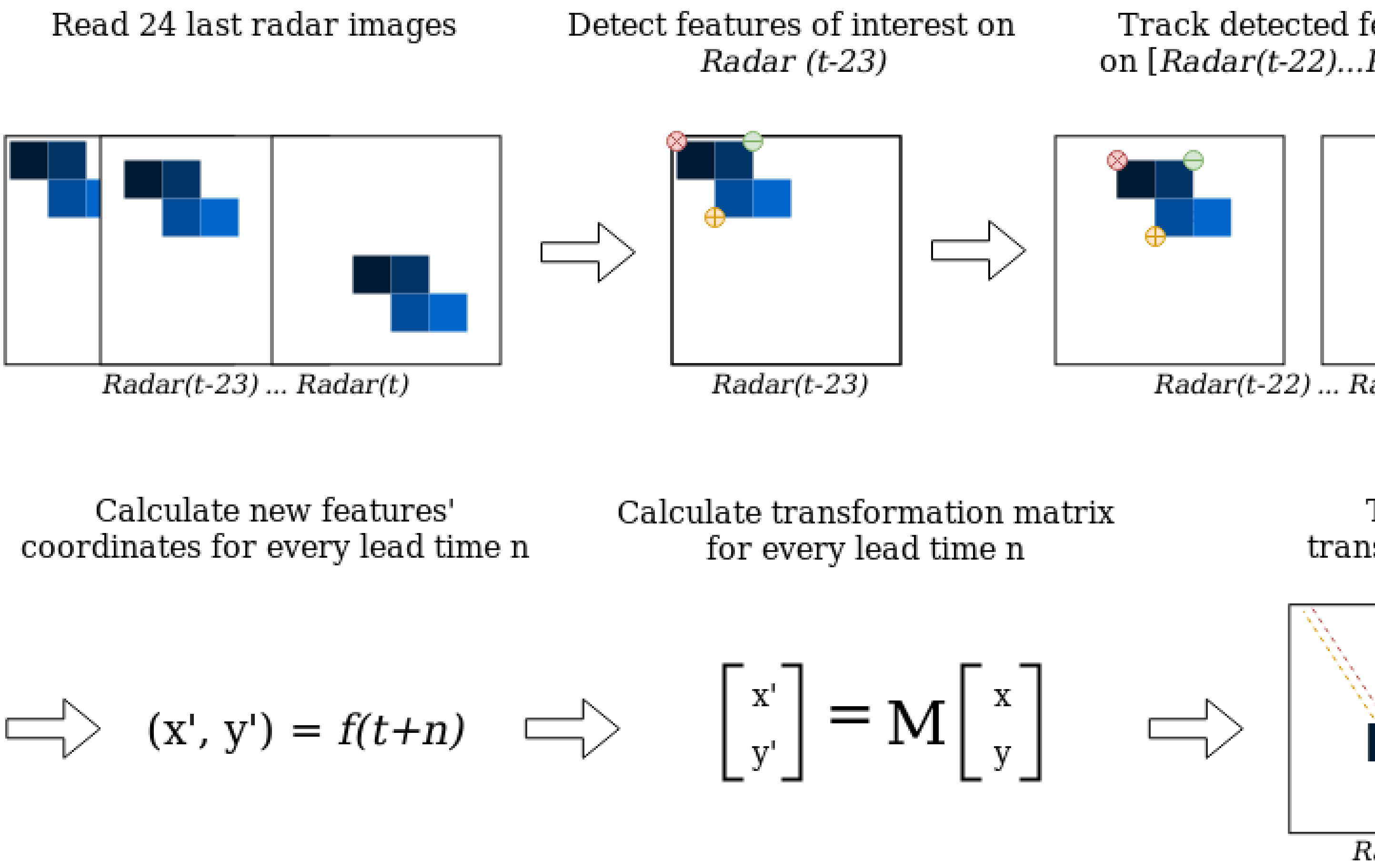
```
# import the model from the rainymotion library
from rainymotion.models import SparseSD

# initialize the model
model = SparseSD()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

2.2.2 The Sparse model

The implementation of the Sparse model can be summarized as follows:

1. Identify features on a radar image at time t-23 using the Shi-Tomasi (Shi and Tomasi, 1994) corner detector;
2. Track these features on radar images at the time from t-22 to t using the local Lukas-Kanade (Lukas and Kanade, 1981) optical flow algorithm;

3. Build linear regression models which independently parametrize changes in coordinates through time (from t-23 to t) for every successfully tracked feature;
4. Linearly extrapolate the features' motion in order to predict the features' locations at each lead time n;
5. Calculate the affine transformation matrix (Schneider and Eberly, 2003) for each lead time n based on the features' locations at time t and t+n;
6. Warp the radar image at time t for each lead time n using the corresponding affine matrix, and linearly interpolate remaining discontinuities (Wolberg, 1990).



The Sparse model usage example:

```
# import the model from the rainymotion library
from rainymotion.models import Sparse

# initialize the model
model = Sparse()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

2.3 Dense optical flow models

Here we will briefly describe and implement the models from the Dense group of the rainymotion library:

1. Dense model
2. DenseRotation model

The Dense group models' implementation can be summarized as follows:

1. Calculate a continuous displacement field using a global DIS optical flow algorithm (Kroeger et al., 2016) based on the radar images at t-1 and t;
2. Use a constant-vector or a semi-Lagrangian scheme (Fig. 1) to advect each pixel according obtained displacement (velocity) field, in one single step for each lead time t+n;
3. Use the intensity of each displaced pixel at its new location at time t+n in order to interpolate the intensity at each grid point of the original (native) radar grid (Liu et al., 2015; Zahraei et al., 2012) (Fig. 2).

