pysteps Reference

Release 1.3.1

PySteps developers

May 18, 2020

Contents

1 Documentation 3

2 Contents 5

Bibliography 223

Index 225

Pysteps is a community-driven initiative for developing and maintaining an easy to use, modular, free and open source Python framework for short-term ensemble prediction systems.

The focus is on probabilistic nowcasting of radar precipitation fields, but pysteps is designed to allow a wider range of uses.

Pysteps is actively developed on GitHub, while a more thorough description of pysteps is available in the pysteps reference publication:

Pulkkinen, S., D. Nerini, A. Perez Hortal, C. Velasco-Forero, U. Germann, A. Seed, and L. Foresti, 2019: Pysteps: an open-source Python library for probabilistic precipitation nowcasting (v1.0). *Geosci. Model Dev.*, **12 (10)**, 4185–4219, doi:10.5194/gmd-12-4185-2019. [source]

Note: Use pysteps to run your first extrapolation nowcast in Google Colab with this interactive notebook.

CHAPTER 1

Documentation

The documentation is separated in three big branches, intended for different audiences.

1.1 User guide

This section is intended for new pysteps users. It provides an introductory overview to the pysteps package, explains how to install it and make use of the most important features.

1.2 pySTEPS reference

Comprehensive description of all the modules and functions available in pysteps.

1.3 pySTEPS developer guide

Resources and guidelines for pysteps developers and contributors.

CHAPTER 2

Contents

2.1 User guide

This guide gives an introductory overview of the pySTEPS package. It explains how to install and make use of the most important features.

For detailed reference documentation of the modules and functions available in the package, see the *pySTEPS reference*.

2.1.1 Installing pysteps

Dependencies

The pysteps package needs the following dependencies

- python >=3.6
- jsonschema
- matplotlib
- netCDF4
- numpy
- opencv
- pillow
- pyproj
- scipy

Additionally, the following packages can be installed for better computational efficiency:

- dask and toolz (for code parallelization)
- pyfftw (for faster FFT computation)

Other optional dependencies include:

- cartopy (for geo-referenced visualization)

- h5py (for importing HDF5 data)
- pygrib (for importing MRMS data)
- pywavelets (for intensity-scale verification)

Anaconda install (recommended)

Conda is an open-source package management system and environment management system that runs on Windows, macOS, and Linux. Conda quickly installs, runs, and updates packages and their dependencies. It also allows you to easily create, save, load, or switch between different environments on your local computer.

Since version 1.0, pySTEPS is available in conda-forge, a community-driven package repository for anaconda.

There are two installation alternatives using anaconda: install pySTEPS in a pre-existing environment or install it new environment.

New environment

In a terminal, to create a new conda environment and install pySTEPS, run:

```
$ conda create -n pysteps
$ source activate pysteps
```

This will create and activate the new python environment. The next step is to add the conda-forge channel where the pySTEPS package is located:

```
$ conda config --env --prepend channels conda-forge
```

Let's set this channel as the priority one:

```
$ conda config --env --set channel_priority strict
```

The latter step is not strictly necessary but is recommended since the conda-forge and the default Anaconda channels are not 100% compatible.

Finally, to install pySTEPS and all its dependencies run:

```
$ conda install pysteps
```

Install from source

The recommended way to install pysteps from the source is using *pip* to adhere to the [PEP517 standards](https://www.python.org/dev/peps/pep-0517/). Using *pip* instead of *setup.py* guarantees that all the package dependencies are properly handled during the installation process.

OSX users

pySTEPS uses Cython extensions that need to be compiled with multi-threading support enabled. The default Apple Clang compiler does not support OpenMP. Hence, using the default compiler would have disabled multi-threading and may raise the following error during the installation:

```
clang: error: unsupported option '-fopenmp'
error: command 'gcc' failed with exit status 1
```

To solve this issue, obtain the latest gcc version with Homebrew that has multi-threading enabled:

```
brew install gcc
```

To make sure that the installer uses the homebrew's gcc, export the following environmental variables in the terminal (supposing that gcc version 8 was installed):

```
export CC=gcc-8
export CXX=g++-8
```

First, check that the homebrew's gcc is detected:

```
which gcc-8
```

This should point to the homebrew's gcc installation.

Under certain circumstances, Homebrew does not add the symbolic links for the gcc executables under /usr/local/bin. If that is the case, specify the CC and CCX variables using the full path to the homebrew installation. For example:

```
export CC=/usr/local/Cellar/gcc/8.3.0/bin/gcc-8
export CXX=/usr/local/Cellar/gcc/8.3.0/bin/g++-8
```

Then, you can continue with the normal installation procedure described next.

Installation

The latest pysteps version in the repository can be installed using pip by simply running in a terminal:

```
pip install git+https://github.com/pySTEPS/pysteps
```

Or, from a local copy of the repo (global installation):

```
git clone https://github.com/pySTEPS/pysteps
cd pysteps
pip install .
```

The above commands install the latest version of the **master** branch, which is continuously under development.

Development mode

The latest version can also be installed in Development Mode, i.e., in such a way that the project appears to be installed, but yet is still editable from the source tree:

```
pip install -e <path to local pysteps repo>
```

Setting up the user-defined configuration file

The pysteps package allows the users to customize the default settings and configuration. The configuration parameters used by default are loaded from a user-defined JSON file and then stored in the **pysteps.rcparams**, a dictionary-like object that can be accessed as attributes or as items.

The pySTEPS configuration file (pystepsrc)

The pysteps package allows the users to customize the default settings and configuration. The configuration parameters used by default are loaded from a user-defined JSON file and then stored in *pysteps.rcparams*, a dictionary-like object that can be accessed as attributes or as items. For example, the default parameters can be obtained using any of the following ways:

```
import pysteps

# Retrieve the colorscale for plots
colorscale = pysteps.rcparams['plot']['colorscale']
colorscale = pysteps.rcparams.plot.colorscale

# Retrieve the the root directory of the fmi data
```

(continues on next page)

(continued from previous page)

```
pysteps.rcparams['data_sources']['fmi']['root_path']
pysteps.rcparams.data_sources.fmi.root_path
```

A less wordy alternative:

```
from pysteps import rcparams
colorscale = rcparams['plot']['colorscale']
colorscale = rcparams.plot.colorscale

fmi_root_path = rcparams['data_sources']['fmi']['root_path']
fmi_root_path = rcparams.data_sources.fmi.root_path
```

Configuration file lookup

When the pysteps package imported, it looks for **pystepsrc** file in the following order:

- **$PWD/pystepsrc** : Looks for the file in the current directory
- **$PYSTEPSRC** : If the system variable $PYSTEPSRC is defined and it points to a file, it is used.
- **$PYSTEPSRC/pystepsrc** : If $PYSTEPSRC points to a directory, it looks for the pystepsrc file inside that directory.
- **$HOME/.pysteps/pystepsrc** (Unix and Mac OS X) : If the system variable $HOME is defined, it looks for the configuration file in this path.
- **%USERPROFILE%\pysteps\pystepsrc** (Windows only): It looks for the configuration file in the pysteps directory located user's home directory (indicated by the %USERPROFILE% system variable).
- Lastly, it looks inside the library in *pysteps\pystepsrc* for a system-defined copy.

The recommended method to setup the configuration files is to edit a copy of the default **pystepsrc** file that is distributed with the package and place that copy inside the user home folder. See the instructions below.

Setting up the user-defined configuration file

Linux and OSX users

For Linux and OSX users, the recommended way to customize the pysteps configuration is placing the pystepsrc parameters file in the users home folder ${HOME} in the following path: **${HOME}/.pysteps/pystepsrc**

To steps to setup up the configuration file in the home directory first, we need to create the directory if it does not exist. In a terminal, run:

```
$ mkdir -p ${HOME}/.pysteps
```

The next step is to find the location of the library's default pystepsrc file. When we import pysteps in a python interpreter, the configuration file loaded is shown:

```
import pysteps
"Pysteps configuration file found at: /path/to/pysteps/library/pystepsrc"
```

Then we copy the library's default configuration file to that directory:

```
$ cp /path/to/pysteps/library/pystepsrc ${HOME}/.pysteps/pystepsrc
```

Edit the file with the text editor of your preference and change the default configurations with your preferences.

Finally, check that the correct configuration file is loaded by the library:

```
import pysteps
"Pysteps configuration file found at: /home/user_name/.pysteps/pystepsrc"
```

Windows

For windows users, the recommended way to customize the pySTEPS configuration is placing the pystepsrc parameters file in the users' folder (defined in the %USERPROFILE% environment variable) in the following path: **%USERPROFILE%\pysteps\pystepsrc**

To setup up the configuration file in the home directory first, we need to create the directory if it does not exist. In a **windows terminal**, run:

```
$ mkdir %USERPROFILE%\pysteps
```

Important

It was reported that the %USERPROFILE% variable may be interpreted as an string literal when the anaconda terminal is used. This will result in a '%USERPROFILE%' folder being created in the current working directory instead of the desired pysteps folder in the user's home. If that is the case, use the explicit path to your home folder instead of *%USERPROFILE%*. For example:

```
$ mkdir C:\Users\your_username\pysteps
```

The next step is to find the location of the library's default pystepsrc file. When we import pysteps in a python interpreter, the configuration file loaded is shown:

```
import pysteps
"Pysteps configuration file found at: C:\path\to\pysteps\library\pystepsrc"
```

Then we copy the library's default configuration file to that directory:

```
$ copy C:\path\to\pysteps\library\pystepsrc %USERPROFILE%\pysteps\pystepsrc
```

Edit the file with the text editor of your preference and change the default configurations with your preferences.

Finally, check that the correct configuration file is loaded by the library:

```
import pysteps
"Pysteps configuration file found at: C:\User\Profile\.pysteps\pystepsrc"
```

More

Example of pystepsrc file

Below you can find the default pystepsrc file. The lines starting with "//" are comments and they are ignored.

```
// pysteps configuration
{
    // "silent_import" : whether to suppress the initial pysteps message
    "silent_import": false,
    "outputs": {
        // path_outputs : path where to save results (figures, forecasts, etc)
        "path_outputs": "./"
    },
    "plot": {
        // "motion_plot" : "streamplot" or "quiver"
        "motion_plot": "quiver",
        // "colorscale" :  "BOM-RF3", "pysteps" or "STEPS-BE"
```

(continues on next page)

(continued from previous page)

```
        "colorscale": "pysteps"
    },
    "data_sources": {
        "bom": {
            "root_path": "./radar/bom",
            "path_fmt": "prcp-cscn/2/%Y/%m/%d",
            "fn_pattern": "2_%Y%m%d_%H%M00.prcp-cscn",
            "fn_ext": "nc",
            "importer": "bom_rf3",
            "timestep": 6,
            "importer_kwargs": {
                "gzipped": true
            }
        },
        "fmi": {
            "root_path": "./radar/fmi",
            "path_fmt": "%Y%m%d",
            "fn_pattern": "%Y%m%d%H%M_fmi.radar.composite.lowest_FIN_SUOMI1",
            "fn_ext": "pgm.gz",
            "importer": "fmi_pgm",
            "timestep": 5,
            "importer_kwargs": {
                "gzipped": true
            }
        },
        "mch": {
            "root_path": "./radar/mch",
            "path_fmt": "%Y%m%d",
            "fn_pattern": "AQC%y%j%H%M?_00005.801",
            "fn_ext": "gif",
            "importer": "mch_gif",
            "timestep": 5,
            "importer_kwargs": {
                "product": "AQC",
                "unit": "mm",
                "accutime": 5
            }
        },
        "opera": {
            "root_path": "./radar/OPERA",
            "path_fmt": "%Y%m%d",
            "fn_pattern": "T_PAAH21_C_EUOC_%Y%m%d%H%M%S",
            "fn_ext": "hdf",
            "importer": "opera_hdf5",
            "timestep": 15,
            "importer_kwargs": {}
        },
        "knmi": {
            "root_path": "./radar/KNMI",
            "path_fmt": "%Y/%m",
            "fn_pattern": "RAD_NL25_RAP_5min_%Y%m%d%H%M",
            "fn_ext": "h5",
            "importer": "knmi_hdf5",
            "timestep": 5,
            "importer_kwargs": {
                "accutime": 5,
                "qty": "ACRR",
                "pixelsize": 1000.0
            }
        },
        "saf": {
```

(continues on next page)

(continued from previous page)

```
            "root_path": "./saf",
            "path_fmt": "%Y%m%d/CRR",
            "fn_pattern": "S_NWC_CRR_MSG4_Europe-VISIR_%Y%m%dT%H%M00Z",
            "fn_ext": "nc",
            "importer": "saf_crri",
            "timestep": 15,
            "importer_kwargs": {
                "gzipped": true
            }
        }
    }
}
```

2.1.2 Installing the Example data

The examples scripts in the user guide, as well as the pySTEPS build-in tests, use the example radar data available in a separate repository: pysteps-data.

The easiest way to install the example data is by using the *download_pysteps_data()* and *create_default_pystepsrc()* functions from the *pysteps.datasets* module.

Installation using the datasets module

Below is a snippet code that can be used to install can configure *pystepsrc* file to point to that example data.

In the example below, the example data is placed in the user's home folder under the **pysteps_data** directory. It also creates a default configuration file that points to the downloaded data and places it in the $HOME/.pysteps (Unix and Mac OS X) or $USERPROFILE/pysteps (Windows). This is one of the default locations where pysteps looks for the configuration file (see *Configuration file lookup* for more information).

```
import os

# Import the helper functions
from pysteps.datasets import download_pysteps_data, create_default_pystepsrc

# In this example we will place it in the user's home folder on the
# `pysteps_data` folder.
home_dir = os.path.expanduser("~")
pysteps_data_dir_path = os.path.join(home_dir, "pysteps_data")

# Download the pysteps data.
download_pysteps_data(pysteps_data_dir_path, force=True)

# Create a default configuration file that points to the downloaded data.
# By default it will place the configuration file in the
# $HOME/.pysteps (unix and Mac OS X) or $USERPROFILE/pysteps (windows).
config_file_path = create_default_pystepsrc(pysteps_data_dir_path)
```

Note that for these changes to take effect you need to restart the python interpreter or use the *pysteps.load_config_file()* function as follows:

```
# Load the new configuration file and replace the default configuration
import pysteps
pysteps.load_config_file(config_file_path, verbose=True)
```

To customize the default configuration file see the *The pySTEPS configuration file (pystepsrc)* section.

Manual installation

Another alternative is to download the data manually into your computer and configure the *pystepsrc* file to point to that example data.

First, download the data from the repository by clicking here.

Unzip the data into a folder of your preference. Once the data is unzipped, the directory structure looks like this:

```
pysteps-data
|
├── radar
       ├── KNMI
       ├── OPERA
       ├── bom
       ├── fmi
       ├── mch
```

The next step is updating the *pystepsrc* file to point to these directories, as described in the *The pySTEPS configuration file (pystepsrc)* section.

2.1.3 pySTEPS examples gallery

Below is a collection of example scripts and tutorials to illustrate the usage of pysteps.

These scripts require the pySTEPS example data. See the installation instructions in the *Installing the Example data* section.

Optical flow

This tutorial offers a short overview of the optical flow routines available in pysteps and it will cover how to compute and plot the motion field from a sequence of radar images.

```
from datetime import datetime
from pprint import pprint
import matplotlib.pyplot as plt
import numpy as np

from pysteps import io, motion, rcparams
from pysteps.utils import conversion, transformation
from pysteps.visualization import plot_precip_field, quiver
```

Read the radar input images

First, we will import the sequence of radar composites. You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201505151630", "%Y%m%d%H%M")
data_source = rcparams.data_sources["mch"]
```

Load the data from the archive

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
```

(continues on next page)

(continued from previous page)

```
importer_kwargs = data_source["importer_kwargs"]
timestep = data_source["timestep"]

# Find the input files from the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_prev_files=9
)

# Read the radar composites
importer = io.get_method(importer_name, "importer")
R, quality, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

del quality  # Not used
```

Preprocess the data

```
# Convert to mm/h
R, metadata = conversion.to_rainrate(R, metadata)

# Store the reference frame
R_ = R[-1, :, :].copy()

# Log-transform the data [dBR]
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)

# Nicely print the metadata
pprint(metadata)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
{'accutime': 5,
 'institution': 'MeteoSwiss',
 'product': 'AQC',
 'projection': '+proj=somerc  +lon_0=7.43958333333333 +lat_0=46.9524055555556 '
               '+k_0=1 +x_0=600000 +y_0=200000 +ellps=bessel '
               '+towgs84=674.374,15.056,405.346,0,0,0,0 +units=m +no_defs',
 'threshold': -10.0,
 'timestamps': array([datetime.datetime(2015, 5, 15, 15, 45),
       datetime.datetime(2015, 5, 15, 15, 50),
       datetime.datetime(2015, 5, 15, 15, 55),
       datetime.datetime(2015, 5, 15, 16, 0),
       datetime.datetime(2015, 5, 15, 16, 5),
       datetime.datetime(2015, 5, 15, 16, 10),
       datetime.datetime(2015, 5, 15, 16, 15),
       datetime.datetime(2015, 5, 15, 16, 20),
       datetime.datetime(2015, 5, 15, 16, 25),
       datetime.datetime(2015, 5, 15, 16, 30)], dtype=object),
 'transform': 'dB',
 'unit': 'mm/h',
 'x1': 255000.0,
 'x2': 965000.0,
 'xpixelsize': 1000.0,
 'y1': -160000.0,
 'y2': 480000.0,
```

(continues on next page)

(continued from previous page)

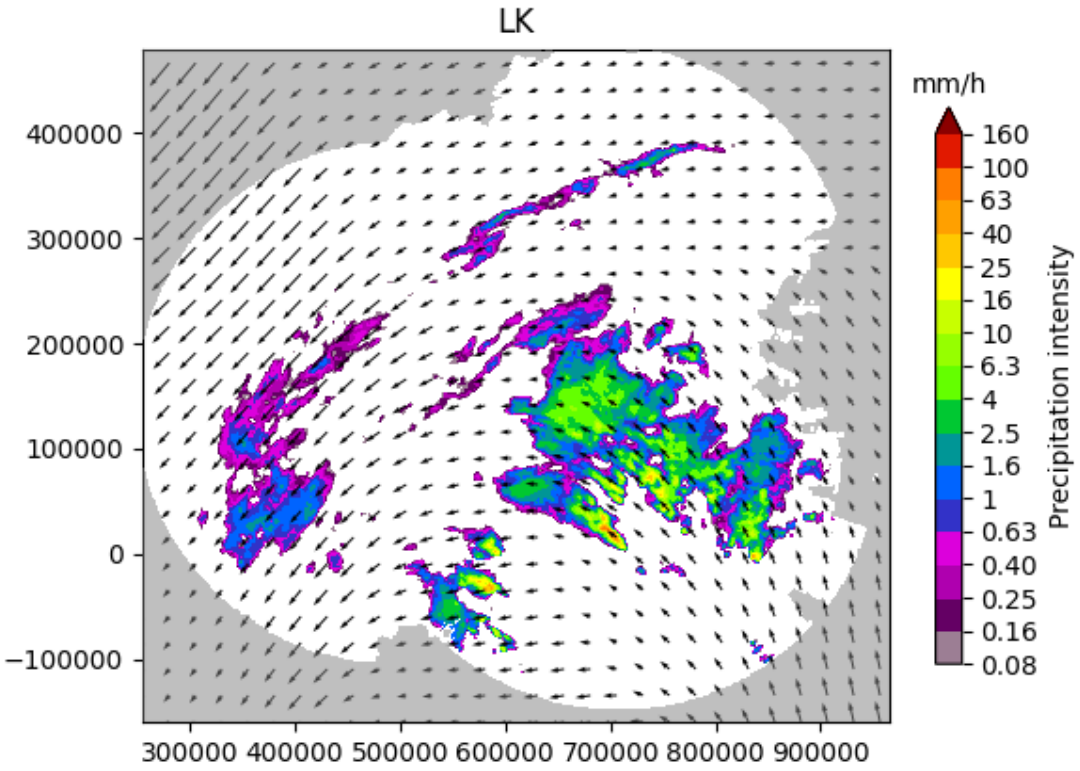
```
 'yorigin': 'upper',
 'ypixelsize': 1000.0,
 'zerovalue': -15.0,
 'zr_a': 316.0,
 'zr_b': 1.5}
```

Lucas-Kanade (LK)

The Lucas-Kanade optical flow method implemented in pysteps is a local tracking approach that relies on the OpenCV package. Local features are tracked in a sequence of two or more radar images. The scheme includes a final interpolation step in order to produce a smooth field of motion vectors.

```
oflow_method = motion.get_method("LK")
V1 = oflow_method(R[-3:, :, :])

# Plot the motion field on top of the reference frame
plot_precip_field(R_, geodata=metadata, title="LK")
quiver(V1, geodata=metadata, step=25)
plt.show()
```



Out:

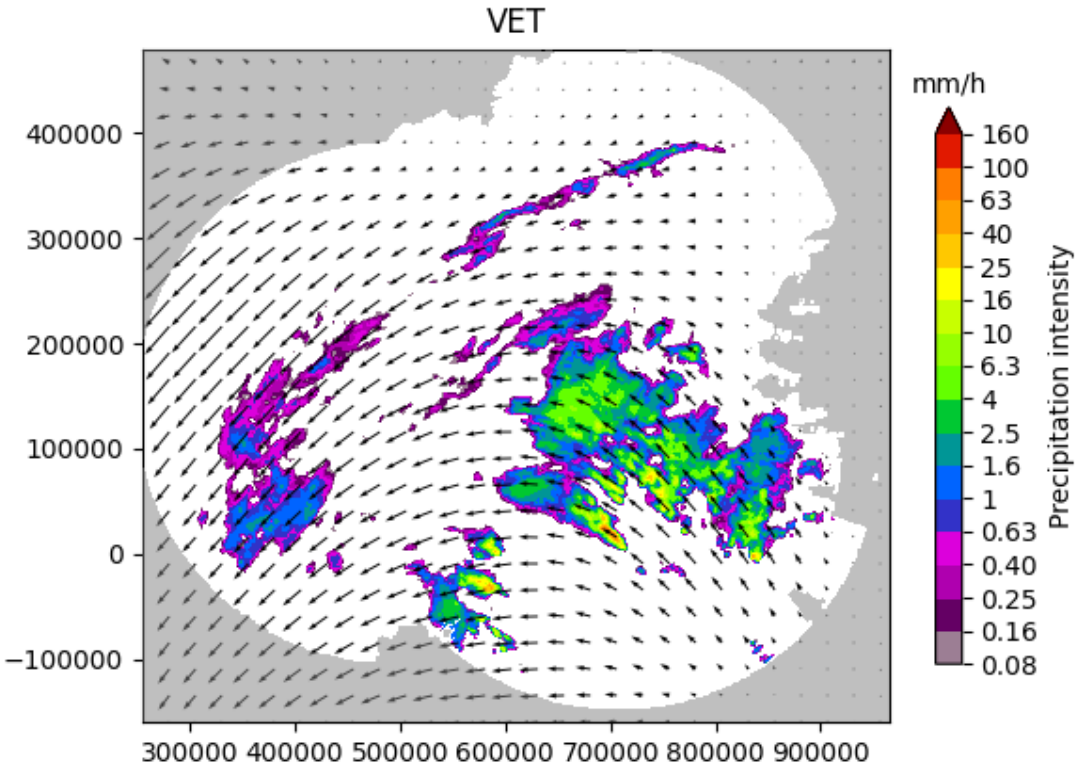
```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
```

Variational echo tracking (VET)

This module implements the VET algorithm presented by Laroche and Zawadzki (1995) and used in the McGill Algorithm for Prediction by Lagrangian Extrapolation (MAPLE) described in Germann and Zawadzki (2002). The approach essentially consists of a global optimization routine that seeks at minimizing a cost function between the displaced and the reference image.

```
oflow_method = motion.get_method("VET")
V2 = oflow_method(R[-3:, :, :])

# Plot the motion field
plot_precip_field(R_, geodata=metadata, title="VET")
quiver(V2, geodata=metadata, step=25)
plt.show()
```



Out:

```
Running VET algorithm
original image shape: (3, 640, 710)
padded image shape: (3, 640, 710)
padded template_image image shape: (3, 640, 710)

Number of sectors: 2,2
Sector Shape: (320, 355)
Minimizing

residuals 3102581.0581715414
smoothness_penalty 0.0
original image shape: (3, 640, 710)
padded image shape: (3, 640, 712)
padded template_image image shape: (3, 640, 712)
```

(continues on next page)

(continued from previous page)

```
Number of sectors: 4,4
Sector Shape: (160, 178)
Minimizing

residuals 2506232.4819293534
smoothness_penalty 0.5027211304402006
original image shape: (3, 640, 710)
padded image shape: (3, 640, 720)
padded template_image image shape: (3, 640, 720)

Number of sectors: 16,16
Sector Shape: (40, 45)
Minimizing

residuals 2267290.5287230993
smoothness_penalty 41.433699598718086
original image shape: (3, 640, 710)
padded image shape: (3, 640, 736)
padded template_image image shape: (3, 640, 736)

Number of sectors: 32,32
Sector Shape: (20, 23)
Minimizing

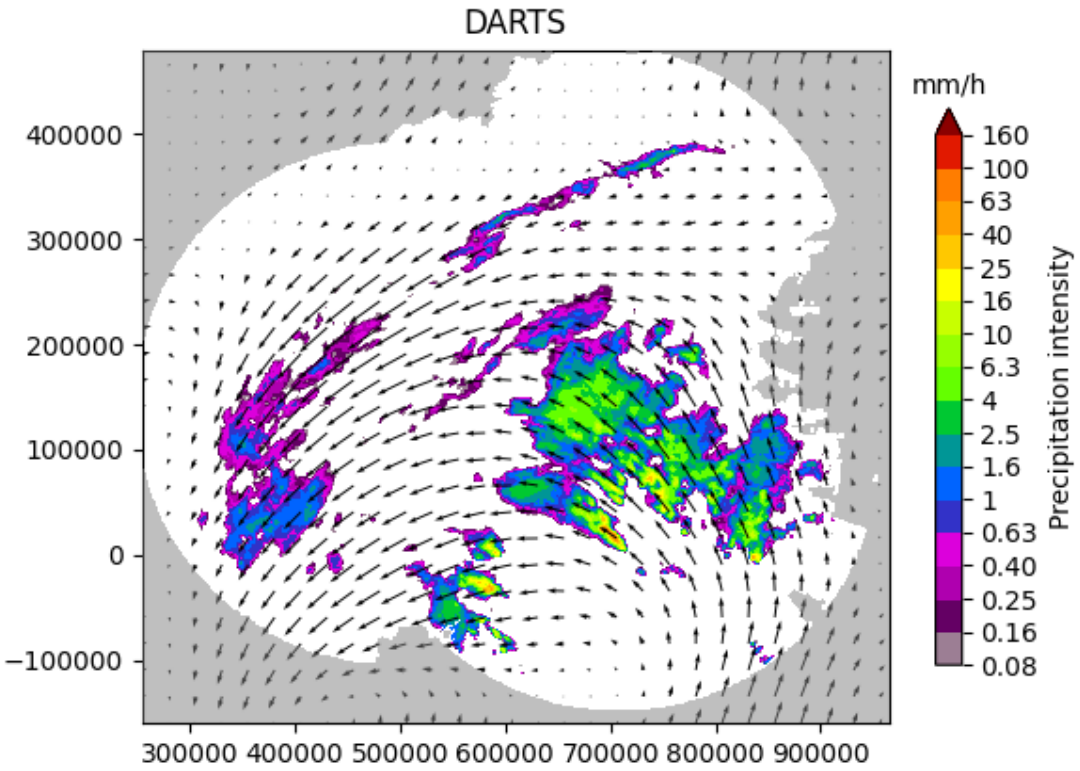
residuals 2288499.8450945
smoothness_penalty 190.92626378296308
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
```

Dynamic and adaptive radar tracking of storms (DARTS)

DARTS uses a spectral approach to optical flow that is based on the discrete Fourier transform (DFT) of a temporal sequence of radar fields. The level of truncation of the DFT coefficients controls the degree of smoothness of the estimated motion field, allowing for an efficient motion estimation. DARTS requires a longer sequence of radar fields for estimating the motion, here we are going to use all the available 10 fields.

```
oflow_method = motion.get_method("DARTS")
R[~np.isfinite(R)] = metadata["zerovalue"]
V3 = oflow_method(R)  # needs longer training sequence

# Plot the motion field
plot_precip_field(R_, geodata=metadata, title="DARTS")
quiver(V3, geodata=metadata, step=25)
plt.show()
```



Out:

```
Computing the motion field with the DARTS method.
-----
DARTS
-----
  Computing the FFT of the reflectivity fields...Done in 0.39 seconds.
  Constructing the y-vector...Done in 0.44 seconds.
  Constructing the H-matrix...Done in 1.62 seconds.
  Solving the linear systems...Done in 0.16 seconds.
--- 2.7054741382598877 seconds ---
```

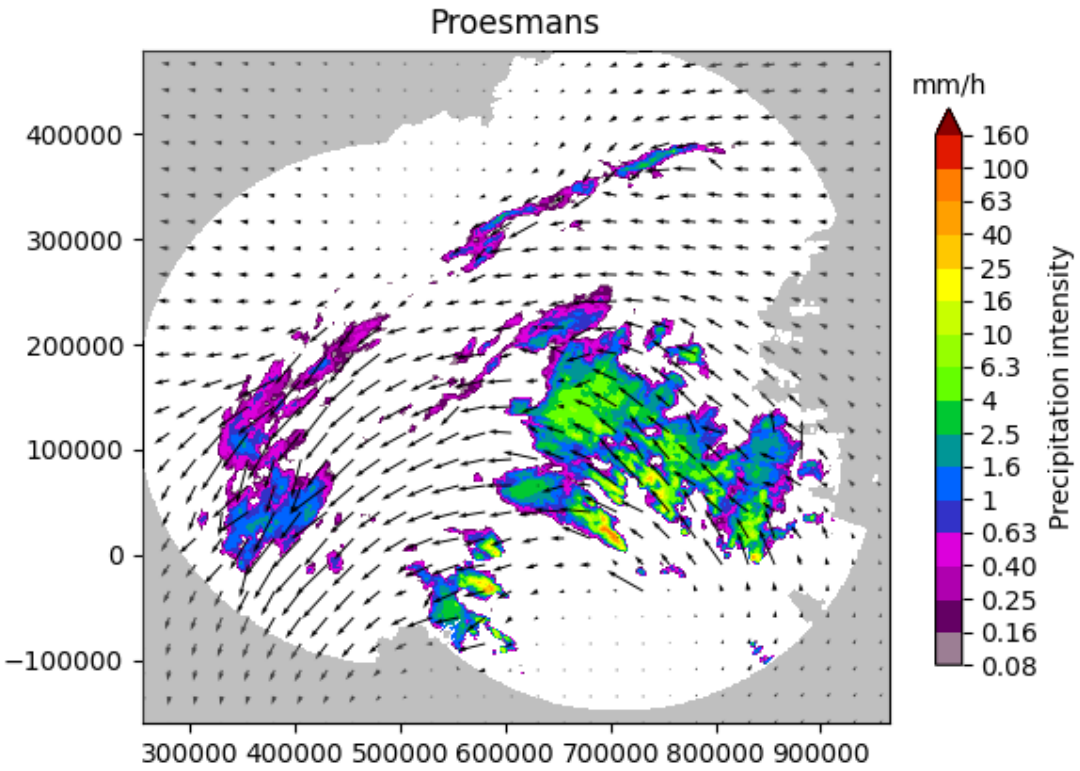
Anisotropic diffusion method (Proesmans et al 1994)

This module implements the anisotropic diffusion method presented in Proesmans et al. (1994), a robust optical flow technique which employs the notion of inconsitency during the solution of the optical flow equations.

```
oflow_method = motion.get_method("proesmans")
R[~np.isfinite(R)] = metadata["zerovalue"]
V4 = oflow_method(R[-2:, :, :])

# Plot the motion field
plot_precip_field(R_, geodata=metadata, title="Proesmans")
quiver(V4, geodata=metadata, step=25)
plt.show()

# sphinx_gallery_thumbnail_number = 1
```



Total running time of the script: (1 minutes 11.210 seconds)

Advection correction

This tutorial shows how to use the optical flow routines of pysteps to implement the advection correction procedure described in Anagnostou and Krajewski (1999).

Advection correction is a temporal interpolation procedure that is often used when estimating rainfall accumulations to correct for the shift of rainfall patterns between consecutive radar rainfall maps. This shift becomes particularly significant for long radar scanning cycles and in presence of fast moving precipitation features.

Note: The code for the advection correction using pysteps was originally written by Daniel Wolfensberger.

```
from datetime import datetime
import matplotlib.pyplot as pl
import numpy as np

from pysteps import io, motion, rcparams
from pysteps.utils import conversion, dimension
from pysteps.visualization import plot_precip_field
from scipy.ndimage import map_coordinates
```

Read the radar input images

First, we import a sequence of 36 images of 5-minute radar composites that we will use to produce a 3-hour rainfall accumulation map. We will keep only one frame every 10 minutes, to simulate a longer scanning cycle and thus better highlight the need for advection correction.

You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201607112100", "%Y%m%d%H%M")
data_source = rcparams.data_sources["mch"]
```

Load the data from the archive

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]
timestep = data_source["timestep"]

# Find the input files from the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_next_files=35,
)

# Read the radar composites
importer = io.get_method(importer_name, "importer")
R, __, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

# Convert to mm/h
R, metadata = conversion.to_rainrate(R, metadata)

# Upscale to 2 km (simply to reduce the memory demand)
R, metadata = dimension.aggregate_fields_space(R, metadata, 2000)

# Keep only one frame every 10 minutes (i.e., every 2 timesteps)
# (to highlight the need for advection correction)
R = R[::2]
```

Advection correction

Now we need to implement the advection correction for a pair of successive radar images. The procedure is based on the algorithm described in Anagnostou and Krajewski (Appendix A, 1999).

To evaluate the advection occurred between two successive radar images, we are going to use the Lucas-Kanade optical flow routine available in pysteps.

```
def advection_correction(R, T=5, t=1):
    """
    R = np.array([qpe_previous, qpe_current])
    T = time between two observations (5 min)
    t = interpolation timestep (1 min)
    """

    # Evaluate advection
    oflow_method = motion.get_method("LK")
    fd_kwargs = {"buffer_mask": 10}  # avoid edge effects
    V = oflow_method(np.log(R), fd_kwargs=fd_kwargs)

    # Perform temporal interpolation
    Rd = np.zeros((R[0].shape))
    x, y = np.meshgrid(
        np.arange(R[0].shape[1], dtype=float), np.arange(R[0].shape[0],␣
↪dtype=float),
```

(continues on next page)

(continued from previous page)

```
    )
    for i in range(1, 1 + int(T / t)):

        pos1 = (y - i / T * V[1], x - i / T * V[0])
        R1 = map_coordinates(R[0], pos1, order=1)

        pos2 = (y + (T - i) / T * V[1], x + (T - i) / T * V[0])
        R2 = map_coordinates(R[1], pos2, order=1)

        Rd += (T - i) * R1 + i * R2

    return 1 / T ** 2 * Rd
```

Finally, we apply the advection correction to the whole sequence of radar images and produce the rainfall accumulation map.

```
R_ac = R[0].copy()
for i in range(R.shape[0] - 1):
    R_ac += advection_correction(R[i : (i + 2)], T=10)
R_ac /= R.shape[0]
```

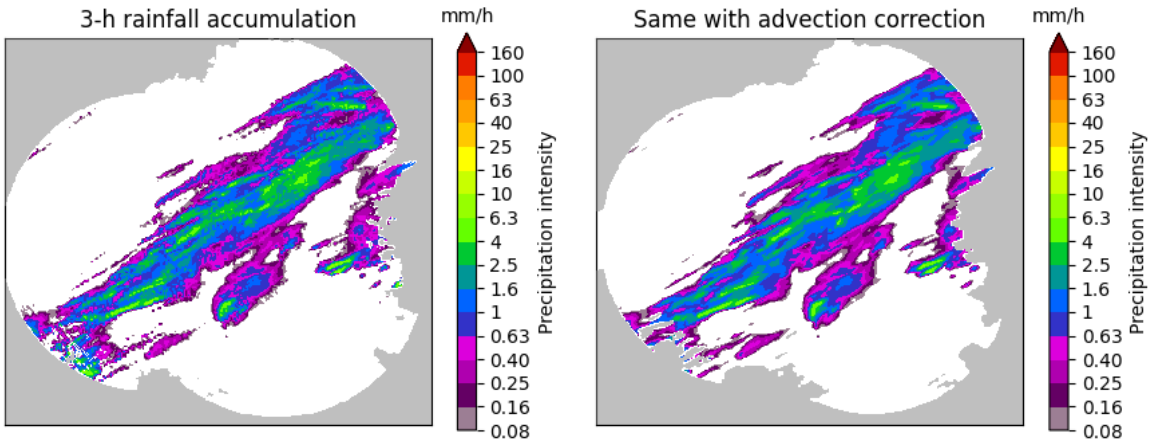
Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/checkouts/v1.3.1/examples/
↪advection_correction.py:97: RuntimeWarning: divide by zero encountered in log
  V = oflow_method(np.log(R), fd_kwargs=fd_kwargs)
```

Results

We compare the two accumulation maps. The first map on the left is computed without advection correction and we can therefore see that the shift between successive images 10 minutes apart produces irregular accumulations. Conversely, the rainfall accumulation of the right is produced using advection correction to account for this spatial shift. The final result is a smoother rainfall accumulation map.

```
pl.figure(figsize=(9, 4))
pl.subplot(121)
plot_precip_field(R.mean(axis=0), title="3-h rainfall accumulation")
pl.subplot(122)
plot_precip_field(R_ac, title="Same with advection correction")
pl.tight_layout()
pl.show()
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
```

Reference

Anagnostou, E. N., and W. F. Krajewski. 1999. "Real-Time Radar Rainfall Estimation. Part I: Algorithm Formulation." Journal of Atmospheric and Oceanic Technology 16: 189–97. https://doi.org/10.1175/1520-0426(1999)016<0189:RTRREP>2.0.CO;2

Total running time of the script: (0 minutes 9.960 seconds)

Extrapolation nowcast

This tutorial shows how to compute and plot an extrapolation nowcast using Finnish radar data.

```
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
from pprint import pprint
from pysteps import io, motion, nowcasts, rcparams, verification
from pysteps.utils import conversion, transformation
from pysteps.visualization import plot_precip_field, quiver
```

Read the radar input images

First, we will import the sequence of radar composites. You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201609281600", "%Y%m%d%H%M")
data_source = rcparams.data_sources["fmi"]
n_leadtimes = 12
```

Load the data from the archive

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]
timestep = data_source["timestep"]

# Find the input files from the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep, num_prev_files=2
)

# Read the radar composites
importer = io.get_method(importer_name, "importer")
Z, _, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

# Convert to rain rate
R, metadata = conversion.to_rainrate(Z, metadata)
```

(continues on next page)

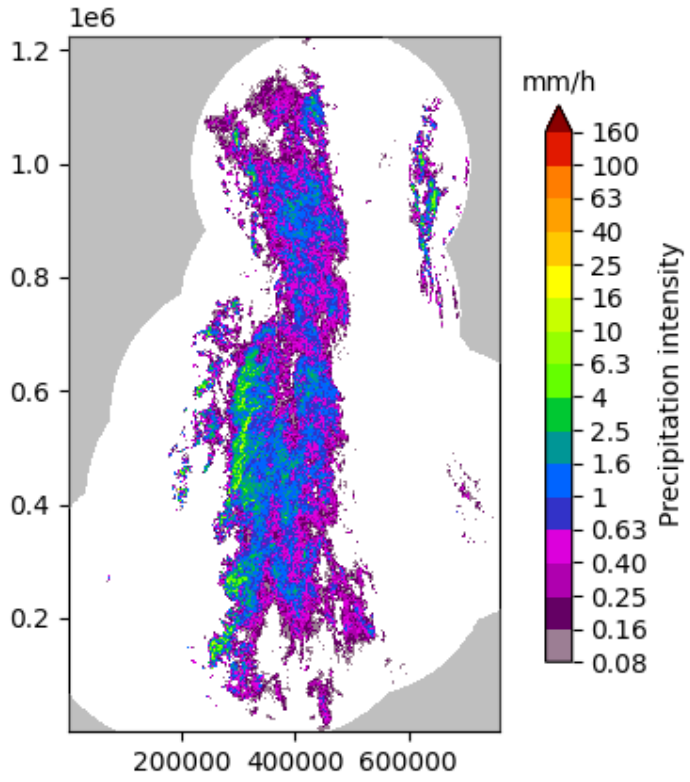
(continued from previous page)

```
# Plot the rainfall field
plot_precip_field(R[-1, :, :], geodata=metadata)
plt.show()

# Store the last frame for plotting it later later
R_ = R[-1, :, :].copy()

# Log-transform the data to unit of dBR, set the threshold to 0.1 mm/h,
# set the fill value to -15 dBR
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)

# Nicely print the metadata
pprint(metadata)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:240: RuntimeWarning: invalid value␣
↪encountered in less
  R[R < threshold] = zerovalue
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
{'accutime': 5.0,
```

(continues on next page)

(continued from previous page)

```
 'institution': 'Finnish Meteorological Institute',
 'projection': '+proj=stere  +lon_0=25E +lat_0=90N +lat_ts=60 +a=6371288 '
               '+x_0=380886.310 +y_0=3395677.920 +no_defs',
 'threshold': -10.0,
 'timestamps': array([datetime.datetime(2016, 9, 28, 15, 50),
       datetime.datetime(2016, 9, 28, 15, 55),
       datetime.datetime(2016, 9, 28, 16, 0)], dtype=object),
 'transform': 'dB',
 'unit': 'mm/h',
 'x1': 0.0049823258887045085,
 'x2': 759752.2852757066,
 'xpixelsize': 999.674053,
 'y1': 0.009731985162943602,
 'y2': 1225544.6588913496,
 'yorigin': 'upper',
 'ypixelsize': 999.62859,
 'zerovalue': -15.0,
 'zr_a': 223.0,
 'zr_b': 1.53}
```

Compute the nowcast

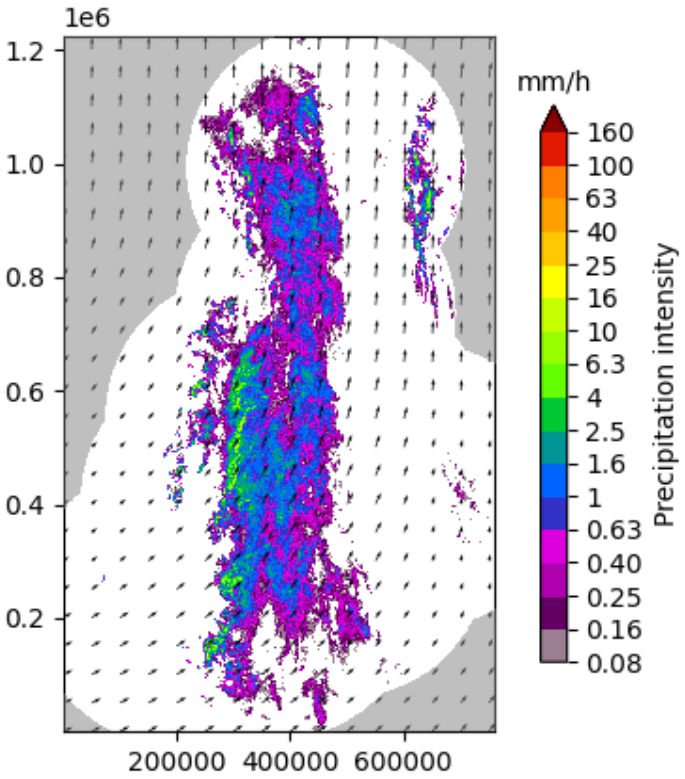
The extrapolation nowcast is based on the estimation of the motion field, which is here performed using a local tracking approach (Lucas-Kanade). The most recent radar rainfall field is then simply advected along this motion field in oder to produce an extrapolation forecast.

```
# Estimate the motion field with Lucas-Kanade
oflow_method = motion.get_method("LK")
V = oflow_method(R[-3:, :, :])

# Extrapolate the last radar observation
extrapolate = nowcasts.get_method("extrapolation")
R[~np.isfinite(R)] = metadata["zerovalue"]
R_f = extrapolate(R[-1, :, :], V, n_leadtimes)

# Back-transform to rain rate
R_f = transformation.dB_transform(R_f, threshold=-10.0, inverse=True)[0]

# Plot the motion field
plot_precip_field(R_, geodata=metadata)
quiver(V, geodata=metadata, step=50)
plt.show()
```



Verify with FSS

The fractions skill score (FSS) provides an intuitive assessment of the dependency of skill on spatial scale and intensity, which makes it an ideal skill score for high-resolution precipitation forecasts.

```
# Find observations in the data archive
fns = io.archive.find_by_date(
    date,
    root_path,
    path_fmt,
    fn_pattern,
    fn_ext,
    timestep,
    num_prev_files=0,
    num_next_files=n_leadtimes,
)
# Read the radar composites
R_o, _, metadata_o = io.read_timeseries(fns, importer, **importer_kwargs)
R_o, metadata_o = conversion.to_rainrate(R_o, metadata_o, 223.0, 1.53)

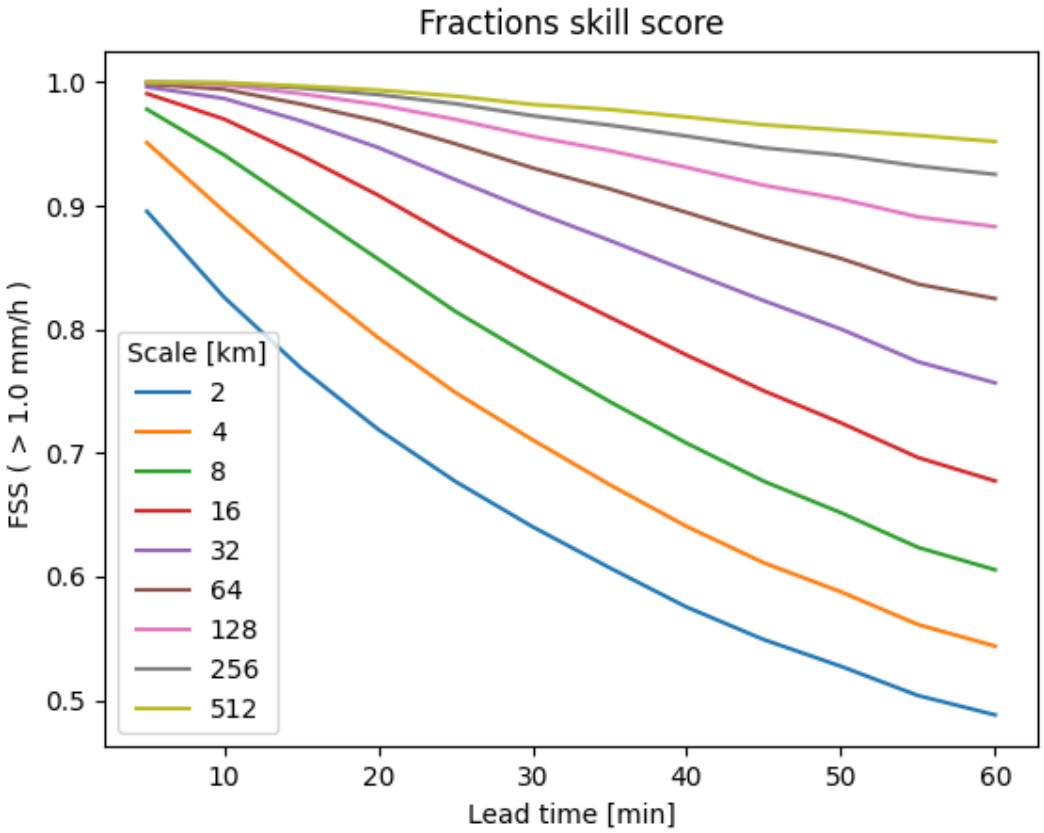
# Compute fractions skill score (FSS) for all lead times, a set of scales and 1 mm/
↪h
fss = verification.get_method("FSS")
scales = [2, 4, 8, 16, 32, 64, 128, 256, 512]
thr = 1.0
score = []
for i in range(n_leadtimes):
    score_ = []
    for scale in scales:
        score_.append(fss(R_f[i, :, :], R_o[i + 1, :, :], thr, scale))
    score.append(score_)
```

(continues on next page)

(continued from previous page)

```
plt.figure()
x = np.arange(1, n_leadtimes + 1) * timestep
plt.plot(x, score)
plt.legend(scales, title="Scale [km]")
plt.xlabel("Lead time [min]")
plt.ylabel("FSS ( > 1.0 mm/h ) ")
plt.title("Fractions skill score")
plt.show()

# sphinx_gallery_thumbnail_number = 3
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:240: RuntimeWarning: invalid value␣
↪encountered in less
  R[R < threshold] = zerovalue
```

Total running time of the script: (0 minutes 17.424 seconds)

Generation of stochastic noise

This example script shows how to run the stochastic noise field generators included in pysteps.

These noise fields are used as perturbation terms during an extrapolation nowcast in order to represent the uncertainty in the evolution of the rainfall field.

```
from matplotlib import cm, pyplot
import numpy as np
import os
from pprint import pprint
from pysteps import io, rcparams
from pysteps.noise.fftgenerators import initialize_param_2d_fft_filter
from pysteps.noise.fftgenerators import initialize_nonparam_2d_fft_filter
from pysteps.noise.fftgenerators import generate_noise_2d_fft_filter
from pysteps.utils import conversion, rapsd, transformation
from pysteps.visualization import plot_precip_field, plot_spectrum1d
```

Read precipitation field

First thing, the radar composite is imported and transformed in units of dB. This image will be used to train the Fourier filters that are necessary to produce the fields of spatially correlated noise.