References: Kroeger, T., Timofte, R., Dai, D., & Van Gool, L. (2016, October). Fast optical flow using dense inverse search. In European Conference on Computer Vision (pp. 471-488). Springer, Cham.

Fig. 1. Advection

Fig. 2. Scheme of Dense group models

2.3.1 The Dense model

The Dense model usage example:

```python
# import the model from the rainymotion library
from rainymotion.models import Dense

# initialize the model
model = Dense()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

2.3.2 The DenseRotation model

The DenseRotation model usage example:

```python
# import the model from the rainymotion library
from rainymotion.models import DenseRotation

# initialize the model
model = DenseRotation()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

2.4 Real case verification: the Braunsbach event

The "Braunsbach event" – the hazardous event which took place in Braunsbach town on 29 May 2016 and was triggered by torrential rainfall.

This event is in strong focus of the Hydrology and Climatology Group led by Axel Bronstert in the University of Potsdam and we tested all of our models on this data span for first. Read more in the paper.

Libraries we use

- rainymotion – models, metrics, pre/postprocessing
- numpy – general manipulations with arrays
- h5py – reading sample data from HDF5 format
- matplotlib – plotting

[1]:

```
# import rainymotion library
from rainymotion import models, metrics, utils

# import accompanying libraries
from collections import OrderedDict
import numpy as np
import h5py
import matplotlib.pyplot as plt

%matplotlib inline
```

[2]:

```
# import sample data
data = h5py.File("../../data/data.h5", mode="r")
# import dictionary with timestep indexes
# dictionary structure: {"t": [ [t-24, t-23,..., t-1], [t+1,...,t+12] ]}
eval_idx = np.load("../../data/eval_dict.npy").item()
```

[3]:

```
# create placeholder (or load previously calculated) results
results = h5py.File("../../data/results.h5")
```

[4]:

```
# create ground truth predictions
def ground_truth(data_instance, eval_instance, results_instance):

    results_instance.create_group("/GT/")

    for key in sorted(list(eval_instance.keys())):

        ground_truth = np.array([ data_instance[key][()] for key in eval_
↪instance[key][1] ])

        results_instance["/GT/"].create_dataset(key,
                                                 data=ground_truth,
                                                 dtype="float16",
                                                 chunks=(12,100,100),
                                                 maxshape=(12,900,900), compression=
↪"gzip")
```

If you want to run calculations by yourself, please, uncomment corresponding cells.

```
%%time
ground_truth(data, eval_idx, results)

"""
Intel Core i7
CPU times: user 34.2 s, sys: 496 ms, total: 34.7 s
Wall time: 35.1 s
"""
```

```
[5]: def persistence(data_instance, eval_instance, results_instance):

         results_instance.create_group("/Persistence/")

         for key in sorted(list(eval_instance.keys())):

             inputs = np.array([ data_instance[key][()] for key in eval_instance[key][0][-
     ↪1:] ])

             model = models.Persistence()

             model.input_data = inputs

             nowcast = model.run()

             results_instance["/Persistence/"].create_dataset(key,
                                                              data=nowcast,
                                                              dtype="float16",
                                                              chunks=(12,100,100),
                                                              maxshape=(12,900,900),
                                                              compression="gzip")
```

```
%%time
persistence(data, eval_idx, results)

"""
Intel Core i7
CPU times: user 24.8 s, sys: 104 ms, total: 24.9 s
Wall time: 24.9 s
"""
```

```
[6]: def optical_flow(data_instance, eval_instance, results_instance, model_name):

         if model_name == "Sparse":
             model = models.Sparse()

         elif model_name == "SparseSD":
             model = models.SparseSD()

         elif model_name == "Dense":
             model = models.Dense()

         elif model_name == "DenseRotation":
             model = models.DenseRotation()

         results_instance.create_group("/{}/".format(model_name))
```

(continues on next page)

(continued from previous page)

```
    for key in sorted(list(eval_instance.keys())):

        inputs = np.array([ data_instance[key][()] for key in eval_instance[key][0] ])

        model.input_data = inputs

        nowcast = model.run()

        results_instance["/{}/".format(model_name)].create_dataset(key,
                                                                   data=nowcast,
                                                                   dtype="float16",
                                                                   chunks=(12,100,
↪100),
                                                                   maxshape=(12,900,
↪900),
                                                                   compression="gzip")
```

```
%%time
optical_flow(data, eval_idx, results, "Sparse")

"""
Intel Core i7
CPU times: user 8min 34s, sys: 5min 55s, total: 14min 29s
Wall time: 5min 42s
"""
```

```
%%time
optical_flow(data, eval_idx, results, "SparseSD")

"""
Intel Core i7
CPU times: user 6min 45s, sys: 5min 53s, total: 12min 39s
Wall time: 5min 13s
"""
```

```
%%time
optical_flow(data, eval_idx, results, "Dense")

"""
Intel Core i7
CPU times: user 50min 39s, sys: 1min 16s, total: 51min 55s
Wall time: 15min 51s
"""
```

```
%%time
optical_flow(data, eval_idx, results, "DenseRotation")

"""
Intel Core i7
CPU times: user 57min 46s, sys: 1min 16s, total: 59min 3s
Wall time: 22min 56s
"""
```

[7]:

```
# load a mask which maps RY product coverage
mask = np.load("../../data/RY_mask.npy")
```

(continues on next page)

(continued from previous page)

```
mask = np.array([mask for i in range(12)])

# Verification block
def calculate_CSI(obs, sim, thresholds=[0.125, 0.250, 0.500, 1.000]):

    result = {}

    for threshold in thresholds:
        result[str(threshold)] = [metrics.CSI(obs[i], sim[i], threshold=threshold)␣
↪for i in range(obs.shape[0])]

    return result

def calculate_MAE(obs, sim):

    return [metrics.MAE(obs[i], sim[i]) for i in range(obs.shape[0])]

def calculate_metrics_dict(eval_instance, results_instance,
                           model_names=["Persistence", "Sparse", "SparseSD", "Dense",
↪"DenseRotation"]):

    metrics_dict = OrderedDict()

    for model_name in model_names:

        metrics_dict[model_name] = OrderedDict()

        for key in sorted(list(eval_instance.keys())):

            metrics_dict[model_name][key] = {model_name: {"CSI": None, "MAE": None}}

            # observed ground-truth
            o = results_instance["GT"][key][()]

            # results of nowcasting
            s = results_instance[model_name][key][()]

            # convert values from depth (mm) to intensity (mm/h)
            o = utils.depth2intensity(o)
            s = utils.depth2intensity(s)

            # mask arrays
            o = np.ma.array(o, mask=mask)
            s = np.ma.array(s, mask=mask)

            metrics_dict[model_name][key][model_name]["CSI"] = calculate_CSI(o, s)
            metrics_dict[model_name][key][model_name]["MAE"] = calculate_MAE(o, s)

    return metrics_dict
```

```
%%time
metrics_dict = calculate_metrics_dict(eval_idx, results)

"""
Intel Core i7
CPU times: user 11min 51s, sys: 16.1 s, total: 12min 7s
```

(continues on next page)

(continued from previous page)

```
Wall time: 12min 7s
"""
```

```
np.save("../../data/results_metrics.npy", metrics_dict)
```

[8]:
```
metrics_dict = np.load("../../data/results_metrics.npy").item()
```

[9]:
```
# Vizualization block
def MAE_simple_plot(metrics_dict, ax, axis=0):

    ### data preparation block ###
    event_name = "Braunsbach"
    # create a useful keys
    model_names = sorted(list(metrics_dict.keys()))

    main_keys = sorted(list(metrics_dict[model_names[0]].keys()))

    # create a holder for averaged MAE results
    mae = {model_name: None for model_name in model_names}

    for model_name in model_names:
        mae[model_name] = np.array( [metrics_dict[model_name][step][model_name]["MAE
↪"] for step in main_keys] )

    ###   ###   ###
    t = range(5, 65, 5)

    #fig, ax = plt.subplots(figsize=(12, 8))

    for model in model_names:

        data = mae[model]

        data_to_plot = np.mean(data, axis=axis)

        ax.plot(t, data_to_plot, label=model)

    ax.grid(linestyle="--")
    ax.legend(fontsize=14)

    ax.set_title("{}: {} -- {}".format(event_name.title(), main_keys[0], main_keys[-
↪1]), fontsize=16)
    ax.set_xlabel("Lead time, min", fontdict={"size": 16})
    ax.set_ylabel("MAE, mm/h", fontdict={"size": 16})

    ax.tick_params(labelsize=14)

    return ax

def CSI_simple_plot(metrics_dict, threshold, ax):

    """
    threshold should be a string from ['0.125', '0.25', '0.5', '1.0']
    """
    event_name = "Braunsbach"
```