```
# Import the example radar composite
root_path = rcparams.data_sources["mch"]["root_path"]
filename = os.path.join(root_path, "20160711", "AQC161932100V_00005.801.gif")
R, _, metadata = io.import_mch_gif(filename, product="AQC", unit="mm", accutime=5.
↪0)

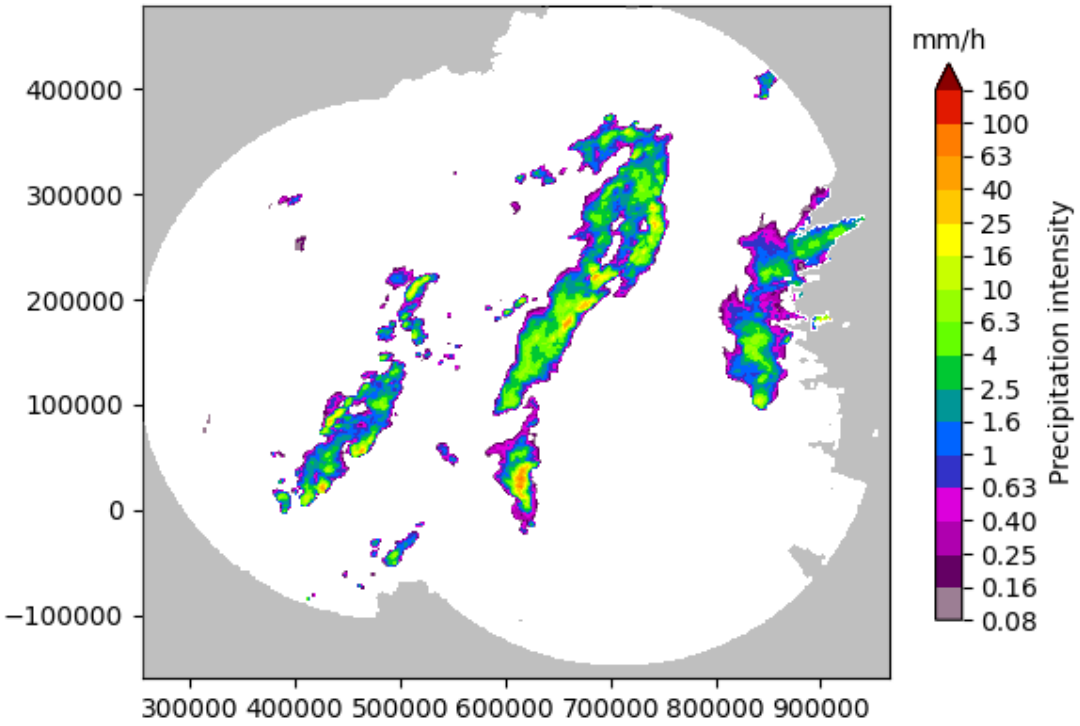
# Convert to mm/h
R, metadata = conversion.to_rainrate(R, metadata)

# Nicely print the metadata
pprint(metadata)

# Plot the rainfall field
plot_precip_field(R, geodata=metadata)

# Log-transform the data
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)

# Assign the fill value to all the Nans
R[~np.isfinite(R)] = metadata["zerovalue"]
```



Out:

```
{'accutime': 5.0,
 'institution': 'MeteoSwiss',
 'product': 'AQC',
 'projection': '+proj=somerc  +lon_0=7.43958333333333 +lat_0=46.9524055555556 '
               '+k_0=1 +x_0=600000 +y_0=200000 +ellps=bessel '
               '+towgs84=674.374,15.056,405.346,0,0,0,0 +units=m +no_defs',
 'threshold': 0.01155375598376629,
 'transform': None,
 'unit': 'mm/h',
 'x1': 255000.0,
 'x2': 965000.0,
 'xpixelsize': 1000.0,
 'y1': -160000.0,
 'y2': 480000.0,
 'yorigin': 'upper',
 'ypixelsize': 1000.0,
 'zerovalue': 0.0,
 'zr_a': 316.0,
 'zr_b': 1.5}
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
```

Parametric filter

In the parametric approach, a power-law model is used to approximate the power spectral density (PSD) of a given rainfall field.

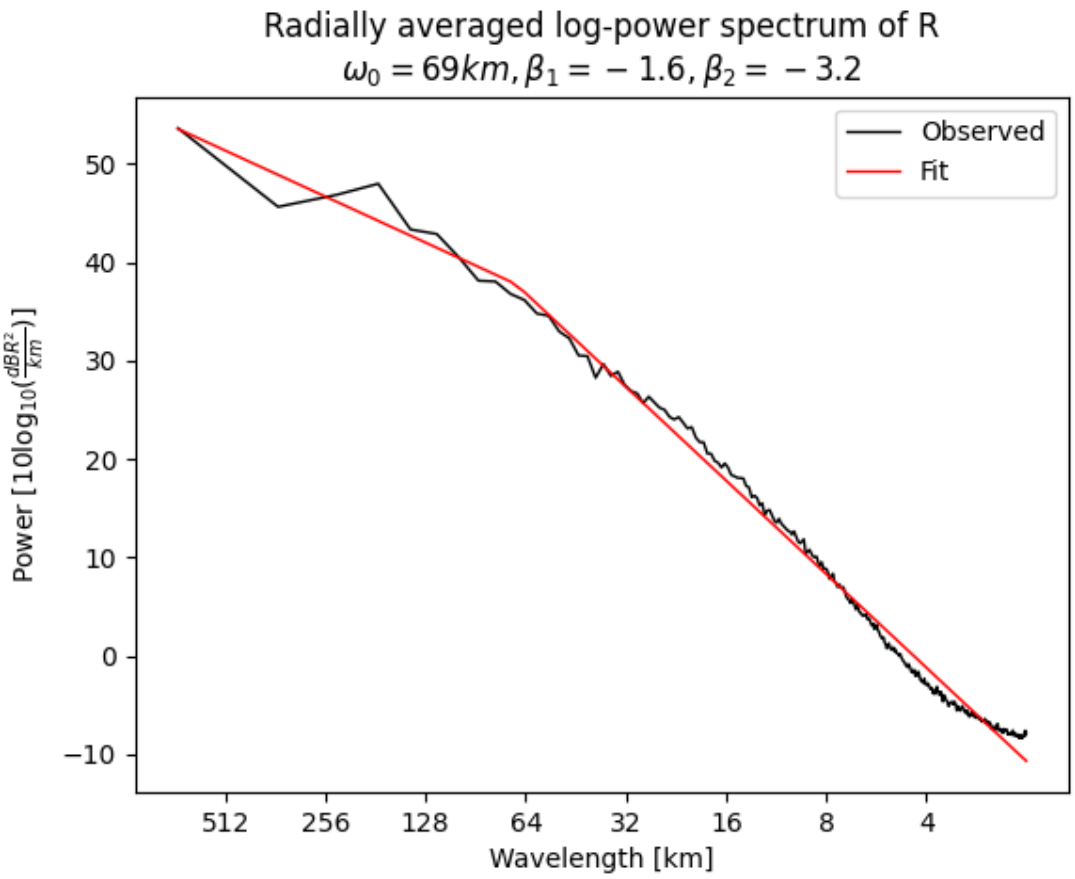
The parametric model uses a piece-wise linear function with two spectral slopes (beta1 and beta2) and one breaking point

```
# Fit the parametric PSD to the observation
Fp = initialize_param_2d_fft_filter(R)

# Compute the observed and fitted 1D PSD
L = np.max(Fp["input_shape"])
if L % 2 == 0:
    wn = np.arange(0, int(L / 2) + 1)
else:
    wn = np.arange(0, int(L / 2))
R_, freq = rapsd(R, fft_method=np.fft, return_freq=True)
f = np.exp(Fp["model"](np.log(wn), *Fp["pars"]))

# Extract the scaling break in km, beta1 and beta2
w0 = L / np.exp(Fp["pars"][0])
b1 = Fp["pars"][2]
b2 = Fp["pars"][3]

# Plot the observed power spectrum and the model
fig, ax = pyplot.subplots()
plot_scales = [512, 256, 128, 64, 32, 16, 8, 4]
plot_spectrum1d(
    freq,
    R_,
    x_units="km",
    y_units="dBR",
    color="k",
    ax=ax,
    label="Observed",
    wavelength_ticks=plot_scales,
)
plot_spectrum1d(
    freq,
    f,
    x_units="km",
    y_units="dBR",
    color="r",
    ax=ax,
    label="Fit",
    wavelength_ticks=plot_scales,
)
pyplot.legend()
ax.set_title(
    "Radially averaged log-power spectrum of R\n"
    r"$\omega_0=%.0f km, \beta_1=%.1f, \beta_2=%.1f$" % (w0, b1, b2)
)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/noise/fftgenerators.py:204: RuntimeWarning: divide by zero␣
↪encountered in log
  F = np.exp(piecewise_linear(np.log(R), *pf))
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/checkouts/v1.3.1/examples/
↪plot_noise_generators.py:74: RuntimeWarning: divide by zero encountered in log
  f = np.exp(Fp["model"](np.log(wn), *Fp["pars"]))
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/spectral.py:85: RuntimeWarning: divide by␣
↪zero encountered in log10
  10.0 * np.log10(fft_freq),
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/spectral.py:85: RuntimeWarning: invalid␣
↪value encountered in log10
  10.0 * np.log10(fft_freq),
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/spectral.py:85: RuntimeWarning: divide by␣
↪zero encountered in log10
  10.0 * np.log10(fft_freq),
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/spectral.py:85: RuntimeWarning: invalid␣
↪value encountered in log10
  10.0 * np.log10(fft_freq),

Text(0.5, 1.0, 'Radially averaged log-power spectrum of R\n$\\omega_0=69 km,␣
↪\\beta_1=-1.6, \\beta_2=-3.2$')
```

Nonparametric filter

In the nonparametric approach, the Fourier filter is obtained directly from the power spectrum of the observed precipitation field R.

```
Fnp = initialize_nonparam_2d_fft_filter(R)
```

Noise generator

The parametric and nonparametric filters obtained above can now be used to produce N realizations of random fields of prescribed power spectrum, hence with the same correlation structure as the initial rainfall field.

```
seed = 42
num_realizations = 3

# Generate noise
Np = []
Nnp = []
for k in range(num_realizations):
    Np.append(generate_noise_2d_fft_filter(Fp, seed=seed + k))
    Nnp.append(generate_noise_2d_fft_filter(Fnp, seed=seed + k))

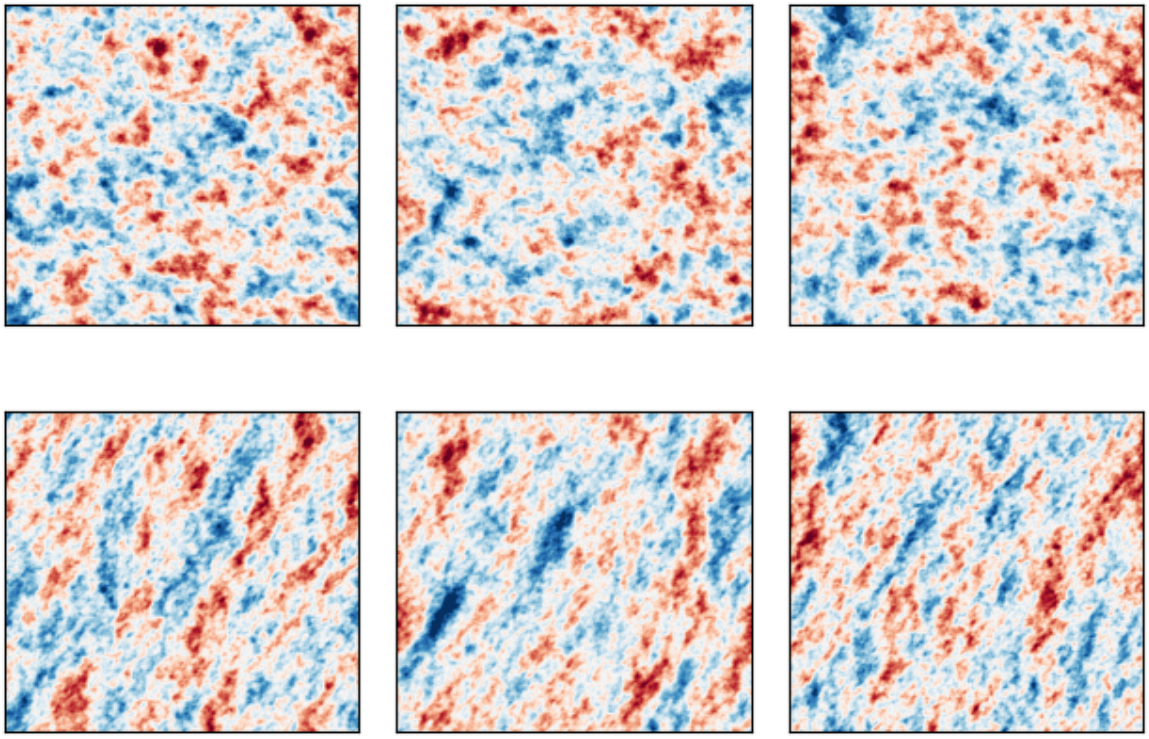
# Plot the generated noise fields

fig, ax = pyplot.subplots(nrows=2, ncols=3)

# parametric noise
ax[0, 0].imshow(Np[0], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[0, 1].imshow(Np[1], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[0, 2].imshow(Np[2], cmap=cm.RdBu_r, vmin=-3, vmax=3)

# nonparametric noise
ax[1, 0].imshow(Nnp[0], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[1, 1].imshow(Nnp[1], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[1, 2].imshow(Nnp[2], cmap=cm.RdBu_r, vmin=-3, vmax=3)

for i in range(2):
    for j in range(3):
        ax[i, j].set_xticks([])
        ax[i, j].set_yticks([])
pyplot.tight_layout()
```



The above figure highlights the main limitation of the parametric approach (top row), that is, the assumption of an isotropic power law scaling relationship, meaning that anisotropic structures such as rainfall bands cannot be represented.

Instead, the nonparametric approach (bottom row) allows generating perturbation fields with anisotropic structures, but it also requires a larger sample size and is sensitive to the quality of the input data, e.g. the presence of residual clutter in the radar image.

In addition, both techniques assume spatial stationarity of the covariance structure of the field.

```
# sphinx_gallery_thumbnail_number = 3
```

Total running time of the script: (0 minutes 1.921 seconds)

Data transformations

The statistics of intermittent precipitation rates are particularly non-Gaussian and display an asymmetric distribution bounded at zero. Such properties restrict the usage of well-established statistical methods that assume symmetric or Gaussian data.

A common workaround is to introduce a suitable data transformation to approximate a normal distribution.

In this example, we test the data transformation methods available in pysteps in order to obtain a more symmetric distribution of the precipitation data (excluding the zeros). The currently available transformations include the Box-Cox, dB, square-root and normal quantile transforms.

```
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
from pysteps import io, rcparams
```

(continues on next page)

(continued from previous page)

```
from pysteps.utils import conversion, transformation
from scipy.stats import skew
```

Read the radar input images

First, we will import the sequence of radar composites. You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201609281600", "%Y%m%d%H%M")
data_source = rcparams.data_sources["fmi"]
```

Load the data from the archive

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]
timestep = data_source["timestep"]

# Get 1 hour of observations in the data archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep, num_next_files=11,
)

# Read the radar composites
importer = io.get_method(importer_name, "importer")
Z, _, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

# Keep only positive rainfall values
Z = Z[Z > metadata["zerovalue"]].flatten()

# Convert to rain rate
R, metadata = conversion.to_rainrate(Z, metadata)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/checkouts/v1.3.1/examples/
↪data_transformations.py:64: RuntimeWarning: invalid value encountered in greater
  Z = Z[Z > metadata["zerovalue"]].flatten()
```

Test data transformations

```
# Define method to visualize the data distribution with boxplots and plot the
# corresponding skewness
def plot_distribution(data, labels, skw):

    N = len(data)
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()

    ax2.plot(np.arange(N + 2), np.zeros(N + 2), ":r")
    ax1.boxplot(data, labels=labels, sym="", medianprops={"color": "k"})

    ymax = []
    for i in range(N):
```

(continues on next page)

(continued from previous page)

```
        y = skw[i]
        x = i + 1
        ax2.plot(x, y, "*r", ms=10, markeredgecolor="k")
        ymax.append(np.max(data[i]))

    # ylims
    ylims = np.percentile(ymax, 50)
    ax1.set_ylim((-1 * ylims, ylims))
    ylims = np.max(np.abs(skw))
    ax2.set_ylim((-1.1 * ylims, 1.1 * ylims))

    # labels
    ax1.set_ylabel(r"Standardized values [$\sigma$]")
    ax2.set_ylabel(r"Skewness []", color="r")
    ax2.tick_params(axis="y", labelcolor="r")
```

Box-Cox transform

The Box-Cox transform is a well-known power transformation introduced by Box and Cox (1964). In its one-parameter version, the Box-Cox transform takes the form T(x) = ln(x) for lambda = 0, or T(x) = (x**lambda - 1)/lambda otherwise.

To find a suitable lambda, we will experiment with a range of values and select the one that produces the most symmetric distribution, i.e., the lambda associated with a value of skewness closest to zero. To visually compare the results, the transformed data are standardized.

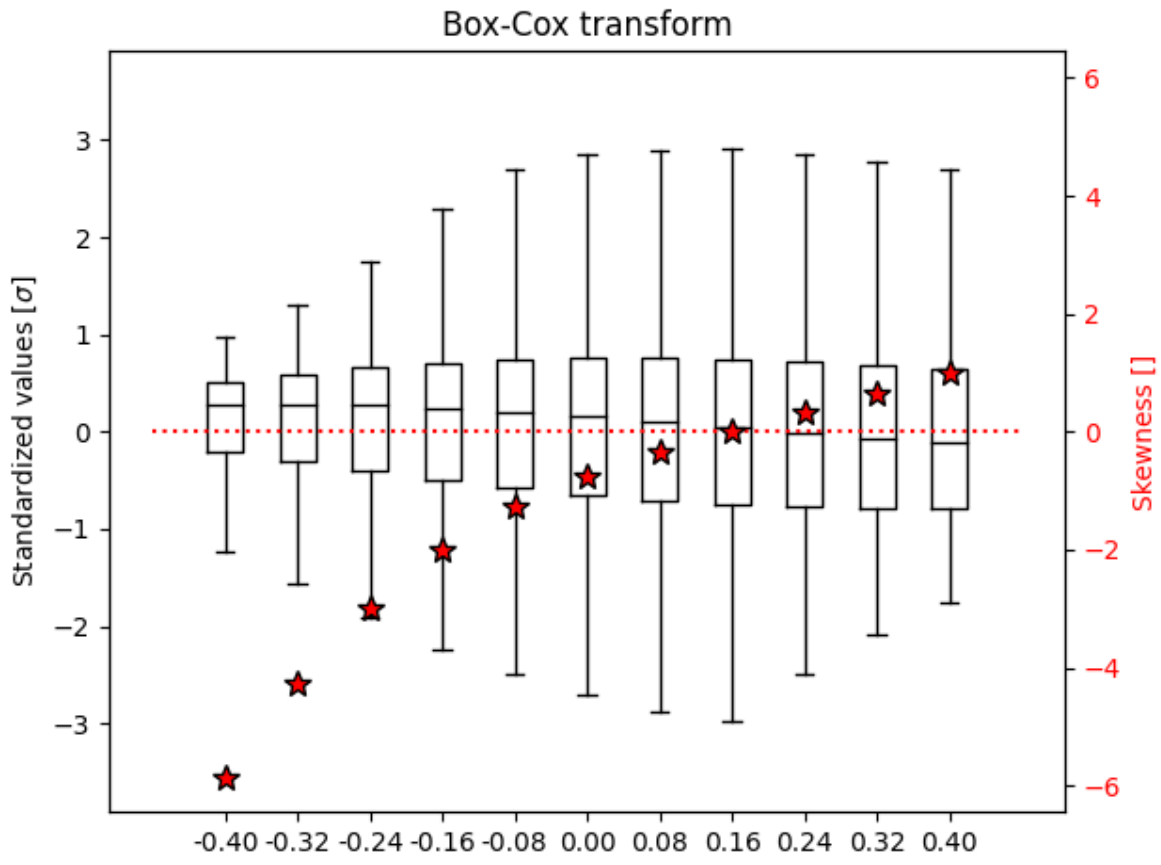
```
data = []
labels = []
skw = []

# Test a range of values for the transformation parameter Lambda
Lambdas = np.linspace(-0.4, 0.4, 11)
for i, Lambda in enumerate(Lambdas):
    R_, _ = transformation.boxcox_transform(R, metadata, Lambda)
    R_ = (R_ - np.mean(R_)) / np.std(R_)
    data.append(R_)
    labels.append("{0:.2f}".format(Lambda))
    skw.append(skew(R_))  # skewness

# Plot the transformed data distribution as a function of lambda
plot_distribution(data, labels, skw)
plt.title("Box-Cox transform")
plt.tight_layout()
plt.show()

# Best lambda
idx_best = np.argmin(np.abs(skw))
Lambda = Lambdas[idx_best]

print("Best parameter lambda: %.2f\n(skewness = %.2f)" % (Lambda, skw[idx_best]))
```



Out:

```
Best parameter lambda: 0.16
(skewness = 0.02)
```

Compare data transformations

```
data = []
labels = []
skw = []
```

Rain rates

First, let's have a look at the original rain rate values.

```
data.append((R - np.mean(R)) / np.std(R))
labels.append("R")
skw.append(skew(R))
```

dB transform

We transform the rainfall data into dB units: 10*log(R)

```
R_, _ = transformation.dB_transform(R, metadata)
data.append((R_ - np.mean(R_)) / np.std(R_))
```

(continues on next page)

(continued from previous page)

```
labels.append("dB")
skw.append(skew(R_))
```

Square-root transform

Transform the data using the square-root: sqrt(R)

```
R_, _ = transformation.sqrt_transform(R, metadata)
data.append((R_ - np.mean(R_)) / np.std(R_))
labels.append("sqrt")
skw.append(skew(R_))
```

Box-Cox transform

We now apply the Box-Cox transform using the best parameter lambda found above.

```
R_, _ = transformation.boxcox_transform(R, metadata, Lambda)
data.append((R_ - np.mean(R_)) / np.std(R_))
labels.append("Box-Cox\n($\lambda=$%.2f)" % Lambda)
skw.append(skew(R_))
```

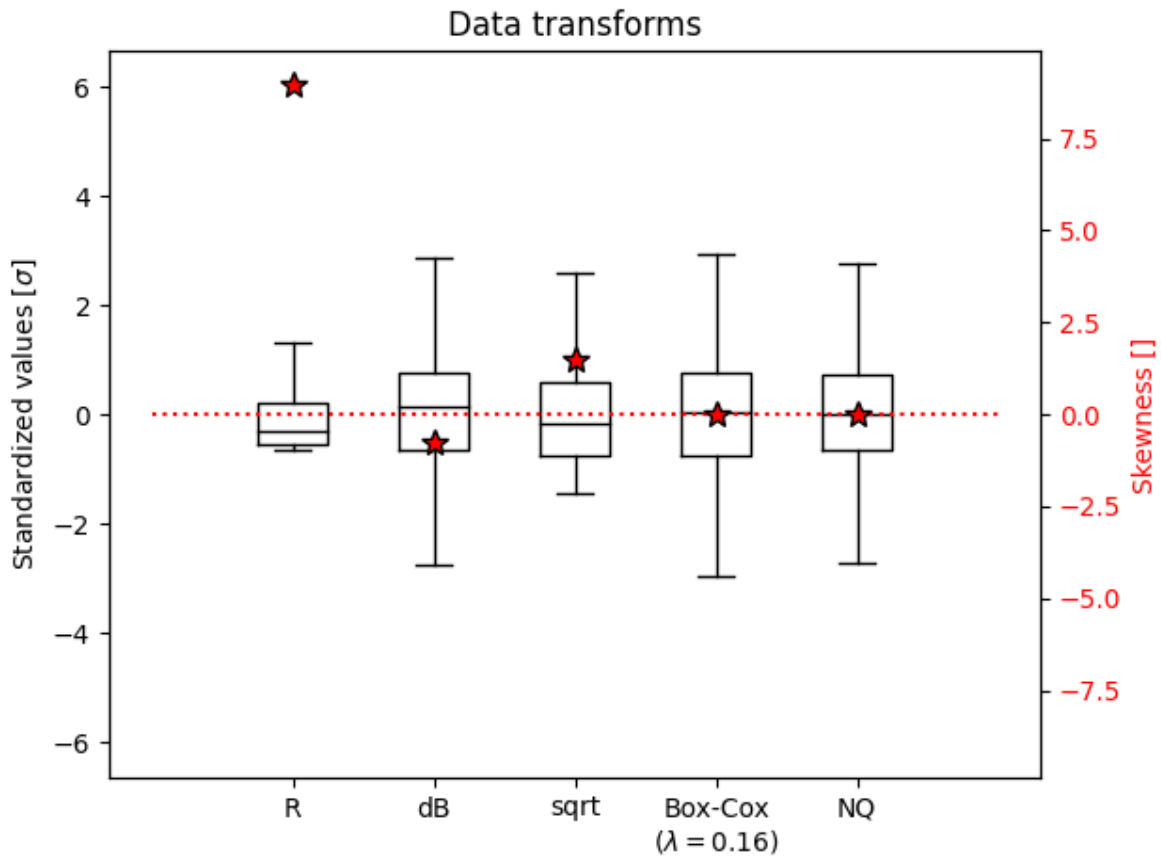
Normal quantile transform

At last, we apply the empirical normal quantile (NQ) transform as described in Bogner et al (2012).

```
R_, _ = transformation.NQ_transform(R, metadata)
data.append((R_ - np.mean(R_)) / np.std(R_))
labels.append("NQ")
skw.append(skew(R_))
```

By plotting all the results, we can notice first of all the strongly asymmetric distribution of the original data (R) and that all transformations manage to reduce its skewness. Among these, the Box-Cox transform (using the best parameter lambda) and the normal quantile (NQ) transform provide the best correction. Despite not producing a perfectly symmetric distribution, the square-root (sqrt) transform has the strong advantage of being defined for zeros, too, while all other transformations need an arbitrary rule for non-positive values.

```
plot_distribution(data, labels, skw)
plt.title("Data transforms")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 9.668 seconds)

Cascade decomposition

This example script shows how to compute and plot the cascade decompositon of a single radar precipitation field in pysteps.

```
from matplotlib import cm, pyplot
import numpy as np
import os
from pprint import pprint
from pysteps.cascade.bandpass_filters import filter_gaussian
from pysteps import io, rcparams
from pysteps.cascade.decomposition import decomposition_fft
from pysteps.utils import conversion, transformation
from pysteps.visualization import plot_precip_field
```

Read precipitation field

First thing, the radar composite is imported and transformed in units of dB.

```
# Import the example radar composite
root_path = rcparams.data_sources["fmi"]["root_path"]
filename = os.path.join(
    root_path, "20160928", "201609281600_fmi.radar.composite.lowest_FIN_SUOMI1.pgm.
↪gz"
)
R, _, metadata = io.import_fmi_pgm(filename, gzipped=True)

# Convert to rain rate
```

(continues on next page)

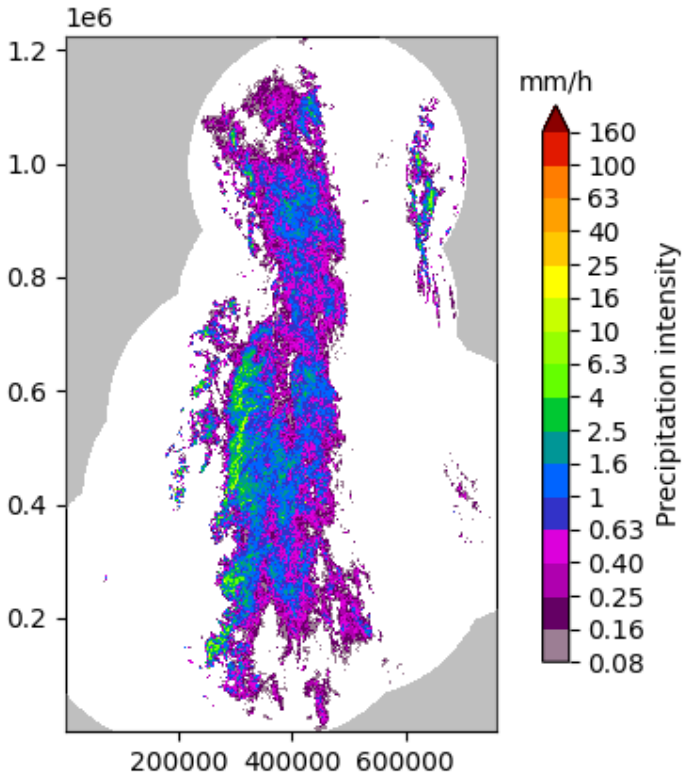
(continued from previous page)

```
R, metadata = conversion.to_rainrate(R, metadata)

# Nicely print the metadata
pprint(metadata)

# Plot the rainfall field
plot_precip_field(R, geodata=metadata)

# Log-transform the data
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:240: RuntimeWarning: invalid value␣
↪encountered in less
  R[R < threshold] = zerovalue
{'accutime': 5.0,
 'institution': 'Finnish Meteorological Institute',
 'projection': '+proj=stere  +lon_0=25E +lat_0=90N +lat_ts=60 +a=6371288 '
               '+x_0=380886.310 +y_0=3395677.920 +no_defs',
 'threshold': 0.00025488054718738 59,
 'transform': None,
 'unit': 'mm/h',
 'x1': 0.0049823258887045085,
 'x2': 759752.2852757066,
 'xpixelsize': 999.674053,
 'y1': 0.009731985162943602,
 'y2': 1225544.6588913496,
```

(continues on next page)

(continued from previous page)

```
 'yorigin': 'upper',
 'ypixelsize': 999.62859,
 'zerovalue': 0.0,
 'zr_a': 223.0,
 'zr_b': 1.53}
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
```

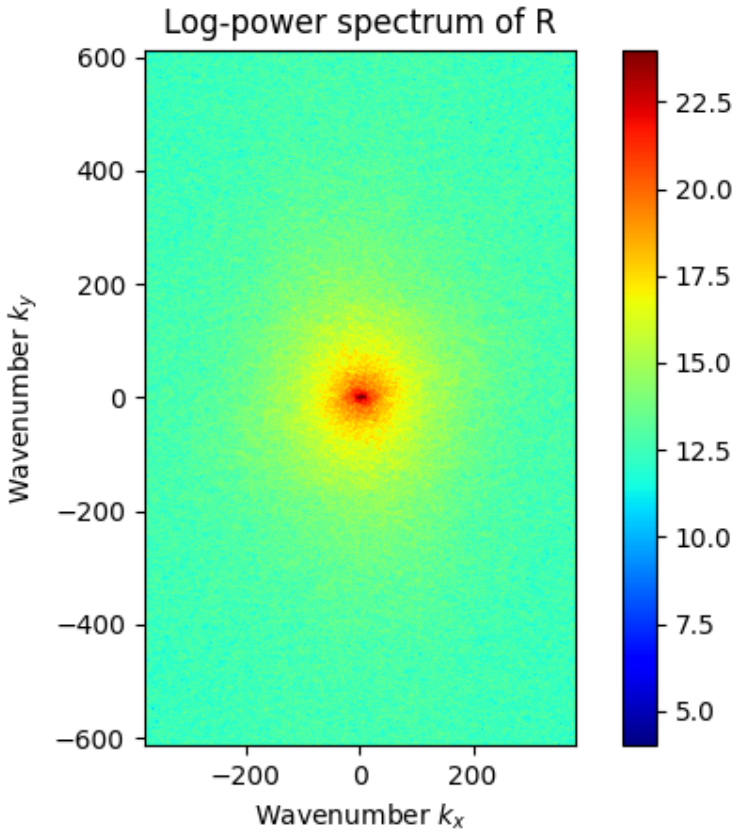
2D Fourier spectrum

Compute and plot the 2D Fourier power spectrum of the precipitaton field.

```
# Set Nans as the fill value
R[~np.isfinite(R)] = metadata["zerovalue"]

# Compute the Fourier transform of the input field
F = abs(np.fft.fftshift(np.fft.fft2(R)))

# Plot the power spectrum
M, N = F.shape
fig, ax = pyplot.subplots()
im = ax.imshow(
    np.log(F ** 2), vmin=4, vmax=24, cmap=cm.jet, extent=(-N / 2, N / 2, -M / 2, M␣
↪/ 2)
)
cb = fig.colorbar(im)
ax.set_xlabel("Wavenumber $k_x$")
ax.set_ylabel("Wavenumber $k_y$")
ax.set_title("Log-power spectrum of R")
```



Out:

```
Text(0.5, 1.0, 'Log-power spectrum of R')
```

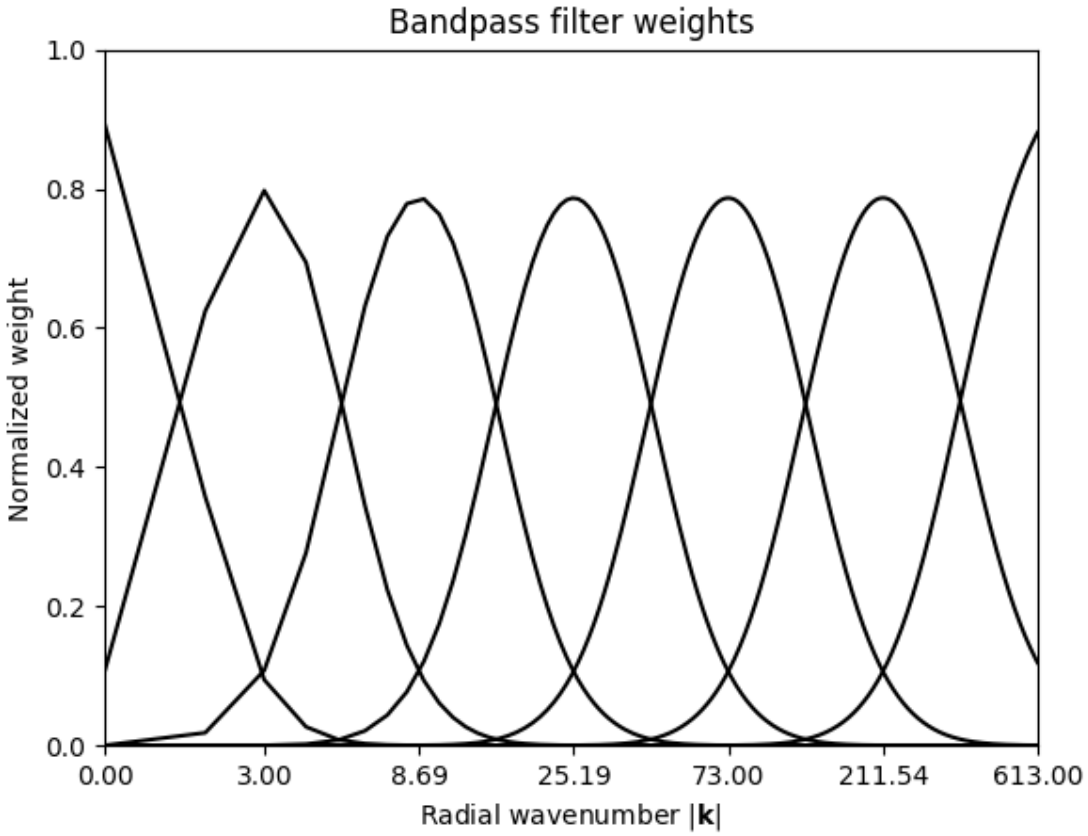
Cascade decomposition

First, construct a set of Gaussian bandpass filters and plot the corresponding 1D filters.

```
num_cascade_levels = 7

# Construct the Gaussian bandpass filters
filter = filter_gaussian(R.shape, num_cascade_levels)

# Plot the bandpass filter weights
L = max(N, M)
fig, ax = pyplot.subplots()
for k in range(num_cascade_levels):
    ax.semilogx(
        np.linspace(0, L / 2, len(filter["weights_1d"][k, :])),
        filter["weights_1d"][k, :],
        "k-",
        basex=pow(0.5 * L / 3, 1.0 / (num_cascade_levels - 2)),
    )
ax.set_xlim(1, L / 2)
ax.set_ylim(0, 1)
xt = np.hstack([[1.0], filter["central_wavenumbers"][1:]])
ax.set_xticks(xt)
ax.set_xticklabels(["%.2f" % cf for cf in filter["central_wavenumbers"]])
ax.set_xlabel("Radial wavenumber $|\mathbf{k}|$")
ax.set_ylabel("Normalized weight")
ax.set_title("Bandpass filter weights")
```



Out:

```
Text(0.5, 1.0, 'Bandpass filter weights')
```

Finally, apply the 2D Gaussian filters to decompose the radar rainfall field into a set of cascade levels of decreasing spatial scale and plot them.

```
decomp = decomposition_fft(R, filter, compute_stats=True)

# Plot the normalized cascade levels
for i in range(num_cascade_levels):
    mu = decomp["means"][i]
    sigma = decomp["stds"][i]
    decomp["cascade_levels"][i] = (decomp["cascade_levels"][i] - mu) / sigma

fig, ax = pyplot.subplots(nrows=2, ncols=4)

ax[0, 0].imshow(R, cmap=cm.RdBu_r, vmin=-5, vmax=5)
ax[0, 1].imshow(decomp["cascade_levels"][0], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[0, 2].imshow(decomp["cascade_levels"][1], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[0, 3].imshow(decomp["cascade_levels"][2], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[1, 0].imshow(decomp["cascade_levels"][3], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[1, 1].imshow(decomp["cascade_levels"][4], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[1, 2].imshow(decomp["cascade_levels"][5], cmap=cm.RdBu_r, vmin=-3, vmax=3)
ax[1, 3].imshow(decomp["cascade_levels"][6], cmap=cm.RdBu_r, vmin=-3, vmax=3)

ax[0, 0].set_title("Observed")
ax[0, 1].set_title("Level 1")
ax[0, 2].set_title("Level 2")
ax[0, 3].set_title("Level 3")
ax[1, 0].set_title("Level 4")
```

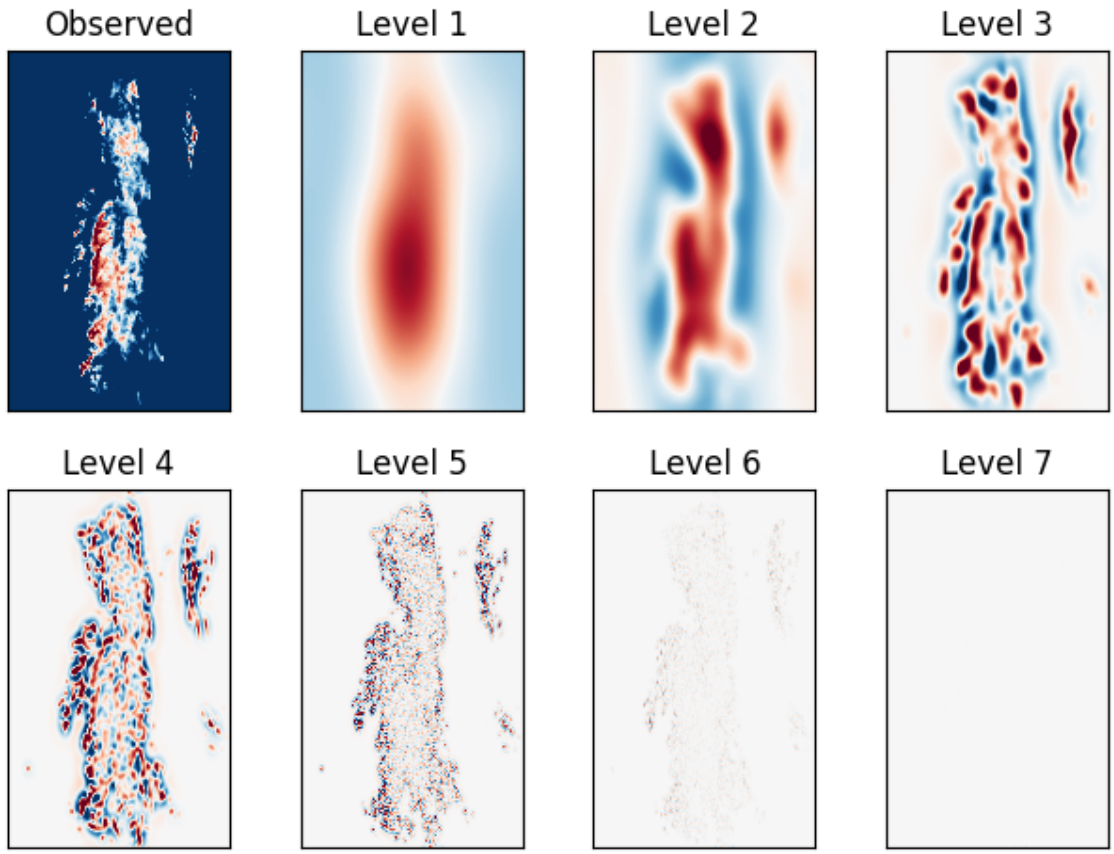
(continues on next page)

(continued from previous page)

```
ax[1, 1].set_title("Level 5")
ax[1, 2].set_title("Level 6")
ax[1, 3].set_title("Level 7")

for i in range(2):
    for j in range(4):
        ax[i, j].set_xticks([])
        ax[i, j].set_yticks([])
pyplot.tight_layout()

# sphinx_gallery_thumbnail_number = 4
```



Total running time of the script: (0 minutes 2.259 seconds)

STEPS nowcast

This tutorial shows how to compute and plot an ensemble nowcast using Swiss radar data.

```
from pylab import *
from datetime import datetime
from pprint import pprint
from pysteps import io, nowcasts, rcparams
from pysteps.motion.lucaskanade import dense_lucaskanade
from pysteps.postprocessing.ensemblestats import excprob
from pysteps.utils import conversion, dimension, transformation
from pysteps.visualization import plot_precip_field

# Set nowcast parameters
```

(continues on next page)

(continued from previous page)

```
n_ens_members = 20
n_leadtimes = 6
seed = 24
```

Read precipitation field

First thing, the sequence of Swiss radar composites is imported, converted and transformed into units of dBR.

```
date = datetime.strptime("201701311200", "%Y%m%d%H%M")
data_source = "mch"

# Load data source config
root_path = rcparams.data_sources[data_source]["root_path"]
path_fmt = rcparams.data_sources[data_source]["path_fmt"]
fn_pattern = rcparams.data_sources[data_source]["fn_pattern"]
fn_ext = rcparams.data_sources[data_source]["fn_ext"]
importer_name = rcparams.data_sources[data_source]["importer"]
importer_kwargs = rcparams.data_sources[data_source]["importer_kwargs"]
timestep = rcparams.data_sources[data_source]["timestep"]

# Find the radar files in the archive
fns = io.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep, num_prev_files=2
)

# Read the data from the archive
importer = io.get_method(importer_name, "importer")
R, _, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

# Convert to rain rate
R, metadata = conversion.to_rainrate(R, metadata)

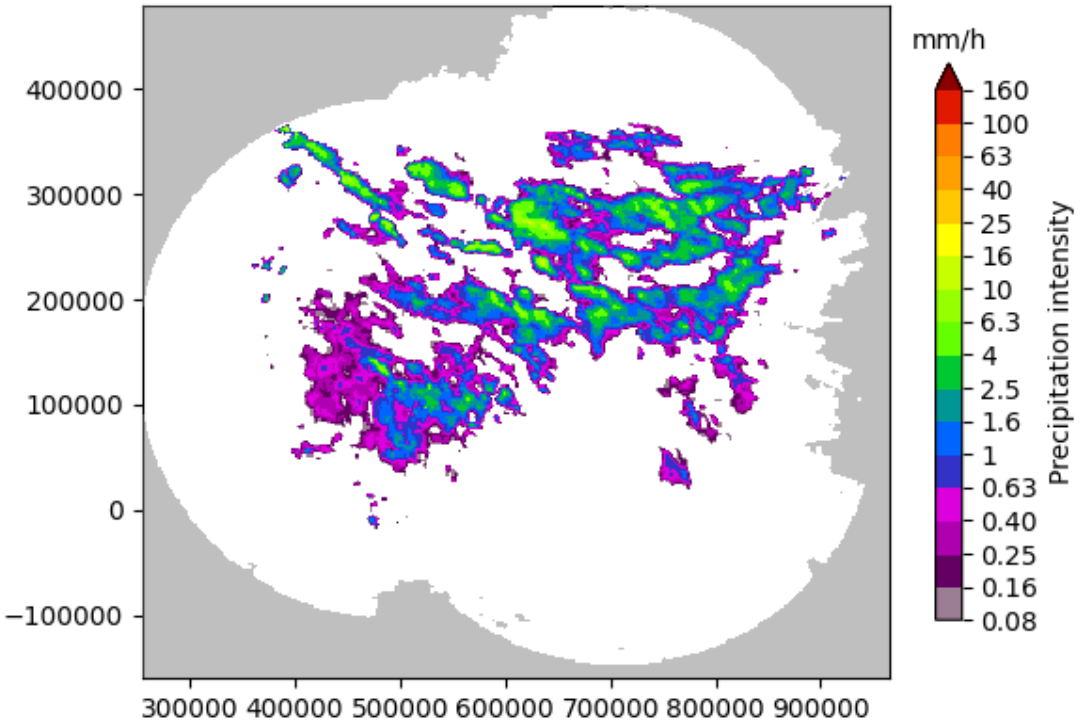
# Upscale data to 2 km to limit memory usage
R, metadata = dimension.aggregate_fields_space(R, metadata, 2000)

# Plot the rainfall field
plot_precip_field(R[-1, :, :], geodata=metadata)

# Log-transform the data to unit of dBR, set the threshold to 0.1 mm/h,
# set the fill value to -15 dBR
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)

# Set missing values with the fill value
R[~np.isfinite(R)] = -15.0

# Nicely print the metadata
pprint(metadata)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
{'accutime': 5,
 'institution': 'MeteoSwiss',
 'product': 'AQC',
 'projection': '+proj=somerc  +lon_0=7.43958333333333 +lat_0=46.9524055555556 '
               '+k_0=1 +x_0=600000 +y_0=200000 +ellps=bessel '
               '+towgs84=674.374,15.056,405.346,0,0,0,0 +units=m +no_defs',
 'threshold': -10.0,
 'timestamps': array([datetime.datetime(2017, 1, 31, 11, 50),
       datetime.datetime(2017, 1, 31, 11, 55),
       datetime.datetime(2017, 1, 31, 12, 0)], dtype=object),
 'transform': 'dB',
 'unit': 'mm/h',
 'x1': 255000.0,
 'x2': 965000.0,
 'xpixelsize': 2000,
 'y1': -160000.0,
 'y2': 480000.0,
 'yorigin': 'upper',
 'ypixelsize': 2000,
 'zerovalue': -15.0,
 'zr_a': 316.0,
```

(continues on next page)

(continued from previous page)

```
 'zr_b': 1.5}
```

Deterministic nowcast with S-PROG

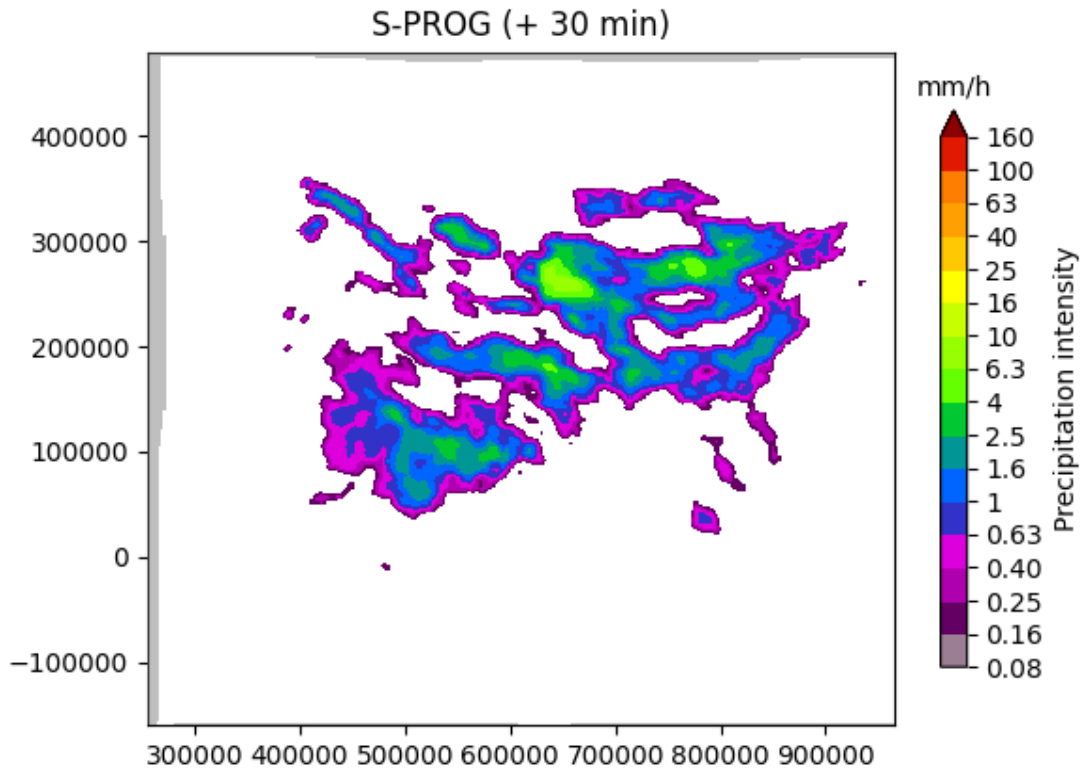
First, the motiong field is estimated using a local tracking approach based on the Lucas-Kanade optical flow. The motion field can then be used to generate a deterministic nowcast with the S-PROG model, which implements a scale filtering appraoch in order to progressively remove the unpredictable spatial scales during the forecast.

```
# Estimate the motion field
V = dense_lucaskanade(R)

# The S-PROG nowcast
nowcast_method = nowcasts.get_method("sprog")
R_f = nowcast_method(
    R[-3:, :, :],
    V,
    n_leadtimes,
    n_cascade_levels=8,
    R_thr=-10.0,
    decomp_method="fft",
    bandpass_filter_method="gaussian",
    probmatching_method="mean",
)

# Back-transform to rain rate
R_f = transformation.dB_transform(R_f, threshold=-10.0, inverse=True)[0]

# Plot the S-PROG forecast
plot_precip_field(
    R_f[-1, :, :],
    geodata=metadata,
    title="S-PROG (+ %i min)" % (n_leadtimes * timestep),
)
```



Out:

```
Computing S-PROG nowcast:
-------------------------

Inputs:
-------
input dimensions: 320x355

Methods:
--------
extrapolation:          semilagrangian
bandpass filter:        gaussian
decomposition:          fft
conditional statistics: no
probability matching:   mean
FFT method:             numpy
domain:                 spatial

Parameters:
-----------
number of time steps:     6
parallel threads:         1
number of cascade levels: 8
order of the AR(p) model: 2
precip. intensity threshold: -10
***********************************************
* Correlation coefficients for cascade levels: *
***********************************************
----------------------------------------
| Level |     Lag-1     |     Lag-2     |
```

(continues on next page)

(continued from previous page)

```
----------------------------------------
| 1     | 0.999323      | 0.997328      |
----------------------------------------
| 2     | 0.997948      | 0.992460      |
----------------------------------------
| 3     | 0.994097      | 0.981508      |
----------------------------------------
| 4     | 0.983362      | 0.951486      |
----------------------------------------
| 5     | 0.945661      | 0.850431      |
----------------------------------------
| 6     | 0.834901      | 0.648814      |
----------------------------------------
| 7     | 0.521708      | 0.334606      |
----------------------------------------
| 8     | 0.147390      | 0.052553      |
----------------------------------------
****************************************
* AR(p) parameters for cascade levels: *
****************------------------------------------------------
| Level |    Phi-1      |    Phi-2      |    Phi-0      |
----------------------------------------------------------
| 1     | 1.927722      | -0.929028     | 0.013613      |
----------------------------------------------------------
| 2     | 1.835646      | -0.839420     | 0.034797      |
----------------------------------------------------------
| 3     | 1.561780      | -0.571054     | 0.089062      |
----------------------------------------------------------
| 4     | 1.445694      | -0.470154     | 0.160328      |
----------------------------------------------------------
| 5     | 1.337822      | -0.414695     | 0.295877      |
----------------------------------------------------------
| 6     | 0.967863      | -0.159255     | 0.543376      |
----------------------------------------------------------
| 7     | 0.476960      | 0.085772      | 0.849980      |
----------------------------------------------------------
| 8     | 0.142745      | 0.031514      | 0.988587      |
----------------------------------------------------------
Starting nowcast computation.
Computing nowcast for time step 1... done.
Computing nowcast for time step 2... done.
Computing nowcast for time step 3... done.
Computing nowcast for time step 4... done.
Computing nowcast for time step 5... done.
Computing nowcast for time step 6... done.
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:240: RuntimeWarning: invalid value␣
↪encountered in less
  R[R < threshold] = zerovalue

<matplotlib.axes._subplots.AxesSubplot object at 0x7fe7d7670d90>
```

As we can see from the figure above, the forecast produced by S-PROG is a smooth field. In other words, the forecast variance is lower than the variance of the original observed field. However, certain applications demand that the forecast retain the same statistical properties of the observations. In such cases, the S-PROG forecasts are of limited use and a stochatic approach might be of more interest.