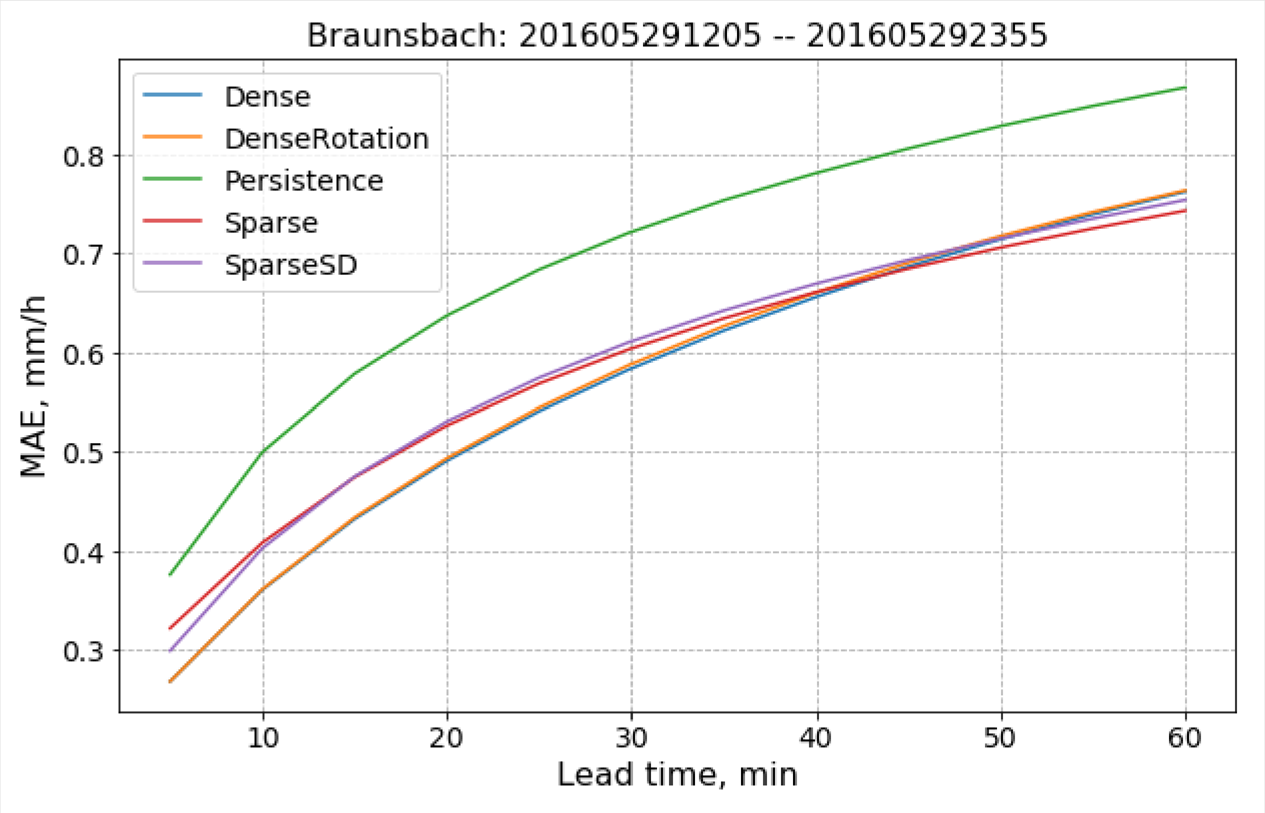
(continues on next page)

(continued from previous page)

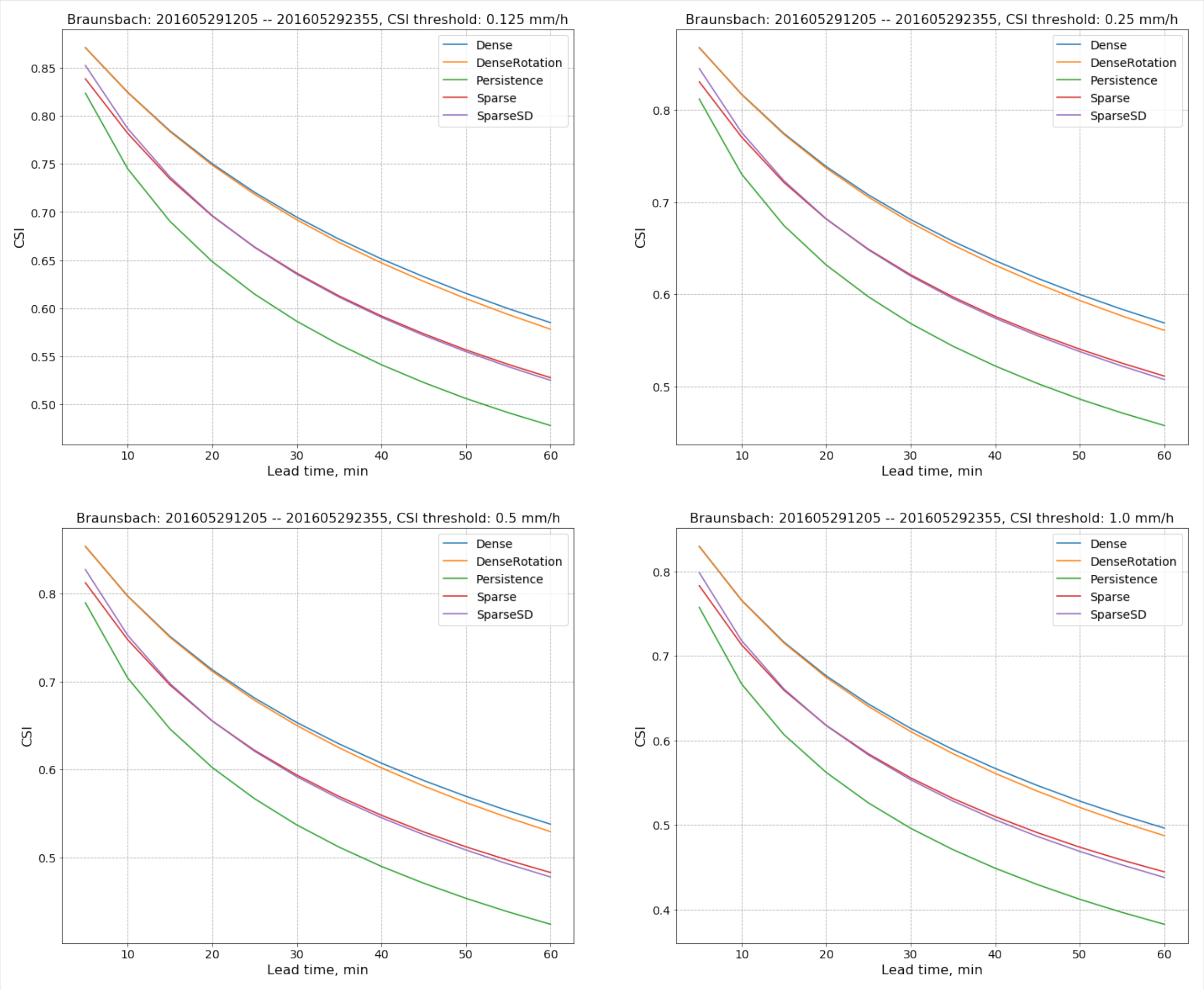
```
    # create a useful keys
    model_names = sorted(list(metrics_dict.keys()))

    main_keys = sorted(list(metrics_dict[model_names[0]].keys()))

    # create a holder for averaged MAE results
    csi = {model_name: None for model_name in model_names}

    for model_name in model_names:
        csi[model_name] = np.array( [metrics_dict[model_name][step][model_name]["CSI
↪"][threshold] for step in main_keys] )

    ###

    t = range(5, 65, 5)

    for model in model_names:

        data = csi[model]
        data_to_plot = np.mean(data, axis=0)
        ax.plot(t, data_to_plot, label=model)

    ax.grid(linestyle="--")
    ax.legend(fontsize=14)

    ax.set_title("{}: {} -- {}, CSI threshold: {} mm/h".format(event_name.title(),
                                                               main_keys[0],
                                                               main_keys[-1],
                                                               threshold),
                 fontsize=16)

    ax.set_xlabel("Lead time, min", fontdict={"size": 16})
    ax.set_ylabel("CSI", fontdict={"size": 16})

    ax.tick_params(labelsize=14)

    return ax
```

[10]:

```
fig, ax = plt.subplots(figsize=(10,6))
ax = MAE_simple_plot(metrics_dict, ax)
```



```
[11]: fig, axes = plt.subplots(2, 2, figsize=(24,20))
      axes = axes.ravel()
      for i, threshold in enumerate(['0.125', '0.25', '0.5', '1.0']):
          axes[i] = CSI_simple_plot(metrics_dict, threshold, axes[i])
```


Braunsbach: 201605291205 -- 201605292355, CSI threshold: 0.125 mm/h
CSI
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
10
20
30
40
50
60
Lead time, min
Dense
DenseRotation
Persistence
Sparse
SparseSD
Braunsbach: 201605291205 -- 201605292355, CSI threshold: 0.25 mm/h
CSI
0.8
0.7
0.6
0.5
10
20
30
40
50
60
Lead time, min
Dense
DenseRotation
Persistence
Sparse
SparseSD
Braunsbach: 201605291205 -- 201605292355, CSI threshold: 0.5 mm/h
CSI
0.8
0.7
0.6
0.5
10
20
30
40
50
60
Lead time, min
Dense
DenseRotation
Persistence
Sparse
SparseSD
Braunsbach: 201605291205 -- 201605292355, CSI threshold: 1.0 mm/h
CSI
0.8
0.7
0.6
0.5
0.4
10
20
30
40
50
60
Lead time, min
Dense
DenseRotation
Persistence
Sparse
SparseSD

CHAPTER 3

Library Reference

3.1 Models

Documentation for all precipitation nowcasting models implemented in `rainymotion.models` module.

For the detailed model description please refer to our paper:

Note: *Ayzel, G., Heistermann, M., and Winterrath, T.: Optical flow models as an open benchmark for radar-based precipitation nowcasting (rainymotion v0.1), Geosci. Model Dev. Discuss., https://doi.org/10.5194/gmd-2018-166, in review, 2018.*

3.1.1 The Sparse group

The central idea around this model group is to identify distinct features in a radar image that are suitable for tracking. In this context, a "feature" is defined as a distinct point ("corner") with a sharp gradient of rainfall intensity. Inside this group, we developed two models that slightly differ with regard to both tracking and extrapolation.

The SparseSD model

The first model (SparseSD, SD stands for Single Delta) uses only the two most recent radar images for identifying, tracking, and extrapolating features. Assuming that *t* denotes both the nowcast issue time and the time of the most recent radar image, the implementation can be summarized as follows (Fig. 1):

- Identify features in a radar image at time *t-1* using the Shi–Tomasi corner detector. This detector determines the most prominent corners in the image based on the calculation of the corner quality measure;
- Track these features at time *t* using the local Lucas–Kanade optical flow algorithm. This algorithm tries to identify the location of feature we previously identified on the radar image at time *t-1* on the radar image at time *t* based on the solving a set of optical flow equations in the local feature neighborhood using the least-squares approach;

- Linearly extrapolate the features' motion in order to predict the features' locations at each lead time *n*;
- Calculate the affine transformation matrix for each lead time *n* based on the all identified features' locations at time *t* and *t+n* using the least-squares approach. This matrix uniquely identifies the required transformation of the last observed radar image at time *t* to obtain nowcasted images at times *t+1*. . . *t+n* providing the smallest possible difference between identified and extrapolated features' locations;
- Warp the radar image at time *t* for each lead time *n* using the corresponding affine matrix, and linearly interpolate remaining discontinuities. Warping procedure uniquely transforms each pixel location of the radar image at time *t* to its new location on the corresponding nowcasted radar images at times *t+1*. . . *t+n* and then performs linear interpolation procedure in order to interpolate nowcasted pixels' intensities to the original grid of the radar image at time *t*.