Stochastic nowcast with STEPS

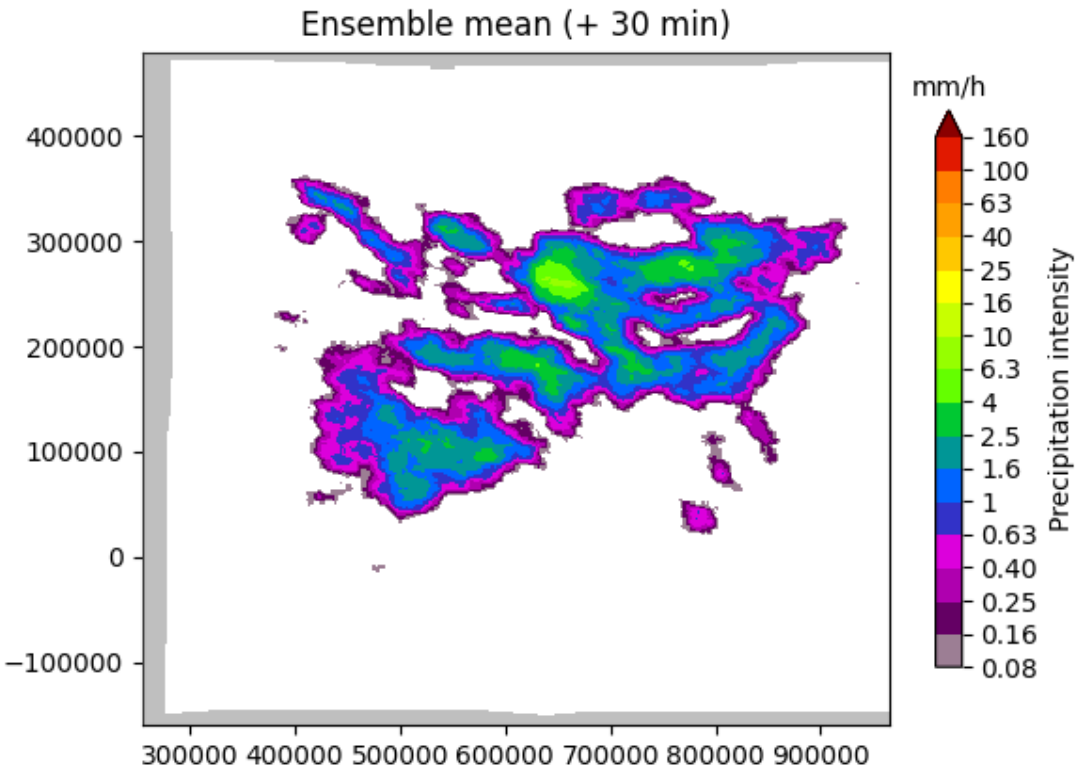
The S-PROG approach is extended to include a stochastic term which represents the variance associated to the unpredictable development of precipitation. This approach is known as STEPS (short-term ensemble prediction

system).

```
# The STEPES nowcast
nowcast_method = nowcasts.get_method("steps")
R_f = nowcast_method(
    R[-3:, :, :],
    V,
    n_leadtimes,
    n_ens_members,
    n_cascade_levels=6,
    R_thr=-10.0,
    kmperpixel=2.0,
    timestep=timestep,
    decomp_method="fft",
    bandpass_filter_method="gaussian",
    noise_method="nonparametric",
    vel_pert_method="bps",
    mask_method="incremental",
    seed=seed,
)

# Back-transform to rain rates
R_f = transformation.dB_transform(R_f, threshold=-10.0, inverse=True)[0]


# Plot the ensemble mean
R_f_mean = np.mean(R_f[:, -1, :, :], axis=0)
plot_precip_field(
    R_f_mean,
    geodata=metadata,
    title="Ensemble mean (+ %i min)" % (n_leadtimes * timestep),
)
```



Out:

```
Computing STEPS nowcast:
------------------------

Inputs:
-------
input dimensions: 320x355
km/pixel:         2
time step:        5 minutes

Methods:
--------
extrapolation:          semilagrangian
bandpass filter:        gaussian
decomposition:          fft
noise generator:        nonparametric
noise adjustment:       no
velocity perturbator:   bps
conditional statistics: no
precip. mask method:    incremental
probability matching:   cdf
FFT method:             numpy
domain:                 spatial

Parameters:
-----------
number of time steps:     6
ensemble size:            20
parallel threads:         1
number of cascade levels: 6
```

(continues on next page)

(continued from previous page)

```
order of the AR(p) model: 2
velocity perturbations, parallel:      10.88,0.23,-7.68
velocity perturbations, perpendicular: 5.76,0.31,-2.72
precip. intensity threshold: -10
*********************************************
* Correlation coefficients for cascade levels: *
*********************************************
----------------------------------------
| Level |     Lag-1     |     Lag-2     |
----------------------------------------
| 1     | 0.999339      | 0.997423      |
----------------------------------------
| 2     | 0.997571      | 0.991287      |
----------------------------------------
| 3     | 0.988960      | 0.967597      |
----------------------------------------
| 4     | 0.944632      | 0.852797      |
----------------------------------------
| 5     | 0.737254      | 0.545578      |
----------------------------------------
| 6     | 0.234025      | 0.107296      |
----------------------------------------
*************************************
* AR(p) parameters for cascade levels: *
*************************************
------------------------------------------------------
| Level |    Phi-1     |    Phi-2     |    Phi-0     |
------------------------------------------------------
| 1     | 1.928549     | -0.929825    | 0.013382     |
------------------------------------------------------
| 2     | 1.791276     | -0.795638    | 0.042197     |
------------------------------------------------------
| 3     | 1.459340     | -0.475631    | 0.130350     |
------------------------------------------------------
| 4     | 1.291461     | -0.367158    | 0.305215     |
------------------------------------------------------
| 5     | 0.733967     | 0.004458     | 0.675609     |
------------------------------------------------------
| 6     | 0.221020     | 0.055572     | 0.970728     |
------------------------------------------------------
Starting nowcast computation.
Computing nowcast for time step 1... done.
Computing nowcast for time step 2... done.
Computing nowcast for time step 3... done.
Computing nowcast for time step 4... done.
Computing nowcast for time step 5... done.
Computing nowcast for time step 6... done.

<matplotlib.axes._subplots.AxesSubplot object at 0x7fe7db968d90>
```

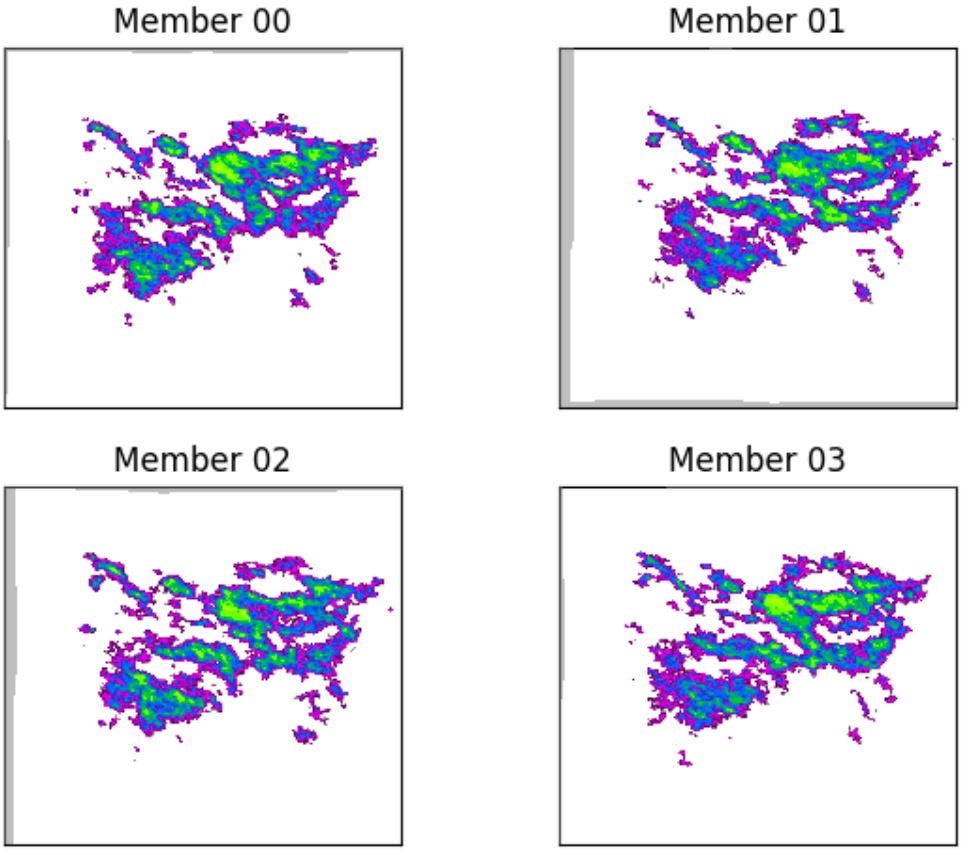
The mean of the ensemble displays similar properties as the S-PROG forecast seen above, although the degree of smoothing also depends on the ensemble size. In this sense, the S-PROG forecast can be seen as the mean of an ensemble of infinite size.

```
# Plot some of the realizations
fig = figure()
for i in range(4):
    ax = fig.add_subplot(221 + i)
    ax.set_title("Member %02d" % i)
    plot_precip_field(R_f[i, -1, :, :], geodata=metadata, colorbar=False, axis="off
↪")
```

(continues on next page)

(continued from previous page)

```
tight_layout()
```



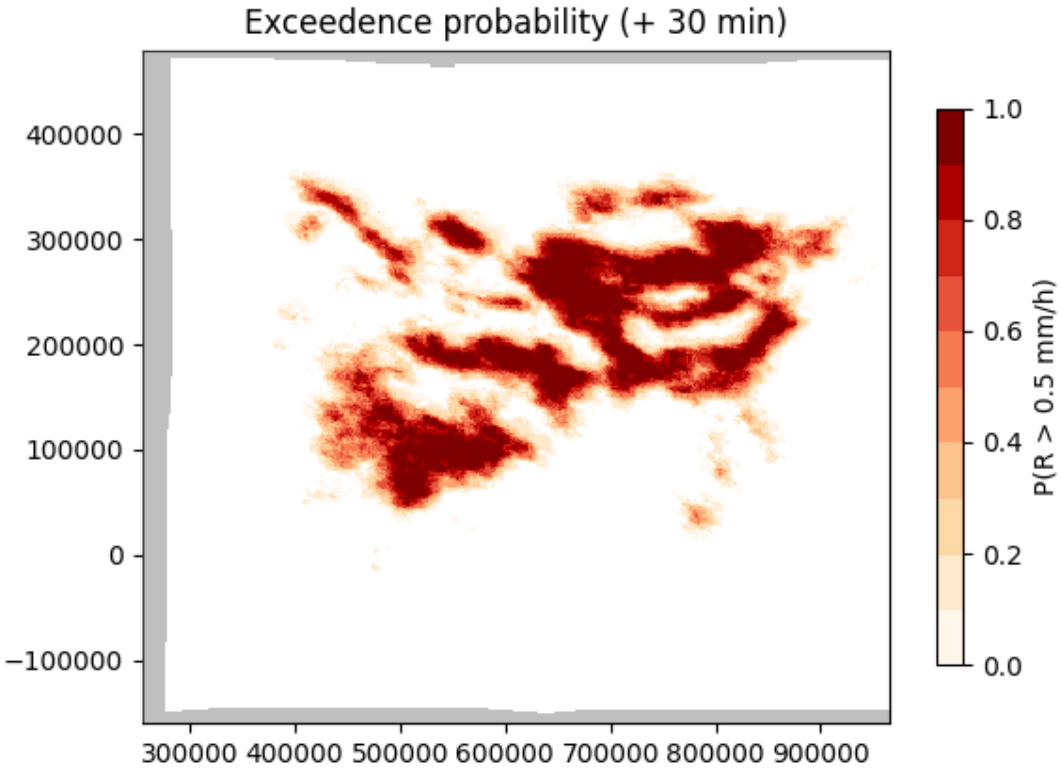
As we can see from these two members of the ensemble, the stochastic forecast mantains the same variance as in the observed rainfall field. STEPS also includes a stochatic perturbation of the motion field in order to quantify the its uncertainty.

Finally, it is possible to derive probabilities from our ensemble forecast.

```
# Compute exceedence probabilities for a 0.5 mm/h threshold
P = excprob(R_f[:, -1, :, :], 0.5)

# Plot the field of probabilities
plot_precip_field(
    P,
    geodata=metadata,
    drawlonlatlines=False,
    type="prob",
    units="mm/h",
    probthr=0.5,
    title="Exceedence probability (+ %i min)" % (n_leadtimes * timestep),
)

# sphinx_gallery_thumbnail_number = 5
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/postprocessing/ensemblestats.py:111: RuntimeWarning:␣
↪invalid value encountered in greater_equal
  X_[X >= x] = 1.0
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/postprocessing/ensemblestats.py:112: RuntimeWarning:␣
↪invalid value encountered in less
  X_[X < x] = 0.0
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:315: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 1e-3] = np.nan

<matplotlib.axes._subplots.AxesSubplot object at 0x7fe7d65e1610>
```

Total running time of the script: (0 minutes 19.553 seconds)

Ensemble verification

In this tutorial we perform a verification of a probabilistic extrapolation nowcast using MeteoSwiss radar data.

```
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
from pprint import pprint
from pysteps import io, nowcasts, rcparams, verification
from pysteps.motion.lucaskanade import dense_lucaskanade
from pysteps.postprocessing import ensemblestats
```

(continues on next page)

(continued from previous page)

```
from pysteps.utils import conversion, dimension, transformation
from pysteps.visualization import plot_precip_field
```

Read precipitation field

First, we will import the sequence of MeteoSwiss ("mch") radar composites. You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201607112100", "%Y%m%d%H%M")
data_source = rcparams.data_sources["mch"]
n_ens_members = 20
n_leadtimes = 6
seed = 24
```

Load the data from the archive

The data are upscaled to 2 km resolution to limit the memory usage and thus be able to afford a larger number of ensemble members.

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]
timestep = data_source["timestep"]

# Find the radar files in the archive
fns = io.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep, num_prev_files=2
)

# Read the data from the archive
importer = io.get_method(importer_name, "importer")
R, _, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

# Convert to rain rate
R, metadata = conversion.to_rainrate(R, metadata)

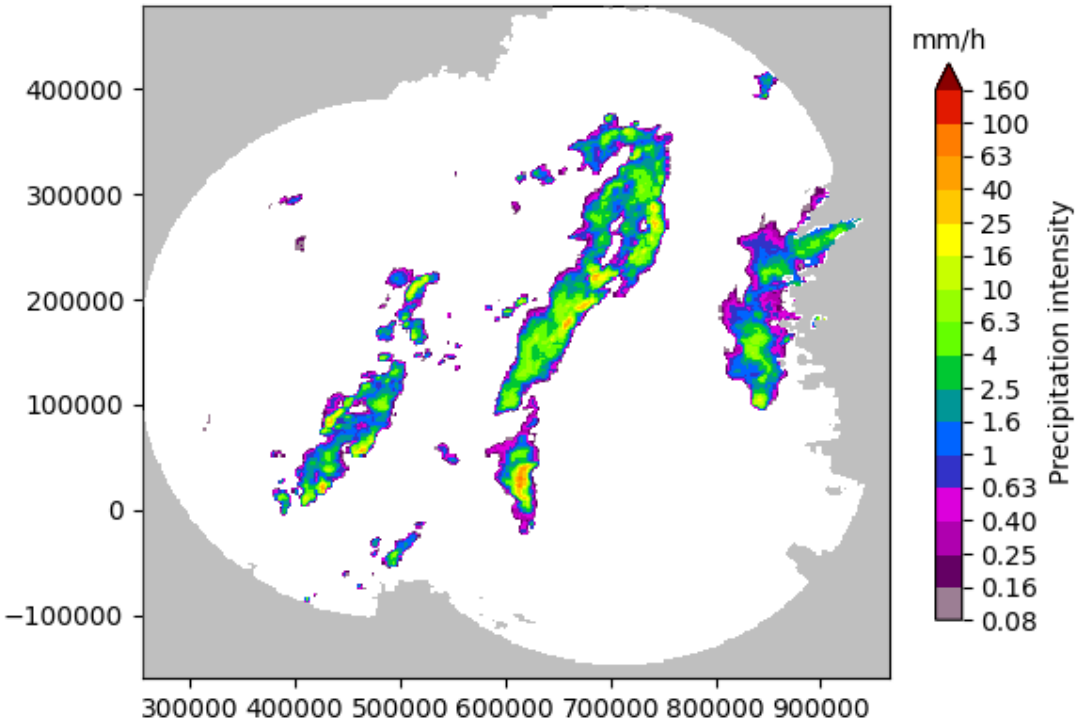
# Upscale data to 2 km
R, metadata = dimension.aggregate_fields_space(R, metadata, 2000)

# Plot the rainfall field
plot_precip_field(R[-1, :, :], geodata=metadata)
plt.show()

# Log-transform the data to unit of dBR, set the threshold to 0.1 mm/h,
# set the fill value to -15 dBR
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)

# Set missing values with the fill value
R[~np.isfinite(R)] = -15.0

# Nicely print the metadata
pprint(metadata)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
{'accutime': 5,
 'institution': 'MeteoSwiss',
 'product': 'AQC',
 'projection': '+proj=somerc  +lon_0=7.43958333333333 +lat_0=46.9524055555556 '
               '+k_0=1 +x_0=600000 +y_0=200000 +ellps=bessel '
               '+towgs84=674.374,15.056,405.346,0,0,0,0 +units=m +no_defs',
 'threshold': -10.0,
 'timestamps': array([datetime.datetime(2016, 7, 11, 20, 50),
       datetime.datetime(2016, 7, 11, 20, 55),
       datetime.datetime(2016, 7, 11, 21, 0)], dtype=object),
 'transform': 'dB',
 'unit': 'mm/h',
 'x1': 255000.0,
 'x2': 965000.0,
 'xpixelsize': 2000,
 'y1': -160000.0,
 'y2': 480000.0,
 'yorigin': 'upper',
 'ypixelsize': 2000,
 'zerovalue': -15.0,
 'zr_a': 316.0,
```

(continues on next page)

(continued from previous page)

```
 'zr_b': 1.5}
```

Forecast

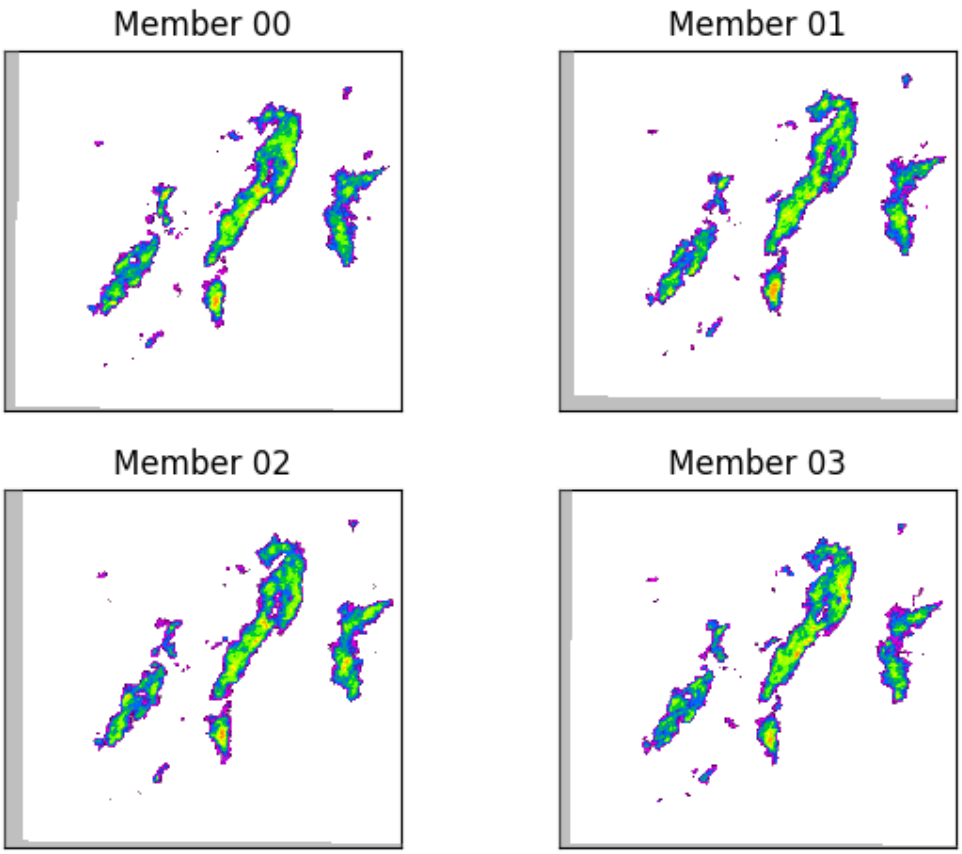
We use the STEPS approach to produce a ensemble nowcast of precipitation fields.

```
# Estimate the motion field
V = dense_lucaskanade(R)

# Perform the ensemble nowcast with STEPS
nowcast_method = nowcasts.get_method("steps")
R_f = nowcast_method(
    R[-3:, :, :],
    V,
    n_leadtimes,
    n_ens_members,
    n_cascade_levels=6,
    R_thr=-10.0,
    kmperpixel=2.0,
    timestep=timestep,
    decomp_method="fft",
    bandpass_filter_method="gaussian",
    noise_method="nonparametric",
    vel_pert_method="bps",
    mask_method="incremental",
    seed=seed,
)

# Back-transform to rain rates
R_f = transformation.dB_transform(R_f, threshold=-10.0, inverse=True)[0]

# Plot some of the realizations
fig = plt.figure()
for i in range(4):
    ax = fig.add_subplot(221 + i)
    ax.set_title("Member %02d" % i)
    plot_precip_field(R_f[i, -1, :, :], geodata=metadata, colorbar=False, axis="off
↪")
plt.tight_layout()
plt.show()
```



Out:

```
Computing STEPS nowcast:
------------------------

Inputs:
-------
input dimensions: 320x355
km/pixel:         2
time step:        5 minutes

Methods:
--------
extrapolation:          semilagrangian
bandpass filter:        gaussian
decomposition:          fft
noise generator:        nonparametric
noise adjustment:       no
velocity perturbator:   bps
conditional statistics: no
precip. mask method:    incremental
probability matching:   cdf
FFT method:             numpy
domain:                 spatial

Parameters:
-----------
number of time steps:     6
ensemble size:            20
parallel threads:         1
number of cascade levels: 6
```

(continues on next page)

(continued from previous page)

```
order of the AR(p) model: 2
velocity perturbations, parallel:      10.88,0.23,-7.68
velocity perturbations, perpendicular: 5.76,0.31,-2.72
precip. intensity threshold: -10
*********************************************
* Correlation coefficients for cascade levels: *
*********************************************
----------------------------------------
| Level |     Lag-1     |     Lag-2     |
----------------------------------------
| 1     | 0.999070      | 0.996933      |
----------------------------------------
| 2     | 0.997782      | 0.994111      |
----------------------------------------
| 3     | 0.993430      | 0.984036      |
----------------------------------------
| 4     | 0.959103      | 0.897505      |
----------------------------------------
| 5     | 0.780676      | 0.600230      |
----------------------------------------
| 6     | 0.302027      | 0.119775      |
----------------------------------------
*************************************
* AR(p) parameters for cascade levels: *
*************************************
------------------------------------------------------
| Level |    Phi-1     |    Phi-2     |    Phi-0     |
------------------------------------------------------
| 1     | 1.647721     | -0.649255    | 0.032797     |
------------------------------------------------------
| 2     | 1.325832     | -0.328780    | 0.062870     |
------------------------------------------------------
| 3     | 1.210969     | -0.218977    | 0.111662     |
------------------------------------------------------
| 4     | 1.226924     | -0.279241    | 0.271798     |
------------------------------------------------------
| 5     | 0.799113     | -0.023618    | 0.624762     |
------------------------------------------------------
| 6     | 0.292537     | 0.031421     | 0.952829     |
------------------------------------------------------
Starting nowcast computation.
Computing nowcast for time step 1... done.
Computing nowcast for time step 2... done.
Computing nowcast for time step 3... done.
Computing nowcast for time step 4... done.
Computing nowcast for time step 5... done.
Computing nowcast for time step 6... done.
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:240: RuntimeWarning: invalid value␣
↪encountered in less
  R[R < threshold] = zerovalue
```

Verification

Pysteps includes a number of verification metrics to help users to analyze the general characteristics of the nowcasts in terms of consistency and quality (or goodness). Here, we will verify our probabilistic forecasts using the ROC curve, reliability diagrams, and rank histograms, as implemented in the verification module of pysteps.

```
# Find the files containing the verifying observations
fns = io.archive.find_by_date(
    date,
```

(continues on next page)

(continued from previous page)

```
    root_path,
    path_fmt,
    fn_pattern,
    fn_ext,
    timestep,
    0,
    num_next_files=n_leadtimes,
)

# Read the observations
R_o, _, metadata_o = io.read_timeseries(fns, importer, **importer_kwargs)

# Convert to mm/h
R_o, metadata_o = conversion.to_rainrate(R_o, metadata_o)

# Upscale data to 2 km
R_o, metadata_o = dimension.aggregate_fields_space(R_o, metadata_o, 2000)

# Compute the verification for the last lead time

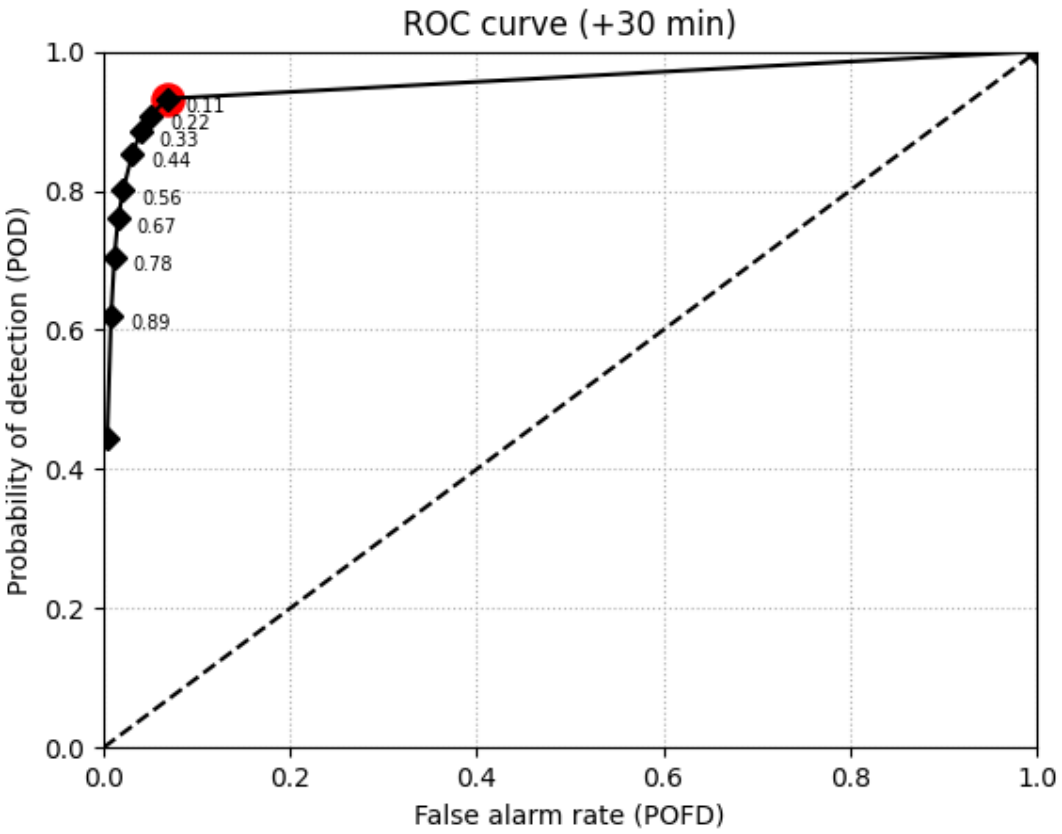
# compute the exceedance probability of 0.1 mm/h from the ensemble
P_f = ensemblestats.excprob(R_f[:, -1, :, :], 0.1, ignore_nan=True)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/postprocessing/ensemblestats.py:111: RuntimeWarning:␣
↪invalid value encountered in greater_equal
  X_[X >= x] = 1.0
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/postprocessing/ensemblestats.py:112: RuntimeWarning:␣
↪invalid value encountered in less
  X_[X < x] = 0.0
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/postprocessing/ensemblestats.py:115: RuntimeWarning: Mean␣
↪of empty slice
  P.append(np.nanmean(X_, axis=0))
```

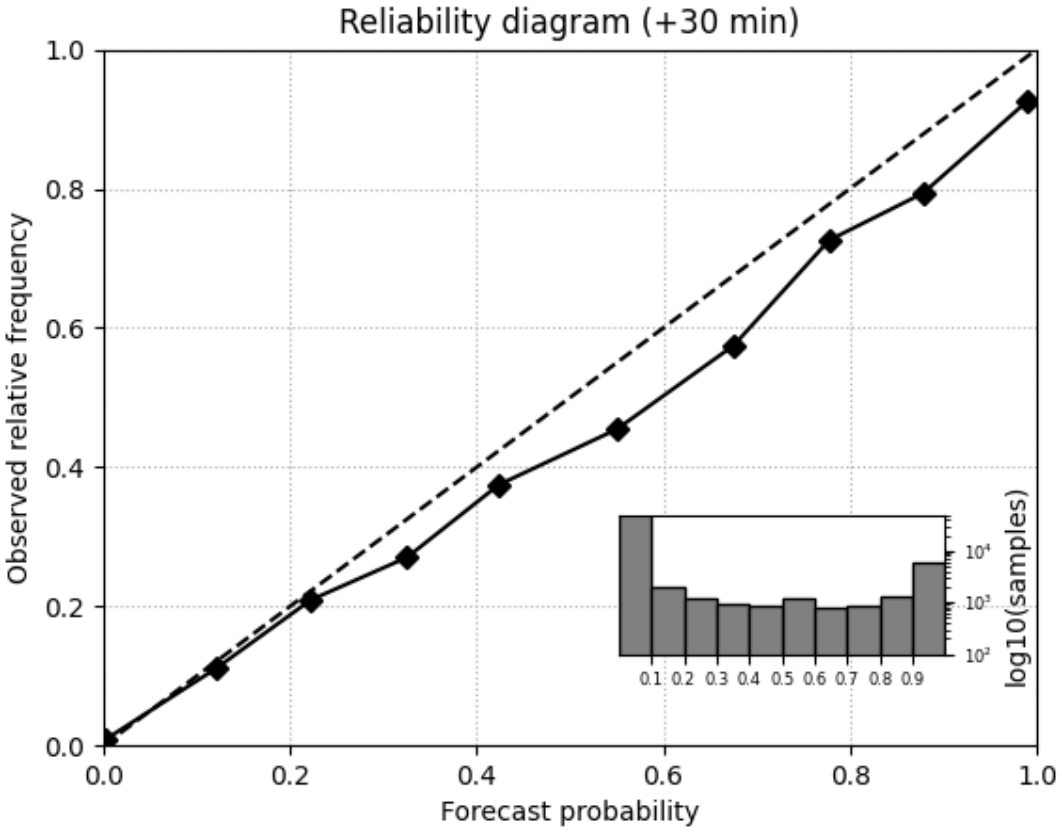
ROC curve

```
roc = verification.ROC_curve_init(0.1, n_prob_thrs=10)
verification.ROC_curve_accum(roc, P_f, R_o[-1, :, :])
fig, ax = plt.subplots()
verification.plot_ROC(roc, ax, opt_prob_thr=True)
ax.set_title("ROC curve (+%i min)" % (n_leadtimes * timestep))
plt.show()
```



Reliability diagram

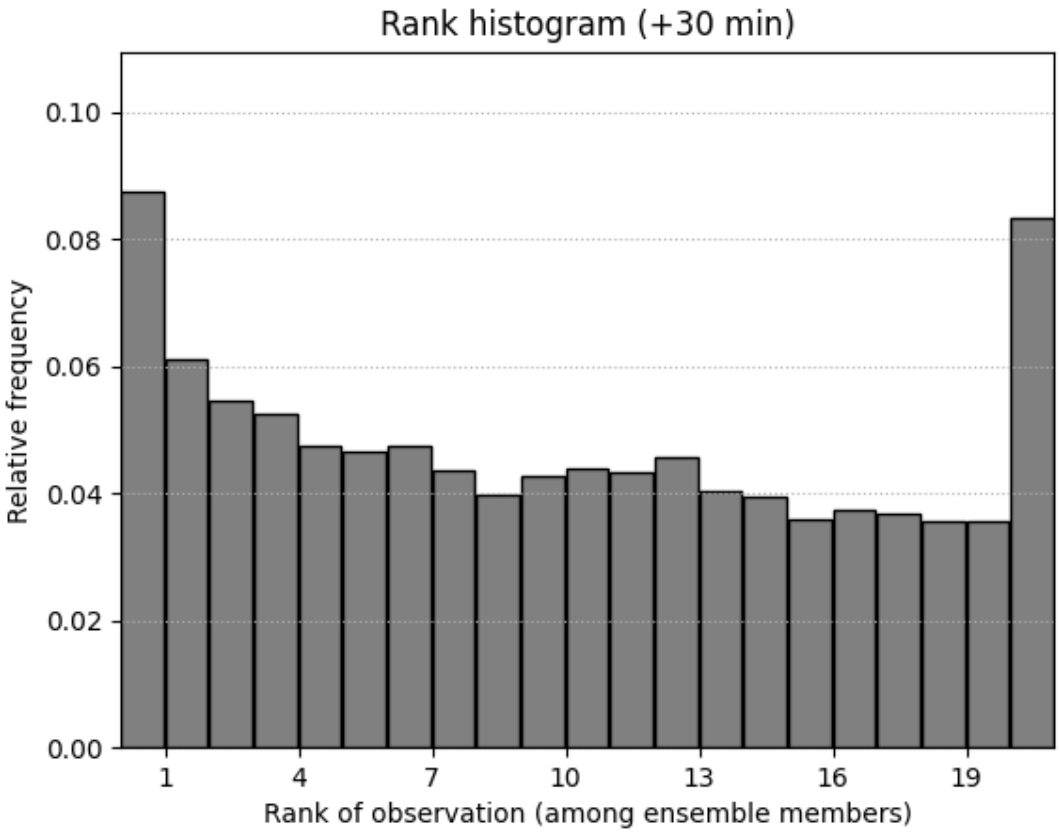
```
reldiag = verification.reldiag_init(0.1)
verification.reldiag_accum(reldiag, P_f, R_o[-1, :, :])
fig, ax = plt.subplots()
verification.plot_reldiag(reldiag, ax)
ax.set_title("Reliability diagram (+%i min)" % (n_leadtimes * timestep))
plt.show()
```



Rank histogram

```
rankhist = verification.rankhist_init(R_f.shape[0], 0.1)
verification.rankhist_accum(rankhist, R_f[:, -1, :, :], R_o[-1, :, :])
fig, ax = plt.subplots()
verification.plot_rankhist(rankhist, ax)
ax.set_title("Rank histogram (+%i min)" % (n_leadtimes * timestep))
plt.show()

# sphinx_gallery_thumbnail_number = 5
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/verification/ensscores.py:216: RuntimeWarning: invalid␣
↪value encountered in greater_equal
  mask_nz = np.logical_or(X_o >= X_min, np.any(X_f >= X_min, axis=1))
```

Total running time of the script: (0 minutes 18.969 seconds)

Handling of no-data in Lucas-Kanade

Areas of missing data in radar images are typically caused by visibility limits such as beam blockage and the radar coverage itself. These artifacts can mislead the echo tracking algorithms. For instance, precipitation leaving the domain might be erroneously detected as having nearly stationary velocity.

This example shows how the Lucas-Kanade algorithm can be tuned to avoid the erroneous interpretation of velocities near the maximum range of the radars by buffering the no-data mask in the radar image in order to exclude all vectors detected nearby no-data areas.

```
from datetime import datetime
from matplotlib import cm, colors

import matplotlib.pyplot as plt
import numpy as np

from pysteps import io, motion, nowcasts, rcparams, verification
from pysteps.utils import conversion, transformation
from pysteps.visualization import plot_precip_field, quiver
```

Read the radar input images

First, we will import the sequence of radar composites. You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201607112100", "%Y%m%d%H%M")
data_source = rcparams.data_sources["mch"]
```

Load the data from the archive

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]
timestep = data_source["timestep"]

# Find the two input files from the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_prev_files=1
)

# Read the radar composites
importer = io.get_method(importer_name, "importer")
R, quality, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

del quality  # Not used
```

Preprocess the data

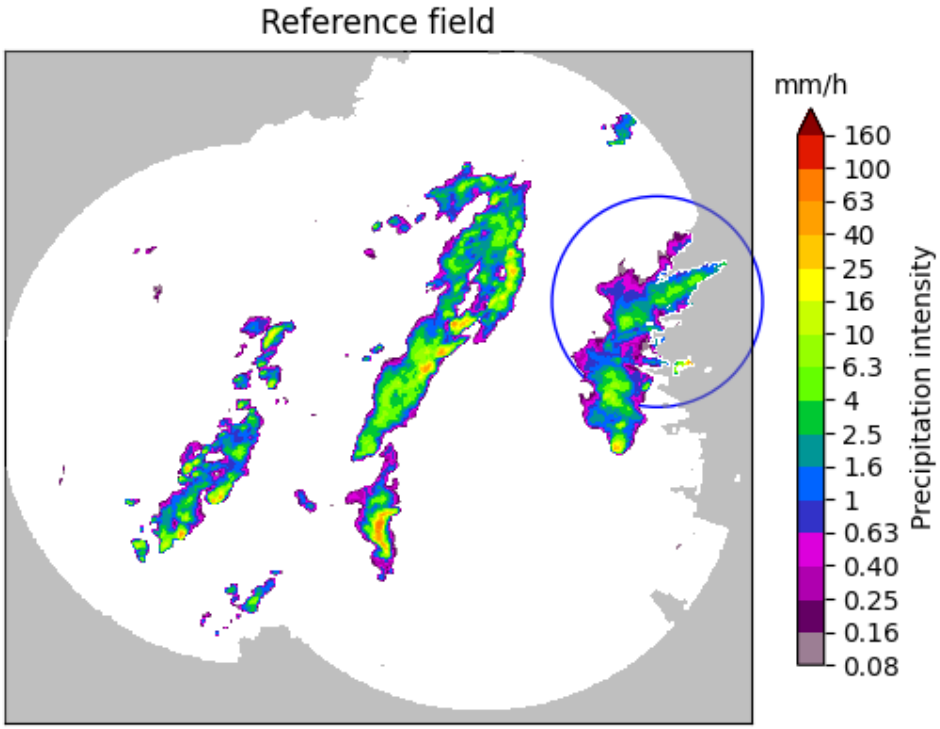
```
# Convert to mm/h
R, metadata = conversion.to_rainrate(R, metadata)

# Keep the reference frame in mm/h and its mask (for plotting purposes)
ref_mm = R[0, :, :].copy()
mask = np.ones(ref_mm.shape)
mask[~np.isnan(ref_mm)] = np.nan

# Log-transform the data [dBR]
R, metadata = transformation.dB_transform(R, metadata, threshold=0.1, zerovalue=-
↪15.0)

# Keep the reference frame in dBR (for plotting purposes)
ref_dbr = R[0].copy()
ref_dbr[ref_dbr < -10] = np.nan

# Plot the reference field
plot_precip_field(ref_mm, title="Reference field")
circle = plt.Circle((620, 400), 100, color="b", clip_on=False, fill=False)
plt.gca().add_artist(circle)
plt.show()
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:210: RuntimeWarning: invalid value␣
↪encountered in less
  zeros = R < threshold
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/checkouts/v1.3.1/examples/
↪LK_buffer_mask.py:79: RuntimeWarning: invalid value encountered in less
  ref_dbr[ref_dbr < -10] = np.nan
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/visualization/precipfields.py:311: RuntimeWarning: invalid␣
↪value encountered in less
  R[R < 0.1] = np.nan
```

Notice the "half-in, half-out" precipitation area within the blue circle. As we are going to show next, the tracking algorithm can erroneously interpret precipitation leaving the domain as stationary motion.

Also note that the radar image includes NaNs in areas of missing data. These are used by the optical flow algorithm to define the radar mask.

Sparse Lucas-Kanade

By setting the optional argument `dense=False` in `xy, uv = dense_lucaskanade(...)`, the LK algorithm returns the motion vectors detected by the Lucas-Kanade scheme without interpolating them on the grid. This allows us to better identify the presence of wrongly detected stationary motion in areas where precipitation is leaving the domain (look for the red dots within the blue circle in the figure below).

```
# Get Lucas-Kanade optical flow method
dense_lucaskanade = motion.get_method("LK")

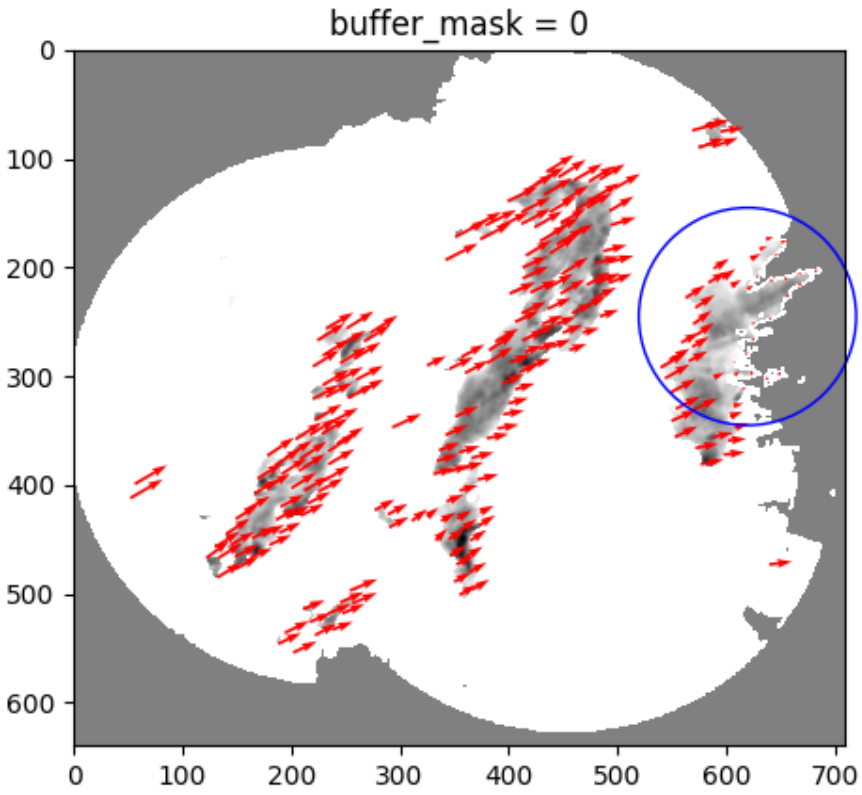
# Mask invalid values
```

(continues on next page)

(continued from previous page)

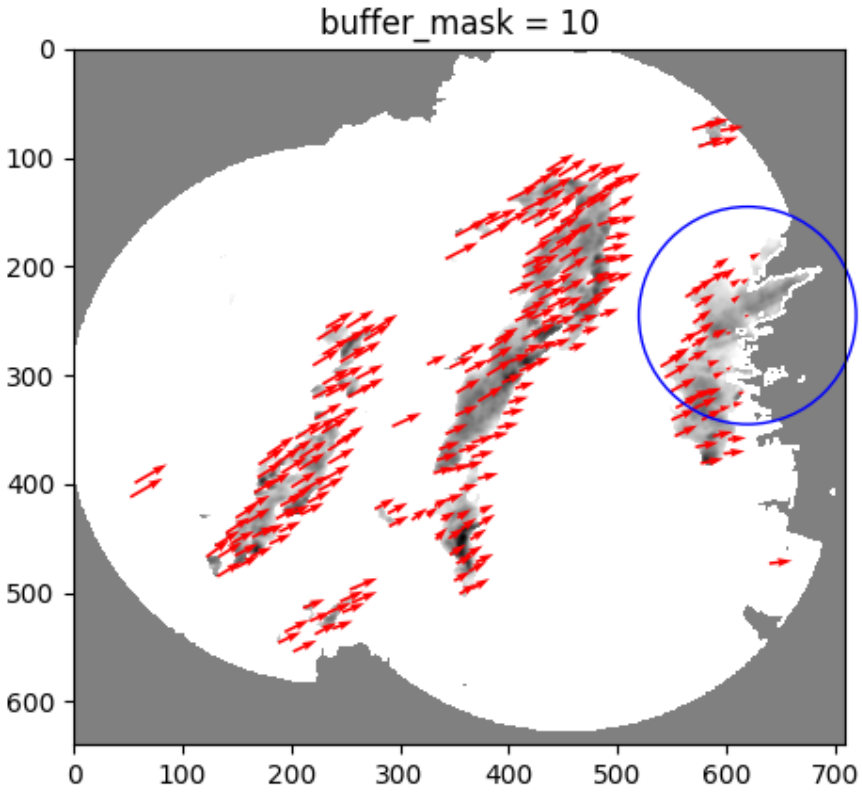
```
R = np.ma.masked_invalid(R)

# Use no buffering of the radar mask
fd_kwargs1 = {"buffer_mask": 0}
xy, uv = dense_lucaskanade(R, dense=False, fd_kwargs=fd_kwargs1)
plt.imshow(ref_dbr, cmap=plt.get_cmap("Greys"))
plt.imshow(mask, cmap=colors.ListedColormap(["black"]), alpha=0.5)
plt.quiver(
    xy[:, 0],
    xy[:, 1],
    uv[:, 0],
    uv[:, 1],
    color="red",
    angles="xy",
    scale_units="xy",
    scale=0.2,
)
circle = plt.Circle((620, 245), 100, color="b", clip_on=False, fill=False)
plt.gca().add_artist(circle)
plt.title("buffer_mask = 0")
plt.show()
```



The LK algorithm cannot distinguish missing values from no precipitation, that is, no-data are the same as no-echoes. As a result, the fixed boundaries produced by precipitation in contact with no-data areas are interpreted as stationary motion. One way to mitigate this effect of the boundaries is to introduce a slight buffer of the no-data mask so that the algorithm will ignore all the portions of the radar domain that are nearby no-data areas. This buffer can be set by the keyword argument `buffer_mask` within the feature detection optional arguments `fd_kwargs`. Note that by default `dense_lucaskanade` uses a 5-pixel buffer.

```
# with buffer
buffer = 10
fd_kwargs2 = {"buffer_mask": buffer}
xy, uv = dense_lucaskanade(R, dense=False, fd_kwargs=fd_kwargs2)
plt.imshow(ref_dbr, cmap=plt.get_cmap("Greys"))
plt.imshow(mask, cmap=colors.ListedColormap(["black"]), alpha=0.5)
plt.quiver(
    xy[:, 0],
    xy[:, 1],
    uv[:, 0],
    uv[:, 1],
    color="red",
    angles="xy",
    scale_units="xy",
    scale=0.2,
)
circle = plt.Circle((620, 245), 100, color="b", clip_on=False, fill=False)
plt.gca().add_artist(circle)
plt.title("buffer_mask = %i" % buffer)
plt.show()
```



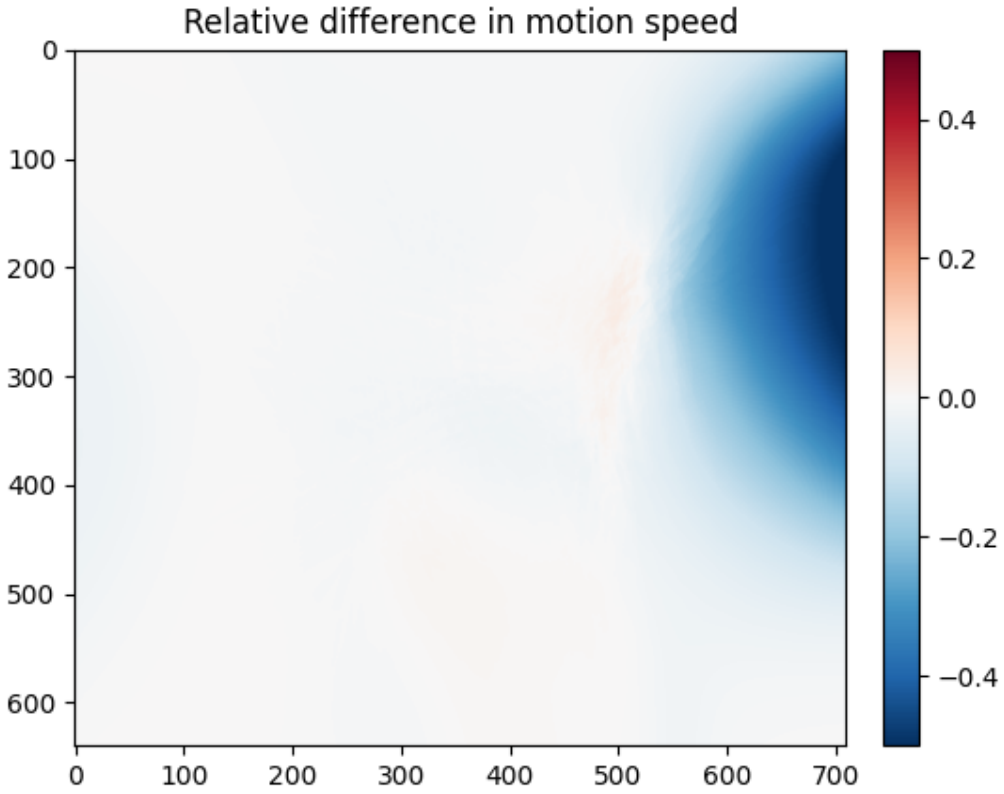
Dense Lucas-Kanade

The above displacement vectors produced by the Lucas-Kanade method are now interpolated to produce a full field of motion (i.e., `dense=True`). By comparing the velocity of the motion fields, we can easily notice the negative bias that is introduced by the the erroneous interpretation of velocities near the maximum range of the radars. Please note that we are setting a small shape parameter `epsilon` for the interpolation routine. This will produce a smoother motion field.

```
interp_kwargs = {"epsilon": 5}  # use a small shape parameter for interpolation
UV1 = dense_lucaskanade(
    R, dense=True, fd_kwargs=fd_kwargs1, interp_kwargs=interp_kwargs
)
UV2 = dense_lucaskanade(
    R, dense=True, fd_kwargs=fd_kwargs2, interp_kwargs=interp_kwargs
)

V1 = np.sqrt(UV1[0] ** 2 + UV1[1] ** 2)
V2 = np.sqrt(UV2[0] ** 2 + UV2[1] ** 2)

plt.imshow((V1 - V2) / V2, cmap=cm.RdBu_r, vmin=-0.5, vmax=0.5)
plt.colorbar(fraction=0.04, pad=0.04)
plt.title("Relative difference in motion speed")
plt.show()
```



Notice how the presence of erroneous velocity vectors produces a significantly slower motion field near the right edge of the domain.

Forecast skill

We are now going to evaluate the benefit of buffering the radar mask by computing the forecast skill in terms of the Spearman correlation coefficient. The extrapolation forecasts are computed using the dense UV motion fields estimated above.

```
# Get the advection routine and extrapolate the last radar frame by 12 time steps
# (i.e., 1 hour lead time)
extrapolate = nowcasts.get_method("extrapolation")
R[~np.isfinite(R)] = metadata["zerovalue"]
R_f1 = extrapolate(R[-1], UV1, 12)
```

(continues on next page)

(continued from previous page)

```
R_f2 = extrapolate(R[-1], UV2, 12)

# Back-transform to rain rate
R_f1 = transformation.dB_transform(R_f1, threshold=-10.0, inverse=True)[0]
R_f2 = transformation.dB_transform(R_f2, threshold=-10.0, inverse=True)[0]

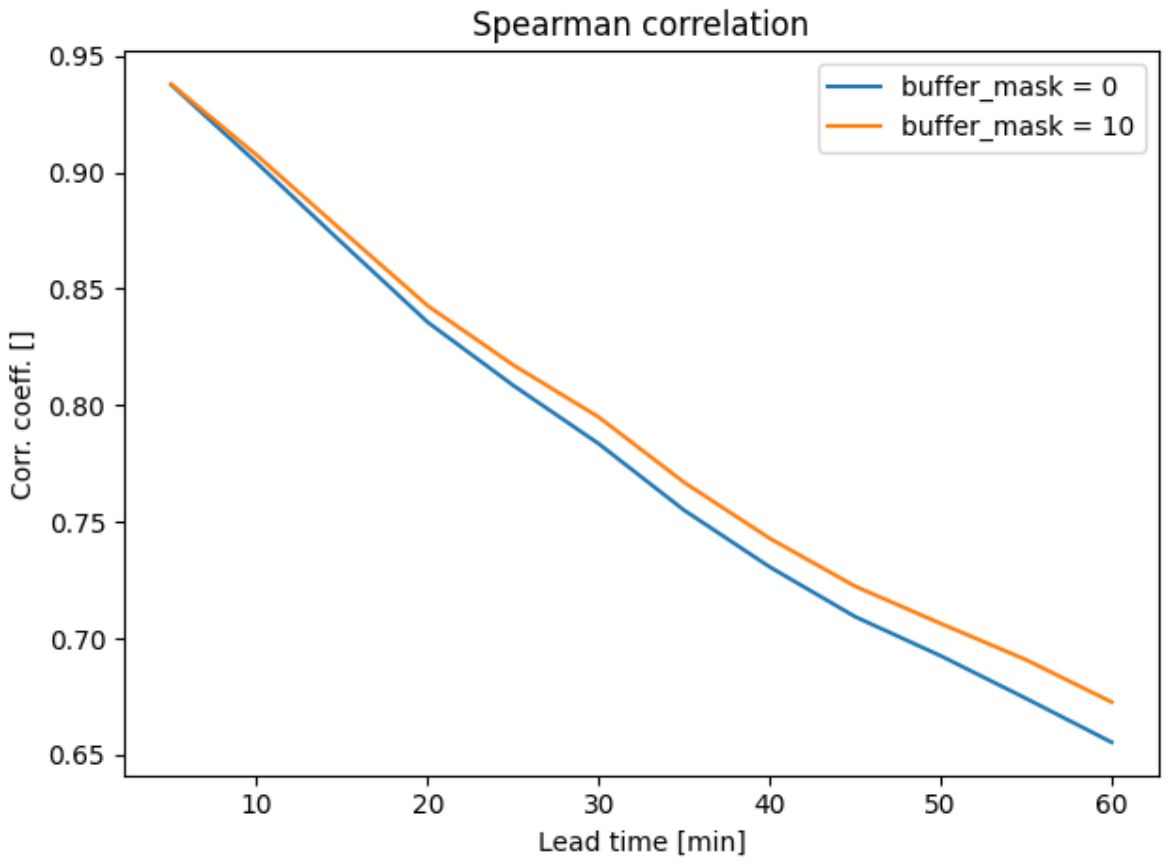
# Find the veriyfing observations in the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_next_files=12,
)

# Read and convert the radar composites
R_o, _, metadata_o = io.read_timeseries(fns, importer, **importer_kwargs)
R_o, metadata_o = conversion.to_rainrate(R_o, metadata_o)

# Compute Spearman correlation
skill = verification.get_method("corr_s")
score_1 = []
score_2 = []
for i in range(12):
    score_1.append(skill(R_f1[i, :, :], R_o[i + 1, :, :])["corr_s"])
    score_2.append(skill(R_f2[i, :, :], R_o[i + 1, :, :])["corr_s"])

x = (np.arange(12) + 1) * 5  # [min]
plt.plot(x, score_1, label="buffer_mask = 0")
plt.plot(x, score_2, label="buffer_mask = %i" % buffer)
plt.legend()
plt.xlabel("Lead time [min]")
plt.ylabel("Corr. coeff. []")
plt.title("Spearman correlation")

plt.tight_layout()
plt.show()
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pysteps/envs/v1.3.1/lib/python3.8/
↪site-packages/pysteps/utils/transformation.py:240: RuntimeWarning: invalid value␣
↪encountered in less
  R[R < threshold] = zerovalue
```

As expected, the corrected motion field produces better forecast skill already within the first hour into the nowcast.

```
# sphinx_gallery_thumbnail_number = 2
```

Total running time of the script: (0 minutes 18.055 seconds)

ANVIL nowcast

This example demonstrates how to use ANVIL and the advantages compared to extrapolation nowcast and S-PROG.

Load the libraries.

```
from datetime import datetime, timedelta
import warnings

warnings.simplefilter("ignore")
import matplotlib.pyplot as plt
import numpy as np
from pysteps import motion, io, rcparams, utils
from pysteps.nowcasts import anvil, extrapolation, sprog
from pysteps.visualization import plot_precip_field
```

Read the input data

ANVIL was originally developed to use vertically integrated liquid (VIL) as the input data, but the model allows using any two-dimensional input fields. Here we use a composite of rain rates.

```
date = datetime.strptime("201505151620", "%Y%m%d%H%M")

# Read the data source information from rcparams
data_source = rcparams.data_sources["mch"]

root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]

# Find the input files in the archive. Use history length of 5 timesteps
filenames = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_prev_files=5
)

# Read the input time series
importer = io.get_method(importer_name, "importer")
rainrate_field, quality, metadata = io.read_timeseries(
    filenames, importer, **importer_kwargs
)

# Convert to rain rate (mm/h)
rainrate_field, metadata = utils.to_rainrate(rainrate_field, metadata)
```

Compute the advection field

Apply the Lucas-Kanade method with the parameters given in Pulkkinen et al. (2020) to compute the advection field.

```
fd_kwargs = {}
fd_kwargs["max_corners"] = 1000
fd_kwargs["quality_level"] = 0.01
fd_kwargs["min_distance"] = 2
fd_kwargs["block_size"] = 8

lk_kwargs = {}
lk_kwargs["winsize"] = (15, 15)

oflow_kwargs = {}
oflow_kwargs["fd_kwargs"] = fd_kwargs
oflow_kwargs["lk_kwargs"] = lk_kwargs
oflow_kwargs["decl_scale"] = 10

oflow = motion.get_method("lucaskanade")

# transform the input data to logarithmic scale
rainrate_field_log, _ = utils.transformation.dB_transform(
    rainrate_field, metadata=metadata
)
velocity = oflow(rainrate_field_log, **oflow_kwargs)
```

Compute the nowcasts and threshold rain rates below 0.5 mm/h

```
forecast_extrap = extrapolation.forecast(
    rainrate_field[-1], velocity, 3, extrap_kwargs={"allow_nonfinite_values": True}
)
forecast_extrap[forecast_extrap < 0.5] = 0.0

rainrate_field_finite = rainrate_field.copy()
rainrate_field_finite[~np.isfinite(rainrate_field_finite)] = 0.0
forecast_sprog = sprog.forecast(
    rainrate_field_finite[-3:], velocity, 3, n_cascade_levels=8, R_thr=0.5
)
forecast_sprog[~np.isfinite(forecast_extrap)] = np.nan
forecast_sprog[forecast_sprog < 0.5] = 0.0

forecast_anvil = anvil.forecast(
    rainrate_field[-4:], velocity, 3, ar_window_radius=25, ar_order=2
)
forecast_anvil[forecast_anvil < 0.5] = 0.0
```

Out:

```
Computing S-PROG nowcast:
-------------------------

Inputs:
-------
input dimensions: 640x710

Methods:
--------
extrapolation:          semilagrangian
bandpass filter:        gaussian
decomposition:          fft
conditional statistics: no
probability matching:   mean
FFT method:             numpy
domain:                 spatial

Parameters:
-----------
number of time steps:     3
parallel threads:         1
number of cascade levels: 8
order of the AR(p) model: 2
precip. intensity threshold: 0.5
***********************************************
* Correlation coefficients for cascade levels: *
***********************************************
-----------------------------------------
| Level |     Lag-1     |     Lag-2     |
-----------------------------------------
| 1     | 0.999397      | 0.997910      |
-----------------------------------------
| 2     | 0.996562      | 0.985136      |
-----------------------------------------
| 3     | 0.963618      | 0.874269      |
-----------------------------------------
| 4     | 0.951529      | 0.844201      |
-----------------------------------------
| 5     | 0.900574      | 0.723940      |
-----------------------------------------
```

(continues on next page)

(continued from previous page)

```
| 6     | 0.710953      | 0.373262      |
----------------------------------------
| 7     | 0.526338      | 0.253881      |
----------------------------------------
| 8     | 0.083348      | -0.049399     |
----------------------------------------
**************************************
* AR(p) parameters for cascade levels: *
**************************************
-------------------------------------------------------
| Level |    Phi-1     |    Phi-2     |    Phi-0     |
-------------------------------------------------------
| 1     | 1.731371     | -0.732417    | 0.023650     |
-------------------------------------------------------
| 2     | 1.840638     | -0.846987    | 0.044041     |
-------------------------------------------------------
| 3     | 1.520762     | -0.578179    | 0.218079     |
-------------------------------------------------------
| 4     | 1.455430     | -0.529569    | 0.260891     |
-------------------------------------------------------
| 5     | 1.255416     | -0.394017    | 0.399536     |
-------------------------------------------------------
| 6     | 0.834825     | -0.174233    | 0.692483     |
-------------------------------------------------------
| 7     | 0.543193     | -0.032022    | 0.849839     |
-------------------------------------------------------
| 8     | 0.083493     | -0.001743    | 0.996519     |
-------------------------------------------------------
Starting nowcast computation.
Computing nowcast for time step 1... done.
Computing nowcast for time step 2... done.
Computing nowcast for time step 3... done.
Computing ANVIL nowcast:
------------------------

Inputs:
-------
input dimensions: 640x710

Methods:
--------
extrapolation:   semilagrangian
FFT:             numpy

Parameters:
-----------
number of time steps:        3
parallel threads:            1
number of cascade levels:    8
order of the ARI(p,1) model: 2
ARI(p,1) window radius:      25
R(VIL) window radius:        3
Starting nowcast computation.
Computing nowcast for time step 1... done.
Computing nowcast for time step 2... done.
Computing nowcast for time step 3... done.
```

Read the reference observation field and threshold rain rates below 0.5 mm/h

```
filenames = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_next_files=3
)

refobs_field, quality, metadata = io.read_timeseries(
    filenames, importer, **importer_kwargs
)

refobs_field, metadata = utils.to_rainrate(refobs_field[-1], metadata)
refobs_field[refobs_field < 0.5] = 0.0
```

Plot the extrapolation, S-PROG and ANVIL nowcasts.

For comparison, the observed rain rate fields are also plotted. Growth and decay areas are marked with red and blue circles, respectively.

```
def plot_growth_decay_circles(ax):
    circle = plt.Circle(
        (360, 300), 25, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (420, 350), 30, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (405, 380), 30, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (420, 500), 25, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (480, 535), 30, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (330, 470), 35, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (505, 205), 30, color="b", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (440, 180), 30, color="r", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (590, 240), 30, color="r", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)
    circle = plt.Circle(
        (585, 160), 15, color="r", clip_on=False, fill=False, zorder=1e9
    )
    ax.add_artist(circle)


fig = plt.figure(figsize=(10, 13))
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(321)
rainrate_field[-1][rainrate_field[-1] < 0.5] = 0.0
plot_precip_field(rainrate_field[-1])
plot_growth_decay_circles(ax)
ax.set_title("Obs. %s" % str(date))

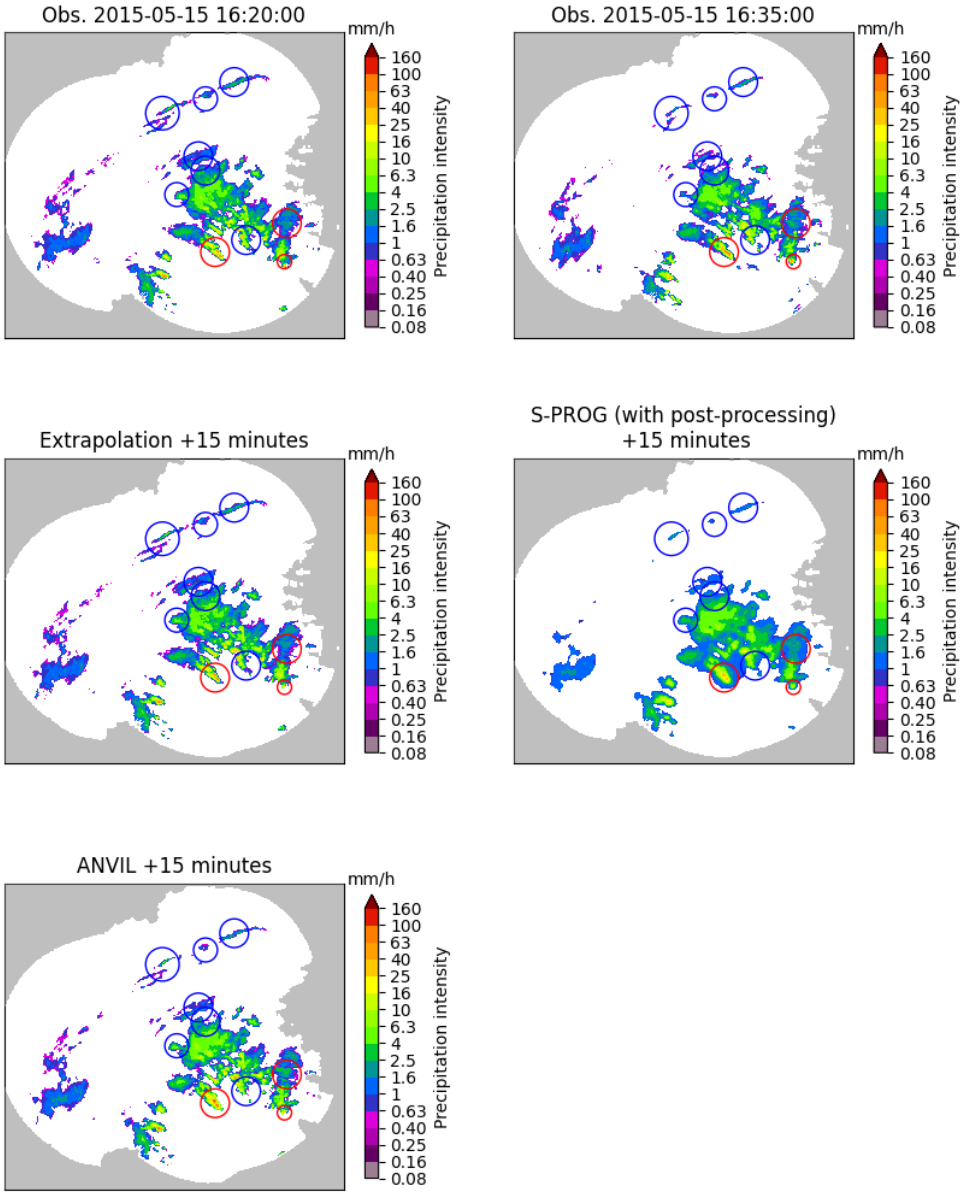
ax = fig.add_subplot(322)
plot_precip_field(refobs_field)
plot_growth_decay_circles(ax)
ax.set_title("Obs. %s" % str(date + timedelta(minutes=15)))

ax = fig.add_subplot(323)
plot_precip_field(forecast_extrap[-1])
plot_growth_decay_circles(ax)
ax.set_title("Extrapolation +15 minutes")

ax = fig.add_subplot(324)
plot_precip_field(forecast_sprog[-1])
plot_growth_decay_circles(ax)
ax.set_title("S-PROG (with post-processing)\n +15 minutes")

ax = fig.add_subplot(325)
plot_precip_field(forecast_anvil[-1])
plot_growth_decay_circles(ax)
ax.set_title("ANVIL +15 minutes")

plt.show()
```



Remarks

The extrapolation nowcast is static, i.e. it does not predict any growth or decay. While S-PROG is to some extent able to predict growth and decay, this this comes with loss of small-scale features. In addition, statistical post-processing needs to be applied to correct the bias and incorrect wet-area ratio introduced by the autoregressive process. ANVIL is able to do both: predict growth and decay and preserve the small-scale structure in a way that post-processing is not necessary.

Total running time of the script: (0 minutes 19.026 seconds)

Optical flow methods convergence

In this example we test the convergence of the optical flow methods available in pySTEPS using idealized motion fields.

To test the convergence, using an example precipitation field we will:

- Read precipitation field from a file
- Morph the precipitation field using a given motion field (linear or rotor) to generate a sequence of moving precipitation patterns.
- Using the available optical flow methods, retrieve the motion field from the precipitation time sequence (synthetic precipitation observations).

Let's first load the libraries that we will use.

```
from datetime import datetime
import time

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.cm import get_cmap
from scipy.ndimage import uniform_filter

import pysteps as stp
from pysteps import motion, io, rcparams
from pysteps.motion.vet import morph
from pysteps.visualization import plot_precip_field, quiver
```

Load the reference precipitation data

First, we will import a radar composite from the archive. You need the pysteps-data archive downloaded and the pystepsrc file configured with the data_source paths pointing to data folders.

```
# Selected case
date = datetime.strptime("201505151630", "%Y%m%d%H%M")
data_source = rcparams.data_sources["mch"]
```

Load the data from the archive

```
root_path = data_source["root_path"]
path_fmt = data_source["path_fmt"]
fn_pattern = data_source["fn_pattern"]
fn_ext = data_source["fn_ext"]
importer_name = data_source["importer"]
importer_kwargs = data_source["importer_kwargs"]

# Find the reference field in the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep=5, num_prev_files=0
)

# Read the reference radar composite
importer = io.get_method(importer_name, "importer")
reference_field, quality, metadata = io.read_timeseries(
    fns, importer, **importer_kwargs
)

del quality  # Not used

reference_field = np.squeeze(reference_field)  # Remove time dimension
```

Preprocess the data

```
# Convert to mm/h
reference_field, metadata = stp.utils.to_rainrate(reference_field, metadata)

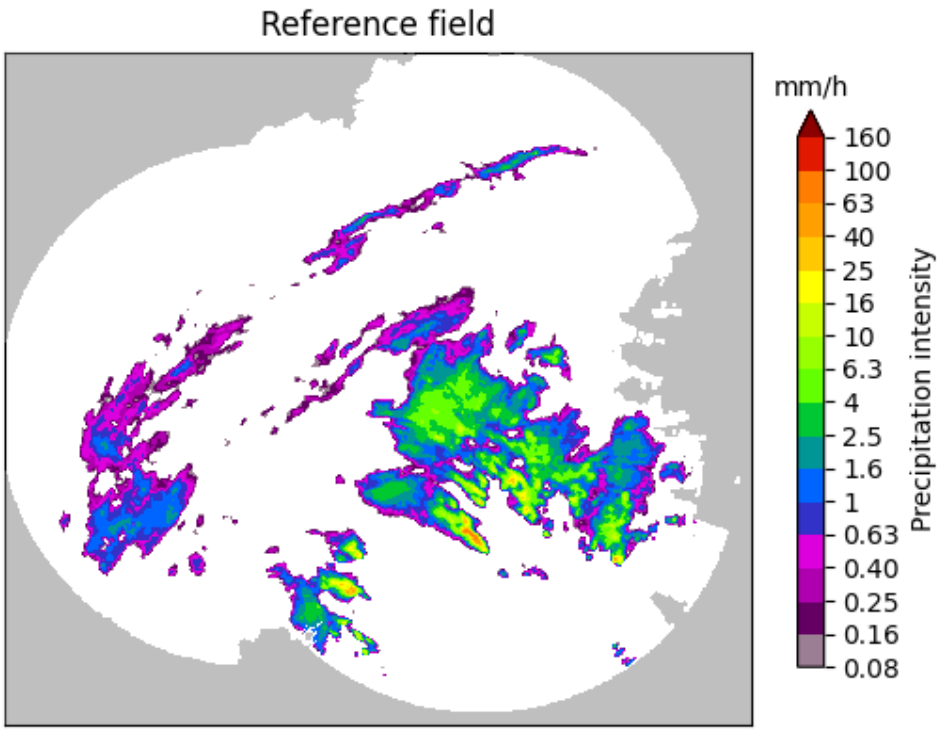
# Mask invalid values
reference_field = np.ma.masked_invalid(reference_field)

# Plot the reference precipitation
plot_precip_field(reference_field, title="Reference field")
plt.show()

# Log-transform the data [dBR]
reference_field, metadata = stp.utils.dB_transform(
    reference_field, metadata, threshold=0.1, zerovalue=-15.0
)

print("Precip. pattern shape: " + str(reference_field.shape))

# This suppress nan conversion warnings in plot functions
reference_field.data[reference_field.mask] = np.nan
```



Out:

```
Precip. pattern shape: (640, 710)
```

Synthetic precipitation observations

Now we need to create a series of precipitation fields by applying the ideal motion field to the reference precipitation field “n” times.

To evaluate the accuracy of the computed_motion vectors, we will use a relative RMSE measure. Relative MSE = <(expected_motion - computed_motion)^2> / <expected_motion^2>

```
# Relative RMSE = Rel_RMSE = sqrt(Relative MSE)
#
# - Rel_RMSE = 0%: no error
# - Rel_RMSE = 100%: The retrieved motion field has an average error equal in
#   magnitude to the motion field.
#
# Relative RMSE is computed over a region surrounding the precipitation
# field, were there is enough information to retrieve the motion field.
# The "precipitation region" includes the precipitation pattern plus a margin of
# approximately 20 grid points.
```

Let's create a function to construct different motion fields.

```
def create_motion_field(input_precip, motion_type):
    """
    Create idealized motion fields to be applied to the reference image.

    Parameters
    ----------

    input_precip: numpy array (lat, lon)

    motion_type : str
        The supported motion fields are:

            - linear_x: (u=2, v=0)
            - linear_y: (u=0, v=2)
            - rotor: rotor field

    Returns
    -------
    ideal_motion : numpy array (u, v)
    """

    # Create an imaginary grid on the image and create a motion field to be
    # applied to the image.
    ny, nx = input_precip.shape

    x_pos = np.arange(nx)
    y_pos = np.arange(ny)
    x, y = np.meshgrid(x_pos, y_pos, indexing="ij")

    ideal_motion = np.zeros((2, nx, ny))

    if motion_type == "linear_x":
        ideal_motion[0, :] = 2  # Motion along x
    elif motion_type == "linear_y":
        ideal_motion[1, :] = 2  # Motion along y
    elif motion_type == "rotor":
        x_mean = x.mean()
        y_mean = y.mean()
        norm = np.sqrt(x * x + y * y)
        mask = norm != 0
        ideal_motion[0, mask] = 2 * (y - y_mean)[mask] / norm[mask]
        ideal_motion[1, mask] = -2 * (x - x_mean)[mask] / norm[mask]
    else:
        raise ValueError("motion_type not supported.")

    # We need to swap the axes because the optical flow methods expect
```

(continues on next page)

(continued from previous page)

```
    # (lat, lon) or (y,x) indexing convention.
    ideal_motion = ideal_motion.swapaxes(1, 2)
    return ideal_motion
```

Let's create another function that construct the temporal series of precipitation observations.

```
def create_observations(input_precip, motion_type, num_times=9):
    """
    Create synthetic precipitation observations by displacing the input field
    using an ideal motion field.

    Parameters
    ----------

    input_precip: numpy array (lat, lon)
        Input precipitation field.

    motion_type : str
        The supported motion fields are:

            - linear_x: (u=2, v=0)
            - linear_y: (u=0, v=2)
            - rotor: rotor field

    num_times: int, optional
        Length of the observations sequence.


    Returns
    -------
    synthetic_observations : numpy array
        Sequence of observations
    """

    ideal_motion = create_motion_field(input_precip, motion_type)

    # The morph function expects (lon, lat) or (x, y) dimensions.
    # Hence, we need to swap the lat,lon axes.

    # NOTE: The motion field passed to the morph function can't have any NaNs.
    # Otherwise, it can result in a segmentation fault.
    morphed_field, mask = morph(
        input_precip.swapaxes(0, 1), ideal_motion.swapaxes(1, 2)
    )

    mask = np.array(mask, dtype=bool)

    synthetic_observations = np.ma.MaskedArray(morphed_field, mask=mask)
    synthetic_observations = synthetic_observations[np.newaxis, :]

    for t in range(1, num_times):
        morphed_field, mask = morph(
            synthetic_observations[t - 1], ideal_motion.swapaxes(1, 2)
        )
        mask = np.array(mask, dtype=bool)

        morphed_field = np.ma.MaskedArray(
            morphed_field[np.newaxis, :], mask=mask[np.newaxis, :]
        )

        synthetic_observations = np.ma.concatenate(
```

(continues on next page)

(continued from previous page)

```
            [synthetic_observations, morphed_field], axis=0
        )

    # Swap  back to (lat, lon)
    synthetic_observations = synthetic_observations.swapaxes(1, 2)

    synthetic_observations = np.ma.masked_invalid(synthetic_observations)

    synthetic_observations.data[np.ma.getmaskarray(synthetic_observations)] = 0

    return ideal_motion, synthetic_observations


def plot_optflow_method_convergence(input_precip, optflow_method_name, motion_
↪type):
    """
    Test the convergence to the actual solution of the optical flow method used.

    Parameters
    ----------

    input_precip: numpy array (lat, lon)
        Input precipitation field.

    optflow_method_name: str
        Optical flow method name

    motion_type : str
        The supported motion fields are:

            - linear_x: (u=2, v=0)
            - linear_y: (u=0, v=2)
            - rotor: rotor field
    """

    if optflow_method_name.lower() != "darts":
        num_times = 2
    else:
        num_times = 9

    ideal_motion, precip_obs = create_observations(
        input_precip, motion_type, num_times=num_times
    )

    oflow_method = motion.get_method(optflow_method_name)

    elapsed_time = time.perf_counter()

    computed_motion = oflow_method(precip_obs, verbose=False)

    print(
        f"{optflow_method_name} computation time: "
        f"{(time.perf_counter() - elapsed_time):.1f} [s]"
    )

    precip_obs, _ = stp.utils.dB_transform(precip_obs, inverse=True)

    precip_data = precip_obs.max(axis=0)
    precip_data.data[precip_data.mask] = 0

    precip_mask = (uniform_filter(precip_data, size=20) > 0.1) & ~precip_obs.mask.
↪any(
```

(continues on next page)

(continued from previous page)

```
        axis=0
    )

    cmap = get_cmap("jet")
    cmap.set_under("grey", alpha=0.25)
    cmap.set_over("none")

    # Compare retrieved motion field with the ideal one
    plt.figure(figsize=(9, 4))
    plt.subplot(1, 2, 1)
    ax = plot_precip_field(precip_obs[0], title="Reference motion")
    quiver(ideal_motion, step=25, ax=ax)

    plt.subplot(1, 2, 2)
    ax = plot_precip_field(precip_obs[0], title="Retrieved motion")
    quiver(computed_motion, step=25, ax=ax)

    # To evaluate the accuracy of the computed_motion vectors, we will use
    # a relative RMSE measure.
    # Relative MSE = < (expected_motion - computed_motion)^2 > / <expected_motion^
↪2 >
    # Relative RMSE = sqrt(Relative MSE)

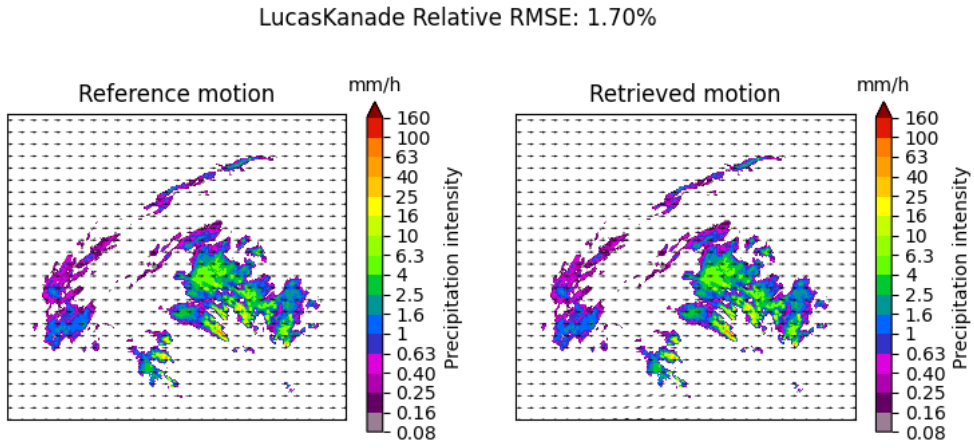
    mse = ((ideal_motion - computed_motion)[:, precip_mask] ** 2).mean()

    rel_mse = mse / (ideal_motion[:, precip_mask] ** 2).mean()
    plt.suptitle(
        f"{optflow_method_name} " f"Relative RMSE: {np.sqrt(rel_mse) * 100:.2f}%"
    )
    plt.show()
```

Lucas-Kanade

Constant motion x-direction

```
plot_optflow_method_convergence(reference_field, "LucasKanade", "linear_x")
```

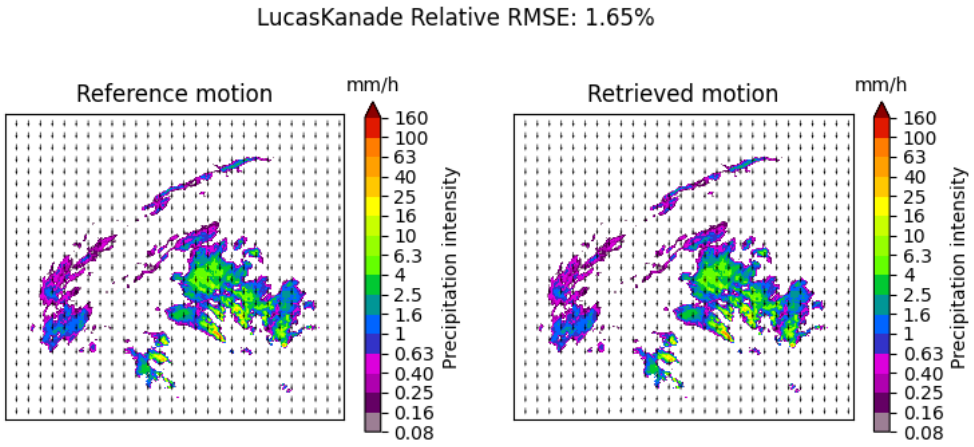


Out:

```
LucasKanade computation time: 3.1 [s]
```

Constant motion y-direction

```
plot_optflow_method_convergence(reference_field, "LucasKanade", "linear_y")
```

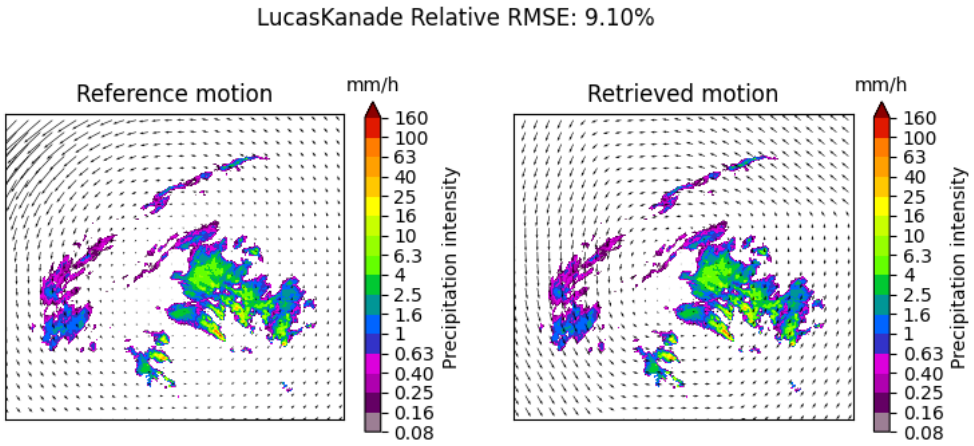


Out:

```
LucasKanade computation time: 3.1 [s]
```

Rotational motion

```
plot_optflow_method_convergence(reference_field, "LucasKanade", "rotor")
```



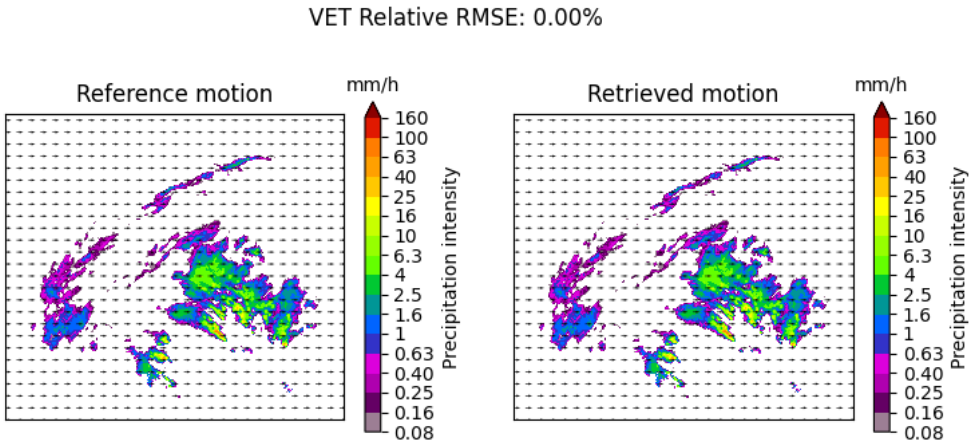
Out:

```
LucasKanade computation time: 3.1 [s]
```

Variational Echo Tracking (VET)

Constant motion x-direction

```
plot_optflow_method_convergence(reference_field, "VET", "linear_x")
```

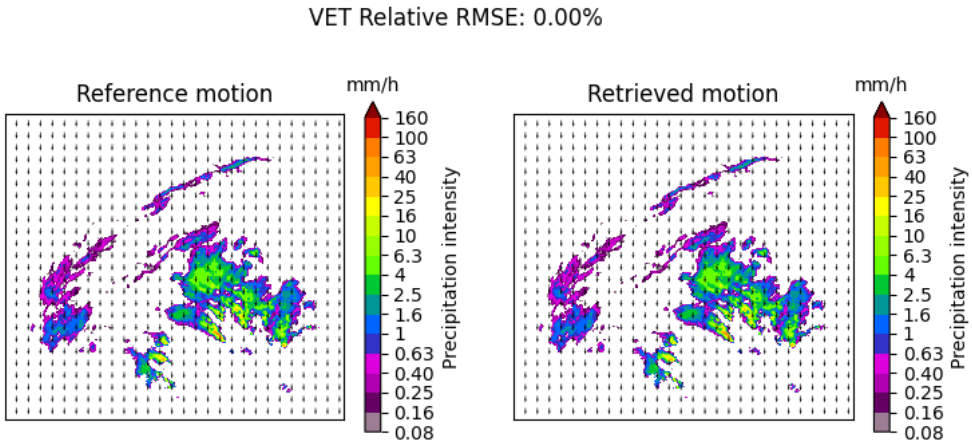


Out:

```
VET computation time: 3.8 [s]
```

Constant motion y-direction

```
plot_optflow_method_convergence(reference_field, "VET", "linear_y")
```

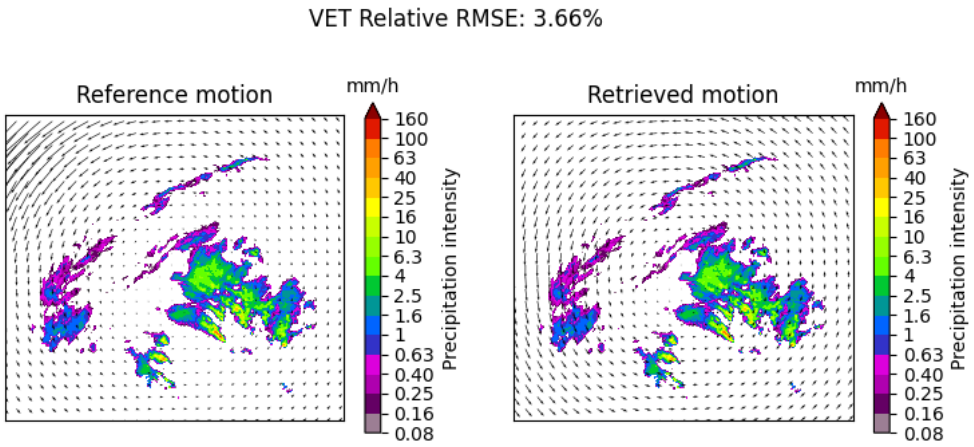


Out:

```
VET computation time: 3.6 [s]
```

Rotational motion

```
plot_optflow_method_convergence(reference_field, "VET", "rotor")
```



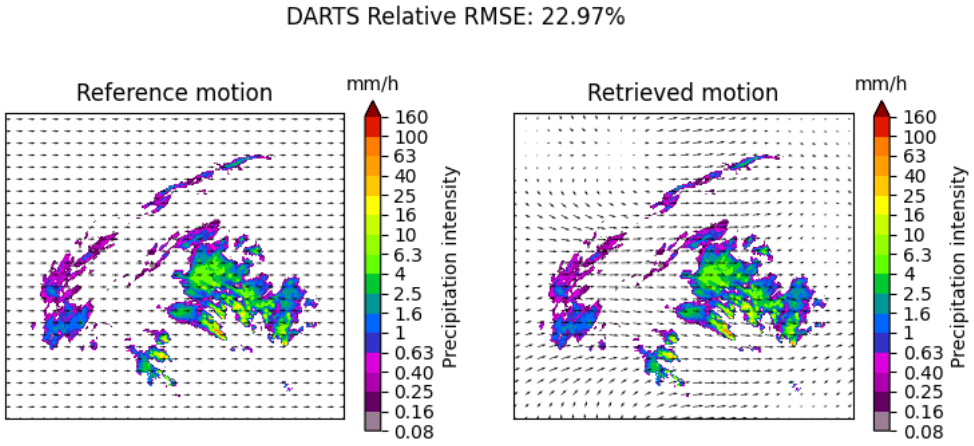
Out:

```
VET computation time: 30.3 [s]
```

DARTS

Constant motion x-direction

```
plot_optflow_method_convergence(reference_field, "DARTS", "linear_x")
```

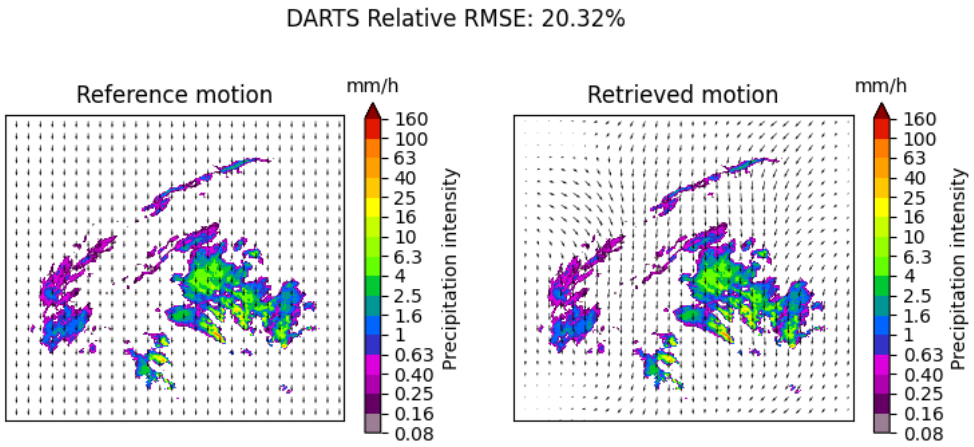


Out:

```
DARTS computation time: 2.7 [s]
```

Constant motion y-direction

```
plot_optflow_method_convergence(reference_field, "DARTS", "linear_y")
```



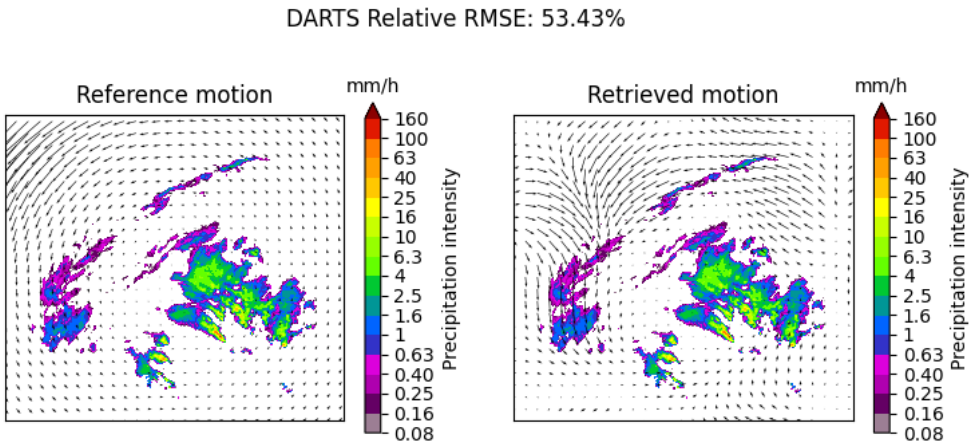
Out:

```
DARTS computation time: 2.6 [s]
```

Rotational motion

```
plot_optflow_method_convergence(reference_field, "DARTS", "rotor")

# sphinx_gallery_thumbnail_number = 5
```



Out:

```
DARTS computation time: 2.6 [s]
```

Total running time of the script: (1 minutes 0.987 seconds)

2.2 pySTEPS reference

Release
1.3.1

Date
May 18, 2020

This page gives an comprehensive description of all the modules and functions available in pySTEPS.

2.2.1 pysteps

Pystep top module utils

pysteps.load_config_file([params_file, ...])	Load the pysteps configuration file.

pysteps.load_config_file

pysteps.**load_config_file**(*params_file=None*, *verbose=False*, *dryrun=False*)
Load the pysteps configuration file. The configuration parameters are available as a DotDictify instance in the *pysteps.rcparams* variable.

Parameters

params_file: str
Path to the parameters file to load. If *params_file=None*, it looks for a configuration file in the default locations.

verbose: bool
Print debugging information. False by default. This flag is overwritten by the silent_import=False in the pysteps configuration file.

dryrun: bool
If False, perform a dry run that does not update the *pysteps.rcparams* attribute.

Returns

rcparams
[_DotDictify] Configuration parameters loaded from file.

2.2.2 pysteps.cascade

Methods for constructing bandpass filters and decomposing 2d precipitation fields into different spatial scales.

pysteps.cascade.interface

Interface for the cascade module.

get_method(name)	Return a callable function for the bandpass filter or cascade decomposition method corresponding to the given name.

pysteps.cascade.interface.get_method

pysteps.cascade.interface.**get_method**(*name*)
Return a callable function for the bandpass filter or cascade decomposition method corresponding to the given name. For the latter, two functions are returned: the first is for the decomposing and the second is for recomposing the cascade.

Filter methods:

Name	Description
gaussian	implementation of bandpass filter using Gaussian weights
uniform	implementation of a filter where all weights are set to one

Decomposition/recomposition methods:

Name	Description
fft	decomposition into multiple spatial scales based on the fast Fourier Transform (FFT) and a set of bandpass filters

pysteps.cascade.bandpass_filters

Bandpass filters for separating different spatial scales from two-dimensional images in the frequency domain.

The methods in this module implement the following interface:

```
filter_xxx(shape, n, optional arguments)
```

where shape is the shape of the input field, respectively, and n is the number of frequency bands to use.

The output of each filter function is a dictionary containing the following key-value pairs:

Key	Value
weights_1d	2d array of shape (n, r) containing 1d filter weights for each frequency band k=1,2,. . . ,n
weights_2d	3d array of shape (n, M, int(N/2)+1) containing the 2d filter weights for each frequency band k=1,2,. . . ,n
central_freqs	1d array of shape n containing the central frequencies of the filters
shape	the shape of the input field in the spatial domain

where r = int(max(N, M)/2)+1

By default, the filter weights are normalized so that for any Fourier wavenumber they sum to one.

Available filters

filter_uniform(shape, n)	A dummy filter with one frequency band covering the whole domain.
filter_gaussian(shape, n[, l_0, . . .])	Implements a set of Gaussian bandpass filters in logarithmic frequency scale.

pysteps.cascade.bandpass_filters.filter_uniform

pysteps.cascade.bandpass_filters.**filter_uniform**(*shape*, *n*)

A dummy filter with one frequency band covering the whole domain. The weights are set to one.

Parameters

shape
[int or tuple] The dimensions (height, width) of the input field. If shape is an int, the domain is assumed to have square shape.

n
[int] Not used. Needed for compatibility with the filter interface.

pysteps.cascade.bandpass_filters.filter_gaussian

pysteps.cascade.bandpass_filters.**filter_gaussian**(*shape*, *n*, *l_0=3*, *gauss_scale=0.5*, *gauss_scale_0=0.5*, *d=1.0*, *normalize=True*)

Implements a set of Gaussian bandpass filters in logarithmic frequency scale.

Parameters

shape
[int or tuple] The dimensions (height, width) of the input field. If shape is an int, the domain is assumed to have square shape.

n
[int] The number of frequency bands to use. Must be greater than 2.

l_0
[int] Central frequency of the second band (the first band is always centered at zero).

gauss_scale
[float] Optional scaling prameter. Proportional to the standard deviation of the Gaussian weight functions.

gauss_scale_0
[float] Optional scaling parameter for the Gaussian function corresponding to the first frequency band.

d
[scalar, optional] Sample spacing (inverse of the sampling rate). Defaults to 1.

normalize
[bool] If True, normalize the weights so that for any given wavenumber they sum to one.

Returns

out
[dict] A dictionary containing the bandpass filters corresponding to the specified frequency bands.

References

[PCH18b]

pysteps.cascade.decomposition

Methods for decomposing two-dimensional fields into multiple spatial scales and recomposing the individual scales to obtain the original field.

The methods in this module implement the following interface:

```
decomposition_xxx(field, bp_filter, **kwargs)
recompose_xxx(decomp, **kwargs)
```

where field is the input field and bp_filter is a dictionary returned by a filter method implemented in *pysteps.cascade.bandpass_filters*. The decomp argument is a decomposition obtained by calling decomposition_xxx. Optional parameters can be passed in the keyword arguments. The output of each method is a dictionary with the following key-value pairs:

Key	Value
cascade_levels	three-dimensional array of shape (k,m,n), where k is the number of cascade levels and the input fields have shape (m,n) if domain is "spectral" and compact output is requested (see the table below), cascade_levels contains a list of one-dimensional arrays
domain	domain of the cascade decomposition: "spatial" or "spectral"
normalized	are the cascade levels normalized: True or False

The following key-value pairs are optional. They are included in the output if `kwargs` contains the "compute_stats" key with value set to True:

Key	Value
means	list of mean values for each cascade level
stds	list of standard deviations for each cascade level

The following key-value pairs are included in the output if `kwargs` contains the key "output_domain" with value set to "spectral":

Key	Value
compact_output	True or False. If set to True, only the parts of the Fourier spectrum with non-negligible filter weights are stored.
weight_masks	Applicable if compact_output is True. Contains a list of masks, where a True value indicates that the corresponding Fourier wavenumber is included in the decomposition

Available methods

decomposition_fft(field, bp_filter, **kwargs)	Decompose a two-dimensional input field into multiple spatial scales by using the Fast Fourier Transform (FFT) and a set of bandpass filters.
recompose_fft(decomp, **kwargs)	Recompose a cascade obtained with decomposition_fft by inverting the normalization and summing the cascade levels.

pysteps.cascade.decomposition.decomposition_fft

pysteps.cascade.decomposition.**decomposition_fft**(*field*, *bp_filter*, ***kwargs*)

Decompose a two-dimensional input field into multiple spatial scales by using the Fast Fourier Transform (FFT) and a set of bandpass filters.

Parameters

field
[array_like] Two-dimensional array containing the input field. All values are required to be finite.

bp_filter
[dict] A filter returned by a method implemented in *pysteps.cascade.bandpass_filters*.

Returns

out
[ndarray] A dictionary described in the module documentation. The num-

ber of cascade levels is determined from the filter (see *pysteps.cascade.bandpass_filters*).

Other Parameters

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see *pysteps.utils.interface.get_method()*). Defaults to "numpy". This option is not used if input_domain and output_domain are both set to "spectral".

normalize
[bool] If True, normalize the cascade levels to zero mean and unit variance. Requires that compute_stats is True. Implies that compute_stats is True. Defaults to False.

mask
[array_like] Optional mask to use for computing the statistics for the cascade levels. Pixels with mask==False are excluded from the computations. This option is not used if output domain is "spectral".

input_domain
[{"spatial", "spectral"}] The domain of the input field. If "spectral", the input is assumed to be in the spectral domain. Defaults to "spatial".

output_domain
[{"spatial", "spectral"}] If "spatial", the output cascade levels are transformed back to the spatial domain by using the inverse FFT. If "spectral", the cascade is kept in the spectral domain. Defaults to "spatial".

compute_stats
[bool] If True, the output dictionary contains the keys "means" and "stds" for the mean and standard deviation of each output cascade level. Defaults to False.

compact_output
[bool] Applicable if output_domain is "spectral". If set to True, only the parts of the Fourier spectrum with non-negligible filter weights are stored. Defaults to False.

pysteps.cascade.decomposition.recompose_fft

`pysteps.cascade.decomposition.`**`recompose_fft`**(*decomp*, ***kwargs*)

Recompose a cascade obtained with decomposition_fft by inverting the normalization and summing the cascade levels.

Parameters

decomp
[dict] A cascade decomposition returned by decomposition_fft.

Returns

out
[numpy.ndarray] A two-dimensional array containing the recomposed cascade.

2.2.3 pysteps.extrapolation

Extrapolation module functions and interfaces.

pysteps.extrapolation.interface

The functions in the extrapolation module implement the following interface:

```
extrapolate(extrap, precip, velocity, timesteps, outval=np.nan, **keywords)
```

where *extrap* is an extrapolator object returned by the initialize function, *precip* is a (m,n) array with input precipitation field to be advected and *velocity* is a (2,m,n) array containing the x- and y-components of the m x n advection field. timesteps is an integer or list specifying the time steps to extrapolate. If an integer is given, a range of uniformly spaced steps 1,2,. . . ,timesteps is created. If a list is given, it is assumed to represent a sequence of monotonously increasing time steps. One time unit is assumed to represent the time step of the advection field. The optional argument *outval* specifies the value for pixels advected from outside the domain. Optional keyword arguments that are specific to a given extrapolation method are passed as a dictionary.

The output of each method is an array R_e that includes the time series of extrapolated fields of shape (num_timesteps, m, n).

get_method(name)	Return two-element tuple for the extrapolation method corresponding to the given name.
eulerian_persistence(precip, velocity, timesteps)	A dummy extrapolation method to apply Eulerian persistence to a two-dimensional precipitation field.

pysteps.extrapolation.interface.get_method

pysteps.extrapolation.interface.**get_method**(*name*)

Return two-element tuple for the extrapolation method corresponding to the given name. The elements of the tuple are callable functions for the initializer of the extrapolator and the extrapolation method, respectively. The available options are:

Name	Description
None	returns None
eulerian	this methods does not apply any advection to the input precipitation field (Eulerian persistence)
semila-grangian	implementation of the semi-Lagrangian method of Germann et al. (2002) [GZ02]

pysteps.extrapolation.interface.eulerian_persistence

pysteps.extrapolation.interface.**eulerian_persistence**(*precip*, *velocity*, *timesteps*, *outval=nan*, ***kwargs*)

A dummy extrapolation method to apply Eulerian persistence to a two-dimensional precipitation field. The method returns the a sequence of the same initial field with no extrapolation applied (i.e. Eulerian persistence).

Parameters

precip
[array-like] Array of shape (m,n) containing the input precipitation field. All values are required to be finite.

velocity
[array-like] Not used by the method.

timesteps
[int or list] Number of time steps or a list of time steps.

outval
[float, optional] Not used by the method.

Returns

out
[array or tuple] If return_displacement=False, return a sequence of the same initial field of shape (num_timesteps,m,n). Otherwise, return a tuple containing the replicated fields and a (2,m,n) array of zeros.

Other Parameters

return_displacement
[bool] If True, return the total advection velocity (displacement) between the initial input field and the advected one integrated along the trajectory. Default : False

References

[GZ02] Germann et al (2002)

pysteps.extrapolation.semilagrangian

Implementation of the semi-Lagrangian method of Germann et al (2002). [GZ02]

extrapolate(precip, velocity, timesteps[, ...])	Apply semi-Lagrangian backward extrapolation to a two-dimensional precipitation field.

pysteps.extrapolation.semilagrangian.extrapolate

pysteps.extrapolation.semilagrangian.**extrapolate**(*precip*, *velocity*, *timesteps*, *outval=nan*, *xy_coords=None*, *allow_nonfinite_values=False*, *vel_timestep=None*, ***kwargs*)

Apply semi-Lagrangian backward extrapolation to a two-dimensional precipitation field.

Parameters

precip
[array-like] Array of shape (m,n) containing the input precipitation field. All values are required to be finite by default.

velocity
[array-like] Array of shape (2,m,n) containing the x- and y-components of the m*n advection field. All values are required to be finite by default.

timesteps
[int or list] If timesteps is integer, it specifies the number of time steps to extrapolate. If a list is given, each element is the desired extrapolation time step from the current time. In this case, the vel_timestep argument must be specified.

outval
[float, optional] Optional argument for specifying the value for pixels advected from outside the domain. If outval is set to 'min', the value is taken as the minimum value of R. Default : np.nan

xy_coords
[ndarray, optional] Array with the coordinates of the grid dimension (2, m, n).

- xy_coords[0] : x coordinates
- xy_coords[1] : y coordinates

By default, the *xy_coords* are computed for each extrapolation.

allow_nonfinite_values
[bool, optional] If True, allow non-finite values in the precipitation and advection fields.

This option is useful if the input fields contain a radar mask (i.e. pixels with no observations are set to nan).

Returns

out
[array or tuple] If return_displacement=False, return a time series extrapolated fields of shape (num_timesteps,m,n). Otherwise, return a tuple containing the extrapolated fields and the total displacement along the advection trajectory.

Other Parameters

D_prev
[array-like] Optional initial displacement vector field of shape (2,m,n) for the extrapolation. Default : None

n_iter
[int] Number of inner iterations in the semi-Lagrangian scheme. If n_iter > 0, the integration is done using the midpoint rule. Otherwise, the advection vectors are taken from the starting point of each interval. Default : 1

return_displacement
[bool] If True, return the total advection velocity (displacement) between the initial input field and the advected one integrated along the trajectory. Default : False

vel_timestep
[float] The time step of the velocity field. It is assumed to have the same unit as the timesteps argument.

References

[GZ02] Germann et al (2002)

2.2.4 pysteps.datasets

Utilities to download the pysteps data and to create a default pysteps configuration file pointing to that data.

download_pysteps_data(dir_path[, force])	Download pysteps data from github.
create_default_pystepsrc(pysteps_data_dir[, ...])	Create a default configuration file pointing to the pysteps data directory.
info()	Describe the available datasets in the pysteps example data.
load_dataset([case, frames])	Load a sequence of radar composites from the pysteps example data.

pysteps.datasets.download_pysteps_data

pysteps.datasets.**download_pysteps_data**(*dir_path*, *force=True*)
Download pysteps data from github.

Parameters

dir_path: str
Path to directory where the psyteps data will be placed.

force
[bool] If the destination directory exits and force=False, a DirectoryNotEmpty excep-

tion if raised. If force=True, the data will we downloaded in the destination directory and may override existing files.

pysteps.datasets.create_default_pystepsrc

pysteps.datasets.**create_default_pystepsrc**(*pysteps_data_dir*, *config_dir=None*, *file_name='pystepsrc'*, *dryrun=False*)

Create a default configuration file pointing to the pysteps data directory.

If the configuration file already exists, it will backup the existing file by appending the extensions '.1', '.2', up to '.5.' to the filename. A maximum of 5 files are kept. .2, up to app.log.5.

File rotation is implemented for the backup files. For example, if the default configuration filename is 'pystepsrc' and the files pystepsrc, pystepsrc.1, pystepsrc.2, etc. exist, they are renamed to respectively pystepsrc.1, pystepsrc.2, pystepsrc.2, etc. Finally, after the existing files are backed up, the new configuration file is written.

Parameters

pysteps_data_dir
[str] Path to the directory with the pysteps data.

config_dir
[str] Destination directory for the configuration file. Default values: $HOME/.pysteps (unix and Mac OS X) or $USERPROFILE/pysteps (windows). The directory is created if it does not exists.

file_name
[str] Configuration file name. *pystepsrc* by default.

dryrun
[bool] Do not create the parameter file, nor create backups of existing files. No changes are made in the file system. It just returns the file path.

Returns

dest_path
[str] Configuration file path.

pysteps.datasets.info

pysteps.datasets.**info**()

Describe the available datasets in the pysteps example data.

```
>>> from pysteps import datasets
>>> datasets.info()
```

pysteps.datasets.load_dataset

pysteps.datasets.**load_dataset**(*case='fmi'*, *frames=14*)

Load a sequence of radar composites from the pysteps example data.

To print the available datasets run

```
>>> from pysteps import datasets
>>> datasets.info()
```

This function load by default 14 composites, corresponding to a 1h and 10min time window. For example, the first two composites can be used to obtain the motion field of the precipitation pattern, while the remaining twelve composites can be used to evaluate the quality of our forecast.

Calling this function requires the pysteps-data installed, otherwise an exception is raised. To install the pysteps example data check the *example_data* section.

Parameters

case
[str] Case to load.

frames
[int] Number composites (radar images). Max allowed value: 24 (35 for MRMS product) Default: 14

Returns

rainrate
[array-like] Precipitation data in mm/h. Dimensions: [time, lat, lon]

metadata
[dict] The metadata observations attributes.

timestep
[number] Time interval between composites in minutes.

2.2.5 pysteps.io

Methods for browsing data archives, reading 2d precipitation fields and writing forecasts into files.

pysteps.io.interface

Interface for the io module.

get_method(name, method_type)	Return a callable function for the method corresponding to the given name.

pysteps.io.interface.get_method

pysteps.io.interface.**get_method**(*name*, *method_type*)
Return a callable function for the method corresponding to the given name.

Parameters

name
[str] Name of the method. The available options are:

Importers:

Name	Description
bom_rf3	NefCDF files used in the Boreau of Meterorology archive containing precipitation intensity composites.
fmi_geotiff	GeoTIFF files used in the Finnish Meteorological Institute (FMI) archive, containing reflectivity composites (dBZ).
fmi_pgm	PGM files used in the Finnish Meteorological Institute (FMI) archive, containing reflectivity composites (dBZ).
knmi_hdf5	HDF5 file format used by KNMI.
mch_gif	GIF files in the MeteoSwiss (MCH) archive containing precipitation composites.
mch_hdf5	HDF5 file format used by MeteoSiss (MCH).
mch_metranet	metranet files in the MeteoSwiss (MCH) archive containing precipitation composites.
mrms_grib	Grib2 files used by the NSSL's MRMS product
opera_hdf5	ODIM HDF5 file format used by Eumetnet/OPERA.
saf_crri	NetCDF SAF CRRI files containing convective rain rate intensity and other

Exporters:

Name	Description
geotiff	Export as GeoTIFF files.
kineros	KINEROS2 Rainfall file as specified in https://www.tucson.ars.ag.gov/kineros/. Grid points are treated as individual rain gauges. A separate file is produced for each ensemble member.
netcdf	NetCDF files conforming to the CF 1.7 specification.

method_type
[{'importer', 'exporter'}] Type of the method (see tables above).

pysteps.io.archive

Utilities for finding archived files that match the given criteria.

find_by_date(date, root_path, path_fmt, ...)	List input files whose timestamp matches the given date.

pysteps.io.archive.find_by_date

pysteps.io.archive.**find_by_date**(*date*, *root_path*, *path_fmt*, *fn_pattern*, *fn_ext*, *timestep*, *num_prev_files=0*, *num_next_files=0*, *silent=False*)

List input files whose timestamp matches the given date.

Parameters

date
[datetime.datetime] The given date.

root_path
[str] The root path to search the input files.

path_fmt
[str] Path format. It may consist of directory names separated by '/', date/time specifiers beginning with '%' (e.g. %Y/%m/%d) and wildcards (?) that match any single character.

fn_pattern
[str] The name pattern of the input files without extension. The pattern can contain time specifiers (e.g. %H, %M and %S).

fn_ext
[str] Extension of the input files.

timestep
[float] Time step between consecutive input files (minutes).

num_prev_files
[int] Optional, number of previous files to find before the given timestamp.

num_next_files
[int] Optional, number of future files to find after the given timestamp.

silent
[bool] Optional, whether to suppress all messages from the method.

Returns

out
[tuple] If num_prev_files=0 and num_next_files=0, return a pair containing the found file name and the corresponding timestamp as a datetime.datetime object. Otherwise, return a tuple of two lists, the first one for the file names and the second one for the corresponding timestemps. The lists are sorted in ascending order with respect to timestamp. A None value is assigned if a file name corresponding to a given timestamp is not found.

pysteps.io.importers

Methods for importing files containing two-dimensional radar mosaics.