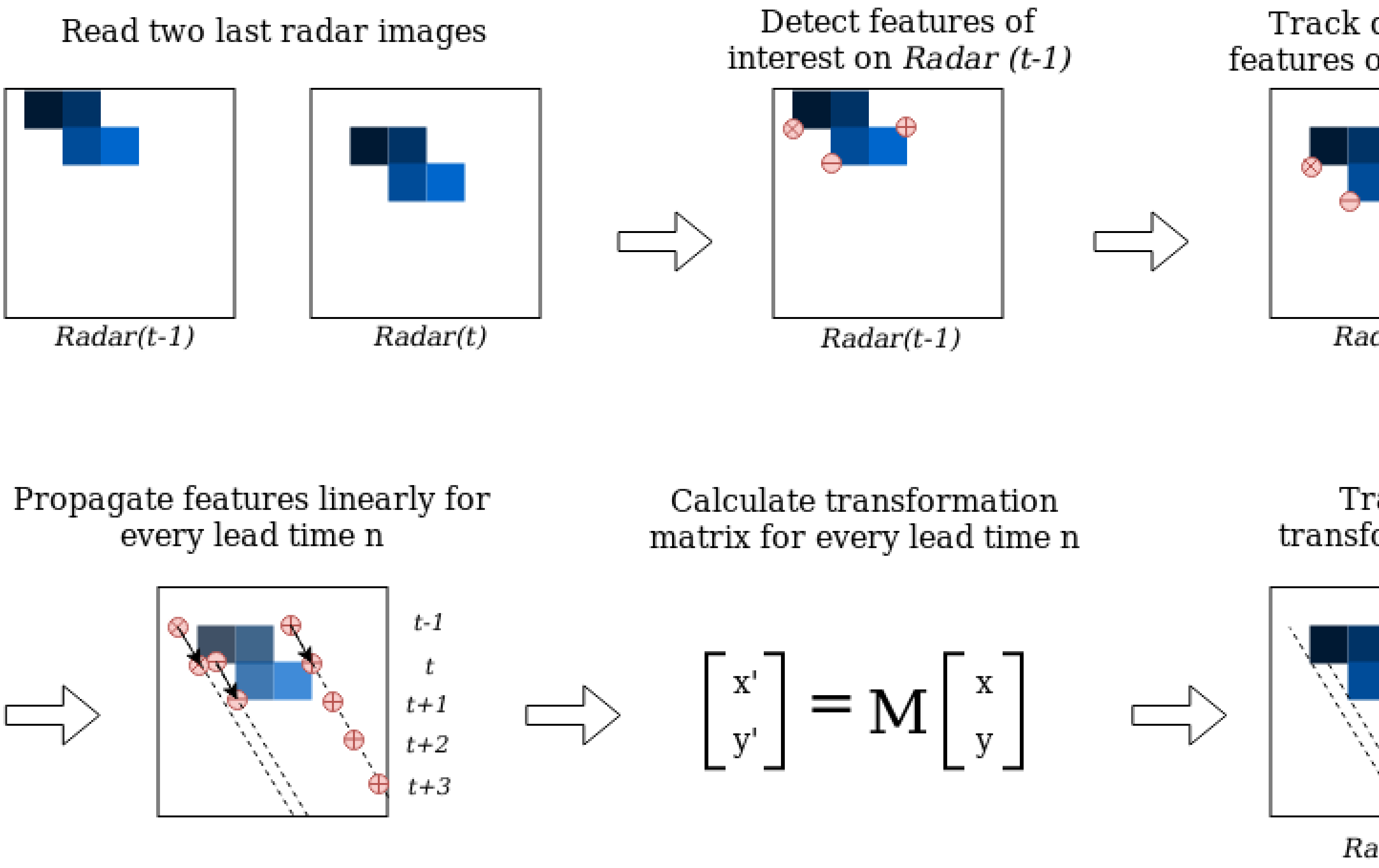


Fig. 1: Figure 1. A visual representation of the SparseSD model routine.

The SparseSD model usage example:

```
# import the model from the rainymotion library
from rainymotion.models import SparseSD

# initialize the model
model = SparseSD()

# upload data to the model instance
model.input_data = np.load("/path/to/data")
```

(continues on next page)

(continued from previous page)

```
# run the model with default parameters
nowcast = model.run()
```

See also:

Tutorials and Examples.

The Sparse model

The Sparse model uses the 24 most recent radar images, and we consider here only features that are persistent over the whole period (of 24 timesteps) for capturing the most steady movement. Its implementation can be summarized as follows (Fig. 2):

- Identify features on a radar image at time *t-23* using the Shi–Tomasi corner detector;
- Track these features on radar images at the time from *t-22* to *t* using the local Lucas–Kanade optical flow algorithm;
- Build linear regression models which independently parametrize changes in coordinates through time (from *t-23* to *t*) for every successfully tracked feature;
- Continue with steps 3-5 of the SparseSD model routine.

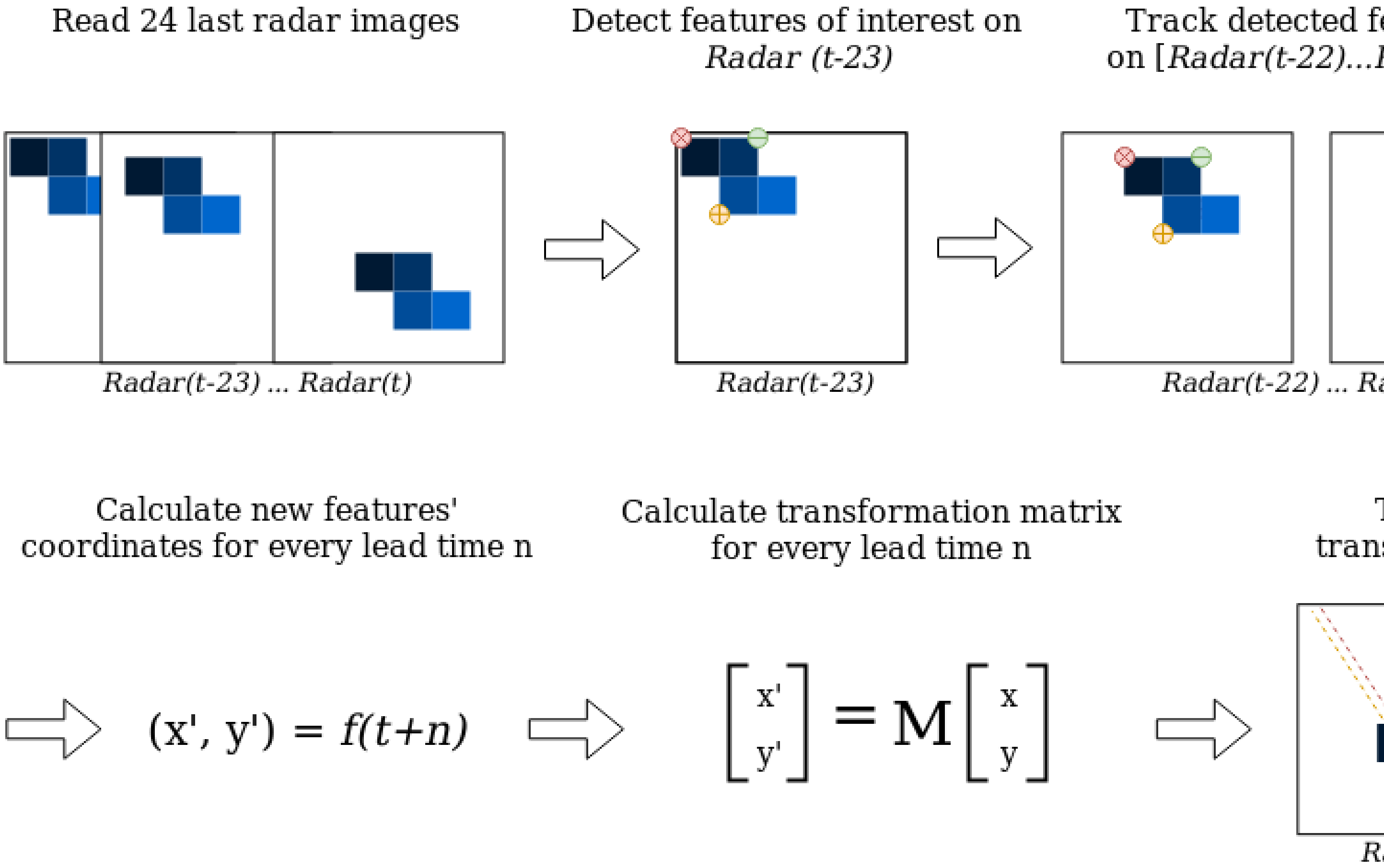

Read 24 last radar images
Detect features of interest on Radar (t-23)
Track detected fe
on [Radar(t-22)...I
Radar(t-23) ... Radar(t)
Radar(t-23)
Radar(t-22) ... Ra
Calculate new features' coordinates for every lead time n
Calculate transformation matrix for every lead time n
T
trans
(x', y') = f(t+n)
x'
y'
= M
x
y
R

Fig. 2: Figure 2. A visual representation of the Sparse model routine.