The methods in this module implement the following interface:

```
import_xxx(filename, optional arguments)
```

where **xxx** is the name (or abbreviation) of the file format and filename is the name of the input file.

The output of each method is a three-element tuple containing a two-dimensional radar mosaic, the corresponding quality field and a metadata dictionary. If the file contains no quality information, the quality field is set to None. Pixels containing missing data are set to nan.

The metadata dictionary contains the following recommended key-value pairs:

Key	Value
projection	PROJ.4-compatible projection definition
x1	x-coordinate of the lower-left corner of the data raster
y1	y-coordinate of the lower-left corner of the data raster
x2	x-coordinate of the upper-right corner of the data raster
y2	y-coordinate of the upper-right corner of the data raster
xpixelsize	grid resolution in x-direction
ypixelsize	grid resolution in y-direction
yorigin	a string specifying the location of the first element in the data raster w.r.t. y-axis: 'upper' = upper border 'lower' = lower border
institution	name of the institution who provides the data
unit	the physical unit of the data: 'mm/h', 'mm' or 'dBZ'
transform	the transformation of the data: None, 'dB', 'Box-Cox' or others
accutime	the accumulation time in minutes of the data, float
threshold	the rain/no rain threshold with the same unit, transformation and accutime of the data.
zerovalue	the value assigned to the no rain pixels with the same unit, transformation and accutime of the data.
zr_a	the Z-R constant a in Z = a*R**b
zr_b	the Z-R exponent b in Z = a*R**b

Available Importers

import_bom_rf3(filename, **kwargs)	Import a NetCDF radar rainfall product from the BoM Rainfields3.
import_fmi_geotiff(filename, **kwargs)	Import a reflectivity field (dBZ) from an FMI GeoTIFF file.
import_fmi_pgm(filename[, gzipped])	Import a 8-bit PGM radar reflectivity composite from the FMI archive.
import_knmi_hdf5(filename[, qty, accutime, ...])	Import a precipitation or reflectivity field (and optionally the quality field) from a HDF5 file conforming to the KNMI Data Centre specification.
import_mch_gif(filename, product, unit, ...)	Import a 8-bit gif radar reflectivity composite from the MeteoSwiss archive.
import_mch_hdf5(filename[, qty])	Import a precipitation field (and optionally the quality field) from a MeteoSwiss HDF5 file conforming to the ODIM specification.
import_mch_metranet(filename, product, unit, ...)	Import a 8-bit bin radar reflectivity composite from the MeteoSwiss archive.
import_mrms_grib(filename[, extent, window_size])	Importer for NSSL's Multi-Radar/Multi-Sensor System ([MRMS](https://www.nssl.noaa.gov/projects/mrms/)) rainrate product (grib format).
import_opera_hdf5(filename[, qty])	Import a precipitation field (and optionally the quality field) from an OPERA HDF5 file conforming to the ODIM specification.
import_saf_crri(filename[, extent])	Import a NetCDF radar rainfall product from the Convective Rainfall Rate Intensity (CRRI) product from the Satellite Application Facilities (SAF).

pysteps.io.importers.import_bom_rf3

pysteps.io.importers.**import_bom_rf3**(*filename*, ***kwargs*)

Import a NetCDF radar rainfall product from the BoM Rainfields3.

Parameters

filename
[str] Name of the file to import.

Returns

out
[tuple] A three-element tuple containing the rainfall field in mm/h imported from the Bureau RF3 netcdf, the quality field and the metadata. The quality field is currently set to None.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.importers.import_fmi_geotiff

pysteps.io.importers.**import_fmi_geotiff**(*filename*, ***kwargs*)
Import a reflectivity field (dBZ) from an FMI GeoTIFF file.

Parameters

filename
[str] Name of the file to import.

Returns

out
[tuple] A three-element tuple containing the precipitation field, the associated quality field and metadata. The quality field is currently set to None.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.importers.import_fmi_pgm

pysteps.io.importers.**import_fmi_pgm**(*filename*, *gzipped=False*, ***kwargs*)
Import a 8-bit PGM radar reflectivity composite from the FMI archive.

Parameters

filename
[str] Name of the file to import.

gzipped
[bool] If True, the input file is treated as a compressed gzip file.

Returns

out
[tuple] A three-element tuple containing the reflectivity composite in dBZ and the associated quality field and metadata. The quality field is currently set to None.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.importers.import_knmi_hdf5

pysteps.io.importers.**import_knmi_hdf5**(*filename*, *qty='ACRR'*, *accutime=5.0*, *pixelsize=1.0*, ***kwargs*)

Import a precipitation or reflectivity field (and optionally the quality field) from a HDF5 file conforming to the KNMI Data Centre specification.

Parameters

filename
[str] Name of the file to import.

qty :
The quantity to read from the file. The currently supported identifiers are: 'ACRR'=hourly rainfall accumulation (mm) and 'DBZH'=max-reflectivity (dBZ). The default value is 'ACRR'.

accutime
[float] The accumulation time of the dataset in minutes. A 5 min accumulation is used as default, but hourly, daily and monthly accumulations are also available.

pixelsize: float
The pixel size of a raster cell in kilometers. The default value for the KNMI datasets is a 1 km grid cell size, but datasets with 2.4 km pixel size are also available.

Returns

out
[tuple] A three-element tuple containing precipitation accumulation [mm] / reflectivity [dBZ] of the KNMI product, the associated quality field and metadata. The quality field is currently set to None.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

Notes

Every KNMI data type has a slightly different naming convention. The standard setup is based on the accumulated rainfall product on 1 km2 spatial and 5 min temporal resolution. See https://data.knmi.nl/datasets?q=radar for a list of all available KNMI radar data.

pysteps.io.importers.import_mch_gif

pysteps.io.importers.**import_mch_gif**(*filename*, *product*, *unit*, *accutime*, ***kwargs*)

Import a 8-bit gif radar reflectivity composite from the MeteoSwiss archive.

Parameters

filename
[str] Name of the file to import.

product :
The name of the MeteoSwiss QPE product.

Currently supported prducts:

Name	Product
AQC	Acquire
CPC	CombiPrecip
RZC	Precip
AZC	RZC accumulation

unit :
the physical unit of the data

accutime
[float] the accumulation time in minutes of the data

Returns

out
[tuple] A three-element tuple containing the precipitation field in mm/h imported from a MeteoSwiss gif file and the associated quality field and metadata. The quality field is currently set to None.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.importers.import_mch_hdf5

pysteps.io.importers.**import_mch_hdf5**(*filename*, *qty='RATE'*, ***kwargs*)

Import a precipitation field (and optionally the quality field) from a MeteoSwiss HDF5 file conforming to the ODIM specification.

Parameters

filename
[str] Name of the file to import.

qty :
The quantity to read from the file. The currently supported identitiers are: 'RATE'=instantaneous rain rate (mm/h), 'ACRR'=hourly rainfall accumulation (mm) and 'DBZH'=max-reflectivity (dBZ). The default value is 'RATE'.

Returns

out
[tuple] A three-element tuple containing the OPERA product for the requested quantity and the associated quality field and metadata. The quality field is read from the file if it contains a dataset whose quantity identifier is 'QIND'.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.importers.import_mch_metranet

pysteps.io.importers.**import_mch_metranet**(*filename*, *product*, *unit*, *accutime*)
Import a 8-bit bin radar reflectivity composite from the MeteoSwiss archive.

Parameters

filename
[str] Name of the file to import.

product :
The name of the MeteoSwiss QPE product.

Currently supported prducts:

Name	Product
AQC	Acquire
CPC	CombiPrecip
RZC	Precip
AZC	RZC accumulation

unit :
the physical unit of the data

accutime
[float] the accumulation time in minutes of the data

Returns

out
[tuple] A three-element tuple containing the precipitation field in mm/h imported from a MeteoSwiss gif file and the associated quality field and metadata. The quality field is currently set to None.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: “float32”, “float64”, “single”, and “double”.

fillna
[float or np.nan] Value used to represent the missing data (“No Coverage”). By default, np.nan is used.

pysteps.io.importers.import_mrms_grib

pysteps.io.importers.**import_mrms_grib**(*filename*, *extent=None*, *window_size=4*, ***kwargs*)

Importer for NSSL’s Multi-Radar/Multi-Sensor System ([MRMS](https://www.nssl.noaa.gov/projects/mrms/)) rainrate product (grib format).

The rainrate values are expressed in mm/h, and the dimensions of the data array are [latitude, longitude]. The first grid point (0,0) corresponds to the upper left corner of the domain, while (last i, last j) denote the lower right corner.

Due to the large size of the dataset (3500 x 7000), a float32 type is used by default to reduce the memory footprint. However, be aware that when this array is passed to a pystep function, it may be converted to double precision, doubling the memory footprint. To change the precision of the data, use the *dtype* keyword.

Also, by default, the original data is downscaled by 4 (resulting in a ~4 km grid spacing). In case that the original grid spacing is needed, use *window_size=1*. But be aware that a single composite in double precipitation will require 186 Mb of memory.

Finally, if desired, the precipitation data can be extracted over a sub region of the full domain using the *extent* keyword. By default, the entire domain is returned.

Parameters

filename
[str] Name of the file to import.

extent: None or array-like
Longitude and latitude range (in degrees) of the data to be retrieved. (min_lon, max_lon, min_lat, max_lat). By default (None), the entire domain is retrieved. The extent can be in any form that can be converted to a flat array of 4 elements array (e.g., lists or tuples).

window_size
[array_like or int] Array containing down-sampling integer factor along each axis. If an integer value is given, the same block shape is used for all the image dimensions. Default: window_size=4.

Returns

precipitation
[2D array, float32] Precipitation field in mm/h. The dimensions are [latitude, longitude]. The first grid point (0,0) corresponds to the upper left corner of the domain, while (last i, last j) denote the lower right corner.

quality
[None] Not implement.

metadata
[dict] Associated metadata (pixel sizes, map projections, etc.).

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: “float32”, “float64”, “single”, and “double”.

fillna
[float or np.nan] Value used to represent the missing data (“No Coverage”). By default, np.nan is used.

Notes

In the MRMS grib files, “-3” is used to represent “No Coverage” or “Missing data”. However, in this reader replace those values by the value specified in the *fillna* argument (NaN by default).

Note that “missing values” are not the same as “no precipitation” values. Missing values indicates regions with no valid measures. While zero precipitation indicates regions with no valid measurements, but with no precipitation detected.

pysteps.io.importers.import_opera_hdf5

pysteps.io.importers.**import_opera_hdf5**(*filename*, *qty='RATE'*, ***kwargs*)
Import a precipitation field (and optionally the quality field) from an OPERA HDF5 file conforming to the ODIM specification.

Parameters

filename
[str] Name of the file to import.

qty :
The quantity to read from the file. The currently supported identitiers are: ‘RATE’=instantaneous rain rate (mm/h), ‘ACRR’=hourly rainfall accumulation (mm) and ‘DBZH’=max-reflectivity (dBZ). The default value is ‘RATE’.

Returns

out
[tuple] A three-element tuple containing the OPERA product for the requested quantity and the associated quality field and metadata. The quality field is read from the file if it contains a dataset whose quantity identifier is ‘QIND’.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: “float32”, “float64”, “single”, and “double”.

fillna
[float or np.nan] Value used to represent the missing data (“No Coverage”). By default, np.nan is used.

pysteps.io.importers.import_saf_crri

pysteps.io.importers.**import_saf_crri**(*filename*, *extent=None*, ***kwargs*)
Import a NetCDF radar rainfall product from the Convective Rainfall Rate Intensity (CRRI) product from the Satellite Application Facilities (SAF).

Product description available on http://www.nwcsaf.org/crr_description (last visited Jan 26, 2020).

Parameters

filename
[str] Name of the file to import.

extent
[scalars (left, right, bottom, top), optional] The spatial extent specified in data coordinates. If None, the full extent is imported.

Returns

out
[tuple] A three-element tuple containing the rainfall field in mm/h, the quality field and the metadata imported from the CRRI SAF netcdf file. The quality field includes values [1, 2, 4, 8, 16, 24, 32] meaning "nodata", "internal_consistency", "temporal_consistency", "good", "questionable", "bad", and "interpolated", respectively.

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.nowcast_importers

Methods for importing nowcast files.

The methods in this module implement the following interface:

```
import_xxx(filename, optional arguments)
```

where xxx is the name (or abbreviation) of the file format and filename is the name of the input file.

The output of each method is a two-element tuple containing the nowcast array and a metadata dictionary.

The metadata dictionary contains the following mandatory key-value pairs:

Key	Value
projection	PROJ.4-compatible projection definition
x1	x-coordinate of the lower-left corner of the data raster
y1	y-coordinate of the lower-left corner of the data raster
x2	x-coordinate of the upper-right corner of the data raster
y2	y-coordinate of the upper-right corner of the data raster
xpixelsize	grid resolution in x-direction
ypixelsize	grid resolution in y-direction
yorigin	a string specifying the location of the first element in the data raster w.r.t. y-axis: 'upper' = upper border 'lower' = lower border
institution	name of the institution who provides the data
timestep	time step of the input data (minutes)
unit	the physical unit of the data: 'mm/h', 'mm' or 'dBZ'
transform	the transformation of the data: None, 'dB', 'Box-Cox' or others
accutime	the accumulation time in minutes of the data, float
threshold	the rain/no rain threshold with the same unit, transformation and accutime of the data.
zerovalue	it is the value assigned to the no rain pixels with the same unit, transformation and accutime of the data.

Available Nowcast Importers

import_netcdf_pysteps(filename[, onerror])	Read a nowcast or an ensemble of nowcasts from a NetCDF file conforming to the CF 1.7 specification.

pysteps.io.nowcast_importers.import_netcdf_pysteps

pysteps.io.nowcast_importers.**import_netcdf_pysteps**(*filename*, *onerror='warn'*, ***kwargs*)

Read a nowcast or an ensemble of nowcasts from a NetCDF file conforming to the CF 1.7 specification.

If an error occurs during the import, the corresponding error message is shown, and (None, None) is returned.

Parameters

filename
[str] Name of the file to import.

onerror
[str] Define the behavior if an exception is raised during the import. - "warn": Print an error message and return (None, None) - "raise": Raise an exception

Returns

precipitation
[2D array, float32] Precipitation field in mm/h. The dimensions are [latitude, longitude]. The first grid point (0,0) corresponds to the upper left corner of the domain, while (last i, last j) denote the lower right corner.

metadata
[dict] Associated metadata (pixel sizes, map projections, etc.).

Other Parameters

dtype
[str] Data-type to which the array is cast. Valid values: "float32", "float64", "single", and "double".

fillna
[float or np.nan] Value used to represent the missing data ("No Coverage"). By default, np.nan is used.

pysteps.io.exporters

Methods for exporting forecasts of 2d precipitation fields into various file formats.

Each exporter method in this module has its own initialization function that implements the following interface:

```
initialize_forecast_exporter_xxx(outpath, outfnprefix, startdate, timestep,
                                 n_timesteps, shape, metadata,
                                 n_ens_members=1,
                                 incremental=None, **kwargs)
```

where xxx specifies the file format.

This function creates the output files and writes the metadata. See the documentation of the initialization methods for the format of the output files and their names. The datasets are written by calling *pysteps.io.exporters.export_forecast_dataset()*, and the files are closed by calling *pysteps.io.exporters.close_forecast_files()*.

The arguments of initialize_forecast_exporter_xxx are described in the following table:

Argument	Type/values	Description
outpath	str	output path
outfnprefix	str	prefix of output file names
startdate	datetime.datetime	start date of the forecast
timestep	int	length of the forecast time step (minutes)
n_timesteps	int	number of time steps in the forecast this argument is ignored if incremental is set to 'timestep'.
shape	tuple	two-element tuple defining the shape (height,width) of the forecast grids
metadata	dict	metadata dictionary containing the projection,x1,x2,y1,y2 and unit attributes described in the documentation of pysteps.io.importers
n_ens_members	int	number of ensemble members in the forecast this argument is ignored if incremental is set to 'member'
incremental	{None, 'timestep', 'member'}	allow incremental writing of datasets the available options are: 'timestep' = write a forecast or a forecast ensemble for a given time step 'member' = write a forecast sequence for a given ensemble member

Optional exporter-specific arguments are passed with `kwargs`. The return value is a dictionary containing an exporter object. This can be used with *pysteps.io.exporters.export_forecast_dataset()* to write the datasets to the output files.

Available Exporters

initialize_forecast_exporter_geotiff(...[,...])	Initialize a GeoTIFF forecast exporter.
initialize_forecast_exporter_kineros(...[,...])	Initialize a KINEROS2 Rainfall .pre file as specified in https://www.tucson.ars.ag.gov/kineros/.
initialize_forecast_exporter_netcdf(outpath,...)	Initialize a netCDF forecast exporter.

pysteps.io.exporters.initialize_forecast_exporter_geotiff

pysteps.io.exporters.**initialize_forecast_exporter_geotiff**(*outpath*, *outfnprefix*, *startdate*, *timestep*, *n_timesteps*, *shape*, *metadata*, *n_ens_members=1*, *incremental=None*, ***kwargs*)

Initialize a GeoTIFF forecast exporter.

The output files are named as '<outfnprefix>_<startdate>_<t>.tif', where startdate is in YYmmddHHMM format and t is lead time (minutes). GDAL needs to be installed to use this exporter.

Parameters

outpath
[str] Output path.

outfnprefix
[str] Prefix for output file names.

startdate
[datetime.datetime] Start date of the forecast.

timestep
[int] Time step of the forecast (minutes).

n_timesteps
[int] Number of time steps in the forecast. This argument is ignored if incremental is set to 'timestep'.

shape
[tuple of int] Two-element tuple defining the shape (height,width) of the forecast grids.

metadata: dict
Metadata dictionary containing the projection,x1,x2,y1,y2 and unit attributes described in the documentation of *pysteps.io.importers*.

n_ens_members
[int] Number of ensemble members in the forecast.

incremental
[{None,'timestep'}, optional] Allow incremental writing of datasets into the GeoTIFF files. Set to 'timestep' to enable writing forecasts or forecast ensembles separately for each time step. If set to None, incremental writing is disabled and the whole forecast is written in a single function call. The 'member' option is not currently implemented.

Returns

exporter
[dict] The return value is a dictionary containing an exporter object. This can be used with *pysteps.io.exporters.export_forecast_dataset()* to write the datasets.

pysteps.io.exporters.initialize_forecast_exporter_kineros

pysteps.io.exporters.**initialize_forecast_exporter_kineros**(*outpath*, *outfnprefix*, *startdate*, *timestep*, *n_timesteps*, *shape*, *metadata*, *n_ens_members=1*, *incremental=None*, ***kwargs*)

Initialize a KINEROS2 Rainfall .pre file as specified in https://www.tucson.ars.ag.gov/kineros/.

Grid points are treated as individual rain gauges and a separate file is produced for each ensemble member. The output files are named as <outfnprefix>_N<n>.pre, where <n> is the index of ensemble member starting from zero.

Parameters

outpath
[str] Output path.

outfnprefix
[str] Prefix for output file names.

startdate
[datetime.datetime] Start date of the forecast.

timestep
[int] Time step of the forecast (minutes).

n_timesteps
[int] Number of time steps in the forecast this argument is ignored if incremental is set to 'timestep'.

shape
[tuple of int] Two-element tuple defining the shape (height,width) of the forecast grids.

metadata: dict
Metadata dictionary containing the projection,x1,x2,y1,y2 and unit attributes described in the documentation of *pysteps.io.importers*.

n_ens_members
[int] Number of ensemble members in the forecast. This argument is ignored if incremental is set to 'member'.

incremental
[{None}, optional] Currently not implemented for this method.

Returns

exporter
[dict] The return value is a dictionary containing an exporter object. This c an be used with *pysteps.io.exporters.export_forecast_dataset()* to write datasets into the given file format.

pysteps.io.exporters.initialize_forecast_exporter_netcdf

pysteps.io.exporters.**initialize_forecast_exporter_netcdf**(*outpath*, *outfnprefix*, *startdate*, *timestep*, *n_timesteps*, *shape*, *metadata*, *n_ens_members=1*, *incremental=None*, ***kwargs*)

Initialize a netCDF forecast exporter. All outputs are written to a single file named as '<outfnprefix>_.nc'.

Parameters

outpath
[str] Output path.

outfnprefix
[str] Prefix for output file names.

startdate
[datetime.datetime] Start date of the forecast.

timestep
[int] Time step of the forecast (minutes).

n_timesteps
[int] Number of time steps in the forecast this argument is ignored if incremental is set to 'timestep'.

shape
[tuple of int] Two-element tuple defining the shape (height,width) of the forecast grids.

metadata: dict
Metadata dictionary containing the projection,x1,x2,y1,y2 and unit attributes described in the documentation of *pysteps.io.importers*.

n_ens_members
[int] Number of ensemble members in the forecast. This argument is ignored if incremental is set to 'member'.

incremental
[{None,'timestep','member'}, optional] Allow incremental writing of datasets into the netCDF files.

The available options are: 'timestep' = write a forecast or a forecast ensemble for a given time step; 'member' = write a forecast sequence for a given ensemble member. If set to None, incremental writing is disabled.

Returns

exporter
[dict] The return value is a dictionary containing an exporter object. This c an be used with *pysteps.io.exporters.export_forecast_dataset()* to write datasets into the given file format.

Generic functions

export_forecast_dataset(field, exporter)	Write a forecast array into a file.
close_forecast_files(exporter)	Close the files associated with a forecast exporter.

pysteps.io.exporters.export_forecast_dataset

pysteps.io.exporters.**export_forecast_dataset**(*field*, *exporter*)

Write a forecast array into a file.

If the exporter was initialized with n_ens_members>1, the written dataset has dimensions (n_ens_members,num_timesteps,shape[0],shape[1]), where shape refers to the shape of the two-dimensional forecast grids. Otherwise, the dimensions are (num_timesteps,shape[0],shape[1]). If the exporter was initialized with incremental!=None, the array is appended to the existing dataset either along the ensemble member or time axis.

Parameters

exporter
[dict] An exporter object created with any initialization method implemented in *pysteps.io.exporters*.

field
[array_like] The array to write. The required shape depends on the choice of the 'incremental' parameter the exporter was initialized with:

incremental	required shape
None	(num_ens_members,num_timesteps,shape[0],shape[1])
'timestep'	(num_ens_members,shape[0],shape[1])
'member'	(num_timesteps,shape[0],shape[1])

If the exporter was initialized with num_ens_members=1, the num_ens_members dimension is dropped.

pysteps.io.exporters.close_forecast_files

pysteps.io.exporters.**close_forecast_files**(*exporter*)

Close the files associated with a forecast exporter.

Finish writing forecasts and close the output files opened by a forecast exporter.

Parameters

exporter
[dict] An exporter object created with any initialization method implemented in *pysteps.io.exporters*.

pysteps.io.readers

Module with the reader functions.

read_timeseries(inputfns, importer, **kwargs)	Read a time series of input files using the methods implemented in the *pysteps.io.importers* module and stack them into a 3d array of shape (num_timesteps, height, width).

pysteps.io.readers.read_timeseries

pysteps.io.readers.**read_timeseries**(*inputfns*, *importer*, ***kwargs*)

Read a time series of input files using the methods implemented in the *pysteps.io.importers* module and stack them into a 3d array of shape (num_timesteps, height, width).

Parameters

inputfns
[tuple] Input files returned by a function implemented in the *pysteps.io.archive* module.

importer
[function] A function implemented in the *pysteps.io.importers* module.

kwargs
[dict] Optional keyword arguments for the importer.

Returns

out
[tuple] A three-element tuple containing the read data and quality rasters and associated metadata. If an input file name is None, the corresponding precipitation and quality fields are filled with nan values. If all input file names are None or if the length of the file name list is zero, a three-element tuple containing None values is returned.

2.2.6 pysteps.motion

Implementations of optical flow methods.

pysteps.motion.interface

Interface for the motion module. It returns a callable optical flow routine for computing the motion field.

The methods in the motion module implement the following interface:

```
motion_method(precip, **keywords)
```

where precip is a (T,m,n) array containing a sequence of T two-dimensional input images of shape (m,n). The first dimension represents the images time dimension and the value of T depends on the type of the method.

The output is a three-dimensional array (2,m,n) containing the dense x- and y-components of the motion field in units of pixels / timestep as given by the input array R.

get_method(name)	Return a callable function for the optical flow method corresponding to the given name.

pysteps.motion.interface.get_method

pysteps.motion.interface.**get_method**(*name*)
Return a callable function for the optical flow method corresponding to the given name. The available options are:

Python-based implementations	
Name	Description
None	returns a zero motion field
constant	constant advection field estimated by maximizing the correlation between two images
darts	implementation of the DARTS method of Ruzanski et al. (2011)
lucaskanade	OpenCV implementation of the Lucas-Kanade method with interpolated motion vectors for areas with no precipitation
proesmans	the anisotropic diffusion method of Proesmans et al. (1994)
vet	implementation of the VET method of Laroche and Zawadzki (1995) and Germann and Zawadzki (2002)

Methods implemented in C (these require separate compilation and linkage)	
Name	Description
brox	implementation of the variational method of Brox et al. (2004) from IPOL (http://www.ipol.im/pub/art/2013/21)
clg	implementation of the Combined Local-Global (CLG) method of Bruhn et al., 2005 from IPOL (http://www.ipol.im/pub/art/2015/44)

pysteps.motion.constant

Implementation of a constant advection field estimation by maximizing the correlation between two images.

constant(R, **kwargs)	Compute a constant advection field by finding a translation vector that maximizes the correlation between two successive images.

pysteps.motion.constant.constant

pysteps.motion.constant.**constant**(*R*, ***kwargs*)

Compute a constant advection field by finding a translation vector that maximizes the correlation between two successive images.

Parameters

R

[array_like] Array of shape (T,m,n) containing a sequence of T two-dimensional input images of shape (m,n). If T > 2, two last elements along axis 0 are used.

Returns

out

[array_like] The constant advection field having shape (2, m, n), where out[0, :, :] contains the x-components of the motion vectors and out[1, :, :] contains the y-components.

pysteps.motion.darts

Implementation of the DARTS algorithm.

DARTS(input_images, **kwargs)	Compute the advection field from a sequence of input images by using the DARTS method.

pysteps.motion.darts.DARTS

pysteps.motion.darts.**DARTS**(*input_images*, ***kwargs*)

Compute the advection field from a sequence of input images by using the DARTS method. [RCW11]

Parameters

input_images

[array-like] Array of shape (T,m,n) containing a sequence of T two-dimensional input images of shape (m,n).

Returns

out

[ndarray] Three-dimensional array (2,m,n) containing the dense x- and y-components

of the motion field in units of pixels / timestep as given by the input array R.

Other Parameters

N_x
[int] Number of DFT coefficients to use for the input images, x-axis (default=50).

N_y
[int] Number of DFT coefficients to use for the input images, y-axis (default=50).

N_t
[int] Number of DFT coefficients to use for the input images, time axis (default=4). N_t must be strictly smaller than T.

M_x
[int] Number of DFT coefficients to compute for the output advection field, x-axis (default=2).

M_y
[int] Number of DFT coefficients to compute for the output advection field, y-axis (default=2).

fft_method
[str] A string defining the FFT method to use, see utils.fft.get_method. Defaults to 'numpy'.

output_type
[{"spatial", "spectral"}] The type of the output: "spatial"=apply the inverse FFT to obtain the spatial representation of the advection field, "spectral"=return the (truncated) DFT representation.

n_threads
[int] Number of threads to use for the FFT computation. Applicable if fft_method is 'pyfftw'.

verbose
[bool] If True, print information messages.

lsq_method
[{1, 2}] The method to use for solving the linear equations in the least squares sense: 1=numpy.linalg.lstsq, 2=explicit computation of the Moore-Penrose pseudoinverse and SVD.

verbose
[bool] if set to True, it prints information about the program

pysteps.motion.lucaskanade

The Lucas-Kanade (LK) local feature tracking module.

This module implements the interface to the local Lucas-Kanade routine available in OpenCV.

For its dense method, it additionally interpolates the sparse vectors over a regular grid to return a motion field.

dense_lucaskanade(input_images[, lk_kwargs, ...])	Run the Lucas-Kanade optical flow routine and interpolate the motion vectors.
track_features(prvs_image, next_image, points)	Interface to the OpenCV Lucas-Kanade features tracking algorithm (cv.calcOpticalFlowPyrLK).

pysteps.motion.lucaskanade.dense_lucaskanade

pysteps.motion.lucaskanade.**dense_lucaskanade**(*input_images*, *lk_kwargs=None*, *fd_method='ShiTomasi'*, *fd_kwargs=None*, *interp_method='rbfinterp2d'*, *interp_kwargs=None*, *dense=True*, *nr_std_outlier=3*, *k_outlier=30*, *size_opening=3*, *decl_scale=20*, *verbose=False*)

Run the Lucas-Kanade optical flow routine and interpolate the motion vectors.

Interface to the OpenCV implementation of the local Lucas-Kanade optical flow method applied in combination to a feature detection routine.

The sparse motion vectors are finally interpolated to return the whole motion field.

Parameters

input_images
[ndarray or MaskedArray] Array of shape (T, m, n) containing a sequence of T two-dimensional input images of shape (m, n). The indexing order in **input_images** is assumed to be (time, latitude, longitude).

T = 2 is the minimum required number of images. With T > 2, all the resulting sparse vectors are pooled together for the final interpolation on a regular grid.

In case of ndarray, invalid values (Nans or infs) are masked, otherwise the mask of the MaskedArray is used. Such mask defines a region where features are not detected for the tracking algorithm.

lk_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the Lucas-Kanade features tracking algorithm. See the documentation of *pysteps.motion.lucaskanade.track_features()*.

fd_method
[{"ShiTomasi", "blob"}, optional] Name of the feature detection routine. See feature detection methods in *pysteps.utils.images*.

fd_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the features detection algorithm. See the documentation of *pysteps.utils.images*.

interp_method
[{"rbfinterp2d"}, optional] Name of the interpolation method to use. See interpolation methods in *pysteps.utils.interpolate*.

interp_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the interpolation algorithm. See the documentation of *pysteps.utils.interpolate*.

dense
[bool, optional] If True, return the three-dimensional array (2, m, n) containing the dense x- and y-components of the motion field.

If False, return the sparse motion vectors as 2-D **xy** and **uv** arrays, where **xy** defines the vector positions, **uv** defines the x and y direction components of the vectors.

nr_std_outlier
[int, optional] Maximum acceptable deviation from the mean in terms of number of standard deviations. Any sparse vector with a deviation larger than this threshold is flagged as outlier and excluded from the interpolation. See the documentation of *pysteps.utils.cleansing.detect_outliers()*.

k_outlier
: [int or None, optional] The number of nearest neighbours used to localize the outlier detection. If set to None, it employs all the data points (global detection). See the documentation of *pysteps.utils.cleansing.detect_outliers()*.

size_opening
: [int, optional] The size of the structuring element kernel in pixels. This is used to perform a binary morphological opening on the input fields in order to filter isolated echoes due to clutter. If set to zero, the filtering is not perfomed. See the documentation of *pysteps.utils.images.morph_opening()*.

decl_scale
: [int, optional] The scale declustering parameter in pixels used to reduce the number of redundant sparse vectors before the interpolation. Sparse vectors within this declustering scale are averaged together. If set to less than 2 pixels, the declustering is not perfomed. See the documentation of *pysteps.utils.cleansing.decluster()*.

verbose
: [bool, optional] If set to True, print some information about the program.

Returns

out
: [ndarray or tuple] If **dense=True** (the default), return the advection field having shape (2, m, n), where out[0, :, :] contains the x-components of the motion vectors and out[1, :, :] contains the y-components. The velocities are in units of pixels / timestep, where timestep is the time difference between the two input images. Return a zero motion field of shape (2, m, n) when no motion is detected.

If **dense=False**, it returns a tuple containing the 2-dimensional arrays **xy** and **uv**, where x, y define the vector locations, u, v define the x and y direction components of the vectors. Return two empty arrays when no motion is detected.

See also:

pysteps.motion.lucaskanade.track_features

References

Bouguet, J.-Y.: Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm, Intel Corp., 5, 4, https://doi.org/10.1109/HPDC.2004.1323531, 2001

Lucas, B. D. and Kanade, T.: An iterative image registration technique with an application to stereo vision, in: Proceedings of the 1981 DARPA Imaging Understanding Workshop, pp. 121–130, 1981.

pysteps.motion.lucaskanade.track_features

`pysteps.motion.lucaskanade.`**`track_features`**(*prvs_image*, *next_image*, *points*, *winsize=(50, 50)*, *nr_levels=3*, *criteria=(3, 10, 0)*, *flags=0*, *min_eig_thr=0.0001*, *verbose=False*)

Interface to the OpenCV Lucas-Kanade features tracking algorithm (cv.calcOpticalFlowPyrLK).

Parameters

prvs_image
: [ndarray or MaskedArray] Array of shape (m, n) containing the first image. Invalid values (Nans or infs) are filled using the min value.

next_image
[ndarray or MaskedArray] Array of shape (m, n) containing the successive image. Invalid values (Nans or infs) are filled using the min value.

points
[array_like] Array of shape (p, 2) indicating the pixel coordinates of the tracking points (corners).

winsize
[tuple of int, optional] The **winSize** parameter in calcOpticalFlowPyrLK. It represents the size of the search window that it is used at each pyramid level.

nr_levels
[int, optional] The **maxLevel** parameter in calcOpticalFlowPyrLK. It represents the 0-based maximal pyramid level number.

criteria
[tuple of int, optional] The **TermCriteria** parameter in calcOpticalFlowPyrLK , which specifies the termination criteria of the iterative search algorithm.

flags
[int, optional] Operation flags, see documentation calcOpticalFlowPyrLK.

min_eig_thr
[float, optional] The **minEigThreshold** parameter in calcOpticalFlowPyrLK.

verbose
[bool, optional] Print the number of vectors that have been found.

Returns

xy
[ndarray] Array of shape (d, 2) with the x- and y-coordinates of $d <= p$ detected sparse motion vectors.

uv
[ndarray] Array of shape (d, 2) with the u- and v-components of $d <= p$ detected sparse motion vectors.

See also:

pysteps.motion.lucaskanade.dense_lucaskanade

Notes

The tracking points can be obtained with the `pysteps.utils.images.ShiTomasi_detection()` routine.

References

Bouguet, J.-Y.: Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm, Intel Corp., 5, 4, https://doi.org/10.1109/HPDC.2004.1323531, 2001

Lucas, B. D. and Kanade, T.: An iterative image registration technique with an application to stereo vision, in: Proceedings of the 1981 DARPA Imaging Understanding Workshop, pp. 121–130, 1981.

pysteps.motion.proesmans

Implementation of the anisotropic diffusion method of Proesmans et al. (1994).

proesmans(input_images[, lam, num_iter, ...])	Implementation of the anisotropic diffusion method of Proesmans et al.

pysteps.motion.proesmans.proesmans

pysteps.motion.proesmans.**proesmans**(*input_images*, *lam=50.0*, *num_iter=100*, *num_levels=6*, *filter_std=0.0*, *verbose=True*)

Implementation of the anisotropic diffusion method of Proesmans et al. (1994).

Parameters

input_images
[array_like] Array of shape (2, m, n) containing the first and second input image.

lam
[float] Multiplier of the smoothness term. Smaller values give a smoother motion field.

num_iter
[float] The number of iterations to use.

num_levels
[int] The number of image pyramid levels to use.

filter_std
[float] Standard deviation of an optional Gaussian filter that is applied before computing the optical flow.

verbose
[bool, optional] Verbosity enabled if True (default).

Returns

out
[ndarray] The advection field having shape (2, m, n), where out[0, :, :] contains the x-components of the motion vectors and out[1, :, :] contains the y-components. The velocities are in units of pixels / timestep, where timestep is the time difference between the two input images.

References

[PvGPO94]

pysteps.motion.vet

Variational Echo Tracking (VET) Module

This module implements the VET algorithm presented by Laroche and Zawadzki (1995) and used in the McGill Algorithm for Prediction by Lagrangian Extrapolation (MAPLE) described in Germann and Zawadzki (2002).

The morphing and the cost functions are implemented in Cython and parallelized for performance.

vet(input_images[, sectors, smooth_gain, ...])	Variational Echo Tracking Algorithm presented in Laroche and Zawadzki (1995) and used in the McGill Algorithm for Prediction by Lagrangian Extrapolation (MAPLE) described in Germann and Zawadzki (2002).
vet_cost_function(sector_displacement_1d, ...)	

Continued on next page

Table 20 – continued from previous page

vet_cost_function_gradient(*args, **kwargs)	Compute the vet cost function gradient.
morph(image, displacement[, gradient])	Morph image by applying a displacement field (Warping).
round_int(scalar)	Round number to nearest integer.
ceil_int(scalar)	Round number to nearest integer.
get_padding(dimension_size, sectors)	Get the padding at each side of the one dimensions of the image so the new image dimensions are divided evenly in the number of *sectors* specified.

pysteps.motion.vet.vet

pysteps.motion.vet.**vet**(*input_images*, *sectors=((32, 16, 4, 2), (32, 16, 4, 2))*, *smooth_gain=1000000.0*, *first_guess=None*, *intermediate_steps=False*, *verbose=True*, *indexing='yx'*, *padding=0*, *options=None*)

Variational Echo Tracking Algorithm presented in Laroche and Zawadzki (1995) and used in the McGill Algorithm for Prediction by Lagrangian Extrapolation (MAPLE) described in Germann and Zawadzki (2002).

This algorithm computes the displacement field between two images (the input_image with respect to the template image). The displacement is sought by minimizing the sum of the residuals of the squared differences of the images pixels and the contribution of a smoothness constraint. In the case that a MaskedArray is used as input, the residuals term in the cost function is only computed over areas with non-masked values. Otherwise, it is computed over the entire domain.

To find the minimum, a scaling guess procedure is applied, from larger to smaller scales. This reduces the chances that the minimization procedure converges to a local minimum. The first scaling guess is defined by the scaling sectors keyword.

The smoothness of the returned displacement field is controlled by the smoothness constraint gain (**smooth_gain** keyword).

If a first guess is not given, zero displacements are used as the first guess.

The cost function is minimized using the scipy minimization function, with the 'CG' method by default. This method proved to give the best results under many different conditions and is the most similar one to the original VET implementation in Laroche and Zawadzki (1995).

The method CG uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method described in Nocedal and Wright (2006), pp. 120-122.

Parameters

input_images
[ndarray or MaskedArray] Input images, sequence of 2D arrays, or 3D arrays. The first dimension represents the images time dimension.

The template_image (first element in first dimensions) denotes the reference image used to obtain the displacement (2D array). The second is the target image.

The expected dimensions are (2,ni,nj).

sectors
[list or array, optional] Number of sectors on each dimension used in the scaling procedure. If dimension is 1, the same sectors will be used both image dimensions (x and y). If **sectors** is a 1D array, the same number of sectors is used in both dimensions.

smooth_gain
[float, optional] Smooth gain factor

first_guess
[ndarray, optional] The shape of the first guess should have the same shape as the initial

sectors shapes used in the scaling procedure. If first_guess is not present zeros are used as first guess.

E.g.:
If the first sector shape in the scaling procedure is (ni,nj), then the first_guess should have (2, ni, nj) shape.

intermediate_steps
[bool, optional] If True, also return a list with the first guesses obtained during the scaling procedure. False, by default.

verbose
[bool, optional] Verbosity enabled if True (default).

indexing
[str, optional] Input indexing order.'ij' and 'xy' indicates that the dimensions of the input are (time, longitude, latitude), while 'yx' indicates (time, latitude, longitude). The displacement field dimensions are ordered accordingly in a way that the first dimension indicates the displacement along x (0) or y (1). That is, UV displacements are always returned.

padding
[int] Padding width in grid points. A border is added to the input array to reduce the effects of the minimization at the border.

options
[dict, optional] A dictionary of solver options. See scipy minimization function for more details.

Returns

displacement_field
[ndarray] Displacement Field (2D array representing the transformation) that warps the template image into the input image. The dimensions are (2,ni,nj), where the first dimension indicates the displacement along x (0) or y (1) in units of pixels / timestep as given by the input_images array.

intermediate_steps
[list of ndarray] List with the first guesses obtained during the scaling procedure.

References

Laroche, S., and I. Zawadzki, 1995: Retrievals of horizontal winds from single-Doppler clear-air data by methods of cross-correlation and variational analysis. J. Atmos. Oceanic Technol., 12, 721–738. doi: http://dx.doi.org/10.1175/1520-0426(1995)012<0721:ROHWFS>2.0.CO;2

Germann, U. and I. Zawadzki, 2002: Scale-Dependence of the Predictability of Precipitation from Continental Radar Images. Part I: Description of the Methodology. Mon. Wea. Rev., 130, 2859–2873, doi: 10.1175/1520-0493(2002)130<2859:SDOTPO>2.0.CO;2.

Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.

pysteps.motion.vet.vet_cost_function

`pysteps.motion.vet.`**`vet_cost_function`**(*sector_displacement_1d*, *input_images*, *blocks_shape*, *mask*, *smooth_gain*, *debug=False*, *gradient=False*)

Variational Echo Tracking Cost Function.

This function is designed to be used with the scipy minimization. The function first argument is the variable to be used in the minimization procedure.

The sector displacement must be a flat array compatible with the dimensions of the input image and sectors shape (see parameters section below for more details).

Parameters

sector_displacement_1d
[ndarray] Array of displacements to apply to each sector. The dimensions are: sector_displacement_2d [x (0) or y (1) displacement, i index of sector, j index of sector]. The shape of the sector displacements must be compatible with the input image and the block shape. The shape should be (2, mx, my) where mx and my are the numbers of sectors in the x and the y dimension.

input_images
[ndarray] Input images, sequence of 2D arrays, or 3D arrays. The first dimension represents the images time dimension.

The template_image (first element in first dimensions) denotes the reference image used to obtain the displacement (2D array). The second is the target image.

The expected dimensions are (2,nx,ny). Be aware the the 2D images dimensions correspond to (lon,lat) or (x,y).

blocks_shape
[ndarray (ndim=2)] Number of sectors in each dimension (x and y). blocks_shape.shape = (mx,my)

mask
[ndarray (ndim=2)] Data mask. If is True, the data is marked as not valid and is not used in the computations.

smooth_gain
[float] Smoothness constrain gain

debug
[bool, optional] If True, print debugging information.

gradient
[bool, optional] If True, the gradient of the morphing function is returned.

Returns

penalty or gradient values.

penalty
[float] Value of the cost function

gradient_values
[ndarray (float64 ,ndim = 3), optional] If gradient keyword is True, the gradient of the function is also returned.

pysteps.motion.vet.vet_cost_function_gradient

`pysteps.motion.vet.`**`vet_cost_function_gradient`**(**args*, ***kwargs*)
Compute the vet cost function gradient. See *`vet_cost_function()`* for more information.

pysteps.motion.vet.morph

`pysteps.motion.vet.`**`morph`**(*image*, *displacement*, *gradient=False*)
Morph image by applying a displacement field (Warping).

The new image is created by selecting for each position the values of the input image at the positions given by the x and y displacements. The routine works in a backward sense. The displacement vectors have to refer to their destination.

For more information in Morphing functions see Section 3 in Beezley and Mandel (2008).

Beezley, J. D., & Mandel, J. (2008). Morphing ensemble Kalman filters. Tellus A, 60(1), 131-140.

The displacement field in x and y directions and the image must have the same dimensions.

The morphing is executed in parallel over x axis.

The value of displaced pixels that fall outside the limits takes the value of the nearest edge. Those pixels are indicated by values greater than 1 in the output mask.

Parameters

image
[ndarray (ndim = 2)] Image to morph

displacement
[ndarray (ndim = 3)] Displacement field to be applied (Warping). The first dimension corresponds to the coordinate to displace.

The dimensions are: displacement [i/x (0) or j/y (1) , i index of pixel, j index of pixel]

gradient
[bool, optional] If True, the gradient of the morphing function is returned.

Returns

image
[ndarray (float64 ,ndim = 2)] Morphed image.

mask
[ndarray (int8 ,ndim = 2)] Invalid values mask. Points outside the boundaries are masked. Values greater than 1, indicate masked values.

gradient_values
[ndarray (float64 ,ndim = 3), optional] If gradient keyword is True, the gradient of the function is also returned.

pysteps.motion.vet.round_int

`pysteps.motion.vet.`**`round_int`**(*scalar*)
Round number to nearest integer. Returns and integer value.

pysteps.motion.vet.ceil_int

`pysteps.motion.vet.`**`ceil_int`**(*scalar*)
Round number to nearest integer. Returns and integer value.

pysteps.motion.vet.get_padding

`pysteps.motion.vet.`**`get_padding`**(*dimension_size*, *sectors*)
Get the padding at each side of the one dimensions of the image so the new image dimensions are divided evenly in the number of *sectors* specified.

Parameters

dimension_size
[int] Actual dimension size.

sectors
[int] number of sectors over which the the image will be divided.

Returns

pad_before , pad_after: int, int
Padding at each side of the image for the corresponding dimension.

2.2.7 pysteps.noise

Implementation of deterministic and ensemble nowcasting methods.

pysteps.noise.interface

Interface for the noise module.

get_method(name)	Return two callable functions to initialize and generate 2d perturbations of precipitation or velocity fields.

pysteps.noise.interface.get_method

pysteps.noise.interface.**get_method**(*name*)

Return two callable functions to initialize and generate 2d perturbations of precipitation or velocity fields.

Methods for precipitation fields:

Name	Description
parametric	this global generator uses parametric Fourier filtering (power-law model)
nonparametric	this global generator uses nonparametric Fourier filtering
ssft	this local generator uses the short-space Fourier filtering
nested	this local generator uses a nested Fourier filtering

Methods for velocity fields:

Name	Description
bps	The method described in [BPS06], where time-dependent velocity perturbations are sampled from the exponential distribution

pysteps.noise.fftgenerators

Methods for noise generators based on FFT filtering of white noise.

The methods in this module implement the following interface for filter initialization depending on their parametric or nonparametric nature:

```
initialize_param_2d_xxx_filter(field, **kwargs)
```

or:

```
initialize_nonparam_2d_xxx_filter(field, **kwargs)
```

where field is an array of shape (m, n) or (t, m, n) that defines the target field and optional parameters are supplied as keyword arguments.

The output of each initialization method is a dictionary containing the keys field and input_shape. The first is a two-dimensional array of shape (m, int(n/2)+1) that defines the filter. The second one is the shape of the input field for the filter.

The methods in this module implement the following interface for the generation of correlated noise:

```
generate_noise_2d_xxx_filter(field, randstate=np.random, seed=None, **kwargs)
```

where field (m, n) is a filter returned from the corresponding initialization method, and randstate and seed can be used to set the random generator and its seed. Additional keyword arguments can be included as a dictionary.

The output of each generator method is a two-dimensional array containing the field of correlated noise cN of shape (m, n).

initialize_param_2d_fft_filter(field, **kwargs)	Takes one ore more 2d input fields, fits two spectral slopes, beta1 and beta2, to produce one parametric, global and isotropic fourier filter.
initialize_nonparam_2d_fft_filter(field, ...)	Takes one ore more 2d input fields and produces one non-parametric, global and anisotropic fourier filter.
initialize_nonparam_2d_nested_filter(field, ...)	Function to compute the local Fourier filters using a nested approach.
initialize_nonparam_2d_ssft_filter(field, ...)	Function to compute the local Fourier filters using the Short-Space Fourier filtering approach.
generate_noise_2d_fft_filter(F[, randstate, ...])	Produces a field of correlated noise using global Fourier filtering.
generate_noise_2d_ssft_filter(F[, ...])	Function to compute the locally correlated noise using a nested approach.

pysteps.noise.fftgenerators.initialize_param_2d_fft_filter

pysteps.noise.fftgenerators.**initialize_param_2d_fft_filter**(*field*, ***kwargs*)

Takes one ore more 2d input fields, fits two spectral slopes, beta1 and beta2, to produce one parametric, global and isotropic fourier filter.

Parameters

field
[array-like] Two- or three-dimensional array containing one or more input fields. All values are required to be finite. If more than one field are passed, the average fourier filter is returned. It assumes that fields are stacked by the first axis: [nr_fields, y, x].

Returns

out
[dict] A dictionary containing the keys field, input_shape, model and pars. The first is a two-dimensional array of shape (m, int(n/2)+1) that defines the filter. The second one is the shape of the input field for the filter. The last two are the model and fitted parameters, respectively.

This dictionary can be passed to *pysteps.noise.fftgenerators.generate_noise_2d_fft_filter()* to generate noise fields.

Other Parameters

win_fun
[{'hann', 'tukey' or None}] Optional tapering function to be applied to the input field, generated with *pysteps.utils.tapering.compute_window_function()*. (default None).

model
[{'power-law'}] The name of the parametric model to be used to fit the power spectrum of the input field (default 'power-law').

weighted
[bool] Whether or not to apply 1/sqrt(power) as weight in the numpy.polyfit() function (default False).

rm_rdisc
[bool] Whether or not to remove the rain/no-rain disconituity (default False). It assumes no-rain pixels are assigned with lowest value.

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see "FFT methods" in *pysteps.utils.interface.get_method()*). Defaults to "numpy".

pysteps.noise.fftgenerators.initialize_nonparam_2d_fft_filter

pysteps.noise.fftgenerators.**initialize_nonparam_2d_fft_filter**(*field*, ***kwargs*)

Takes one ore more 2d input fields and produces one non-parametric, global and anisotropic fourier filter.

Parameters

field
[array-like] Two- or three-dimensional array containing one or more input fields. All values are required to be finite. If more than one field are passed, the average fourier filter is returned. It assumes that fields are stacked by the first axis: [nr_fields, y, x].

Returns

out
[dict] A dictionary containing the keys field and input_shape. The first is a two-dimensional array of shape (m, int(n/2)+1) that defines the filter. The second one is the shape of the input field for the filter.

It can be passed to *pysteps.noise.fftgenerators.generate_noise_2d_fft_filter()*.

Other Parameters

win_fun
[{'hann', 'tukey', None}] Optional tapering function to be applied to the input field, generated with *pysteps.utils.tapering.compute_window_function()* (default 'tukey').

donorm
[bool] Option to normalize the real and imaginary parts. Default : False

rm_rdisc
[bool] Whether or not to remove the rain/no-rain disconituity (default True). It assumes no-rain pixels are assigned with lowest value.

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see "FFT methods" in *pysteps.utils.interface.get_method()*). Defaults to "numpy".

pysteps.noise.fftgenerators.initialize_nonparam_2d_nested_filter

pysteps.noise.fftgenerators.**initialize_nonparam_2d_nested_filter**(*field*, *gridres=1.0*, ***kwargs*)

Function to compute the local Fourier filters using a nested approach.

Parameters

field
[array-like] Two- or three-dimensional array containing one or more input fields. All values are required to be finite. If more than one field are passed, the average fourier filter is returned. It assumes that fields are stacked by the first axis: [nr_fields, y, x].

gridres
[float] Grid resolution in km.

Returns

field
[array-like] Four-dimensional array containing the 2d fourier filters distributed over a 2d spatial grid. It can be passed to *pysteps.noise.fftgenerators.generate_noise_2d_ssft_filter()*.

Other Parameters

max_level
[int] Localization parameter. 0: global noise, >0: increasing degree of localization (default 3).

win_fun
[{'hann', 'tukey', None}] Optional tapering function to be applied to the input field, generated with *pysteps.utils.tapering.compute_window_function()* (default 'tukey').

war_thr
[float [0;1]] Threshold for the minimum fraction of rain needed for computing the FFT (default 0.1).

rm_rdisc
[bool] Whether or not to remove the rain/no-rain disconituity. It assumes no-rain pixels are assigned with lowest value.

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see "FFT methods" in *pysteps.utils.interface.get_method()*). Defaults to "numpy".

pysteps.noise.fftgenerators.initialize_nonparam_2d_ssft_filter

pysteps.noise.fftgenerators.**initialize_nonparam_2d_ssft_filter**(*field*, ***kwargs*)

Function to compute the local Fourier filters using the Short-Space Fourier filtering approach.

Parameters

field
[array-like] Two- or three-dimensional array containing one or more input fields. All values are required to be finite. If more than one field are passed, the average fourier filter is returned. It assumes that fields are stacked by the first axis: [nr_fields, y, x].

Returns

field
[array-like] Four-dimensional array containing the 2d fourier filters distributed over a 2d spatial grid. It can be passed to *pysteps.noise.fftgenerators.generate_noise_2d_ssft_filter()*.

Other Parameters

win_size
[int or two-element tuple of ints] Size-length of the window to compute the SSFT (default (128, 128)).

win_fun
[{'hann', 'tukey', None}] Optional tapering function to be applied to the input field, generated with *pysteps.utils.tapering.compute_window_function()* (default 'tukey').

overlap
[float [0,1[] The proportion of overlap to be applied between successive windows (default 0.3).

war_thr
[float [0,1]] Threshold for the minimum fraction of rain needed for computing the FFT (default 0.1).

rm_rdisc
[bool] Whether or not to remove the rain/no-rain disconituity. It assumes no-rain pixels are assigned with lowest value.

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see "FFT methods" in *pysteps.utils.interface.get_method()*). Defaults to "numpy".

References

[NBS+17]

pysteps.noise.fftgenerators.generate_noise_2d_fft_filter

pysteps.noise.fftgenerators.**generate_noise_2d_fft_filter**(*F*, *randstate=None*, *seed=None*, *fft_method=None*, *domain='spatial'*)

Produces a field of correlated noise using global Fourier filtering.

Parameters

F
[dict] A filter object returned by *pysteps.noise.fftgenerators.initialize_param_2d_fft_filter()* or *pysteps.noise.fftgenerators.initialize_nonparam_2d_fft_filter()*. All values in the filter array are required to be finite.

randstate
[mtrand.RandomState] Optional random generator to use. If set to None, use numpy.random.

seed
[int] Value to set a seed for the generator. None will not set the seed.

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see "FFT methods" in *pysteps.utils.interface.get_method()*). Defaults to "numpy".

domain
[{"spatial", "spectral"}] The domain for the computations: If "spatial", the noise is generated in the spatial domain and transformed back to spatial domain after the Fourier filtering. If "spectral", the noise field is generated and kept in the spectral domain.