The Sparse model usage example:

```
# import the model from the rainymotion library
from rainymotion.models import Sparse

# initialize the model
model = Sparse()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

See also:

Tutorials and Examples.

3.1.2 The Dense group

The Dense group of models uses the Dense Inverse Search algorithm (DIS) which allows us to explicitly estimate the velocity of each image pixel based on an analysis of two consecutive radar images.

The two models in this group differ only with regard to the extrapolation (or advection) step. The first model (the Dense) uses a constant-vector advection scheme, while the second model (the DenseRotation) uses a semi-Lagrangian advection scheme (Fig. 3).

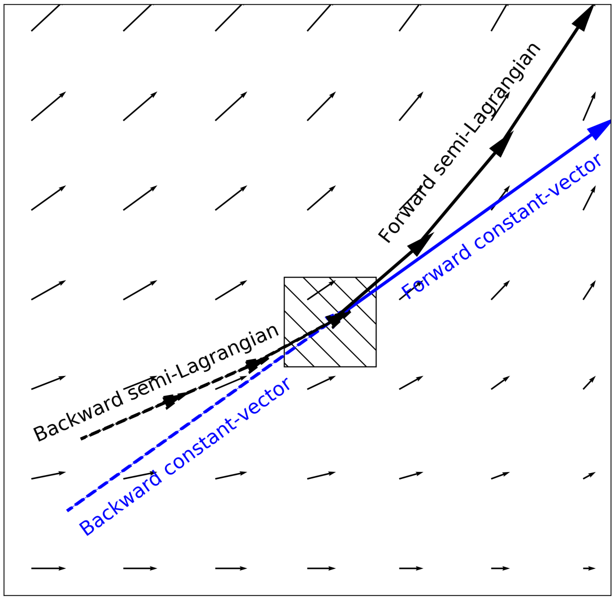


Fig. 3: Figure 3. Advection schemes representation.

Both the Dense and DenseRotation models utilize a linear interpolation procedure (we use Inverse Distance Weightning approach by default) in order to interpolate advected rainfall intensities at their predicted locations to the original radar grid (Fig. 4).

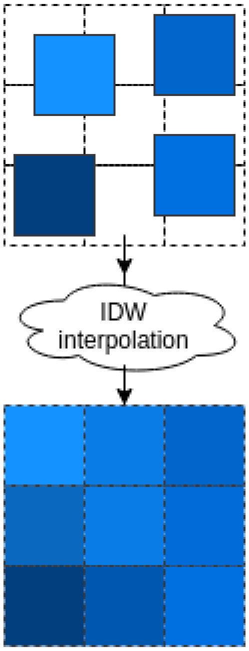


Fig. 4: Figure 4. Interpolation of the advected pixels

The Dense model

The Dense model implementation can be summarized as follows:

- Calculate a continuous displacement field using a global DIS optical flow algorithm based on the radar images at time *t-1* and *t*;
- Use a backward constant-vector approach to extrapolate (advect) each pixel according to the obtained displacement (velocity) field, in one single step for each lead time *t+n*;
- As a result of the advection step, we basically obtain an irregular point cloud that consists of the original radar pixels displaced from their original location. We use the intensity of each displaced pixel at its predicted location at time *t+n* in order to interpolate the intensity at each grid point of the original (native) radar grid using the inverse distance weighting interpolation technique (Fig. 4).

The Dense model usage example:

```
# import the model from the rainymotion library
from rainymotion.models import Dense

# initialize the model
model = Dense()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

See also:

Tutorials and Examples.

The DenseRotation model

The routine for the DenseRotation model is almost the same as for the Dense model, except differences in advection approach (the second step of the Dense model routine), which can be summarized as follows:

- Instead of using the backward constant-vector approach, we use the backward semi-Lagrangian approach to extrapolate (advect) each pixel according to the obtained displacement (velocity) field, in one single step for each lead time *t+n*. For the semi-Lagrangian scheme, we update the velocity of displaced pixels at each prediction time step by implementing a linear interpolation of obtained displacement field at time *t* to displaced pixels' locations at this (current) time step.

The DenseRoration model usage example:

```
# import the model from the rainymotion library
from rainymotion.models import DenseRotation

# initialize the model
model = DenseRotation()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

See also:

Tutorials and Examples.

3.1.3 The Eulerian Persistence

The (trivial) benchmark model of Eulerian persistence assumes that for any lead time *n*, the precipitation field is the same as for time *t*.

The Persistence model usage example:

```
# import the model from the rainymotion library
from rainymotion.models import Persistence

# initialize the model
model = Persistence()

# upload data to the model instance
model.input_data = np.load("/path/to/data")

# run the model with default parameters
nowcast = model.run()
```

See also:

Tutorials and Examples.

3.2 Metrics

The `rainymotion` library provides the extensive list of goodness-of-fit statistical metrics to evaluate nowcasting models performance.

Metric	Description
Regression	
R	Correlation coefficient
R2	Coefficient of determination
RMSE	Root mean squared error
MAE	Mean absolute error
QPN specific	
CSI	Critical Success Index
FAR	False Alarm Rate
POD	Probability Of Detection
HSS	Heidke Skill Score
ETS	Equitable Threat Score
BSS	Brier Skill Score
ML specific	
ACC	Accuracy
precision	Precision
recall	Recall
FSC	F1-score
MCC	Matthews Correlation Coefficient

You can easily use any metric for verification of your nowcasts:

```
# import the specific metric from the rainymotion library
from rainymotion.metrics import CSI

# read your observations and simulations
obs = np.load("/path/to/observations")
sim = np.load("/path/to/simulations")

# calculate the corresponding metric
csi = CSI(obs, sim, threshold=1.0)
```

See also:

Tutorials and Examples.

3.3 Utils

The `rainymotion` library provides some useful utils to help user with a data preprocessing workflow which is usually needed to perform radar-based precipitation nowcasting. At the moment, we have utils that only deal with the RY radar product by DWD. By the way, you can use such utils as an example to construct your own data preprocessing pipeline.

Function	Description
depth2intensity	Convert rainfall depth (in mm) to rainfall intensity (mm/h)
intensity2depth	Convert rainfall intensity (mm/h) back to rainfall depth (mm)
RYScaler	Scale RY data from mm (in float64) to brightness (in uint8)
inv_RYScaler	Scale brightness (in uint8) back to RY data (in mm).

See also:

Tutorials and Examples for how to use `rainymotion.utils` in the nowcasting workflow.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search