Returns

N
[array-like] A two-dimensional field of stationary correlated noise. The noise field is normalized to zero mean and unit variance.

pysteps.noise.fftgenerators.generate_noise_2d_ssft_filter

pysteps.noise.fftgenerators.**generate_noise_2d_ssft_filter**(*F*, *randstate=None*, *seed=None*, ***kwargs*)

Function to compute the locally correlated noise using a nested approach.

Parameters

F
[array-like] A filter object returned by *pysteps.noise.fftgenerators.initialize_nonparam_2d_ssft_filter()* or *pysteps.noise.fftgenerators.initialize_nonparam_2d_nested_filter()*. The filter is a four-dimensional array containing the 2d fourier filters distributed over a 2d spatial grid.

randstate
[mtrand.RandomState] Optional random generator to use. If set to None, use numpy.random.

seed
[int] Value to set a seed for the generator. None will not set the seed.

Returns

N
[array-like] A two-dimensional numpy array of non-stationary correlated noise.

Other Parameters

overlap
[float] Percentage overlap [0-1] between successive windows (default 0.2).

win_fun
[{'hann', 'tukey', None}] Optional tapering function to be applied to the input field, generated with *pysteps.utils.tapering.compute_window_function()* (default 'tukey').

fft_method
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see "FFT methods" in *pysteps.utils.interface.get_method()*). Defaults to "numpy".

pysteps.noise.motion

Methods for generating perturbations of two-dimensional motion fields.

The methods in this module implement the following interface for initialization:

```
inizialize_xxx(V, pixelsperkm, timestep, optional arguments)
```

where V (2,m,n) is the motion field and pixelsperkm and timestep describe the spatial and temporal resolution of the motion vectors. The output of each initialization method is a dictionary containing the perturbator that can be supplied to generate_xxx.

The methods in this module implement the following interface for the generation of a motion perturbation field:

```
generate_xxx(perturbator, t, randstate=np.random, seed=None)
```

where perturbator is a dictionary returned by an initialize_xxx method. Optional random generator can be specified with the randstate and seed arguments, respectively. The output of each generator method is an array of shape (2,m,n) containing the x- and y-components of the motion vector perturbations, where m and n are determined from the perturbator.

get_default_params_bps_par()	Return a tuple containing the default velocity perturbation parameters given in [BPS06] for the parallel component.
get_default_params_bps_perp()	Return a tuple containing the default velocity perturbation parameters given in [BPS06] for the perpendicular component.
initialize_bps(V, pixelsperkm, timestep[, ...])	Initialize the motion field perturbator described in [BPS06].
generate_bps(perturbator, t)	Generate a motion perturbation field by using the method described in [BPS06].

pysteps.noise.motion.get_default_params_bps_par

pysteps.noise.motion.**get_default_params_bps_par**()

Return a tuple containing the default velocity perturbation parameters given in [BPS06] for the parallel component.

pysteps.noise.motion.get_default_params_bps_perp

pysteps.noise.motion.**get_default_params_bps_perp**()

Return a tuple containing the default velocity perturbation parameters given in [BPS06] for the perpendicular component.

pysteps.noise.motion.initialize_bps

pysteps.noise.motion.**initialize_bps**(*V*, *pixelsperkm*, *timestep*, *p_par=None*, *p_perp=None*, *randstate=None*, *seed=None*)

Initialize the motion field perturbator described in [BPS06]. For simplicity, the bias adjustment procedure described there has not been implemented. The perturbator generates a field whose magnitude increases with respect to lead time.

Parameters

V
[array_like] Array of shape (2,m,n) containing the x- and y-components of the m*n motion field to perturb.

p_par
[tuple] Tuple containing the parameters a,b and c for the standard deviation of the perturbations in the direction parallel to the motion vectors. The standard deviations are modeled by the function f_par(t) = a*t**b+c, where t is lead time. The default values are taken from [BPS06].

p_perp
[tuple] Tuple containing the parameters a,b and c for the standard deviation of the perturbations in the direction perpendicular to the motion vectors. The standard deviations are modeled by the function f_par(t) = a*t**b+c, where t is lead time. The default values are taken from [BPS06].

pixelsperkm
[float] Spatial resolution of the motion field (pixels/kilometer).

timestep
[float] Time step for the motion vectors (minutes).

randstate
[mtrand.RandomState] Optional random generator to use. If set to None, use numpy.random.

seed
[int] Optional seed number for the random generator.

Returns

out
[dict] A dictionary containing the perturbator that can be supplied to generate_motion_perturbations_bps.

See also:

pysteps.noise.motion.generate_bps

pysteps.noise.motion.generate_bps

pysteps.noise.motion.**generate_bps**(*perturbator*, *t*)
Generate a motion perturbation field by using the method described in [BPS06].

Parameters

perturbator
[dict] A dictionary returned by initialize_motion_perturbations_bps.

t
[float] Lead time for the perturbation field (minutes).

Returns

out
[ndarray] Array of shape (2,m,n) containing the x- and y-components of the motion vector perturbations, where m and n are determined from the perturbator.

See also:

pysteps.noise.motion.initialize_bps

pysteps.noise.utils

Miscellaneous utility functions related to generation of stochastic perturbations.

compute_noise_stddev_adjs(R, R_thr_1, ...[, ...])	Apply a scale-dependent adjustment factor to the noise fields used in STEPS.

pysteps.noise.utils.compute_noise_stddev_adjs

pysteps.noise.utils.**compute_noise_stddev_adjs**(*R*, *R_thr_1*, *R_thr_2*, *F*, *decomp_method*, *noise_filter*, *noise_generator*, *num_iter*, *conditional=True*, *num_workers=1*, *seed=None*)
Apply a scale-dependent adjustment factor to the noise fields used in STEPS.

Simulates the effect of applying a precipitation mask to a Gaussian noise field obtained by the nonparametric filter method. The idea is to decompose the masked noise field into a cascade and compare the standard

deviations of each level into those of the observed precipitation intensity field. This gives correction factors for the standard deviations [BPS06]. The calculations are done for n realizations of the noise field, and the correction factors are calculated from the average values of the standard deviations.

Parameters

R
[array_like] The input precipitation field, assumed to be in logarithmic units (dBR or reflectivity).

R_thr_1
[float] Intensity threshold for precipitation/no precipitation.

R_thr_2
[float] Intensity values below R_thr_1 are set to this value.

F
[dict] A bandpass filter dictionary returned by a method defined in pysteps.cascade.bandpass_filters. This defines the filter to use and the number of cascade levels.

decomp_method
[function] A function defined in pysteps.cascade.decomposition. Specifies the method to use for decomposing the observed precipitation field and noise field into different spatial scales.

num_iter
[int] The number of noise fields to generate.

conditional
[bool] If set to True, compute the statistics conditionally by excluding areas of no precipitation.

num_workers
[int] The number of workers to use for parallel computation. Applicable if dask is installed.

seed
[int] Optional seed number for the random generators.

Returns

out
[list] A list containing the standard deviation adjustment factor for each cascade level.

2.2.8 pysteps.nowcasts

Implementation of deterministic and ensemble nowcasting methods.

pysteps.nowcasts.interface

Interface for the nowcasts module. It returns a callable function for computing nowcasts.

The methods in the nowcasts module implement the following interface:

```
forecast(precip, velocity, num_timesteps, **keywords)
```

where precip is a (m,n) array with input precipitation field to be advected and velocity is a (2,m,n) array containing the x- and y-components of the m x n advection field. num_timesteps is an integer specifying the number of time steps to forecast. The interface accepts optional keyword arguments specific to the given method.

The output depends on the type of the method. For deterministic methods, the output is a three-dimensional array of shape (num_timesteps,m,n) containing a time series of nowcast precipitation fields.

For stochastic methods that produce an ensemble, the output is a four-dimensional array of shape (num_ensemble_members,num_timesteps,m,n). The time step of the output is taken from the inputs.

get_method(name)	Return a callable function for computing nowcasts.

pysteps.nowcasts.interface.get_method

pysteps.nowcasts.interface.**get_method**(*name*)

Return a callable function for computing nowcasts.

Description: Return a callable function for computing deterministic or ensemble precipitation nowcasts.

Implemented methods:

Name	Description
anvil	the autoregressive nowcasting using VIL (ANVIL) nowcasting method developed in [PCvLH20]
eulerian	this approach keeps the last observation frozen (Eulerian persistence)
lagrangian or extrapolation	this approach extrapolates the last observation using the motion field (Lagrangian persistence)
sprog	the S-PROG method described in [See03]
steps	the STEPS stochastic nowcasting method described in [See03], [BPS06] and [SPN13]
sseps	short-space ensemble prediction system (SSEPS). Essentially, this is a localization of STEPS.

steps and sseps produce stochastic nowcasts, and the other methods are deterministic.

pysteps.nowcasts.anvil

Implementation of the autoregressive nowcasting using VIL (ANVIL) nowcasting method developed in [PCvLH20]. Compared to S-PROG, the main improvements are using an autoregressive integrated (ARI) model and the option to use vertically integrated liquid (VIL) as the input variable. Using the ARI model avoids biasedness and loss of small-scale features in the forecast field, and no statistical post-processing is needed. In addition, the model allows localization of parameter estimates. It was shown in [PCvLH20] that due to the above improvements, ANVIL produces more reliable deterministic nowcasts than S-PROG.

forecast(vil, velocity, n_timesteps[, ...])	Generate a nowcast by using the autoregressive nowcasting using VIL (ANVIL) method.

pysteps.nowcasts.anvil.forecast

pysteps.nowcasts.anvil.**forecast**(*vil*, *velocity*, *n_timesteps*, *rainrate=None*, *n_cascade_levels=8*, *extrap_method='semilagrangian'*, *ar_order=2*, *ar_window_radius=50*, *r_vil_window_radius=3*, *fft_method='numpy'*, *num_workers=1*, *extrap_kwargs=None*, *filter_kwargs=None*, *measure_time=False*)

Generate a nowcast by using the autoregressive nowcasting using VIL (ANVIL) method. The key features of ANVIL are:

1) Extrapolation-based nowcast.

2) Growth and decay: implemented by using a cascade decomposition and a multiscale autoregressive integrated ARI(p,1) model. Instead of the original time series, the ARI model is applied to the differenced one corresponding to time derivatives.

4) Originally designed for using integrated liquid (VIL) as the input data. In this case, the rain rate (R) is obtained from VIL via an empirical relation. This implementation is more general so that the input

can be any two-dimensional precipitation field.

3) The parameters of the ARI model and the R(VIL) relation are allowed to be spatially variable. The estimation is done using a moving window.

Parameters

vil
[array_like] Array of shape (ar_order+2,m,n) containing the input fields ordered by timestamp from oldest to newest. The time steps between the inputs are assumed to be regular.

velocity
[array_like] Array of shape (2,m,n) containing the x- and y-components of the advection field. The velocities are assumed to represent one time step between the inputs. All values are required to be finite.

n_timesteps
[int] Number of time steps to forecast.

rainrate
[array_like] Array of shape (m,n) containing the most recently observed rain rate field. If set to None, no R(VIL) conversion is done and the outputs are in the same units as the inputs.

n_cascade_levels
[int, optional] The number of cascade levels to use.

extrap_method
[str, optional] Name of the extrapolation method to use. See the documentation of pysteps.extrapolation.interface.

ar_order
[int, optional] The order of the autoregressive model to use. The recommended values are 1 or 2. Using a higher-order model is strongly discouraged because the stationarity of the AR process cannot be guaranteed.

ar_window_radius
[int, optional] The radius of the window to use for determining the parameters of the autoregressive model.

r_vil_window_radius
[int, optional] The radius of the window to use for determining the R(VIL) relation. Applicable if rainrate is not None.

fft_method
[str, optional] A string defining the FFT method to use (see utils.fft.get_method). Defaults to 'numpy' for compatibility reasons. If pyFFTW is installed, the recommended method is 'pyfftw'.

num_workers
[int, optional] The number of workers to use for parallel computation. Applicable if dask is installed or pyFFTW is used for computing the FFT. When num_workers>1, it is advisable to disable OpenMP by setting the environment variable OMP_NUM_THREADS to 1. This avoids slowdown caused by too many simultaneous threads.

extrap_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the extrapolation method. See the documentation of pysteps.extrapolation.

filter_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the filter method. See the documentation of pysteps.cascade.bandpass_filters.py.

measure_time
[bool, optional] If True, measure, print and return the computation time.

Returns

out
[ndarray] A three-dimensional array of shape (n_timesteps,m,n) containing a time series of forecast precipitation fields. The time series starts from t0+timestep, where timestep is taken from the input VIL/rain rate fields. If measure_time is True, the return value is a three-element tuple containing the nowcast array, the initialization time of the nowcast generator and the time used in the main loop (seconds).

References

[PCvLH20]

pysteps.nowcasts.extrapolation

Implementation of extrapolation-based nowcasting methods.

forecast(precip, velocity, num_timesteps[, ...])	Generate a nowcast by applying a simple advection-based extrapolation to the given precipitation field.

pysteps.nowcasts.extrapolation.forecast

pysteps.nowcasts.extrapolation.**forecast**(*precip*, *velocity*, *num_timesteps*, *extrap_method='semilagrangian'*, *extrap_kwargs=None*, *measure_time=False*)

Generate a nowcast by applying a simple advection-based extrapolation to the given precipitation field.

Parameters

precip
[array-like] Two-dimensional array of shape (m,n) containing the input precipitation field.

velocity
[array-like] Array of shape (2,m,n) containing the x- and y-components of the advection field. The velocities are assumed to represent one time step between the inputs.

num_timesteps
[int] Number of time steps to forecast.

extrap_method
[str, optional] Name of the extrapolation method to use. See the documentation of pysteps.extrapolation.interface.

extrap_kwargs
[dict, , optional] Optional dictionary that is expanded into keyword arguments for the extrapolation method.

measure_time
[bool, optional] If True, measure, print, and return the computation time.

Returns

out
[ndarray] Three-dimensional array of shape (num_timesteps, m, n) containing a time series of nowcast precipitation fields. The time series starts from t0 + timestep, where

timestep is taken from the advection field velocity. If *measure_time* is True, the return value is a two-element tuple containing this array and the computation time (seconds).

See also:

pysteps.extrapolation.interface

pysteps.nowcasts.sprog

Implementation of the S-PROG method described in [See03]

forecast(R, V, n_timesteps[, ...])	Generate a nowcast by using the Spectral Prognosis (S-PROG) method.

pysteps.nowcasts.sprog.forecast

pysteps.nowcasts.sprog.**forecast**(*R*, *V*, *n_timesteps*, *n_cascade_levels=6*, *R_thr=None*, *extrap_method='semilagrangian'*, *decomp_method='fft'*, *bandpass_filter_method='gaussian'*, *ar_order=2*, *conditional=False*, *probmatching_method='mean'*, *num_workers=1*, *fft_method='numpy'*, *domain='spatial'*, *extrap_kwargs=None*, *filter_kwargs=None*, *measure_time=False*)

Generate a nowcast by using the Spectral Prognosis (S-PROG) method.

Parameters

R
[array-like] Array of shape (ar_order+1,m,n) containing the input precipitation fields ordered by timestamp from oldest to newest. The time steps between the inputs are assumed to be regular, and the inputs are required to have finite values.

V
[array-like] Array of shape (2,m,n) containing the x- and y-components of the advection field. The velocities are assumed to represent one time step between the inputs. All values are required to be finite.

n_timesteps
[int] Number of time steps to forecast.

n_cascade_levels
[int, optional] The number of cascade levels to use.

R_thr
[float] The threshold value for minimum observable precipitation intensity.

extrap_method
[str, optional] Name of the extrapolation method to use. See the documentation of pysteps.extrapolation.interface.

decomp_method
[{'fft'}, optional] Name of the cascade decomposition method to use. See the documentation of pysteps.cascade.interface.

bandpass_filter_method
[{'gaussian', 'uniform'}, optional] Name of the bandpass filter method to use with the cascade decomposition. See the documentation of pysteps.cascade.interface.

ar_order
[int, optional] The order of the autoregressive model to use. Must be >= 1.

conditional
[bool, optional] If set to True, compute the statistics of the precipitation field conditionally by excluding pixels where the values are below the threshold R_thr.

probmatching_method
[{'cdf','mean',None}, optional] Method for matching the conditional statistics of the forecast field (areas with precipitation intensity above the threshold R_thr) with those of the most recently observed one. 'cdf'=map the forecast CDF to the observed one, 'mean'=adjust only the mean value, None=no matching applied.

num_workers
[int, optional] The number of workers to use for parallel computation. Applicable if dask is enabled or pyFFTW is used for computing the FFT. When num_workers>1, it is advisable to disable OpenMP by setting the environment variable OMP_NUM_THREADS to 1. This avoids slowdown caused by too many simultaneous threads.

fft_method
[str, optional] A string defining the FFT method to use (see utils.fft.get_method). Defaults to 'numpy' for compatibility reasons. If pyFFTW is installed, the recommended method is 'pyfftw'.

domain
[{"spatial", "spectral"}] If "spatial", all computations are done in the spatial domain (the classical S-PROG model). If "spectral", the AR(2) models are applied directly in the spectral domain to reduce memory footprint and improve performance [PCH18a].

extrap_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the extrapolation method. See the documentation of pysteps.extrapolation.

filter_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the filter method. See the documentation of pysteps.cascade.bandpass_filters.py.

measure_time
[bool] If set to True, measure, print and return the computation time.

Returns

out
[ndarray] A three-dimensional array of shape (n_timesteps,m,n) containing a time series of forecast precipitation fields. The time series starts from t0+timestep, where timestep is taken from the input precipitation fields R. If measure_time is True, the return value is a three-element tuple containing the nowcast array, the initialization time of the nowcast generator and the time used in the main loop (seconds).

See also:

pysteps.extrapolation.interface, *pysteps.cascade.interface*

References

[See03], [PCH18a]

pysteps.nowcasts.sseps

Implementation of the Short-space ensemble prediction system (SSEPS) method. Essentially, SSEPS is a localized version of STEPS.

For localization we intend the use of a subset of the observations in order to estimate model parameters that are distributed in space. The short-space approach used in [NBS+17] is generalized to the whole nowcasting system. This essenially boils down to a moving window localization of the nowcasting procedure, whereby all parameters are estimated over a subdomain of prescribed size.

forecast(R, metadata, V, n_timesteps[, ...])	Generate a nowcast ensemble by using the Short-space ensemble prediction system (SSEPS) method.

pysteps.nowcasts.sseps.forecast

pysteps.nowcasts.sseps.**forecast**(*R*, *metadata*, *V*, *n_timesteps*, *n_ens_members=24*, *n_cascade_levels=6*, *win_size=256*, *overlap=0.1*, *war_thr=0.1*, *extrap_method='semilagrangian'*, *decomp_method='fft'*, *bandpass_filter_method='gaussian'*, *noise_method='ssft'*, *ar_order=2*, *vel_pert_method=None*, *probmatching_method='cdf'*, *mask_method='incremental'*, *callback=None*, *fft_method='numpy'*, *return_output=True*, *seed=None*, *num_workers=1*, *extrap_kwargs=None*, *filter_kwargs=None*, *noise_kwargs=None*, *vel_pert_kwargs=None*, *mask_kwargs=None*, *measure_time=False*)

Generate a nowcast ensemble by using the Short-space ensemble prediction system (SSEPS) method. This is an experimental version of STEPS which allows for localization by means of a window function.

Parameters

R
[array-like] Array of shape (ar_order+1,m,n) containing the input precipitation fields ordered by timestamp from oldest to newest. The time steps between the inputs are assumed to be regular, and the inputs are required to have finite values.

metadata
[dict] Metadata dictionary containing the accutime, xpixelsize, threshold and zerovalue attributes as described in the documentation of *pysteps.io.importers*. xpixelsize is assumed to be in meters.

V
[array-like] Array of shape (2,m,n) containing the x- and y-components of the advection field. The velocities are assumed to represent one time step between the inputs. All values are required to be finite.

win_size
[int or two-element sequence of ints] Size-length of the localization window.

overlap
[float [0,1[] A float between 0 and 1 prescribing the level of overlap between successive windows. If set to 0, no overlap is used.

war_thr
[float] Threshold for the minimum fraction of rain in a given window.

n_timesteps
[int] Number of time steps to forecast.

n_ens_members
[int] The number of ensemble members to generate.

n_cascade_levels
[int] The number of cascade levels to use.

extrap_method
[{'semilagrangian'}] Name of the extrapolation method to use. See the documentation of pysteps.extrapolation.interface.

decomp_method
[{'fft'}] Name of the cascade decomposition method to use. See the documentation of pysteps.cascade.interface.

bandpass_filter_method
[{'gaussian', 'uniform'}] Name of the bandpass filter method to use with the cascade decomposition.

noise_method
[{'parametric','nonparametric','ssft','nested',None}] Name of the noise generator to use for perturbating the precipitation field. See the documentation of pysteps.noise.interface. If set to None, no noise is generated.

ar_order: int
The order of the autoregressive model to use. Must be >= 1.

vel_pert_method: {'bps',None}
Name of the noise generator to use for perturbing the advection field. See the documentation of pysteps.noise.interface. If set to None, the advection field is not perturbed.

mask_method
[{'incremental', None}] The method to use for masking no precipitation areas in the forecast field. The masked pixels are set to the minimum value of the observations. 'incremental' = iteratively buffer the mask with a certain rate (currently it is 1 km/min), None=no masking.

probmatching_method
[{'cdf', None}] Method for matching the statistics of the forecast field with those of the most recently observed one. 'cdf'=map the forecast CDF to the observed one, None=no matching applied. Using 'mean' requires that mask_method is not None.

callback
[function] Optional function that is called after computation of each time step of the nowcast. The function takes one argument: a three-dimensional array of shape (n_ens_members,h,w), where h and w are the height and width of the input field R, respectively. This can be used, for instance, writing the outputs into files.

return_output
[bool] Set to False to disable returning the outputs as numpy arrays. This can save memory if the intermediate results are written to output files using the callback function.

seed
[int] Optional seed number for the random generators.

num_workers
[int] The number of workers to use for parallel computation. Applicable if dask is enabled or pyFFTW is used for computing the FFT. When num_workers>1, it is advisable to disable OpenMP by setting the environment variable OMP_NUM_THREADS to 1. This avoids slowdown caused by too many simultaneous threads.

fft_method
[str] A string defining the FFT method to use (see utils.fft.get_method). Defaults to 'numpy' for compatibility reasons. If pyFFTW is installed, the recommended method is 'pyfftw'.

extrap_kwargs
[dict] Optional dictionary containing keyword arguments for the extrapolation method. See the documentation of pysteps.extrapolation.

filter_kwargs
[dict] Optional dictionary containing keyword arguments for the filter method. See the

documentation of pysteps.cascade.bandpass_filters.py.

noise_kwargs
[dict] Optional dictionary containing keyword arguments for the initializer of the noise generator. See the documentation of pysteps.noise.fftgenerators.

vel_pert_kwargs
[dict] Optional dictionary containing keyword arguments "p_pert_par" and "p_pert_perp" for the initializer of the velocity perturbator. See the documentation of pysteps.noise.motion.

mask_kwargs
[dict] Optional dictionary containing mask keyword arguments 'mask_f' and 'mask_rim', the factor defining the the mask increment and the rim size, respectively. The mask increment is defined as mask_f*timestep/kmperpixel.

measure_time
[bool] If set to True, measure, print and return the computation time.

Returns

out
[ndarray] If return_output is True, a four-dimensional array of shape (n_ens_members,n_timesteps,m,n) containing a time series of forecast precipitation fields for each ensemble member. Otherwise, a None value is returned. The time series starts from t0+timestep, where timestep is taken from the input precipitation fields R.

See also:

pysteps.extrapolation.interface, *pysteps.cascade.interface*

pysteps.noise.interface, *pysteps.noise.utils.compute_noise_stddev_adjs*

Notes

Please be aware that this represents a (very) experimental implementation.

References

[See03], [BPS06], [SPN13], [NBS+17]

pysteps.nowcasts.steps

Implementation of the STEPS stochastic nowcasting method as described in [See03], [BPS06] and [SPN13].

forecast(R, V, n_timesteps[, n_ens_members, ...])	Generate a nowcast ensemble by using the Short-Term Ensemble Prediction System (STEPS) method.

pysteps.nowcasts.steps.forecast

pysteps.nowcasts.steps.**forecast**(*R*, *V*, *n_timesteps*, *n_ens_members=24*, *n_cascade_levels=6*, *R_thr=None*, *kmperpixel=None*, *timestep=None*, *extrap_method='semilagrangian'*, *decomp_method='fft'*, *bandpass_filter_method='gaussian'*, *noise_method='nonparametric'*, *noise_stddev_adj=None*, *ar_order=2*, *vel_pert_method='bps'*, *conditional=False*, *probmatching_method='cdf'*, *mask_method='incremental'*, *callback=None*, *return_output=True*, *seed=None*, *num_workers=1*, *fft_method='numpy'*, *domain='spatial'*, *extrap_kwargs=None*, *filter_kwargs=None*, *noise_kwargs=None*, *vel_pert_kwargs=None*, *mask_kwargs=None*, *measure_time=False*)

Generate a nowcast ensemble by using the Short-Term Ensemble Prediction System (STEPS) method.

Parameters

R
[array-like] Array of shape (ar_order+1,m,n) containing the input precipitation fields ordered by timestamp from oldest to newest. The time steps between the inputs are assumed to be regular, and the inputs are required to have finite values.

V
[array-like] Array of shape (2,m,n) containing the x- and y-components of the advection field. The velocities are assumed to represent one time step between the inputs. All values are required to be finite.

n_timesteps
[int] Number of time steps to forecast.

n_ens_members
[int, optional] The number of ensemble members to generate.

n_cascade_levels
[int, optional] The number of cascade levels to use.

R_thr
[float, optional] Specifies the threshold value for minimum observable precipitation intensity. Required if mask_method is not None or conditional is True.

kmperpixel
[float, optional] Spatial resolution of the input data (kilometers/pixel). Required if vel_pert_method is not None or mask_method is 'incremental'.

timestep
[float, optional] Time step of the motion vectors (minutes). Required if vel_pert_method is not None or mask_method is 'incremental'.

extrap_method
[str, optional] Name of the extrapolation method to use. See the documentation of pysteps.extrapolation.interface.

decomp_method
[{'fft'}, optional] Name of the cascade decomposition method to use. See the documentation of pysteps.cascade.interface.

bandpass_filter_method
[{'gaussian', 'uniform'}, optional] Name of the bandpass filter method to use with the cascade decomposition. See the documentation of pysteps.cascade.interface.

noise_method
[{'parametric','nonparametric','ssft','nested',None}, optional] Name of the noise gen-

erator to use for perturbating the precipitation field. See the documentation of pysteps.noise.interface. If set to None, no noise is generated.

noise_stddev_adj
[{'auto','fixed',None}, optional] Optional adjustment for the standard deviations of the noise fields added to each cascade level. This is done to compensate incorrect std. dev. estimates of casace levels due to presence of no-rain areas. 'auto'=use the method implemented in pysteps.noise.utils.compute_noise_stddev_adjs. 'fixed'= use the formula given in [BPS06] (eq. 6), None=disable noise std. dev adjustment.

ar_order
[int, optional] The order of the autoregressive model to use. Must be >= 1.

vel_pert_method
[{'bps',None}, optional] Name of the noise generator to use for perturbing the advection field. See the documentation of pysteps.noise.interface. If set to None, the advection field is not perturbed.

conditional
[bool, optional] If set to True, compute the statistics of the precipitation field conditionally by excluding pixels where the values are below the threshold R_thr.

mask_method
[{'obs','sprog','incremental',None}, optional] The method to use for masking no precipitation areas in the forecast field. The masked pixels are set to the minimum value of the observations. 'obs' = apply R_thr to the most recently observed precipitation intensity field, 'sprog' = use the smoothed forecast field from S-PROG, where the AR(p) model has been applied, 'incremental' = iteratively buffer the mask with a certain rate (currently it is 1 km/min), None=no masking.

probmatching_method
[{'cdf','mean',None}, optional] Method for matching the statistics of the forecast field with those of the most recently observed one. 'cdf'=map the forecast CDF to the observed one, 'mean'=adjust only the conditional mean value of the forecast field in precipitation areas, None=no matching applied. Using 'mean' requires that mask_method is not None.

callback
[function, optional] Optional function that is called after computation of each time step of the nowcast. The function takes one argument: a three-dimensional array of shape (n_ens_members,h,w), where h and w are the height and width of the input field R, respectively. This can be used, for instance, writing the outputs into files.

return_output
[bool, optional] Set to False to disable returning the outputs as numpy arrays. This can save memory if the intermediate results are written to output files using the callback function.

seed
[int, optional] Optional seed number for the random generators.

num_workers
[int, optional] The number of workers to use for parallel computation. Applicable if dask is enabled or pyFFTW is used for computing the FFT. When num_workers>1, it is advisable to disable OpenMP by setting the environment variable OMP_NUM_THREADS to 1. This avoids slowdown caused by too many simultaneous threads.

fft_method
[str, optional] A string defining the FFT method to use (see utils.fft.get_method). Defaults to 'numpy' for compatibility reasons. If pyFFTW is installed, the recommended method is 'pyfftw'.

domain
[{"spatial", "spectral"}] If "spatial", all computations are done in the spatial domain (the classical STEPS model). If "spectral", the AR(2) models and stochastic perturbations are applied directly in the spectral domain to reduce memory footprint and improve performance [PCH19].

extrap_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the extrapolation method. See the documentation of pysteps.extrapolation.

filter_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the filter method. See the documentation of pysteps.cascade.bandpass_filters.py.

noise_kwargs
[dict, optional] Optional dictionary containing keyword arguments for the initializer of the noise generator. See the documentation of pysteps.noise.fftgenerators.

vel_pert_kwargs
[dict, optional] Optional dictionary containing keyword arguments 'p_par' and 'p_perp' for the initializer of the velocity perturbator. The choice of the optimal parameters depends on the domain and the used optical flow method.

Default parameters from [BPS06]: p_par = [10.88, 0.23, -7.68] p_perp = [5.76, 0.31, -2.72]

Parameters fitted to the data (optical flow/domain):

darts/fmi: p_par = [13.71259667, 0.15658963, -16.24368207] p_perp = [8.26550355, 0.17820458, -9.54107834]

darts/mch: p_par = [24.27562298, 0.11297186, -27.30087471] p_perp = [-7.80797846e+01, -3.38641048e-02, 7.56715304e+01]

darts/fmi+mch: p_par = [16.55447057, 0.14160448, -19.24613059] p_perp = [14.75343395, 0.11785398, -16.26151612]

lucaskanade/fmi: p_par = [2.20837526, 0.33887032, -2.48995355] p_perp = [2.21722634, 0.32359621, -2.57402761]

lucaskanade/mch: p_par = [2.56338484, 0.3330941, -2.99714349] p_perp = [1.31204508, 0.3578426, -1.02499891]

lucaskanade/fmi+mch: p_par = [2.31970635, 0.33734287, -2.64972861] p_perp = [1.90769947, 0.33446594, -2.06603662]

vet/fmi: p_par = [0.25337388, 0.67542291, 11.04895538] p_perp = [0.02432118, 0.99613295, 7.40146505]

vet/mch: p_par = [0.5075159, 0.53895212, 7.90331791] p_perp = [0.68025501, 0.41761289, 4.73793581]

vet/fmi+mch: p_par = [0.29495222, 0.62429207, 8.6804131] p_perp = [0.23127377, 0.59010281, 5.98180004]

fmi=Finland, mch=Switzerland, fmi+mch=both pooled into the same data set

The above parameters have been fitten by using run_vel_pert_analysis.py and fit_vel_pert_params.py located in the scripts directory.

See pysteps.noise.motion for additional documentation.

mask_kwargs
[dict] Optional dictionary containing mask keyword arguments 'mask_f' and 'mask_rim', the factor defining the the mask increment and the rim size, respectively. The mask increment is defined as mask_f*timestep/kmperpixel.

measure_time
[bool] If set to True, measure, print and return the computation time.

Returns

out
[ndarray] If return_output is True, a four-dimensional array of shape (n_ens_members,n_timesteps,m,n) containing a time series of forecast precipitation fields for each ensemble member. Otherwise, a None value is returned. The time series starts from t0+timestep, where timestep is taken from the input precipitation fields R. If measure_time is True, the return value is a three-element tuple containing the nowcast array, the initialization time of the nowcast generator and the time used in the main loop (seconds).

See also:

pysteps.extrapolation.interface, *pysteps.cascade.interface*

pysteps.noise.interface, *pysteps.noise.utils.compute_noise_stddev_adjs*

References

[See03], [BPS06], [SPN13], [PCH19]

pysteps.nowcasts.utils

Module with common utilities used by nowcasts methods.

print_ar_params(PHI)	Print the parameters of an AR(p) model.
print_corrcoefs(GAMMA)	Print the parameters of an AR(p) model.
stack_cascades(R_d, n_levels[, ...])	Stack the given cascades into a larger array.

pysteps.nowcasts.utils.print_ar_params

pysteps.nowcasts.utils.**print_ar_params**(*PHI*)
Print the parameters of an AR(p) model.

Parameters

PHI
[array_like] Array of shape (n, p) containing the AR(p) parameters for n cascade levels.

pysteps.nowcasts.utils.print_corrcoefs

pysteps.nowcasts.utils.**print_corrcoefs**(*GAMMA*)
Print the parameters of an AR(p) model.

Parameters

GAMMA
[array_like] Array of shape (m, n) containing n correlation coefficients for m cascade levels.

pysteps.nowcasts.utils.stack_cascades

pysteps.nowcasts.utils.**stack_cascades**(*R_d*, *n_levels*, *convert_to_full_arrays=False*)
Stack the given cascades into a larger array.

Parameters

R_d
[list] List of cascades obtained by calling a method implemented in pysteps.cascade.decomposition.

n_levels
[int] The number of cascade levels.

Returns

out
[tuple] A list of three-dimensional arrays containing the stacked cascade levels.

2.2.9 pysteps.postprocessing

Methods for post-processing of forecasts.

pysteps.postprocessing.ensemblestats

Methods for the computation of ensemble statistics.

mean(X[, ignore_nan, X_thr])	Compute the mean value from a forecast ensemble field.
excprob(X, X_thr[, ignore_nan])	For a given forecast ensemble field, compute exceedance probabilities for the given intensity thresholds.
banddepth(X[, thr, norm])	Compute the modified band depth (Lopez-Pintado and Romo, 2009) for a k-member ensemble data set.

pysteps.postprocessing.ensemblestats.mean

pysteps.postprocessing.ensemblestats.**mean**(*X*, *ignore_nan=False*, *X_thr=None*)
Compute the mean value from a forecast ensemble field.

Parameters

X
[array_like] Array of shape (k, m, n) containing a k-member ensemble of forecast fields of shape (m, n).

ignore_nan
[bool] If True, ignore nan values.

X_thr
[float] Optional threshold for computing the ensemble mean. Values below **X_thr** are ignored.

Returns

out
[ndarray] Array of shape (m, n) containing the ensemble mean.

pysteps.postprocessing.ensemblestats.excprob

pysteps.postprocessing.ensemblestats.**excprob**(*X*, *X_thr*, *ignore_nan=False*)
For a given forecast ensemble field, compute exceedance probabilities for the given intensity thresholds.

Parameters

X
[array_like] Array of shape (k, m, n, ...) containing an k-member ensemble of forecasts with shape (m, n, ...).

X_thr
[float or a sequence of floats] Intensity threshold(s) for which the exceedance probabilities are computed.

ignore_nan
[bool] If True, ignore nan values.

Returns

out
[ndarray] Array of shape (len(X_thr), m, n) containing the exceedance probabilities for the given intensity thresholds. If len(X_thr)=1, the first dimension is dropped.

pysteps.postprocessing.ensemblestats.banddepth

`pysteps.postprocessing.ensemblestats.`**`banddepth`**(*X*, *thr=None*, *norm=False*)
Compute the modified band depth (Lopez-Pintado and Romo, 2009) for a k-member ensemble data set.

Implementation of the exact fast algorithm for computing the modified band detpth as described in Sun et al (2012).

Parameters

X
[array_like] Array of shape (k, m, ...) representing an ensemble of *k* members (i.e., samples) with shape (m, ...).

thr
[float] Optional threshold for excluding pixels that have no samples equal or above the **thr** value.

Returns

out
[array_like] Array of shape *k* containing the (normalized) band depth values for each ensemble member.

References

Lopez-Pintado, Sara, and Juan Romo. 2009. "On the Concept of Depth for Functional Data." Journal of the American Statistical Association 104 (486): 718–34. https://doi.org/10.1198/jasa.2009.0108.

Sun, Ying, Marc G. Genton, and Douglas W. Nychka. 2012. "Exact Fast Computation of Band Depth for Large Functional Datasets: How Quickly Can One Million Curves Be Ranked?" Stat 1 (1): 68–74. https://doi.org/10.1002/sta4.8.

pysteps.postprocessing.probmatching

Methods for matching the probability distribution of two data sets.

compute_empirical_cdf(bin_edges, hist)	Compute an empirical cumulative distribution function from the given histogram.

Continued on next page

Table 33 – continued from previous page

nonparam_match_empirical_cdf(initial_array, ...)	Matches the empirical CDF of the initial array with the empirical CDF of a target array.
pmm_init(bin_edges_1, cdf_1, bin_edges_2, cdf_2)	Initialize a probability matching method (PMM) object from binned cumulative distribution functions (CDF).
pmm_compute(pmm, x)	For a given PMM object and x-coordinate, compute the probability matched value (i.e.
shift_scale(R, f, rain_fraction_trg, ...)	Find shift and scale that is needed to return the required second_moment and rain area.

pysteps.postprocessing.probmatching.compute_empirical_cdf

pysteps.postprocessing.probmatching.**compute_empirical_cdf**(*bin_edges*, *hist*)

Compute an empirical cumulative distribution function from the given histogram.

Parameters

bin_edges
[array_like] Coordinates of left edges of the histogram bins.

hist
[array_like] Histogram counts for each bin.

Returns

out
[ndarray] CDF values corresponding to the bin edges.

pysteps.postprocessing.probmatching.nonparam_match_empirical_cdf

pysteps.postprocessing.probmatching.**nonparam_match_empirical_cdf**(*initial_array*, *target_array*)

Matches the empirical CDF of the initial array with the empirical CDF of a target array. Initial ranks are conserved, but empirical distribution matches the target one. Zero-pixels (i.e. pixels having the minimum value) in the initial array are conserved.

Parameters

initial_array
[array_like] The initial array whose CDF is to be matched with the target.

target_array
[array_like] The target array.

Returns

output_array
[ndarray] The matched array of the same shape as the initial array.

pysteps.postprocessing.probmatching.pmm_init

pysteps.postprocessing.probmatching.**pmm_init**(*bin_edges_1*, *cdf_1*, *bin_edges_2*, *cdf_2*)

Initialize a probability matching method (PMM) object from binned cumulative distribution functions (CDF).

Parameters

bin_edges_1
[array_like] Coordinates of the left bin edges of the source cdf.

cdf_1
[array_like] Values of the source CDF at the bin edges.

bin_edges_2
[array_like] Coordinates of the left bin edges of the target cdf.

cdf_2
[array_like] Values of the target CDF at the bin edges.

pysteps.postprocessing.probmatching.pmm_compute

pysteps.postprocessing.probmatching.**pmm_compute**(*pmm*, *x*)
For a given PMM object and x-coordinate, compute the probability matched value (i.e. the x-coordinate for which the target CDF has the same value as the source CDF).

Parameters

pmm
[dict] A PMM object returned by pmm_init.

x
[float] The coordinate for which to compute the probability matched value.

pysteps.postprocessing.probmatching.shift_scale

pysteps.postprocessing.probmatching.**shift_scale**(*R*, *f*, *rain_fraction_trg*, *second_moment_trg*, ***kwargs*)
Find shift and scale that is needed to return the required second_moment and rain area. The optimization is performed with the Nelder-Mead algorithm available in scipy. It assumes a forward transformation ln_rain = ln(rain)-ln(min_rain) if rain > min_rain, else 0.

Parameters

R
[array_like] The initial array to be shift and scaled.

f
[function] The inverse transformation that is applied after the shift and scale.

rain_fraction_trg
[float] The required rain fraction to be matched by shifting.

second_moment_trg
[float] The required second moment to be matched by scaling. The second_moment is defined as second_moment = var + mean^2.

Returns

shift
[float] The shift value that produces the required rain fraction.

scale
[float] The scale value that produces the required second_moment.

R
[array_like] The shifted, scaled and back-transformed array.

Other Parameters

scale
[float] Optional initial value of the scale parameter for the Nelder-Mead optimisation.

Typically, this would be the scale parameter estimated the previous time step. Default : 1.

max_iterations
[int] Maximum allowed number of iterations and function evaluations. More details: https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html Deafult: 100.

tol
[float] Tolerance for termination. More details: https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html Default: 0.05*second_moment_trg, i.e. terminate the search if the error is less than 5% since the second moment is a bit unstable.

2.2.10 pysteps.timeseries

Methods and models for time series analysis.

pysteps.timeseries.autoregression

Methods related to autoregressive AR(p) models.

adjust_lag2_corrcoef1(gamma_1, gamma_2)	A simple adjustment of lag-2 temporal autocorrelation coefficient to ensure that the resulting AR(2) process is stationary when the parameters are estimated from the Yule-Walker equations.
adjust_lag2_corrcoef2(gamma_1, gamma_2)	A more advanced adjustment of lag-2 temporal autocorrelation coefficient to ensure that the resulting AR(2) process is stationary when the parameters are estimated from the Yule-Walker equations.
ar_acf(gamma[, n])	Compute theoretical autocorrelation function (ACF) from the AR(p) model with lag-l, l=1,2,. . . ,p temporal autocorrelation coefficients.
estimate_ar_params_ols(x, p[, d, . . .])	Estimate the parameters of an autoregressive AR(p) model
estimate_ar_params_ols_localized(x, p, . . .)	Estimate the parameters of a localized AR(p) model
estimate_ar_params_yw(gamma[, d, . . .])	Estimate the parameters of an AR(p) model
estimate_ar_params_yw_localized(gamma[, d])	Estimate the parameters of a localized AR(p) model
estimate_var_params_ols(x, p[, d, . . .])	Estimate the parameters of a vector autoregressive VAR(p) model
estimate_var_params_ols_localized(x, p, . . .)	Estimate the parameters of a vector autoregressive VAR(p) model
estimate_var_params_yw(gamma[, d, . . .])	Estimate the parameters of a VAR(p) model
iterate_ar_model(x, phi[, eps])	Apply an AR(p) model
iterate_var_model(x, phi[, eps])	Apply a VAR(p) model

pysteps.timeseries.autoregression.adjust_lag2_corrcoef1

pysteps.timeseries.autoregression.**adjust_lag2_corrcoef1**(*gamma_1*, *gamma_2*)
A simple adjustment of lag-2 temporal autocorrelation coefficient to ensure that the resulting AR(2) process is stationary when the parameters are estimated from the Yule-Walker equations.

Parameters

gamma_1

[float] Lag-1 temporal autocorrelation coeffient.

gamma_2
[float] Lag-2 temporal autocorrelation coeffient.

Returns

out
[float] The adjusted lag-2 correlation coefficient.

pysteps.timeseries.autoregression.adjust_lag2_corrcoef2

pysteps.timeseries.autoregression.**adjust_lag2_corrcoef2**(*gamma_1*, *gamma_2*)
A more advanced adjustment of lag-2 temporal autocorrelation coefficient to ensure that the resulting AR(2) process is stationary when the parameters are estimated from the Yule-Walker equations.

Parameters

gamma_1
[float] Lag-1 temporal autocorrelation coeffient.

gamma_2
[float] Lag-2 temporal autocorrelation coeffient.

Returns

out
[float] The adjusted lag-2 correlation coefficient.

pysteps.timeseries.autoregression.ar_acf

pysteps.timeseries.autoregression.**ar_acf**(*gamma*, *n=None*)
Compute theoretical autocorrelation function (ACF) from the AR(p) model with lag-l, l=1,2,...,p temporal autocorrelation coefficients.

Parameters

gamma
[array-like] Array of length p containing the lag-l, l=1,2,...p, temporal autocorrelation coefficients. The correlation coefficients are assumed to be in ascending order with respect to time lag.

n
[int] Desired length of ACF array. Must be greater than len(gamma).

Returns

out
[array-like] Array containing the ACF values.

pysteps.timeseries.autoregression.estimate_ar_params_ols

pysteps.timeseries.autoregression.**estimate_ar_params_ols**(*x*, *p*, *d=0*, *check_stationarity=True*, *include_constant_term=False*, *h=0*, *lam=0.0*)
Estimate the parameters of an autoregressive AR(p) model

$x_{k+1} = c + \phi_1 x_k + \phi_2 x_{k-1} + \cdots + \phi_p x_{k-p} + \phi_{p+1}\epsilon$

by using ordinary least squares (OLS). If $d \geq 1$, the parameters are estimated for a d times differenced time series that is integrated back to the original one by summation of the differences.

Parameters

x
[array_like] Array of shape (n,...) containing a time series of length n=p+d+h+1. The remaining dimensions are flattened. The rows and columns of x represent time steps and samples, respectively.

p
[int] The order of the model.

d
[{0,1}] The order of differencing to apply to the time series.

check_stationarity
[bool] Check the stationarity of the estimated model.

include_constant_term
[bool] Include the constant term c to the model.

h
[int] If h>0, the fitting is done by using a history of length h in addition to the minimal required number of time steps n=p+d+1.

lam
[float] If lam>0, the regression is regularized by adding a penalty term (i.e. ridge regression).

Returns

out
[list] The estimated parameter matrices $\mathbf{\Phi}_1, \mathbf{\Phi}_2, \ldots, \mathbf{\Phi}_{p+1}$. If include_constant_term is True, the constant term c is added to the beginning of the list.

Notes

Estimation of the innovation term parameter ϕ_{p+1} is currently implemented for p<=2. If p > 2, ϕ_{p+1} is set to zero.

pysteps.timeseries.autoregression.estimate_ar_params_ols_localized

pysteps.timeseries.autoregression.**estimate_ar_params_ols_localized**(*x*, *p*, *window_radius*, *d=0*, *include_constant_term=False*, *h=0*, *lam=0.0*, *window='gaussian'*)

Estimate the parameters of a localized AR(p) model

$x_{k+1,i} = c_i + \phi_{1,i}x_{k,i} + \phi_{2,i}x_{k-1,i} + \cdots + \phi_{p,i}x_{k-p,i} + \phi_{p+1,i}\epsilon$

by using ordinary least squares (OLS), where i denote spatial coordinates with arbitrary dimension. If $d \geq 1$, the parameters are estimated for a d times differenced time series that is integrated back to the original one by summation of the differences.

Parameters

x
[array_like] Array of shape (n,...) containing a time series of length n=p+d+h+1. The

remaining dimensions are flattened. The rows and columns of x represent time steps and samples, respectively.

p
[int] The order of the model.

window_radius
[float] Radius of the moving window. If window is 'gaussian', window_radius is the standard deviation of the Gaussian filter. If window is 'uniform', the size of the window is 2*window_radius+1.

d
[{0,1}] The order of differencing to apply to the time series.

include_constant_term
[bool] Include the constant term c_i to the model.

h
[int] If h>0, the fitting is done by using a history of length h in addition to the minimal required number of time steps n=p+d+1.

lam
[float] If lam>0, the regression is regularized by adding a penalty term (i.e. ridge regression).

window
[{"gaussian", "uniform"}] The weight function to use for the moving window. Applicable if window_radius < np.inf. Defaults to 'gaussian'.

Returns

out
[list] List of length p+1 containing the AR(p) parameter fields for for the lag-p terms and the innovation term. The parameter fields have the same shape as the elements of gamma. Nan values are assigned, where the sample size for estimating the parameters is too small. If include_constant_term is True, the constant term c_i is added to the beginning of the list.

Notes

Estimation of the innovation term parameter ϕ_{p+1} is currently implemented for p<=2. If p > 2, ϕ_{p+1} is set to a zero array.

pysteps.timeseries.autoregression.estimate_ar_params_yw

pysteps.timeseries.autoregression.**estimate_ar_params_yw**(*gamma*, *d=0*, *check_stationarity=True*)

Estimate the parameters of an AR(p) model

$x_{k+1} = \phi_1 x_k + \phi_2 x_{k-1} + \cdots + \phi_p x_{k-p} + \phi_{p+1}\epsilon$

from the Yule-Walker equations using the given set of autocorrelation coefficients.

Parameters

gamma
[array_like] Array of length p containing the lag-l temporal autocorrelation coefficients for l=1,2,... p. The correlation coefficients are assumed to be in ascending order with respect to time lag.

d
[{0,1}] The order of differencing. If d=1, the correlation coefficients gamma are as-

sumed to be computed from the differenced time series, which is also done for the resulting parameter estimates.

check_stationarity
[bool] If True, the stationarity of the resulting VAR(p) process is tested. An exception is thrown if the process is not stationary.

Returns

out
[ndarray] Array of length p+1 containing the AR(p) parameters for for the lag-p terms and the innovation term.

Notes

To estimate the parameters of an integrated ARI(p,d) model, compute the correlation coefficients gamma by calling *pysteps.timeseries.correlation.temporal_autocorrelation()* with d>0.

pysteps.timeseries.autoregression.estimate_ar_params_yw_localized

pysteps.timeseries.autoregression.**estimate_ar_params_yw_localized**(*gamma*, *d=0*)

Estimate the parameters of a localized AR(p) model

$x_{k+1,i} = \phi_{1,i}x_{k,i} + \phi_{2,i}x_{k-1,i} + \cdots + \phi_{p,i}x_{k-p,i} + \phi_{p+1}\epsilon$

from the Yule-Walker equations using the given set of autocorrelation coefficients :math‘gamma_{l,i}‘, where :math‘l‘ denotes time lag and i denote spatial coordinates with arbitrary dimension.

Parameters

gamma
[array_like] A list containing the lag-l temporal autocorrelation coefficient fields for l=1,2,...p. The correlation coefficients are assumed to be in ascending order with respect to time lag.

d
[{0,1}] The order of differencing. If d=1, the correlation coefficients gamma are assumed to be computed from the differenced time series, which is also done for the resulting parameter estimates.

Returns

out
[list] List of length p+1 containing the AR(p) parameter fields for for the lag-p terms and the innovation term. The parameter fields have the same shape as the elements of gamma.

Notes

To estimate the parameters of an integrated ARI(p,d) model, compute the correlation coefficients gamma by calling *pysteps.timeseries.correlation.temporal_autocorrelation()* with d>0 and window_radius<np.inf.

pysteps.timeseries.autoregression.estimate_var_params_ols

pysteps.timeseries.autoregression.**estimate_var_params_ols**(*x*, *p*, *d=0*, *check_stationarity=True*, *include_constant_term=False*, *h=0*, *lam=0.0*)

Estimate the parameters of a vector autoregressive VAR(p) model

$$\mathbf{x}_{k+1} = \mathbf{c} + \mathbf{\Phi}_1\mathbf{x}_k + \mathbf{\Phi}_2\mathbf{x}_{k-1} + \cdots + \mathbf{\Phi}_p\mathbf{x}_{k-p} + \mathbf{\Phi}_{p+1}\epsilon$$

by using ordinary least squares (OLS). If $d \geq 1$, the parameters are estimated for a d times differenced time series that is integrated back to the original one by summation of the differences.

Parameters

x
[array_like] Array of shape (n, q, :) containing a time series of length n=p+d+h+1 with q-dimensional variables. The remaining dimensions are flattened. The remaining dimensions starting from the third one represent the samples.

p
[int] The order of the model.

d
[{0,1}] The order of differencing to apply to the time series.

check_stationarity
[bool] If True, the stationarity of the resulting VAR(p) process is tested. An exception is thrown if the process is not stationary.

include_constant_term
[bool] Include the constant term **c** to the model.

h
[int] If h>0, the fitting is done by using a history of length h in addition to the minimal required number of time steps n=p+d+1.

lam
[float] If lam>0, the regression is regularized by adding a penalty term (i.e. ridge regression).

Returns

out
[list] The estimated parameter matrices $\mathbf{\Phi}_1, \mathbf{\Phi}_2, \ldots, \mathbf{\Phi}_{p+1}$. If include_constant_term is True, the constant term **c** is added to the beginning of the list.

Notes

Estimation of the innovation parameter $\mathbf{\Phi}_{p+1}$ is not currently implemented, and it is set to a zero matrix.

pysteps.timeseries.autoregression.estimate_var_params_ols_localized

pysteps.timeseries.autoregression.**estimate_var_params_ols_localized**(*x*, *p*, *window_radius*, *d=0*, *include_constant_term=False*, *h=0*, *lam=0.0*, *window='gaussian'*)

Estimate the parameters of a vector autoregressive VAR(p) model

$$\mathbf{x}_{k+1,i} = \mathbf{c}_i + \mathbf{\Phi}_{1,i}\mathbf{x}_{k,i} + \mathbf{\Phi}_{2,i}\mathbf{x}_{k-1,i} + \cdots + \mathbf{\Phi}_{p,i}\mathbf{x}_{k-p,i} + \mathbf{\Phi}_{p+1,i}\epsilon$$

by using ordinary least squares (OLS), where i denote spatial coordinates with arbitrary dimension. If $d \geq 1$, the parameters are estimated for a d times differenced time series that is integrated back to the original one by summation of the differences.

Parameters

x
[array_like] Array of shape (n, q, :) containing a time series of length n=p+d+h+1 with q-dimensional variables. The remaining dimensions are flattened. The remaining dimensions starting from the third one represent the samples.

p
[int] The order of the model.

window_radius
[float] Radius of the moving window. If window is 'gaussian', window_radius is the standard deviation of the Gaussian filter. If window is 'uniform', the size of the window is 2*window_radius+1.

d
[{0,1}] The order of differencing to apply to the time series.

include_constant_term
[bool] Include the constant term **c** to the model.

h
[int] If h>0, the fitting is done by using a history of length h in addition to the minimal required number of time steps n=p+d+1.

lam
[float] If lam>0, the regression is regularized by adding a penalty term (i.e. ridge regression).

window
[{"gaussian", "uniform"}] The weight function to use for the moving window. Applicable if window_radius < np.inf. Defaults to 'gaussian'.

Returns

out
[list] The estimated parameter matrices $\mathbf{\Phi}_{1,i}, \mathbf{\Phi}_{2,i}, \ldots, \mathbf{\Phi}_{p+1,i}$. If include_constant_term is True, the constant term $\mathbf{c}_i$ is added to the beginning of the list. Each element of the list is a matrix of shape (x.shape[2:], q, q).

Notes

Estimation of the innovation parameter $\mathbf{\Phi}_{p+1}$ is not currently implemented, and it is set to a zero matrix.

pysteps.timeseries.autoregression.estimate_var_params_yw

pysteps.timeseries.autoregression.**estimate_var_params_yw**(*gamma*, *d=0*, *check_stationarity=True*)

Estimate the parameters of a VAR(p) model

$$\mathbf{x}_{k+1} = \mathbf{\Phi}_1\mathbf{x}_k + \mathbf{\Phi}_2\mathbf{x}_{k-1} + \cdots + \mathbf{\Phi}_p\mathbf{x}_{k-p} + \mathbf{\Phi}_{p+1}\epsilon$$

from the Yule-Walker equations using the given correlation matrices $\mathbf{\Gamma}_0, \mathbf{\Gamma}_1, \ldots, \mathbf{\Gamma}_n$, where n=p.

Parameters

gamma
[list] List of correlation matrices $\mathbf{\Gamma}_0, \mathbf{\Gamma}_1, \ldots, \mathbf{\Gamma}_n$. To obtain these matrices, use *pysteps.timeseries.correlation.temporal_autocorrelation_multivariate()* with window_radius=np.inf.

d
[{0,1}] The order of differencing. If d=1, the correlation coefficients gamma are assumed to be computed from the differenced time series, which is also done for the resulting parameter estimates.

check_stationarity
[bool] If True, the stationarity of the resulting VAR(p) process is tested. An exception is thrown if the process is not stationary.

Returns

out
[list] List of VAR(p) coefficient matrices $\mathbf{\Phi}_1, \mathbf{\Phi}_2, \ldots \mathbf{\Phi}_{p+1}$, where the last matrix corresponds to the innovation term.

Notes

To estimate the parameters of an integrated VARI(p,d) model, compute the correlation coefficients gamma by calling *pysteps.timeseries.correlation.temporal_autocorrelation_multivariate()* with d>0. Estimation of the innovation parameter $\mathbf{\Phi}_{p+1}$ is not currently implemented, and it is set to a zero matrix.

pysteps.timeseries.autoregression.iterate_ar_model

pysteps.timeseries.autoregression.**iterate_ar_model**(*x*, *phi*, *eps=None*)

Apply an AR(p) model

$$x_{k+1} = \phi_1 x_k + \phi_2 x_{k-1} + \cdots + \phi_p x_{k-p} + \phi_{p+1}\epsilon$$

to a time series x_k.

Parameters

x
[array_like] Array of shape (n,...), n>=p, containing a time series of a input variable x. The elements of x along the first dimension are assumed to be in ascending order by time, and the time intervals are assumed to be regular.

phi
[list] List or array of length p+1 specifying the parameters of the AR(p) model. The parameters are in ascending order by increasing time lag, and the last element is the parameter corresponding to the innovation term eps.

eps

[array_like] Optional innovation term for the AR(p) process. The shape of eps is expected to be a scalar or x.shape[1:] if len(x.shape)>1. If eps is None, the innovation term is not added.

pysteps.timeseries.autoregression.iterate_var_model

pysteps.timeseries.autoregression.**iterate_var_model**(*x*, *phi*, *eps=None*)

Apply a VAR(p) model

$$\mathbf{x}_{k+1} = \mathbf{\Phi}_1 \mathbf{x}_k + \mathbf{\Phi}_2 \mathbf{x}_{k-1} + \cdots + \mathbf{\Phi}_p \mathbf{x}_{k-p} + \mathbf{\Phi}_{p+1} \epsilon$$

to a q-variate time series $\mathbf{x}_k$.

Parameters

x

[array_like] Array of shape (n,q,...), n>=p, containing a q-variate time series of a input variable x. The elements of x along the first dimension are assumed to be in ascending order by time, and the time intervals are assumed to be regular.

phi

[list] List of parameter matrices $\mathbf{\Phi}_1, \mathbf{\Phi}_2, \ldots, \mathbf{\Phi}_{p+1}$.

eps

[array_like] Optional innovation term for the AR(p) process. The shape of eps is expected to be (x.shape[1],) or (x.shape[1],x.shape[2:]) if len(x.shape)>2. If eps is None, the innovation term is not added.

pysteps.timeseries.correlation

Methods for computing spatial and temporal correlation of time series of two-dimensional fields.

temporal_autocorrelation(x[, d, domain, ...])	Compute lag-l temporal autocorrelation coefficients $\gamma_l = \mathrm{corr}(x(t), x(t-l))$, $l = 1, 2, \ldots, n-1$, from a time series $x_1, x_2, \ldots, x_n$.
temporal_autocorrelation_multivariate(x[, ...])	For a q-variate time series $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, compute the lag-l correlation matrices $\mathbf{\Gamma}_l$, where $\Gamma_{l,i,j} = \gamma_{l,i,j}$ and $\gamma_{l,i,j} = \mathrm{corr}(x_i(t), x_j(t-l))$ for $i, j = 1, 2, \ldots, q$ and $l = 0, 1, \ldots, n-1$.

pysteps.timeseries.correlation.temporal_autocorrelation

pysteps.timeseries.correlation.**temporal_autocorrelation**(*x*, *d=0*, *domain='spatial'*, *x_shape=None*, *mask=None*, *use_full_fft=False*, *window='gaussian'*, *window_radius=inf*)

Compute lag-l temporal autocorrelation coefficients $\gamma_l = \mathrm{corr}(x(t), x(t-l))$, $l = 1, 2, \ldots, n-1$, from a time series $x_1, x_2, \ldots, x_n$. If a multivariate time series is given, each element of x_i is treated as one sample from the process generating the time series. Use *temporal_autocorrelation_multivariate()* if cross-correlations between different elements of the time series are desired.

Parameters

x

[array_like] Array of shape (n, ...), where each row contains one sample from the time series x_i. The inputs are assumed to be in increasing order with respect to time, and the

time step is assumed to be regular. All inputs are required to have finite values. The remaining dimensions after the first one are flattened before computing the correlation coefficients.

d
[{0,1}] The order of differencing. If d=1, the input time series is differenced before computing the correlation coefficients. In this case, a time series of length n+1 is needed for computing the n-1 coefficients.

domain
[{"spatial", "spectral"}] The domain of the time series x. If domain is "spectral", the elements of x are assumed to represent the FFTs of the original elements.

x_shape
[tuple] The shape of the original arrays in the spatial domain before applying the FFT. Required if domain is "spectral".

mask
[array_like] Optional mask to use for computing the correlation coefficients. Input elements with mask==False are excluded from the computations. The shape of the mask is expected to be x.shape[1:]. Applicable if domain is "spatial".

use_full_fft
[bool] If True, x represents the full FFTs of the original arrays. Otherwise, the elements of x are assumed to contain only the symmetric part, i.e. in the format returned by numpy.fft.rfft2. Applicable if domain is 'spectral'. Defaults to False.

window
[{"gaussian", "uniform"}] The weight function to use for the moving window. Applicable if window_radius < np.inf. Defaults to 'gaussian'.

window_radius
[float] If window_radius < np.inf, the correlation coefficients are computed in a moving window. Defaults to np.inf (i.e. the coefficients are computed over the whole domain). If window is 'gaussian', window_radius is the standard deviation of the Gaussian filter. If window is 'uniform', the size of the window is 2*window_radius+1.

Returns

out
[list] List of length n-1 containing the temporal autocorrelation coefficients γ_i for time lags $l = 1, 2, ..., n-1$. If window_radius<np.inf, the elements of the list are arrays of shape x.shape[1:]. In this case, nan values are assigned, when the sample size for computing the correlation coefficients is too small.

Notes

Computation of correlation coefficients in the spectral domain is currently implemented only for two-dimensional fields.

pysteps.timeseries.correlation.temporal_autocorrelation_multivariate

pysteps.timeseries.correlation.**temporal_autocorrelation_multivariate**(*x*, *d=0*, *mask=None*, *window='gaussian'*, *window_radius=inf*)

For a q-variate time series $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, compute the lag-l correlation matrices $\mathbf{\Gamma}_l$, where $\Gamma_{l,i,j} = \gamma_{l,i,j}$ and $\gamma_{l,i,j} = \text{corr}(x_i(t), x_j(t-l))$ for $i, j = 1, 2, \ldots, q$ and $l = 0, 1, \ldots, n-1$.

Parameters

x
[array_like] Array of shape (n, q, ...) containing the time series $\mathbf{x}_i$. The inputs are assumed to be in increasing order with respect to time, and the time step is assumed to be regular. All inputs are required to have finite values. The remaining dimensions after the second one are flattened before computing the correlation coefficients.

d
[{0,1}] The order of differencing. If d=1, the input time series is differenced before computing the correlation coefficients. In this case, a time series of length n+1 is needed for computing the n-1 coefficients.

mask
[array_like] Optional mask to use for computing the correlation coefficients. Input elements with mask==False are excluded from the computations. The shape of the mask is expected to be x.shape[2:].

window
[{"gaussian", "uniform"}] The weight function to use for the moving window. Applicable if window_radius < np.inf. Defaults to 'gaussian'.

window_radius
[float] If window_radius < np.inf, the correlation coefficients are computed in a moving window. Defaults to np.inf (i.e. the correlations are computed over the whole domain). If window is 'gaussian', window_radius is the standard deviation of the Gaussian filter. If window is 'uniform', the size of the window is 2*window_radius+1.

Returns

out
[list] List of correlation matrices $\Gamma_0, \Gamma_1, \ldots, \Gamma_{n-1}$ of shape (q,q). If window_radius<np.inf, the elements of the list are arrays of shape (x.shape[2:],q,q). In this case, nan values are assigned, when the sample size for computing the correlation coefficients is too small.

References

[CP02]

2.2.11 pysteps.utils

Implementation of miscellaneous utility functions.

pysteps.utils.interface

Interface for the utils module.

get_method(name, **kwargs)	Return a callable function for the utility method corresponding to the given name.

pysteps.utils.interface.get_method

pysteps.utils.interface.**get_method**(*name*, ***kwargs*)
Return a callable function for the utility method corresponding to the given name.

Arrays methods:

Name	Description
centred_coord	compute a 2D coordinate array

Cleansing methods:

Name	Description
decluster	decluster a set of sparse data points
detect_outliers	detect outliers in a dataset

Conversion methods:

Name	Description
mm/h or rainrate	convert to rain rate [mm/h]
mm or raindepth	convert to rain depth [mm]
dbz or reflectivity	convert to reflectivity [dBZ]

Dimension methods:

Name	Description
accumulate	aggregate fields in time
clip	resize the field domain by geographical coordinates
square	either pad or crop the data to get a square domain
upscale	upscale the field

FFT methods (wrappers to different implementations):

Name	Description
numpy	numpy.fft
scipy	scipy.fftpack
pyfftw	pyfftw.interfaces.numpy_fft

Image processing methods:

Name	Description
blob	blob detection in scale space
ShiTomasi	Shi-Tomasi corner detection
morph_opening	filter small scale noise

Interpolation methods:

Name	Description
rbfinterp2d	fast kernel interpolation of a (multivariate) array over a 2D grid using a radial basis function

Additional keyword arguments are passed to the initializer of the FFT methods, see utils.fft.

Spectral methods:

Name	Description
rapsd	Compute radially averaged power spectral density
rm_rdisc	remove the rain / no-rain discontinuity

Tapering methods:

Name	Description
compute_mask_window_function	Compute window function for a two-dimensional area defined by a non-rectangular mask.
compute_window_function	Compute window function for a two-dimensional rectangular area.

Transformation methods:

Name	Description
boxcox or box-cox	one-parameter Box-Cox transform
db or decibel	transform to units of decibel
log	log transform
nqt	Normal Quantile Transform
sqrt	square-root transform

pysteps.utils.arrays

Utility methods for creating and processing arrays.

compute_centred_coord_array(M, N)	Compute a 2D coordinate array, where the origin is at the center.

pysteps.utils.arrays.compute_centred_coord_array

pysteps.utils.arrays.**compute_centred_coord_array**(*M*, *N*)

Compute a 2D coordinate array, where the origin is at the center.

Parameters

M
[int] The height of the array.

N
[int] The width of the array.

Returns

out
[ndarray] The coordinate array.

Examples

```
>>> compute_centred_coord_array(2, 2)
```

(array([[-2],

[-1],

[0],

[1],

[2]]), array([[-2, -1, 0, 1, 2]]))

pysteps.utils.cleansing

Data cleansing routines for pysteps.

decluster(coord, input_array, scale[, ...])	Decluster a set of sparse data points by aggregating, that is, taking the median value of all values lying within a certain distance (i.e., a cluster).
detect_outliers(input_array, thr[, coord, ...])	Detect outliers in a (multivariate and georeferenced) dataset.

pysteps.utils.cleansing.decluster

pysteps.utils.cleansing.**decluster**(*coord*, *input_array*, *scale*, *min_samples=1*, *verbose=False*)

Decluster a set of sparse data points by aggregating, that is, taking the median value of all values lying within a certain distance (i.e., a cluster).

Parameters

coord
[array_like] Array of shape (n, d) containing the coordinates of the input data into a space of *d* dimensions.

input_array
[array_like] Array of shape (n) or (n, m), where *n* is the number of samples and *m* the number of variables. All values in `input_array` are required to have finite values.

scale
[float or array_like] The `scale` parameter in the same units of `coord`. It can be a scalar or an array_like of shape (d). Data points within the declustering `scale` are aggregated.

min_samples
[int, optional] The minimum number of samples for computing the median within a given cluster.

verbose
[bool, optional] Print out information.

Returns

out
[tuple of ndarrays] A two-element tuple (`out_coord`, `output_array`) containing the declustered coordinates (l, d) and input array (l, m), where *l* is the new number of samples with *l* <= *n*.

pysteps.utils.cleansing.detect_outliers

pysteps.utils.cleansing.**detect_outliers**(*input_array*, *thr*, *coord=None*, *k=None*, *verbose=False*)

Detect outliers in a (multivariate and georeferenced) dataset.

Assume a (multivariate) Gaussian distribution and detect outliers based on the number of standard deviations from the mean.

If spatial information is provided through coordinates, the outlier detection can be localized by considering only the k-nearest neighbours when computing the local mean and standard deviation.

Parameters

input_array
[array_like] Array of shape (n) or (n, m), where *n* is the number of samples and *m* the number of variables. If *m* > 1, the Mahalanobis distance is used. All values in `input_array` are required to have finite values.

thr
[float] The number of standard deviations from the mean used to define an outlier.

coord
[array_like, optional] Array of shape (n, d) containing the coordinates of the input data into a space of *d* dimensions. Passing `coord` requires that `k` is not None.

k
[int or None, optional] The number of nearest neighbours used to localize the outlier detection. If set to None (the default), it employs all the data points (global detection). Setting `k` requires that `coord` is not None.

verbose
[bool, optional] Print out information.

Returns

out
[array_like] A 1-D boolean array of shape (n) with True values indicating the outliers detected in `input_array`.

pysteps.utils.conversion

Methods for converting physical units.

to_rainrate(R, metadata[, zr_a, zr_b])	Convert to rain rate [mm/h].
to_raindepth(R, metadata[, zr_a, zr_b])	Convert to rain depth [mm].
to_reflectivity(R, metadata[, zr_a, zr_b])	Convert to reflectivity [dBZ].

pysteps.utils.conversion.to_rainrate

pysteps.utils.conversion.**to_rainrate**(*R*, *metadata*, *zr_a=None*, *zr_b=None*)
Convert to rain rate [mm/h].

Parameters

R
[array-like] Array of any shape to be (back-)transformed.

metadata
[dict] Metadata dictionary containing the accutime, transform, unit, threshold and zerovalue attributes as described in the documentation of *pysteps.io.importers*.

Additionally, in case of conversion to/from reflectivity units, the zr_a and zr_b attributes are also required, but only if zr_a = zr_b = None. If missing, it defaults to Marshall–Palmer relation, that is, zr_a = 200.0 and zr_b = 1.6.

zr_a, zr_b
[float, optional] The a and b coefficients of the Z-R relationship (Z = a*R^b).

Returns

R
[array-like] Array of any shape containing the converted units.

metadata
[dict] The metadata with updated attributes.

pysteps.utils.conversion.to_raindepth

pysteps.utils.conversion.**to_raindepth**(*R*, *metadata*, *zr_a=None*, *zr_b=None*)
Convert to rain depth [mm].

Parameters

R
[array-like] Array of any shape to be (back-)transformed.

metadata
[dict] Metadata dictionary containing the accutime, transform, unit, threshold and zerovalue attributes as described in the documentation of *pysteps.io.importers*.

Additionally, in case of conversion to/from reflectivity units, the zr_a and zr_b attributes are also required, but only if zr_a = zr_b = None. If missing, it defaults to Marshall–Palmer relation, that is, zr_a = 200.0 and zr_b = 1.6.

zr_a, zr_b
[float, optional] The a and b coefficients of the Z-R relationship (Z = a*R^b).

Returns

R
[array-like] Array of any shape containing the converted units.

metadata
[dict] The metadata with updated attributes.

pysteps.utils.conversion.to_reflectivity

pysteps.utils.conversion.**to_reflectivity**(*R*, *metadata*, *zr_a=None*, *zr_b=None*)
Convert to reflectivity [dBZ].

Parameters

R
[array-like] Array of any shape to be (back-)transformed.

metadata
[dict] Metadata dictionary containing the accutime, transform, unit, threshold and zerovalue attributes as described in the documentation of *pysteps.io.importers*.

Additionally, in case of conversion to/from reflectivity units, the zr_a and zr_b attributes are also required, but only if zr_a = zr_b = None. If missing, it defaults to Marshall–Palmer relation, that is, zr_a = 200.0 and zr_b = 1.6.

zr_a, zr_b
[float, optional] The a and b coefficients of the Z-R relationship (Z = a*R^b).

Returns

R
[array-like] Array of any shape containing the converted units.

metadata
[dict] The metadata with updated attributes.

pysteps.utils.dimension

Functions to manipulate array dimensions.

aggregate_fields(data, window_size[, axis, ...])	Aggregate fields along a given direction.
aggregate_fields_time(R, metadata, ...[, ...])	Aggregate fields in time.
aggregate_fields_space(R, metadata, space_window)	Upscale fields in space.
clip_domain(R, metadata[, extent])	Clip the field domain by geographical coordinates.
square_domain(R, metadata[, method, inverse])	Either pad or crop a field to obtain a square domain.

pysteps.utils.dimension.aggregate_fields

pysteps.utils.dimension.**aggregate_fields**(*data*, *window_size*, *axis=0*, *method='mean'*, *trim=False*)

Aggregate fields along a given direction.

It attempts to aggregate the given R axis in an integer number of sections of length = `window_size`. If such a aggregation is not possible, an error is raised unless `trim` set to True, in which case the axis is trimmed (from the end) to make it perfectly divisible".

Parameters

data
[array-like] Array of any shape containing the input fields.

window_size
[int or tuple of ints] The length of the window that is used to aggregate the fields. If a single integer value is given, the same window is used for all the selected axis.

If `window_size` is a 1D array-like, each element indicates the length of the window that is used to aggregate the fields along each axis. In this case, the number of elements of 'window_size' must be the same as the elements in the `axis` argument.

axis
[int or array-like of ints] Axis or axes where to perform the aggregation. If this is a tuple of ints, the aggregation is performed over multiple axes, instead of a single axis

method
[string, optional] Optional argument that specifies the operation to use to aggregate the values within the window. Default to mean operator.

trim
[bool] In case that the `data` is not perfectly divisible by `window_size` along the selected axis:

- trim=True: the data will be trimmed (from the end) along that axis to make it perfectly divisible.
- trim=False: a ValueError exception is raised.

Returns

new_array
[array-like] The new aggregated array with shape[axis] = k, where k = R.shape[axis] / window_size.

See also:

pysteps.utils.dimension.aggregate_fields_time

pysteps.utils.dimension.aggregate_fields_space

pysteps.utils.dimension.aggregate_fields_time

pysteps.utils.dimension.**aggregate_fields_time**(*R*, *metadata*, *time_window_min*, *ignore_nan=False*)

Aggregate fields in time.

Parameters

R
[array-like] Array of shape (t,m,n) or (l,t,m,n) containing a time series of (ensemble) input fields. They must be evenly spaced in time.

metadata
[dict] Metadata dictionary containing the timestamps and unit attributes as described in the documentation of *pysteps.io.importers*.

time_window_min
[float or None] The length in minutes of the time window that is used to aggregate the fields. The time spanned by the t dimension of R must be a multiple of time_window_min. If set to None, it returns a copy of the original R and metadata.

ignore_nan
[bool, optional] If True, ignore nan values.

Returns

outputarray
[array-like] The new array of aggregated fields of shape (k,m,n) or (l,k,m,n), where k = t*delta/time_window_min and delta is the time interval between two successive timestamps.

metadata
[dict] The metadata with updated attributes.

See also:

pysteps.utils.dimension.aggregate_fields_space

pysteps.utils.dimension.aggregate_fields

pysteps.utils.dimension.aggregate_fields_space

pysteps.utils.dimension.**aggregate_fields_space**(*R*, *metadata*, *space_window*, *ignore_nan=False*)

Upscale fields in space.

Parameters

R
[array-like] Array of shape (m,n), (t,m,n) or (l,t,m,n) containing a single field or a time series of (ensemble) input fields.

metadata
[dict] Metadata dictionary containing the xpixelsize, ypixelsize and unit attributes as described in the documentation of *pysteps.io.importers*.

space_window
[float or None] The length of the space window that is used to upscale the fields. The space_window unit is the same used in the geographical projection of R and hence the same as for the xpixelsize and ypixelsize attributes. The space spanned by the m and n dimensions of R must be a multiple of space_window. If set to None, it returns a copy of the original R and metadata.

ignore_nan
[bool, optional] If True, ignore nan values.

Returns

outputarray
[array-like] The new array of aggregated fields of shape (k,j), (t,k,j) or (l,t,k,j), where k = m*ypixelsize/space_window and j = n*xpixelsize/space_window.

metadata
[dict] The metadata with updated attributes.

See also:

pysteps.utils.dimension.aggregate_fields_time

pysteps.utils.dimension.aggregate_fields

pysteps.utils.dimension.clip_domain

pysteps.utils.dimension.**clip_domain**(*R*, *metadata*, *extent=None*)
Clip the field domain by geographical coordinates.

Parameters

R
[array-like] Array of shape (m,n) or (t,m,n) containing the input fields.

metadata
[dict] Metadata dictionary containing the x1, x2, y1, y2, xpixelsize, ypixelsize, zerovalue and yorigin attributes as described in the documentation of *pysteps.io.importers*.

extent
[scalars (left, right, bottom, top), optional] The extent of the bounding box in data coordinates to be used to clip the data. Note that the direction of the vertical axis and thus the default values for top and bottom depend on origin. We follow the same convention as in the imshow method of matplotlib: https://matplotlib.org/tutorials/intermediate/imshow_extent.html

Returns

R
[array-like] the clipped array

metadata
[dict] the metadata with updated attributes.

pysteps.utils.dimension.square_domain

pysteps.utils.dimension.**square_domain**(*R*, *metadata*, *method='pad'*, *inverse=False*)
Either pad or crop a field to obtain a square domain.

Parameters

R
[array-like] Array of shape (m,n) or (t,m,n) containing the input fields.

metadata
[dict] Metadata dictionary containing the x1, x2, y1, y2, xpixelsize, ypixelsize, attributes as described in the documentation of *pysteps.io.importers*.

method
[{'pad', 'crop'}, optional] Either pad or crop. If pad, an equal number of zeros is added to both ends of its shortest side in order to produce a square domain. If crop, an equal number of pixels is removed to both ends of its longest side in order to produce a square domain. Note that the crop method involves an irreversible loss of data.

inverse
[bool, optional] Perform the inverse method to recover the original domain shape. After a crop, the inverse is performed by padding the field with zeros.

Returns

R
[array-like] the reshape dataset

metadata
[dict] the metadata with updated attributes.

pysteps.utils.fft

Interface module for different FFT methods.

get_numpy(shape[, fftn_shape])	
get_scipy(shape[, fftn_shape])	
get_pyfftw(shape[, fftn_shape, n_threads])	

pysteps.utils.fft.get_numpy

pysteps.utils.fft.**get_numpy**(*shape*, *fftn_shape=None*, ***kwargs*)

pysteps.utils.fft.get_scipy

pysteps.utils.fft.**get_scipy**(*shape*, *fftn_shape=None*, ***kwargs*)

pysteps.utils.fft.get_pyfftw

pysteps.utils.fft.**get_pyfftw**(*shape*, *fftn_shape=None*, *n_threads=1*, ***kwargs*)

pysteps.utils.images

Image processing routines for pysteps.

blob_detection(input_image[, method, ...])	Interface to the **'feature.blob_*'_** methods implemented in scikit-image.
shitomasi_detection(input_image[, ...])	Interface to the OpenCV Shi-Tomasi features detection method to detect corners in an image.
morph_opening(input_image, thr, n)	Filter out small scale noise on the image by applying a binary morphological opening, that is, erosion followed by dilation.

pysteps.utils.images.blob_detection

pysteps.utils.images.**blob_detection**(*input_image*, *method='log'*, *threshold=0.5*, *min_sigma=3*, *max_sigma=20*, *overlap=0.5*, *return_sigmas=False*, ***kwargs*)

Interface to the feature.blob_* methods implemented in scikit-image. A blob is defined as local a maximum

of a Gaussian-filtered image.

Parameters

input_image
[array_like] Array of shape (m, n) containing the input image. Nan values are ignored.

method
[{'log', 'dog', 'doh'}, optional] The method to use: 'log' = Laplacian of Gaussian, 'dog' = Difference of Gaussian, ''

threshold
[float, optional] Detection threshold.

min_sigma
[float, optional] The minimum standard deviation for the Gaussian kernel.

max_sigma
[float, optional] The maximum standard deviation for the Gaussian kernel.

overlap
[float, optional] A value between 0 and 1. If the area of two blobs overlaps by a fraction greater than threshold, the smaller blob is eliminated.

return_sigmas
[bool, optional] If True, the return array has a third column indicating the standard deviations of the Gaussian kernels that detected the blobs.

pysteps.utils.images.shitomasi_detection

pysteps.utils.images.**shitomasi_detection**(*input_image*, *max_corners=1000*, *quality_level=0.01*, *min_distance=10*, *block_size=5*, *buffer_mask=5*, *use_harris=False*, *k=0.04*, *verbose=False*, ***kwargs*)

Interface to the OpenCV Shi-Tomasi features detection method to detect corners in an image.

Corners are used for local tracking methods.

Parameters

input_image
[ndarray or MaskedArray] Array of shape (m, n) containing the input image.

In case of ndarray, invalid values (Nans or infs) are masked, otherwise the mask of the MaskedArray is used. Such mask defines a region where features are not detected.

The fill value for the masked pixels is taken as the minimum of all valid pixels.

max_corners
[int, optional] The `maxCorners` parameter in the Shi-Tomasi corner detection method. It represents the maximum number of points to be tracked (corners). If set to zero, all detected corners are used.

quality_level
[float, optional] The `qualityLevel` parameter in the Shi-Tomasi corner detection method. It represents the minimal accepted quality for the image corners.

min_distance
[int, optional] The `minDistance` parameter in the Shi-Tomasi corner detection method. It represents minimum possible Euclidean distance in pixels between corners.

block_size
[int, optional] The `blockSize` parameter in the Shi-Tomasi corner detection method.

It represents the window size in pixels used for computing a derivative covariation matrix over each pixel neighbourhood.

use_harris
[bool, optional] Whether to use a Harris detector or cornerMinEigenVal.

k
[float, optional] Free parameter of the Harris detector.

buffer_mask
[int, optional] A mask buffer width in pixels. This extends the input mask (if any) to limit edge effects.

verbose
[bool, optional] Print the number of features detected.

Returns

points
[ndarray] Array of shape (p, 2) indicating the pixel coordinates of *p* detected corners.

References

Jianbo Shi and Carlo Tomasi. Good features to track. In Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on, pages 593–600. IEEE, 1994.

pysteps.utils.images.morph_opening

pysteps.utils.images.**morph_opening**(*input_image*, *thr*, *n*)

Filter out small scale noise on the image by applying a binary morphological opening, that is, erosion followed by dilation.

Parameters

input_image
[ndarray or MaskedArray] Array of shape (m, n) containing the input image.

thr
[float] The threshold used to convert the image into a binary image.

n
[int] The structuring element size [pixels].

Returns

input_image
[ndarray or MaskedArray] Array of shape (m,n) containing the filtered image.

pysteps.utils.interpolate

Interpolation routines for pysteps.

rbfinterp2d(coord, input_array, xgrid, ygrid)	Fast 2-D grid interpolation of a sparse (multivariate) array using a radial basis function.

pysteps.utils.interpolate.rbfinterp2d

pysteps.utils.interpolate.**rbfinterp2d**(*coord*, *input_array*, *xgrid*, *ygrid*, *rbfunction='gaussian'*, *epsilon=10*, *k=50*, *nchunks=5*)

Fast 2-D grid interpolation of a sparse (multivariate) array using a radial basis function.

Parameters

coord
[array_like] Array of shape (n, 2) containing the coordinates of the data points into a 2-dimensional space.

input_array
[array_like] Array of shape (n) or (n, m) containing the values of the data points, where *n* is the number of data points and *m* the number of co-located variables. All values in `input_array` are required to have finite values.

xgrid, ygrid
[array_like] 1D arrays representing the coordinates of the 2-D output grid.

rbfunction
[{"gaussian", "multiquadric", "inverse quadratic", "inverse multiquadric", "bump"}, optional] The name of one of the available radial basis function based on a normalized Euclidian norm as defined in the **Notes** section below.

More details provided in the wikipedia reference page linked below.

epsilon
[float, optional] The shape parameter used to scale the input to the radial kernel.

A smaller value for `epsilon` produces a smoother interpolation. More details provided in the wikipedia reference page linked below.

k
[int or None, optional] The number of nearest neighbours used for each target location. This can also be useful to to speed-up the interpolation. If set to None, it interpolates using all the data points at once.

nchunks
[int, optional] The number of chunks in which the grid points are split to limit the memory usage during the interpolation.

Returns

output_array
[ndarray] The interpolated field(s) having shape (*m*, `ygrid.size`, `xgrid.size`).

Notes

The coordinates are normalized before computing the Euclidean norms:

x = (x - min(x)) / max[max(x) - min(x), max(y) - min(y)],

y = (y - min(y)) / max[max(x) - min(x), max(y) - min(y)],

where the min and max values are taken as the 2nd and 98th percentiles.

References

Wikipedia contributors, "Radial basis function," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Radial_basis_function&oldid=906155047 (accessed August 19, 2019).

pysteps.utils.spectral

Utility methods for processing and analyzing precipitation fields in the Fourier domain.

corrcoef(X, Y, shape[, use_full_fft])	Compute the correlation coefficient between two-dimensional arrays in the spectral domain.
mean(X, shape)	Compute the mean value of a two-dimensional array in the spectral domain.
rapsd(Z[, fft_method, return_freq, d, normalize])	Compute radially averaged power spectral density (RAPSD) from the given 2D input field.
remove_rain_norain_discontinuity(R)	Function to remove the rain/no-rain discontinuity.
std(X, shape[, use_full_fft])	Compute the standard deviation of a two-dimensional array in the spectral domain.

pysteps.utils.spectral.corrcoef

pysteps.utils.spectral.**corrcoef**(*X*, *Y*, *shape*, *use_full_fft=False*)

Compute the correlation coefficient between two-dimensional arrays in the spectral domain.

Parameters

X
[array_like] A complex array representing the Fourier transform of a two-dimensional array.

Y
[array_like] A complex array representing the Fourier transform of a two-dimensional array.

shape
[tuple] A two-element tuple specifying the shape of the original input arrays in the spatial domain.

use_full_fft
[bool] If True, X and Y represent the full FFTs of the original arrays. Otherwise, they are assumed to contain only the symmetric part, i.e. in the format returned by numpy.fft.rfft2.

Returns

out
[float] The correlation coefficient. Gives the same result as numpy.corrcoef(X.flatten(), Y.flatten())[0, 1].

pysteps.utils.spectral.mean

pysteps.utils.spectral.**mean**(*X*, *shape*)

Compute the mean value of a two-dimensional array in the spectral domain.

Parameters

X
[array_like] A complex array representing the Fourier transform of a two-dimensional array.

shape
[tuple] A two-element tuple specifying the shape of the original input array in the spatial domain.

Returns

out
[float] The mean value.

pysteps.utils.spectral.rapsd

pysteps.utils.spectral.**rapsd**(*Z*, *fft_method=None*, *return_freq=False*, *d=1.0*, *normalize=False*, ***fft_kwargs*)

Compute radially averaged power spectral density (RAPSD) from the given 2D input field.

Parameters

Z
[array_like] A 2d array of shape (M,N) containing the input field.

fft_method
[object] A module or object implementing the same methods as numpy.fft and scipy.fftpack. If set to None, Z is assumed to represent the shifted discrete Fourier transform of the input field, where the origin is at the center of the array (see numpy.fft.fftshift or scipy.fftpack.fftshift).

return_freq: bool
Whether to also return the Fourier frequencies.

d
[scalar] Sample spacing (inverse of the sampling rate). Defaults to 1. Applicable if return_freq is 'True'.

normalize
[bool] If True, normalize the power spectrum so that it sums to one.

Returns

out
[ndarray] One-dimensional array containing the RAPSD. The length of the array is int(L/2)+1 (if L is even) or int(L/2) (if L is odd), where L=max(M,N).

freq
[ndarray] One-dimensional array containing the Fourier frequencies.

References

[RC11]

pysteps.utils.spectral.remove_rain_norain_discontinuity

pysteps.utils.spectral.**remove_rain_norain_discontinuity**(*R*)

Function to remove the rain/no-rain discontinuity. It can be used before computing Fourier filters to reduce the artificial increase of power at high frequencies caused by the discontinuity.

Parameters

R
[array-like] Array of any shape to be transformed.

Returns

R
[array-like] Array of any shape containing the transformed data.

pysteps.utils.spectral.std

pysteps.utils.spectral.**std**(*X*, *shape*, *use_full_fft=False*)

Compute the standard deviation of a two-dimensional array in the spectral domain.

Parameters

X
[array_like] A complex array representing the Fourier transform of a two-dimensional array.

shape
[tuple] A two-element tuple specifying the shape of the original input array in the spatial domain.

use_full_fft
[bool] If True, X represents the full FFT of the original array. Otherwise, it is assumed to contain only the symmetric part, i.e. in the format returned by numpy.fft.rfft2.

Returns

out
[float] The standard deviation.

pysteps.utils.tapering

Implementations of window functions for computing of the FFT.

compute_mask_window_function(mask, func, ...)	Compute window function for a two-dimensional area defined by a non-rectangular mask.
compute_window_function(m, n, func, **kwargs)	Compute window function for a two-dimensional rectangular region.

pysteps.utils.tapering.compute_mask_window_function

pysteps.utils.tapering.**compute_mask_window_function**(*mask*, *func*, ***kwargs*)
Compute window function for a two-dimensional area defined by a non-rectangular mask. The window function is computed based on the distance to the nearest boundary point of the mask. Window function-specific parameters are given as keyword arguments.

Parameters

mask
[array_like] Two-dimensional boolean array containing the mask. Pixels with True/False are inside/outside the mask.

func
[str] The name of the window function. The currently implemented function is 'tukey'.

Returns

out
[array] Array containing the tapering weights.

pysteps.utils.tapering.compute_window_function

pysteps.utils.tapering.**compute_window_function**(*m*, *n*, *func*, ***kwargs*)
Compute window function for a two-dimensional rectangular region. Window function-specific parameters are given as keyword arguments.

Parameters

m
[int] Height of the array.

n
[int] Width of the array.

func
[str] The name of the window function. The currently implemented functions are 'hann' and 'tukey'.

Returns

out
[array] Array of shape (m, n) containing the tapering weights.

Other Parameters

alpha
[float] Applicable if func is 'tukey'.

Notes

Two-dimensional tapering weights are computed from one-dimensional window functions using w(r), where r is the distance from the center of the region.

pysteps.utils.transformation

Methods for transforming data values.

boxcox_transform(R[, metadata, Lambda, ...])	The one-parameter Box-Cox transformation.
dB_transform(R[, metadata, threshold, ...])	Methods to transform precipitation intensities to/from dB units.
NQ_transform(R[, metadata, inverse])	The normal quantile transformation as in Bogner et al (2012).
sqrt_transform(R[, metadata, inverse])	Square-root transform.

pysteps.utils.transformation.boxcox_transform

pysteps.utils.transformation.**boxcox_transform**(*R*, *metadata=None*, *Lambda=None*, *threshold=None*, *zerovalue=None*, *inverse=False*)

The one-parameter Box-Cox transformation.

The Box-Cox transform is a well-known power transformation introduced by Box and Cox (1964). In its one-parameter version, the Box-Cox transform takes the form T(x) = ln(x) for Lambda = 0, or T(x) = (x**Lambda - 1)/Lambda otherwise.

Default parameters will produce a log transform (i.e. Lambda=0).

Parameters

R
[array-like] Array of any shape to be transformed.

metadata
[dict, optional] Metadata dictionary containing the transform, zerovalue and threshold attributes as described in the documentation of *pysteps.io.importers*.

Lambda
[float, optional] Parameter Lambda of the Box-Cox transformation. It is 0 by default, which produces the log transformation.

Choose Lambda < 1 for positively skewed data, Lambda > 1 for negatively skewed data.

threshold
[float, optional] The value that is used for thresholding with the same units as R. If None, the threshold contained in metadata is used. If no threshold is found in the metadata, a value of 0.1 is used as default.

zerovalue
[float, optional] The value to be assigned to no rain pixels as defined by the threshold. It is equal to the threshold - 1 by default.

inverse
[bool, optional] If set to True, it performs the inverse transform. False by default.

Returns

R
[array-like] Array of any shape containing the (back-)transformed units.

metadata
[dict] The metadata with updated attributes.

References

Box, G. E. and Cox, D. R. (1964), An Analysis of Transformations. Journal of the Royal Statistical Society: Series B (Methodological), 26: 211-243. doi:10.1111/j.2517-6161.1964.tb00553.x

pysteps.utils.transformation.dB_transform

`pysteps.utils.transformation.`**`dB_transform`**(*R*, *metadata=None*, *threshold=None*, *zerovalue=None*, *inverse=False*)

Methods to transform precipitation intensities to/from dB units.

Parameters

R
[array-like] Array of any shape to be (back-)transformed.

metadata
[dict, optional] Metadata dictionary containing the transform, zerovalue and threshold attributes as described in the documentation of *pysteps.io.importers*.

threshold
[float, optional] Optional value that is used for thresholding with the same units as R. If None, the threshold contained in metadata is used. If no threshold is found in the metadata, a value of 0.1 is used as default.

zerovalue
[float, optional] The value to be assigned to no rain pixels as defined by the threshold. It is equal to the threshold - 1 by default.

inverse
[bool, optional] If set to True, it performs the inverse transform. False by default.

Returns

R
[array-like] Array of any shape containing the (back-)transformed units.

metadata
[dict] The metadata with updated attributes.

pysteps.utils.transformation.NQ_transform

pysteps.utils.transformation.**NQ_transform**(*R*, *metadata=None*, *inverse=False*, ***kwargs*)

The normal quantile transformation as in Bogner et al (2012). Zero rain vales are set to zero in norm space.

Parameters

R
[array-like] Array of any shape to be transformed.

metadata
[dict, optional] Metadata dictionary containing the transform, zerovalue and threshold attributes as described in the documentation of *pysteps.io.importers*.

inverse
[bool, optional] If set to True, it performs the inverse transform. False by default.

Returns

R
[array-like] Array of any shape containing the (back-)transformed units.

metadata
[dict] The metadata with updated attributes.

Other Parameters

a
[float, optional] The offset fraction to be used for plotting positions; typically in (0,1). The default is 0., that is, it spaces the points evenly in the uniform distribution.

References

Bogner, K., Pappenberger, F., and Cloke, H. L.: Technical Note: The normal quantile transformation and its application in a flood forecasting system, Hydrol. Earth Syst. Sci., 16, 1085-1094, https://doi.org/10.5194/hess-16-1085-2012, 2012.

pysteps.utils.transformation.sqrt_transform

pysteps.utils.transformation.**sqrt_transform**(*R*, *metadata=None*, *inverse=False*, ***kwargs*)

Square-root transform.

Parameters

R
[array-like] Array of any shape to be transformed.

metadata
[dict, optional] Metadata dictionary containing the transform, zerovalue and threshold attributes as described in the documentation of *pysteps.io.importers*.

inverse
[bool, optional] If set to True, it performs the inverse transform. False by default.

Returns

R
[array-like] Array of any shape containing the (back-)transformed units.

metadata
[dict] The metadata with updated attributes.

2.2.12 pysteps.verification

Methods for verification of deterministic, probabilistic and ensemble forecasts.

pysteps.verification.interface

Interface for the verification module.

get_method(name[, type])	Return a callable function for the method corresponding to the given verification score.

pysteps.verification.interface.get_method

pysteps.verification.interface.**get_method**(*name*, *type='deterministic'*)
Return a callable function for the method corresponding to the given verification score.

Parameters

name
[str] Name of the verification method. The available options are:

type: deterministic

Name	Description
ACC	accuracy (proportion correct)
BIAS	frequency bias
CSI	critical success index (threat score)
F1	the harmonic mean of precision and sensitivity
FA	false alarm rate (prob. of false detection, fall-out, false positive rate)
FAR	false alarm ratio (false discovery rate)
GSS	Gilbert skill score (equitable threat score)
HK	Hanssen-Kuipers discriminant (Pierce skill score)
HSS	Heidke skill score
MCC	Matthews correlation coefficient
POD	probability of detection (hit rate, sensitivity, recall, true positive rate)
SEDI	symmetric extremal dependency index
beta1	linear regression slope (type 1 conditional bias)
beta2	linear regression slope (type 2 conditional bias)
corr_p	pearson's correleation coefficien (linear correlation)
corr_s*	spearman's correlation coefficient (rank correlation)
DRMSE	debiased root mean squared error
MAE	mean absolute error of residuals
ME	mean error or bias of residuals
MSE	mean squared error
NMSE	normalized mean squared error
RMSE	root mean squared error
RV	reduction of variance (Brier Score, Nash-Sutcliffe Efficiency)
scatter*	half the distance between the 16% and 84% percentiles of the weighted cumulative error distribution, where error = dB(pred/obs), as in Germann et al. (2006)
binary_mse	binary MSE
FSS	fractions skill score

type: ensemble

Name	Description
ens_skill	mean ensemble skill
ens_spread	mean ensemble spread
rankhist	rank histogram

type: probabilistic

Name	Description
CRPS	continuous ranked probability score
reldiag	reliability diagram
ROC	ROC curve

type
[{'deterministic', 'ensemble', 'probabilistic'}, optional] Type of the verification method.

Notes

Multiplicative scores can be computed by passing log-tranformed values. Note that "scatter" is the only score that will be computed in dB units of the multiplicative error, i.e.: 10*log10(pred/obs).

beta1 measures the degree of conditional bias of the observations given the forecasts (type 1).

beta2 measures the degree of conditional bias of the forecasts given the observations (type 2).

The normalized MSE is computed as NMSE = E[(pred - obs)^2]/E[(pred + obs)^2].

The debiased RMSE is computed as DRMSE = sqrt(RMSE - ME^2).

The reduction of variance score is computed as RV = 1 - MSE/Var(obs).

Score names denoted by * can only be computed offline, meaning that the these cannot be computed using _init, _accum and _compute methods of this module.

References

Germann, U. , Galli, G. , Boscacci, M. and Bolliger, M. (2006), Radar precipitation measurement in a mountainous region. Q.J.R. Meteorol. Soc., 132: 1669-1692. doi:10.1256/qj.05.190

Potts, J. (2012), Chapter 2 - Basic concepts. Forecast verification: a practitioner's guide in atmospheric sciences, I. T. Jolliffe, and D. B. Stephenson, Eds., Wiley-Blackwell, 11–29.

pysteps.verification.detcatscores

Forecast evaluation and skill scores for deterministic categorial (dichotomous) forecasts.

det_cat_fct(pred, obs, thr[, scores, axis])	Calculate simple and skill scores for deterministic categorical (dichotomous) forecasts.
det_cat_fct_init(thr[, axis])	Initialize a contingency table object.
det_cat_fct_accum(contab, pred, obs)	Accumulate the frequency of "yes" and "no" forecasts and observations in the contingency table.
det_cat_fct_merge(contab_1, contab_2)	Merge two contingency table objects.
det_cat_fct_compute(contab[, scores])	Compute simple and skill scores for deterministic categorical (dichotomous) forecasts from a contingency table object.

pysteps.verification.detcatscores.det_cat_fct

pysteps.verification.detcatscores.**det_cat_fct**(*pred*, *obs*, *thr*, *scores=''*, *axis=None*)

Calculate simple and skill scores for deterministic categorical (dichotomous) forecasts.

Parameters

pred
[array_like] Array of predictions. NaNs are ignored.

obs
[array_like] Array of verifying observations. NaNs are ignored.

thr
[float] The threshold that is applied to predictions and observations in order to define events vs no events (yes/no).

scores
[{string, list of strings}, optional] The name(s) of the scores. The default, scores="", will compute all available scores. The available score names are:

Name	Description
ACC	accuracy (proportion correct)
BIAS	frequency bias
CSI	critical success index (threat score)
F1	the harmonic mean of precision and sensitivity
FA	false alarm rate (prob. of false detection, fall-out, false positive rate)
FAR	false alarm ratio (false discovery rate)
GSS	Gilbert skill score (equitable threat score)
HK	Hanssen-Kuipers discriminant (Pierce skill score)
HSS	Heidke skill score
MCC	Matthews correlation coefficient
POD	probability of detection (hit rate, sensitivity, recall, true positive rate)
SEDI	symmetric extremal dependency index

axis
[None or int or tuple of ints, optional] Axis or axes along which a score is integrated. The default, axis=None, will integrate all of the elements of the input arrays.

If axis is -1 (or any negative integer), the integration is not performed and scores are computed on all of the elements in the input arrays.

If axis is a tuple of ints, the integration is performed on all of the axes specified in the tuple.

Returns

result
[dict] Dictionary containing the verification results.

See also:

pysteps.verification.detcontscores.det_cont_fct

pysteps.verification.detcatscores.det_cat_fct_init

pysteps.verification.detcatscores.**det_cat_fct_init**(*thr*, *axis=None*)

Initialize a contingency table object.

Parameters

thr
[float] threshold that is applied to predictions and observations in order to define events vs no events (yes/no).

axis
[None or int or tuple of ints, optional] Axis or axes along which a score is integrated. The default, axis=None, will integrate all of the elements of the input arrays.

If axis is -1 (or any negative integer), the integration is not performed and scores are computed on all of the elements in the input arrays.

If axis is a tuple of ints, the integration is performed on all of the axes specified in the tuple.

Returns

out
[dict] The contingency table object.

pysteps.verification.detcatscores.det_cat_fct_accum

`pysteps.verification.detcatscores.`**`det_cat_fct_accum`**(*contab*, *pred*, *obs*)

Accumulate the frequency of "yes" and "no" forecasts and observations in the contingency table.

Parameters

contab
[dict] A contingency table object initialized with pysteps.verification.detcatscores.det_cat_fct_init.

pred
[array_like] Array of predictions. NaNs are ignored.

obs
[array_like] Array of verifying observations. NaNs are ignored.

pysteps.verification.detcatscores.det_cat_fct_merge

`pysteps.verification.detcatscores.`**`det_cat_fct_merge`**(*contab_1*, *contab_2*)

Merge two contingency table objects.

Parameters

contab_1
[dict] A contingency table object initialized with *pysteps.verification.detcatscores.det_cat_fct_init()* and populated with *pysteps.verification.detcatscores.det_cat_fct_accum()*.

contab_2
[dict] Another contingency table object initialized with *pysteps.verification.detcatscores.det_cat_fct_init()* and populated with *pysteps.verification.detcatscores.det_cat_fct_accum()*.

Returns

out
[dict] The merged contingency table object.

pysteps.verification.detcatscores.det_cat_fct_compute

pysteps.verification.detcatscores.**det_cat_fct_compute**(*contab*, *scores=''*)

Compute simple and skill scores for deterministic categorical (dichotomous) forecasts from a contingency table object.

Parameters

contab
[dict] A contingency table object initialized with pysteps.verification.detcatscores.det_cat_fct_init and populated with pysteps.verification.detcatscores.det_cat_fct_accum.

scores
[{string, list of strings}, optional] The name(s) of the scores. The default, scores="", will compute all available scores. The available score names a

Name	Description
ACC	accuracy (proportion correct)
BIAS	frequency bias
CSI	critical success index (threat score)
F1	the harmonic mean of precision and sensitivity
FA	false alarm rate (prob. of false detection, fall-out, false positive rate)
FAR	false alarm ratio (false discovery rate)
GSS	Gilbert skill score (equitable threat score)
HK	Hanssen-Kuipers discriminant (Pierce skill score)
HSS	Heidke skill score
MCC	Matthews correlation coefficient
POD	probability of detection (hit rate, sensitivity, recall, true positive rate)
SEDI	symmetric extremal dependency index

Returns

result
[dict] Dictionary containing the verification results.

pysteps.verification.detcontscores

Forecast evaluation and skill scores for deterministic continuous forecasts.

det_cont_fct(pred, obs[, scores, axis, ...])	Calculate simple and skill scores for deterministic continuous forecasts.
det_cont_fct_init([axis, conditioning, thr])	Initialize a verification error object.
det_cont_fct_accum(err, pred, obs)	Accumulate the forecast error in the verification error object.
det_cont_fct_merge(err_1, err_2)	Merge two verification error objects.
det_cont_fct_compute(err[, scores])	Compute simple and skill scores for deterministic continuous forecasts from a verification error object.

pysteps.verification.detcontscores.det_cont_fct

pysteps.verification.detcontscores.**det_cont_fct**(*pred*, *obs*, *scores=''*, *axis=None*, *conditioning=None*, *thr=0.0*)

Calculate simple and skill scores for deterministic continuous forecasts.

Parameters

pred
[array_like] Array of predictions. NaNs are ignored.

obs
[array_like] Array of verifying observations. NaNs are ignored.

scores
[{string, list of strings}, optional] The name(s) of the scores. The default, scores="", will compute all available scores. The available score names are:

Name	Description
beta1	linear regression slope (type 1 conditional bias)
beta2	linear regression slope (type 2 conditional bias)
corr_p	pearson's correleation coefficien (linear correlation)
corr_s*	spearman's correlation coefficient (rank correlation)
DRMSE	debiased root mean squared error
MAE	mean absolute error
ME	mean error or bias
MSE	mean squared error
NMSE	normalized mean squared error
RMSE	root mean squared error
RV	reduction of variance (Brier Score, Nash-Sutcliffe Efficiency)
scatter*	half the distance between the 16% and 84% percentiles of the weighted cumulative error distribution, where error = dB(pred/obs), as in Germann et al. (2006)

axis
[{int, tuple of int, None}, optional] Axis or axes along which a score is integrated. The default, axis=None, will integrate all of the elements of the input arrays.

If axis is -1 (or any negative integer), the integration is not performed and scores are computed on all of the elements in the input arrays.

If axis is a tuple of ints, the integration is performed on all of the axes specified in the tuple.

conditioning
[{None, "single", "double"}, optional] The type of conditioning used for the verification. The default, conditioning=None, includes all pairs. With conditioning="single", only pairs with either pred or obs > thr are included. With conditioning="double", only pairs with both pred and obs > thr are included.

thr
[float] Optional threshold value for conditioning. Defaults to 0.

Returns

result
[dict] Dictionary containing the verification results.

See also:

pysteps.verification.detcatscores.det_cat_fct

Notes

Multiplicative scores can be computed by passing log-tranformed values. Note that "scatter" is the only score that will be computed in dB units of the multiplicative error, i.e.: 10*log10(pred/obs).

beta1 measures the degree of conditional bias of the observations given the forecasts (type 1).

beta2 measures the degree of conditional bias of the forecasts given the observations (type 2).

The normalized MSE is computed as NMSE = E[(pred - obs)^2]/E[(pred + obs)^2].

The debiased RMSE is computed as DRMSE = sqrt(MSE - ME^2).

The reduction of variance score is computed as RV = 1 - MSE/Var(obs).

Score names denoted by * can only be computed offline, meaning that the these cannot be computed using _init, _accum and _compute methods of this module.

References

Germann, U. , Galli, G. , Boscacci, M. and Bolliger, M. (2006), Radar precipitation measurement in a mountainous region. Q.J.R. Meteorol. Soc., 132: 1669-1692. doi:10.1256/qj.05.190

Potts, J. (2012), Chapter 2 - Basic concepts. Forecast verification: a practitioner's guide in atmospheric sciences, I. T. Jolliffe, and D. B. Stephenson, Eds., Wiley-Blackwell, 11–29.

pysteps.verification.detcontscores.det_cont_fct_init

pysteps.verification.detcontscores.**det_cont_fct_init**(*axis=None*, *conditioning=None*, *thr=0.0*)

Initialize a verification error object.

Parameters

axis
[{int, tuple of int, None}, optional] Axis or axes along which a score is integrated. The default, axis=None, will integrate all of the elements of the input arrays.

If axis is -1 (or any negative integer), the integration is not performed and scores are computed on all of the elements in the input arrays.

If axis is a tuple of ints, the integration is performed on all of the axes specified in the tuple.

conditioning
[{None, "single", "double"}, optional] The type of conditioning used for the verification. The default, conditioning=None, includes all pairs. With conditioning="single", only pairs with either pred or obs > thr are included. With conditioning="double", only pairs with both pred and obs > thr are included.

thr
[float] Optional threshold value for conditioning. Defaults to 0.

Returns

out
[dict] The verification error object.

pysteps.verification.detcontscores.det_cont_fct_accum

pysteps.verification.detcontscores.**det_cont_fct_accum**(*err*, *pred*, *obs*)

Accumulate the forecast error in the verification error object.

Parameters

err
[dict] A verification error object initialized with *pysteps.verification.detcontscores.det_cont_fct_init()*.

pred
[array_like] Array of predictions. NaNs are ignored.

obs
[array_like] Array of verifying observations. NaNs are ignored.

References

Chan, Tony F.; Golub, Gene H.; LeVeque, Randall J. (1979), "Updating Formulae and a Pairwise Algorithm for Computing Sample Variances.", Technical Report STAN-CS-79-773, Department of Computer Science, Stanford University.

Schubert, Erich; Gertz, Michael (2018-07-09). "Numerically stable parallel computation of (co-)variance". ACM: 10. doi:10.1145/3221269.3223036.

pysteps.verification.detcontscores.det_cont_fct_merge

pysteps.verification.detcontscores.**det_cont_fct_merge**(*err_1*, *err_2*)
Merge two verification error objects.

Parameters

err_1
[dict] A verification error object initialized with *pysteps.verification.detcontscores.det_cont_fct_init()* and populated with *pysteps.verification.detcontscores.det_cont_fct_accum()*.

err_2
[dict] Another verification error object initialized with *pysteps.verification.detcontscores.det_cont_fct_init()* and populated with *pysteps.verification.detcontscores.det_cont_fct_accum()*.

Returns

out
[dict] The merged verification error object.

pysteps.verification.detcontscores.det_cont_fct_compute

pysteps.verification.detcontscores.**det_cont_fct_compute**(*err*, *scores=''*)
Compute simple and skill scores for deterministic continuous forecasts from a verification error object.

Parameters

err
[dict] A verification error object initialized with *pysteps.verification.detcontscores.det_cont_fct_init()* and populated with *pysteps.verification.detcontscores.det_cont_fct_accum()*.

scores
[{string, list of strings}, optional] The name(s) of the scores. The default, scores="", will compute all available scores. The available score names are:

Name	Description
beta1 | linear regression slope (type 1 conditional bias)	
beta2 | linear regression slope (type 2 conditional bias)	
corr_p	pearson's correleation coefficien (linear correlation)
DRMSE	debiased root mean squared error, i.e. $DRMSE = \sqrt{RMSE - ME^2}$
MAE	mean absolute error
ME	mean error or bias
MSE	mean squared error
NMSE	normalized mean squared error
RMSE	root mean squared error
RV	reduction of variance (Brier Score, Nash-Sutcliffe Efficiency), i.e. $RV = 1 - \frac{MSE}{s_o^2}$

Returns

result
[dict] Dictionary containing the verification results.

pysteps.verification.ensscores

Evaluation and skill scores for ensemble forecasts.

ensemble_skill(X_f, X_o, metric, **kwargs)	Compute mean ensemble skill for a given skill metric.
ensemble_spread(X_f, metric, **kwargs)	Compute mean ensemble spread for a given skill metric.
rankhist(X_f, X_o[, X_min, normalize])	Compute a rank histogram counts and optionally normalize the histogram.
rankhist_init(num_ens_members[, X_min])	Initialize a rank histogram object.
rankhist_accum(rankhist, X_f, X_o)	Accumulate forecast-observation pairs to the given rank histogram.
rankhist_compute(rankhist[, normalize])	Return the rank histogram counts and optionally normalize the histogram.

pysteps.verification.ensscores.ensemble_skill

pysteps.verification.ensscores.**ensemble_skill**(*X_f*, *X_o*, *metric*, ***kwargs*)
Compute mean ensemble skill for a given skill metric.

Parameters

X_f
[array-like] Array of shape (l,m,n) containing the forecast fields of shape (m,n) from l ensemble members.

X_o
[array_like] Array of shape (m,n) containing the observed field corresponding to the forecast.

metric
[str] The deterministic skill metric to be used (list available in *get_method()*).

Returns

out
[float] The mean skill of all ensemble members that is used as defintion of ensemble

skill (as in Zacharov and Rezcova 2009 with the FSS).

References

[ZR09]

pysteps.verification.ensscores.ensemble_spread

pysteps.verification.ensscores.**ensemble_spread**(*X_f*, *metric*, ***kwargs*)

Compute mean ensemble spread for a given skill metric.

Parameters

X_f
[array-like] Array of shape (l,m,n) containing the forecast fields of shape (m,n) from l ensemble members.

metric
[str] The deterministic skill metric to be used (list available in *get_method()*).

Returns

out
[float] The mean skill compted between all possible pairs of the ensemble members, which can be used as definition of mean ensemble spread (as in Zacharov and Rezcova 2009 with the FSS).

References

[ZR09]

pysteps.verification.ensscores.rankhist

pysteps.verification.ensscores.**rankhist**(*X_f*, *X_o*, *X_min=None*, *normalize=True*)

Compute a rank histogram counts and optionally normalize the histogram.

Parameters

X_f
[array-like] Array of shape (k,m,n,...) containing the values from an ensemble forecast of k members with shape (m,n,...).

X_o
[array_like] Array of shape (m,n,...) containing the observed values corresponding to the forecast.

X_min
[{float,None}] Threshold for minimum intensity. Forecast-observation pairs, where all ensemble members and verifying observations are below X_min, are not counted in the rank histogram. If set to None, thresholding is not used.

normalize
[{bool, True}] If True, normalize the rank histogram so that the bin counts sum to one.

pysteps.verification.ensscores.rankhist_init

pysteps.verification.ensscores.**rankhist_init**(*num_ens_members*, *X_min=None*)

Initialize a rank histogram object.

Parameters

num_ens_members
[int] Number ensemble members in the forecasts to accumulate into the rank histogram.

X_min
[{float,None}] Threshold for minimum intensity. Forecast-observation pairs, where all ensemble members and verifying observations are below X_min, are not counted in the rank histogram. If set to None, thresholding is not used.

Returns

out
[dict] The rank histogram object.

pysteps.verification.ensscores.rankhist_accum

pysteps.verification.ensscores.**rankhist_accum**(*rankhist*, *X_f*, *X_o*)
Accumulate forecast-observation pairs to the given rank histogram.

Parameters

rankhist
[dict] The rank histogram object.

X_f
[array-like] Array of shape (k,m,n,...) containing the values from an ensemble forecast of k members with shape (m,n,...).

X_o
[array_like] Array of shape (m,n,...) containing the observed values corresponding to the forecast.

pysteps.verification.ensscores.rankhist_compute

pysteps.verification.ensscores.**rankhist_compute**(*rankhist*, *normalize=True*)
Return the rank histogram counts and optionally normalize the histogram.

Parameters

rankhist
[dict] A rank histogram object created with rankhist_init.

normalize
[bool] If True, normalize the rank histogram so that the bin counts sum to one.

Returns

out
[array_like] The counts for the n+1 bins in the rank histogram, where n is the number of ensemble members.

pysteps.verification.lifetime

Estimation of precipitation lifetime from a decaying verification score function (e.g. autocorrelation function).

lifetime(X_s, X_t[, rule])	Compute the average lifetime by integrating the correlation function as a function of lead time.
lifetime_init([rule])	Initialize a lifetime object.

Continued on next page

Table 51 – continued from previous page

lifetime_accum(lifetime, X_s, X_t)	Compute the lifetime by integrating the correlation function and accumulate the result into the given lifetime object.
lifetime_compute(lifetime)	Compute the average value from the lifetime object.

pysteps.verification.lifetime.lifetime

pysteps.verification.lifetime.**lifetime**(*X_s*, *X_t*, *rule='1/e'*)

Compute the average lifetime by integrating the correlation function as a function of lead time. When not using the 1/e rule, the correlation function must be long enough to converge to 0, otherwise the lifetime is underestimated. The correlation function can be either empirical or theoretical, e.g. derived using the function 'ar_acf' in timeseries/autoregression.py.

Parameters

X_s
[array-like] Array with the correlation function. Works also with other decaying scores that are defined in the range [0,1]=[min_skill,max_skill].

X_t
[array-like] Array with the forecast lead times in the desired unit, e.g. [min, hour].

rule
[str {'1/e', 'trapz', 'simpson'}, optional] Name of the method to integrate the correlation curve.

'1/e' uses the 1/e rule and assumes an exponential decay. It linearly interpolates the time when the correlation goes below the value 1/e. When all values are > 1/e it returns the max lead time. When all values are < 1/e it returns the min lead time.

'trapz' uses the trapezoidal rule for integration.

'simpson' uses the Simpson's rule for integration.

Returns

lf
[float] Estimated lifetime with same units of X_t.

pysteps.verification.lifetime.lifetime_init

pysteps.verification.lifetime.**lifetime_init**(*rule='1/e'*)

Initialize a lifetime object.

Parameters

rule
[str {'1/e', 'trapz', 'simpson'}, optional] Name of the method to integrate the correlation curve.

'1/e' uses the 1/e rule and assumes an exponential decay. It linearly interpolates the time when the correlation goes below the value 1/e. When all values are > 1/e it returns the max lead time. When all values are < 1/e it returns the min lead time.

'trapz' uses the trapezoidal rule for integration.

'simpson' uses the Simpson's rule for integration.

Returns

out
[dict] The lifetime object.

pysteps.verification.lifetime.lifetime_accum

pysteps.verification.lifetime.**lifetime_accum**(*lifetime*, *X_s*, *X_t*)

Compute the lifetime by integrating the correlation function and accumulate the result into the given lifetime object.

Parameters

X_s
[array-like] Array with the correlation function. Works also with other decaying scores that are defined in the range [0,1]=[min_skill,max_skill].

X_t
[array-like] Array with the forecast lead times in the desired unit, e.g. [min, hour].

pysteps.verification.lifetime.lifetime_compute

pysteps.verification.lifetime.**lifetime_compute**(*lifetime*)

Compute the average value from the lifetime object.

Parameters

lifetime
[dict] A lifetime object created with lifetime_init.

Returns

out
[float] The computed lifetime.

pysteps.verification.plots

Methods for plotting verification results.

plot_intensityscale(intscale[, fig, ...])	Plot a intensity-scale verification table with a color bar and axis labels.
plot_rankhist(rankhist[, ax])	Plot a rank histogram.
plot_reldiag(reldiag[, ax])	Plot a reliability diagram.
plot_ROC(ROC[, ax, opt_prob_thr])	Plot a ROC curve.

pysteps.verification.plots.plot_intensityscale

pysteps.verification.plots.**plot_intensityscale**(*intscale*, *fig=None*, *vminmax=None*, *kmperpixel=None*, *unit=None*)

Plot a intensity-scale verification table with a color bar and axis labels.

Parameters

intscale
[dict] The intensity-scale object initialized with *pysteps.verification.spatialscores.intensity_scale_init()* and accumulated with *pysteps.verification.spatialscores.intensity_scale_accum()*.

fig
[matplotlib.figure.Figure, optional] The figure object to use for plotting. If not supplied, a new figure is created.

vminmax
[tuple of floats, optional] The minimum and maximum values for the intensity-scale

skill score in the plot. Defaults to the data extent.

kmperpixel
[float, optional] The conversion factor from pixels to kilometers. If supplied, the unit of the shown spatial scales is km instead of pixels.

unit
[string, optional] The unit of the intensity thresholds.

pysteps.verification.plots.plot_rankhist

pysteps.verification.plots.**plot_rankhist**(*rankhist*, *ax=None*)
Plot a rank histogram.

Parameters

rankhist
[dict] A rank histogram object created by ensscores.rankhist_init.

ax
[axis handle, optional] Axis handle for the figure. If set to None, the handle is taken from the current figure (matplotlib.pylab.gca()).

pysteps.verification.plots.plot_reldiag

pysteps.verification.plots.**plot_reldiag**(*reldiag*, *ax=None*)
Plot a reliability diagram.

Parameters

reldiag
[dict] A reldiag object created by probscores.reldiag_init.

ax
[axis handle, optional] Axis handle for the figure. If set to None, the handle is taken from the current figure (matplotlib.pylab.gca()).

pysteps.verification.plots.plot_ROC

pysteps.verification.plots.**plot_ROC**(*ROC*, *ax=None*, *opt_prob_thr=False*)
Plot a ROC curve.

Parameters

ROC
[dict] A ROC curve object created by probscores.ROC_curve_init.

ax
[axis handle, optional] Axis handle for the figure. If set to None, the handle is taken from the current figure (matplotlib.pylab.gca()).

opt_prob_thr
[bool, optional] If set to True, plot the optimal probability threshold that maximizes the difference between the hit rate (POD) and false alarm rate (POFD).

pysteps.verification.probscores

Evaluation and skill scores for probabilistic forecasts.

CRPS(X_f, X_o)	Compute the continuous ranked probability score (CRPS).

Continued on next page

Table 53 – continued from previous page

CRPS_init()	Initialize a CRPS object.
CRPS_accum(CRPS, X_f, X_o)	Compute the average continuous ranked probability score (CRPS) for a set of forecast ensembles and the corresponding observations and accumulate the result to the given CRPS object.
CRPS_compute(CRPS)	Compute the averaged values from the given CRPS object.
reldiag(P_f, X_o, X_min[, n_bins, min_count])	Compute the x- and y- coordinates of the points in the reliability diagram.
reldiag_init(X_min[, n_bins, min_count])	Initialize a reliability diagram object.
reldiag_accum(reldiag, P_f, X_o)	Accumulate the given probability-observation pairs into the reliability diagram.
reldiag_compute(reldiag)	Compute the x- and y- coordinates of the points in the reliability diagram.
ROC_curve(P_f, X_o, X_min[, n_prob_thrs, ...])	Compute the ROC curve and its area from the given ROC object.
ROC_curve_init(X_min[, n_prob_thrs])	Initialize a ROC curve object.
ROC_curve_accum(ROC, P_f, X_o)	Accumulate the given probability-observation pairs into the given ROC object.
ROC_curve_compute(ROC[, compute_area])	Compute the ROC curve and its area from the given ROC object.

pysteps.verification.probscores.CRPS

pysteps.verification.probscores.**CRPS**(*X_f*, *X_o*)

Compute the continuous ranked probability score (CRPS).

Parameters

X_f
[array_like] Array of shape (k,m,n,...) containing the values from an ensemble forecast of k members with shape (m,n,...).

X_o
[array_like] Array of shape (m,n,...) containing the observed values corresponding to the forecast.

Returns

out
[float] The computed CRPS.

References

[Her00]

pysteps.verification.probscores.CRPS_init

pysteps.verification.probscores.**CRPS_init**()

Initialize a CRPS object.

Returns

out
[dict] The CRPS object.

pysteps.verification.probscores.CRPS_accum

pysteps.verification.probscores.**CRPS_accum**(*CRPS*, *X_f*, *X_o*)

Compute the average continuous ranked probability score (CRPS) for a set of forecast ensembles and the corresponding observations and accumulate the result to the given CRPS object.

Parameters

CRPS
[dict] The CRPS object.

X_f
[array_like] Array of shape (k,m,n,...) containing the values from an ensemble forecast of k members with shape (m,n,...).

X_o
[array_like] Array of shape (m,n,...) containing the observed values corresponding to the forecast.

References

[Her00]

pysteps.verification.probscores.CRPS_compute

pysteps.verification.probscores.**CRPS_compute**(*CRPS*)

Compute the averaged values from the given CRPS object.

Parameters

CRPS
[dict] A CRPS object created with CRPS_init.

Returns

out
[float] The computed CRPS.

pysteps.verification.probscores.reldiag

pysteps.verification.probscores.**reldiag**(*P_f*, *X_o*, *X_min*, *n_bins=10*, *min_count=10*)

Compute the x- and y- coordinates of the points in the reliability diagram.

Parameters

P_f
[array-like] Forecast probabilities for exceeding the intensity threshold specified in the reliability diagram object.

X_o
[array-like] Observed values.

X_min
[float] Precipitation intensity threshold for yes/no prediction.

n_bins
[int] Number of bins to use in the reliability diagram.

min_count
[int] Minimum number of samples required for each bin. A zero value is assigned if the number of samples in a bin is smaller than bin_count.

Returns

out
[tuple] Two-element tuple containing the x- and y-coordinates of the points in the reliability diagram.

pysteps.verification.probscores.reldiag_init

`pysteps.verification.probscores.`**`reldiag_init`**(*X_min*, *n_bins=10*, *min_count=10*)
Initialize a reliability diagram object.

Parameters

X_min
[float] Precipitation intensity threshold for yes/no prediction.

n_bins
[int] Number of bins to use in the reliability diagram.

min_count
[int] Minimum number of samples required for each bin. A zero value is assigned if the number of samples in a bin is smaller than bin_count.

Returns

out
[dict] The reliability diagram object.

References

[BrockerS07]

pysteps.verification.probscores.reldiag_accum

`pysteps.verification.probscores.`**`reldiag_accum`**(*reldiag*, *P_f*, *X_o*)
Accumulate the given probability-observation pairs into the reliability diagram.

Parameters

reldiag
[dict] A reliability diagram object created with reldiag_init.

P_f
[array-like] Forecast probabilities for exceeding the intensity threshold specified in the reliability diagram object.

X_o
[array-like] Observed values.

pysteps.verification.probscores.reldiag_compute

`pysteps.verification.probscores.`**`reldiag_compute`**(*reldiag*)
Compute the x- and y- coordinates of the points in the reliability diagram.

Parameters

reldiag
[dict] A reliability diagram object created with reldiag_init.

Returns

out
[tuple] Two-element tuple containing the x- and y-coordinates of the points in the reliability diagram.

pysteps.verification.probscores.ROC_curve

pysteps.verification.probscores.**ROC_curve**(*P_f*, *X_o*, *X_min*, *n_prob_thrs=10*, *compute_area=False*)

Compute the ROC curve and its area from the given ROC object.

Parameters

P_f
[array_like] Forecasted probabilities for exceeding the threshold specified in the ROC object. Non-finite values are ignored.

X_o
[array_like] Observed values. Non-finite values are ignored.

X_min
[float] Precipitation intensity threshold for yes/no prediction.

n_prob_thrs
[int] The number of probability thresholds to use. The interval [0,1] is divided into n_prob_thrs evenly spaced values.

compute_area
[bool] If True, compute the area under the ROC curve (between 0.5 and 1).

Returns

out
[tuple] A two-element tuple containing the probability of detection (POD) and probability of false detection (POFD) for the probability thresholds specified in the ROC curve object. If compute_area is True, return the area under the ROC curve as the third element of the tuple.

pysteps.verification.probscores.ROC_curve_init

pysteps.verification.probscores.**ROC_curve_init**(*X_min*, *n_prob_thrs=10*)

Initialize a ROC curve object.

Parameters

X_min
[float] Precipitation intensity threshold for yes/no prediction.

n_prob_thrs
[int] The number of probability thresholds to use. The interval [0,1] is divided into n_prob_thrs evenly spaced values.

Returns

out
[dict] The ROC curve object.

pysteps.verification.probscores.ROC_curve_accum

pysteps.verification.probscores.**ROC_curve_accum**(*ROC*, *P_f*, *X_o*)

Accumulate the given probability-observation pairs into the given ROC object.

Parameters

ROC
[dict] A ROC curve object created with ROC_curve_init.

P_f
[array_like] Forecasted probabilities for exceeding the threshold specified in the ROC object. Non-finite values are ignored.

X_o
[array_like] Observed values. Non-finite values are ignored.

pysteps.verification.probscores.ROC_curve_compute

pysteps.verification.probscores.**ROC_curve_compute**(*ROC*, *compute_area=False*)
Compute the ROC curve and its area from the given ROC object.

Parameters

ROC
[dict] A ROC curve object created with ROC_curve_init.

compute_area
[bool] If True, compute the area under the ROC curve (between 0.5 and 1).

Returns

out
[tuple] A two-element tuple containing the probability of detection (POD) and probability of false detection (POFD) for the probability thresholds specified in the ROC curve object. If compute_area is True, return the area under the ROC curve as the third element of the tuple.

pysteps.verification.spatialscores

Skill scores for spatial forecasts.

intensity_scale(X_f, X_o, name, thrs[, ...])	Compute an intensity-scale verification score.
intensity_scale_init(name, thrs[, scales, ...])	Initialize an intensity-scale verification object.
intensity_scale_accum(intscale, X_f, X_o)	Compute and update the intensity-scale verification scores.
intensity_scale_merge(intscale_1, intscale_2)	Merge two intensity-scale verification objects.
intensity_scale_compute(intscale)	Return the intensity scale matrix.
binary_mse(X_f, X_o, thr[, wavelet, ...])	Compute the MSE of the binary error as a function of spatial scale.
binary_mse_init(thr[, wavelet])	Initialize a binary MSE (BMSE) verification object.
binary_mse_accum(bmse, X_f, X_o)	Accumulate forecast-observation pairs to an BMSE object.
binary_mse_merge(bmse_1, bmse_2)	Merge two BMSE objects.
binary_mse_compute(bmse[, return_scales])	Compute the BMSE.
fss(X_f, X_o, thr, scale)	Compute the fractions skill score (FSS) for a deterministic forecast field and the corresponding observation field.
fss_init(thr, scale)	Initialize a fractions skill score (FSS) verification object.
fss_accum(fss, X_f, X_o)	Accumulate forecast-observation pairs to an FSS object.
fss_merge(fss_1, fss_2)	Merge two FSS objects.

Continued on next page

Table 54 – continued from previous page

fss_compute(fss)	Compute the FSS.

pysteps.verification.spatialscores.intensity_scale

pysteps.verification.spatialscores.**intensity_scale**(*X_f*, *X_o*, *name*, *thrs*, *scales=None*, *wavelet='Haar'*)

Compute an intensity-scale verification score.

Parameters

X_f
[array_like] Array of shape (m, n) containing the forecast field.

X_o
[array_like] Array of shape (m, n) containing the verification observation field.

name
[string] A string indicating the name of the spatial verification score to be used:

Name	Description
FSS	Fractions skill score
BMSE	Binary mean squared error

thrs
[float or array_like] Scalar or 1-D array of intensity thresholds for which to compute the verification.

scales
[float or array_like, optional] Scalar or 1-D array of spatial scales in pixels, required if `name="FSS"`.

wavelet
[str, optional] The name of the wavelet function to use in the BMSE. Defaults to the Haar wavelet, as described in Casati et al. 2004. See the documentation of PyWavelets for a list of available options.

Returns

out
[array_like] The two-dimensional array containing the intensity-scale skill scores for each spatial scale and intensity threshold.

See also:

pysteps.verification.spatialscores.binary_mse

pysteps.verification.spatialscores.fss

References

[CRS04], [RL08], [EWW+13]

pysteps.verification.spatialscores.intensity_scale_init

pysteps.verification.spatialscores.**intensity_scale_init**(*name*, *thrs*, *scales=None*, *wavelet='Haar'*)

Initialize an intensity-scale verification object.

Parameters

name
[string] A string indicating the name of the spatial verification score to be used:

Name	Description
FSS	Fractions skill score
BMSE	Binary mean squared error

thrs
[float or array_like] Scalar or 1-D array of intensity thresholds for which to compute the verification.

scales
[float or array_like, optional] Scalar or 1-D array of spatial scales in pixels, required if `name="FSS"`.

wavelet
[str, optional] The name of the wavelet function, required if `name="BMSE"`. Defaults to the Haar wavelet, as described in Casati et al. 2004. See the documentation of PyWavelets for a list of available options.

Returns

out
[dict] The intensity-scale object.

pysteps.verification.spatialscores.intensity_scale_accum

`pysteps.verification.spatialscores.`**`intensity_scale_accum`**(*intscale*, *X_f*, *X_o*)
Compute and update the intensity-scale verification scores.

Parameters

intscale
[dict] The intensity-scale object initialized with *pysteps.verification.spatialscores.intensity_scale_init()*.

X_f
[array_like] Array of shape (m, n) containing the forecast field.

X_o
[array_like] Array of shape (m, n) containing the verification observation field.

pysteps.verification.spatialscores.intensity_scale_merge

`pysteps.verification.spatialscores.`**`intensity_scale_merge`**(*intscale_1*, *intscale_2*)
Merge two intensity-scale verification objects.

Parameters

intscale_1
[dict] Am intensity-scale object initialized with *pysteps.verification.spatialscores.intensity_scale_init()* and populated with *pysteps.verification.spatialscores.intensity_scale_accum()*.

intscale_2
[dict] Another intensity-scale object initialized with *pysteps.verification.spatialscores.intensity_scale_init()* and populated with *pysteps.verification.spatialscores.intensity_scale_accum()*.

Returns

out
[dict] The merged intensity-scale object.

pysteps.verification.spatialscores.intensity_scale_compute

pysteps.verification.spatialscores.**intensity_scale_compute**(*intscale*)
Return the intensity scale matrix.

Parameters

intscale
[dict] The intensity-scale object initialized with *pysteps.verification.spatialscores.intensity_scale_init()* and accumulated with *pysteps.verification.spatialscores.intensity_scale_accum()*.

Returns

out
[array_like] The two-dimensional array of shape (j, k) containing the intensity-scale skill scores for **j** spatial scales and **k** intensity thresholds.

pysteps.verification.spatialscores.binary_mse

pysteps.verification.spatialscores.**binary_mse**(*X_f*, *X_o*, *thr*, *wavelet='haar'*, *return_scales=True*)
Compute the MSE of the binary error as a function of spatial scale.

This method uses PyWavelets for decomposing the error field between the forecasts and observations into multiple spatial scales.

Parameters

X_f
[array_like] Array of shape (m, n) containing the forecast field.

X_o
[array_like] Array of shape (m, n) containing the verification observation field.

thr
[sequence] The intensity threshold for which to compute the verification.

wavelet
[str, optional] The name of the wavelet function to use. Defaults to the Haar wavelet, as described in Casati et al. 2004. See the documentation of PyWavelets for a list of available options.

return_scales
[bool, optional] Whether to return the spatial scales resulting from the wavelet decomposition.

Returns

SS
[array] One-dimensional array containing the binary MSE for each spatial scale.

scales
[list, optional] If `return_scales=True`, return the spatial scales in pixels resulting from the wavelet decomposition.

References

[CRS04]

pysteps.verification.spatialscores.binary_mse_init

pysteps.verification.spatialscores.**binary_mse_init**(*thr*, *wavelet='haar'*)

Initialize a binary MSE (BMSE) verification object.

Parameters

thr
[float] The intensity threshold.

wavelet
[str, optional] The name of the wavelet function to use. Defaults to the Haar wavelet, as described in Casati et al. 2004. See the documentation of PyWavelets for a list of available options.

Returns

bmse
[dict] The initialized BMSE verification object.

pysteps.verification.spatialscores.binary_mse_accum

pysteps.verification.spatialscores.**binary_mse_accum**(*bmse*, *X_f*, *X_o*)

Accumulate forecast-observation pairs to an BMSE object.

Parameters

bmse
[dict] The BMSE object initialized with *pysteps.verification.spatialscores.binary_mse_init()*.

X_f
[array_like] Array of shape (m, n) containing the forecast field.

X_o
[array_like] Array of shape (m, n) containing the observation field.

pysteps.verification.spatialscores.binary_mse_merge

pysteps.verification.spatialscores.**binary_mse_merge**(*bmse_1*, *bmse_2*)

Merge two BMSE objects.

Parameters

bmse_1
[dict] A BMSE object initialized with *pysteps.verification.spatialscores.binary_mse_init()*. and populated with *pysteps.verification.spatialscores.binary_mse_accum()*.

bmse_2
[dict] Another BMSE object initialized with *pysteps.verification.spatialscores.binary_mse_init()*. and populated with *pysteps.verification.spatialscores.binary_mse_accum()*.

Returns

out
[dict] The merged BMSE object.

pysteps.verification.spatialscores.binary_mse_compute

pysteps.verification.spatialscores.**binary_mse_compute**(*bmse*, *return_scales=True*)

Compute the BMSE.

Parameters

bmse
[dict] The BMSE object initialized with *pysteps.verification.spatialscores.binary_mse_init()* and accumulated with *pysteps.verification.spatialscores.binary_mse_accum()*.

return_scales
[bool, optional] Whether to return the spatial scales resulting from the wavelet decomposition.

Returns

BMSE
[array_like] One-dimensional array containing the binary MSE for each spatial scale.

scales
[list, optional] If `return_scales=True`, return the spatial scales in pixels resulting from the wavelet decomposition.

pysteps.verification.spatialscores.fss

pysteps.verification.spatialscores.**fss**(*X_f*, *X_o*, *thr*, *scale*)

Compute the fractions skill score (FSS) for a deterministic forecast field and the corresponding observation field.

Parameters

X_f
[array_like] Array of shape (m, n) containing the forecast field.

X_o
[array_like] Array of shape (m, n) containing the observation field.

thr
[float] The intensity threshold.

scale
[int] The spatial scale in pixels. In practice, the scale represents the size of the moving window that it is used to compute the fraction of pixels above the threshold.

Returns

out
[float] The fractions skill score between 0 and 1.

References

[RL08], [EWW+13]

pysteps.verification.spatialscores.fss_init

pysteps.verification.spatialscores.**fss_init**(*thr*, *scale*)

Initialize a fractions skill score (FSS) verification object.

Parameters

thr
[float] The intensity threshold.

scale
[float] The spatial scale in pixels. In practice, the scale represents the size of the moving window that it is used to compute the fraction of pixels above the threshold.

Returns

fss
[dict] The initialized FSS verification object.

pysteps.verification.spatialscores.fss_accum

pysteps.verification.spatialscores.**fss_accum**(*fss*, *X_f*, *X_o*)
Accumulate forecast-observation pairs to an FSS object.

Parameters

fss
[dict] The FSS object initialized with *pysteps.verification.spatialscores.fss_init()*.

X_f
[array_like] Array of shape (m, n) containing the forecast field.

X_o
[array_like] Array of shape (m, n) containing the observation field.

pysteps.verification.spatialscores.fss_merge

pysteps.verification.spatialscores.**fss_merge**(*fss_1*, *fss_2*)
Merge two FSS objects.

Parameters

fss_1
[dict] A FSS object initialized with *pysteps.verification.spatialscores.fss_init()*. and populated with *pysteps.verification.spatialscores.fss_accum()*.

fss_2
[dict] Another FSS object initialized with *pysteps.verification.spatialscores.fss_init()*. and populated with *pysteps.verification.spatialscores.fss_accum()*.

Returns

out
[dict] The merged FSS object.

pysteps.verification.spatialscores.fss_compute

pysteps.verification.spatialscores.**fss_compute**(*fss*)
Compute the FSS.

Parameters

fss
[dict] An FSS object initialized with *pysteps.verification.spatialscores.fss_init()* and accumulated with *pysteps.verification.spatialscores.fss_accum()*.

Returns

out
[float] The computed FSS value.

2.2.13 pysteps.visualization

Methods for plotting precipitation and motion fields.

pysteps.visualization.animations

Functions to produce animations for pysteps.

animate(R_obs[, nloops, timestamps, R_fct, ...])	Function to animate observations and forecasts in pysteps.

pysteps.visualization.animations.animate

pysteps.visualization.animations.**animate**(*R_obs*, *nloops=2*, *timestamps=None*, *R_fct=None*, *timestep_min=5*, *UV=None*, *motion_plot='quiver'*, *geodata=None*, *map=None*, *colorscale='pysteps'*, *units='mm/h'*, *colorbar=True*, *type='ensemble'*, *prob_thr=None*, *plotanimation=True*, *savefig=False*, *fig_dpi=150*, *fig_format='png'*, *path_outputs=''*, ***kwargs*)

Function to animate observations and forecasts in pysteps.

Parameters

R_obs
[array-like] Three-dimensional array containing the time series of observed precipitation fields.

Returns

ax
[fig axes] Figure axes. Needed if one wants to add e.g. text inside the plot.

Other Parameters

nloops
[int] Optional, the number of loops in the animation.

R_fct
[array-like] Optional, the three or four-dimensional (for ensembles) array containing the time series of forecasted precipitation field.

timestep_min
[float] The time resolution in minutes of the forecast.

UV
[array-like] Optional, the motion field used for the forecast.

motion_plot
[string] The method to plot the motion field.

geodata
[dictionary] Optional dictionary containing geographical information about the field. If geodata is not None, it must contain the following key-value pairs:

Key	Value
projection	PROJ.4-compatible projection definition
x1	x-coordinate of the lower-left corner of the data raster
y1	y-coordinate of the lower-left corner of the data raster
x2	x-coordinate of the upper-right corner of the data raster
y2	y-coordinate of the upper-right corner of the data raster
yorigin	a string specifying the location of the first element in the data raster w.r.t. y-axis: 'upper' = upper border, 'lower' = lower border

map
[str] Optional method for plotting a map. See pysteps.visualization.precipifields.plot_precip.field.

units
[str] Units of the input array (mm/h or dBZ)

colorscale
[str] Which colorscale to use.

title
[str] If not None, print the title on top of the plot.

colorbar
[bool] If set to True, add a colorbar on the right side of the plot.

type
[{'ensemble', 'mean', 'prob'}, str] Type of the map to animate. 'ensemble' = ensemble members, 'mean' = ensemble mean, 'prob' = exceedance probability (using threshold defined in prob_thrs).

prob_thr
[float] Intensity threshold for the exceedance probability maps. Applicable if type = 'prob'.

plotanimation
[bool] If set to True, visualize the animation (useful when one is only interested in saving the individual frames).

savefig
[bool] If set to True, save the individual frames to path_outputs.

fig_dpi
[scalar > 0] Resolution of the output figures, see the documentation of matplotlib.pyplot.savefig. Applicable if savefig is True.

path_outputs
[string] Path to folder where to save the frames.

kwargs
[dict] Optional keyword arguments that are supplied to plot_precip_field and quiver/streamplot.

pysteps.visualization.motionfields

Functions to plot motion fields.

quiver(UV[, ax, map, geodata, ...])	Function to plot a motion field as arrows.
streamplot(UV[, ax, map, geodata, ...])	Function to plot a motion field as streamlines.

pysteps.visualization.motionfields.quiver

pysteps.visualization.motionfields.**quiver**(*UV*, *ax=None*, *map=None*, *geodata=None*, *drawlonlatlines=False*, *basemap_resolution='l'*, *cartopy_scale='50m'*, *lw=0.5*, *cartopy_subplot=(1, 1, 1)*, *axis='on'*, ***kwargs*)

Function to plot a motion field as arrows.

Parameters

UV
[array-like] Array of shape (2,m,n) containing the input motion field.

ax
[axis object] Optional axis object to use for plotting.

map
[{'basemap', 'cartopy'}, optional] Optional method for plotting a map: 'basemap' or 'cartopy'. The former uses mpl_toolkits.basemap, while the latter uses cartopy.

geodata
[dictionary] Optional dictionary containing geographical information about the field. If geodata is not None, it must contain the following key-value pairs:

Key	Value
projection	PROJ.4-compatible projection definition
x1	x-coordinate of the lower-left corner of the data raster
y1	y-coordinate of the lower-left corner of the data raster
x2	x-coordinate of the upper-right corner of the data raster
y2	y-coordinate of the upper-right corner of the data raster
yorigin	location of the first element in the data raster element in the data raster w.r.t. y-axis: 'upper' = upper border, 'lower' = lower border

drawlonlatlines
[bool, optional] If set to True, draw longitude and latitude lines. Applicable if map is 'basemap' or 'cartopy'.

basemap_resolution
[str, optional] The resolution of the basemap, see the documentation of mpl_toolkits.basemap. Applicable if map is 'basemap'.

cartopy_scale
[{'10m', '50m', '110m'}, optional] The scale (resolution) of the map. The available options are '10m', '50m', and '110m'. Applicable if map is 'cartopy'.

lw: float, optional
Linewidth of the map (administrative boundaries and coastlines).

cartopy_subplot
[tuple or SubplotSpec instance, optional] Cartopy subplot. Applicable if map is 'cartopy'.

axis
[{'off','on'}, optional] Whether to turn off or on the x and y axis.

Returns

out
[axis object] Figure axes. Needed if one wants to add e.g. text inside the plot.

Other Parameters

step
[int] Optional resample step to control the density of the arrows. Default : 20

color
[string] Optional color of the arrows. This is a synonym for the PolyCollection facecolor kwarg in matplotlib.collections. Default : black

pysteps.visualization.motionfields.streamplot

pysteps.visualization.motionfields.**streamplot**(*UV*, *ax=None*, *map=None*, *geodata=None*, *drawlonlatlines=False*, *basemap_resolution='l'*, *cartopy_scale='50m'*, *lw=0.5*, *cartopy_subplot=(1, 1, 1)*, *axis='on'*, ***kwargs*)

Function to plot a motion field as streamlines.

Parameters

UV
[array-like] Array of shape (2, m,n) containing the input motion field.

ax
[axis object] Optional axis object to use for plotting.

map
[{'basemap', 'cartopy'}, optional] Optional method for plotting a map: 'basemap' or 'cartopy'. The former uses mpl_toolkits.basemap, while the latter uses cartopy.

geodata
[dictionary] Optional dictionary containing geographical information about the field. If geodata is not None, it must contain the following key-value pairs:

Key	Value
projection	PROJ.4-compatible projection definition
x1	x-coordinate of the lower-left corner of the data raster
y1	y-coordinate of the lower-left corner of the data raster
x2	x-coordinate of the upper-right corner of the data raster
y2	y-coordinate of the upper-right corner of the data raster
yorigin	location of the first element in the data raster element in the data raster w.r.t. y-axis: 'upper' = upper border, 'lower' = lower border

drawlonlatlines
[bool, optional] If set to True, draw longitude and latitude lines. Applicable if map is

'basemap' or 'cartopy'.

basemap_resolution
[str, optional] The resolution of the basemap, see the documentation of mpl_toolkits.basemap. Applicable if map is 'basemap'.

cartopy_scale
[{'10m', '50m', '110m'}, optional] The scale (resolution) of the map. The available options are '10m', '50m', and '110m'. Applicable if map is 'cartopy'.

lw: float, optional
Linewidth of the map (administrative boundaries and coastlines).

cartopy_subplot
[tuple or SubplotSpec instance, optional] Cartopy subplot. Applicable if map is 'cartopy'.

axis
[{'off','on'}, optional] Whether to turn off or on the x and y axis.

Returns

out
[axis object] Figure axes. Needed if one wants to add e.g. text inside the plot.

Other Parameters

density
[float] Controls the closeness of streamlines. Default : 1.5

color
[string] Optional streamline color. This is a synonym for the PolyCollection facecolor kwarg in matplotlib.collections. Default : black

pysteps.visualization.precipfields

Methods for plotting precipitation fields.

plot_precip_field(R[, type, map, geodata, ...])	Function to plot a precipitation intensity or probability field with a colorbar.
get_colormap(type[, units, colorscale])	Function to generate a colormap (cmap) and norm.

pysteps.visualization.precipfields.plot_precip_field

pysteps.visualization.precipfields.**plot_precip_field**(*R*, *type='intensity'*, *map=None*, *geodata=None*, *units='mm/h'*, *bbox=None*, *colorscale='pysteps'*, *probthr=None*, *title=None*, *colorbar=True*, *drawlonlatlines=False*, *lw=0.5*, *axis='on'*, *cax=None*, ***kwargs*)

Function to plot a precipitation intensity or probability field with a colorbar.

Parameters

R
[array-like] Two-dimensional array containing the input precipitation field or an ex-

ceedance probability map.

type
[{'intensity', 'depth', 'prob'}, optional] Type of the map to plot: 'intensity' = precipitation intensity field, 'depth' = precipitation depth (accumulation) field, 'prob' = exceedance probability field.

map
[{'basemap', 'cartopy'}, optional] Optional method for plotting a map: 'basemap' or 'cartopy'. The former uses mpl_toolkits.basemap, while the latter uses cartopy.

geodata
[dictionary, optional] Optional dictionary containing geographical information about the field. Required is map is not None.

If geodata is not None, it must contain the following key-value pairs:

Key	Value
projection	PROJ.4-compatible projection definition
x1	x-coordinate of the lower-left corner of the data raster
y1	y-coordinate of the lower-left corner of the data raster
x2	x-coordinate of the upper-right corner of the data raster
y2	y-coordinate of the upper-right corner of the data raster
yorigin	a string specifying the location of the first element in the data raster w.r.t. y-axis: 'upper' = upper border, 'lower' = lower border

units
[{'mm/h', 'mm', 'dBZ'}, optional] Units of the input array. If type is 'prob', this specifies the unit of the intensity threshold.

bbox
[tuple, optional] Four-element tuple specifying the coordinates of the bounding box. Use this for plotting a subdomain inside the input grid. The coordinates are of the form (lower left x,lower left y,upper right x,upper right y). If map is not None, the x- and y-coordinates are longitudes and latitudes. Otherwise they represent image pixels.

colorscale
[{'pysteps', 'STEPS-BE', 'BOM-RF3'}, optional] Which colorscale to use. Applicable if units is 'mm/h', 'mm' or 'dBZ'.

probthr
[float, optional] Intensity threshold to show in the color bar of the exceedance probability map. Required if type is "prob" and colorbar is True.

title
[str, optional] If not None, print the title on top of the plot.

colorbar
[bool, optional] If set to True, add a colorbar on the right side of the plot.

drawlonlatlines
[bool, optional] If set to True, draw longitude and latitude lines. Applicable if map is 'basemap' or 'cartopy'.

lw: float, optional
Linewidth of the map (administrative boundaries and coastlines).

axis
[{'off','on'}, optional] Whether to turn off or on the x and y axis.

cax
[Axes object, optional] Axes into which the colorbar will be drawn. If no axes is provided the colorbar axes are created next to the plot.

Returns

ax
[fig Axes] Figure axes. Needed if one wants to add e.g. text inside the plot.

Other Parameters

Optional parameters are contained in **kwargs. See basemaps.plot_geography.

pysteps.visualization.precipfields.get_colormap

pysteps.visualization.precipfields.**get_colormap**(*type*, *units='mm/h'*, *colorscale='pysteps'*)

Function to generate a colormap (cmap) and norm.

Parameters

type
[{'intensity', 'depth', 'prob'}, optional] Type of the map to plot: 'intensity' = precipitation intensity field, 'depth' = precipitation depth (accumulation) field, 'prob' = exceedance probability field.

units
[{'mm/h', 'mm', 'dBZ'}, optional] Units of the input array. If type is 'prob', this specifies the unit of the intensity threshold.

colorscale
[{'pysteps', 'STEPS-BE', 'BOM-RF3'}, optional] Which colorscale to use. Applicable if units is 'mm/h', 'mm' or 'dBZ'.

Returns

cmap
[Colormap instance] colormap

norm
[colors.Normalize object] Colors norm

clevs: list(float)
List of precipitation values defining the color limits.

clevsStr: list(str)
List of precipitation values defining the color limits (with correct number of decimals).

pysteps.visualization.spectral

Methods for plotting Fourier spectra.

plot_spectrum1d(fft_freq, fft_power[, ...])	Function to plot in log-log a radially averaged Fourier spectrum.

pysteps.visualization.spectral.plot_spectrum1d

pysteps.visualization.spectral.**plot_spectrum1d**(*fft_freq*, *fft_power*, *x_units=None*, *y_units=None*, *wavelength_ticks=None*, *color='k'*, *lw=1.0*, *label=None*, *ax=None*, ***kwargs*)

Function to plot in log-log a radially averaged Fourier spectrum.

Parameters

fft_freq: array-like
1d array containing the Fourier frequencies computed with the function *pysteps.utils.spectral.rapsd()*.

fft_power: array-like
1d array containing the radially averaged Fourier power spectrum computed with the function *pysteps.utils.spectral.rapsd()*.

x_units: str, optional
Units of the X variable (distance, e.g. "km").

y_units: str, optional
Units of the Y variable (amplitude, e.g. "dBR").

wavelength_ticks: array-like, optional
List of wavelengths where to show xticklabels.

color: str, optional
Line color.

lw: float, optional
Line width.

label: str, optional
Label (for legend).

ax: Axes, optional
Plot axes.

Returns

ax: Axes
Plot axes

pysteps.visualization.utils

Miscellaneous utility functions for the visualization module.

parse_proj4_string(proj4str)	Construct a dictionary from a PROJ.4 projection string.
proj4_to_basemap(proj4str)	Convert a PROJ.4 projection string into a dictionary that can be expanded as keyword arguments to mpl_toolkits.basemap.Basemap.__init__.
proj4_to_cartopy(proj4str)	Convert a PROJ.4 projection string into a Cartopy coordinate reference system (crs) object.
reproject_geodata(geodata, t_proj4str[, ...])	Reproject geodata and optionally create a grid in a new projection.

pysteps.visualization.utils.parse_proj4_string

pysteps.visualization.utils.**parse_proj4_string**(*proj4str*)
Construct a dictionary from a PROJ.4 projection string.

Parameters

proj4str
[str] A PROJ.4-compatible projection string.

Returns

out
[dict] Dictionary, where keys and values are parsed from the projection parameter tokens beginning with '+'.

pysteps.visualization.utils.proj4_to_basemap

pysteps.visualization.utils.**proj4_to_basemap**(*proj4str*)

Convert a PROJ.4 projection string into a dictionary that can be expanded as keyword arguments to mpl_toolkits.basemap.Basemap.__init__.

Parameters

proj4str
[str] A PROJ.4-compatible projection string.

Returns

out
[dict] The output dictionary.

pysteps.visualization.utils.proj4_to_cartopy

pysteps.visualization.utils.**proj4_to_cartopy**(*proj4str*)

Convert a PROJ.4 projection string into a Cartopy coordinate reference system (crs) object.

Parameters

proj4str
[str] A PROJ.4-compatible projection string.

Returns

out
[object] Instance of a crs class defined in cartopy.crs.

pysteps.visualization.utils.reproject_geodata

pysteps.visualization.utils.**reproject_geodata**(*geodata*, *t_proj4str*, *return_grid=None*)

Reproject geodata and optionally create a grid in a new projection.

Parameters

geodata
[dictionary] Dictionary containing geographical information about the field. It must contain the attributes projection, x1, x2, y1, y2, xpixelsize, ypixelsize, as defined in the documentation of pysteps.io.importers.

t_proj4str: str
The target PROJ.4-compatible projection string (fallback).

return_grid
[{None, 'coords', 'quadmesh'}, optional] Whether to return the coordinates of the projected grid. The default return_grid=None does not compute the grid, return_grid='coords' returns the centers of projected grid points, return_grid='quadmesh' returns the coordinates of the quadrilaterals (e.g. to be used by pcolormesh).

Returns

geodata
: [dictionary] Dictionary containing the reprojected geographical information and optionally the required X_grid and Y_grid.

  It also includes a fixed boolean attribute regular_grid=False to indicate that the reprojected grid has no regular spacing.

2.3 pySTEPS developer guide

In this section you can find a series of guidelines and tutorials for contributing to the pySTEPS project.

2.3.1 Contributing to Pysteps

Welcome! pySTEPS is a community-driven initiative for developing and maintaining an easy to use, modular, free and open source Python framework for short-term ensemble prediction systems.

If you haven't already, take a look at the project's README.rst file and the pysteps documentation.

There are many ways to contribute to pysteps:

- contributing bug reports and feature requests
- contributing documentation
- code contributions new features or bug fixes
- contribute with usage examples

Our main forum for discussion is the project's GitHub issue tracker. This is the right place to start a discussion, report a bug, or request a new feature.

Workflow for code contributions

We welcome all kind of contributions, from documentation updates, a bug fix, or a new feature. If your new feature will take a lot of work, we recommend creating an issue with the **enhancement** tag to encourage discussions.

We use the usual GitHub pull-request flow, which may be familiar to you if you've contributed to other projects on GitHub.

First Time Contributors

If you are interested in helping to improve pysteps, the best way to get started is by looking for "Good First Issue" in the issue tracker.

In a nutshell, the main steps to follow for contributing to pysteps are:

- Setting up the development environment
- Fork the repository
- Install pre-commit hooks
- Create a new branch for each contribution
- Read the Code Style guide
- Work on your changes
- Test your changes
- Push to your fork repository and create a new PR in GitHub.

Setting up the Development environment

The recommended way to setup up the developer environment is the Anaconda (commonly referred as Conda). Conda quickly installs, runs and updates packages and their dependencies. It also allows to easily create, save, load and switch between different environments on your local computer.

The developer environment can be created using the environment_dev.yml file in the project's root directory running the command:

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

This will create the **pysteps_dev** environment that can be activated using:

```
conda activate pysteps_dev
```

Once the environment is created, the package can be installed in development mode, in such a way that the project appears to be installed, but yet is still editable from the source tree. See instructions in the *Installing pysteps* section.

Fork the repository

Once you have set the development environment, the next step is creating your local copy of the repository where you will commit your modifications. The steps to follow are:

1. Set up Git in your computer.
2. Create a GitHub account (if you don't have one).
3. Fork the repository in your GitHub.
4. Clone local copy of your fork. For example:

   ```
   git clone https://github.com/<your-account>/pysteps.git
   ```

Done!, now you have a local copy of pysteps git repository. If you are new to GitHub, below you can find a list of useful tutorials:

- http://rogerdudler.github.io/git-guide/index.html
- https://www.atlassian.com/git/tutorials

Install pre-commit hooks

Once you cloned the repository and the development environment is installed (and active), install the git pre-commit hook by executing the following command in the repository's root:

```
pre-commit install
```

The pre-commit hooks are scripts that are executed in every commit automatically to identify simple issues with the code. If an issue is identified (the pre-commit script exits with non-zero status), the hook aborts the commit and print the error. Currently, Pysteps only test the code to be committed complies with black's autoformat. In case that the commit is aborted, you only need to run black in the entire source code. This can be done by running `black .` or `pre-commit run --all-files`. The latter is recommended since it indicates if the commit contained any formatting errors (that are automatically corrected). To ensure that the same configuration is used in every development environment, black's configuration is stored in the *pyproject.toml* file. This configuration is automatically load when black is run from any directory in the pysteps project.

For more information about git hooks and the pre-commit package see:

- https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
- https://pre-commit.com/

Create a new branch

As a collaborator, all the new contributions that you want should be done in a new branch under your forked repository. Working on the master branch is reserved for Core Contributors only. Core Contributors are developers that actively work and maintain the repository. They are the only ones who accept pull requests and push commits directly to the pysteps repository.

For more information on how to create and work with branches, see "Branches in a Nutshell" in the Git documentation

Code Style

We strongly suggest to follow the PEP8 coding standards. Note that this is not strictly enforced yet, since many source files in pysteps are not PEP8 compliant. However, we encourage new contributions to be compliant at least with the coding style summarized next.

Coding style summary

For quick reference, these are the most important good coding practices to follow:

- Always use 4 spaces for indentation (don't use tabs).
- Write UTF-8 (add **# -*- coding: utf-8 -*-** at the top of each file).
- Max line-length: 88 characters (note that we don't use the PEP8's 79 value).
- Always indent wrapped code for readability.
- Avoid extraneous whitespace.
- Don't use whitespace to line up assignment operators (=, :).
- Spaces around = for assignment.
- No spaces around = for default parameter values (keywords).
- Spaces around mathematical operators, but group them sensibly.
- No multiple statements on the same line.
- Naming conventions:

  Function names, variable names, and filenames should be descriptive and self explanatory. Avoid using abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word. Avoid single letter variables if possible and use more verbose names for clarity. An exception for this are indexes in loops (*i, j, k, etc*).

  The following table summarizes the conventions:

  Source: Google's python style guide

- Ignore returned variables:

  If you need to ignore part to the variables returned by a function, use "_" (single underscore) or __ (double underscore):

```
precip, __, metadata = import_bom_rf3('example_file.bom')
precip2, _, metadata2 = import_bom_rf3('example_file2.bom')
```

- Zen of Python (PEP 20), the guiding principles for Python's design:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

(continues on next page)

(continued from previous page)

```
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

For a detailed description of a pythonic code style check these guidelines:

- The Hitchhiker's Guide to Python
- Google's python style guide
- PEP8

Auto-formatters

Hand-formatting code according the PEP8 style guide can be a tedious process if it is done manually. To make our lives easy, in pysteps, we use black to auto-format the code using its default configuration. Black can be installed using any of the following:

```
conda install black

#For the latest version:
conda install -c conda-forge black

pip install black
```

Check the official documentation for more information.

Docstrings

Every module, function, or class must have a docstring that describe its purpose and how to use it, following the conventions described in the PEP 257 and the Numpy's docstrings format.

Here is a summary of the most important rules:

- One-line docstrings Triple quotes are used even though the string fits on one line. This makes it easy to later expand it.
- A one-line docstring is a phrase ending in a period.
- All docstrings should be written in imperative (“”“Return some value.”“”) mood rather than descriptive mood (“”“Returns some value.”“”).

Here is an example of a docstring:

```
def adjust_lag2_corrcoef1(gamma_1, gamma_2):
    """A simple adjustment of lag-2 temporal autocorrelation coefficient to
    ensure that the resulting AR(2) process is stationary when the parameters
    are estimated from the Yule-Walker equations.

    Parameters
    ----------
    gamma_1 : float
```

(continues on next page)

(continued from previous page)

```
    Lag-1 temporal autocorrelation coeffient.
  gamma_2 : float
    Lag-2 temporal autocorrelation coeffient.

  Returns
  -------
  out : float
    The adjusted lag-2 correlation coefficient.
  """
```

Working on changes

IMPORTANT

If your changes will take a significant amount of work, we highly recommend opening an issue first, explaining what you want to do and why. It is better to start the discussions early, in case other contributors disagree with what you would like to do or have ideas that will help you do it.

Collaborators guidelines

As a collaborator, all the new contributions that you want should be done in a new branch under your forked repository. Working on the master branch is reserved for Core Contributors only. Core Contributors are developers that actively work and maintain the repository. They are the only ones who accept pull requests and push commits directly to the **pysteps** repository.

To include the contributions for collaborators, we use the usual GitHub pull-request flow. In their simplest form, pull requests are a mechanism for a collaborator to notify to the pysteps project about a completed feature".

Once your proposed changes are ready, you need to create a pull request via your GitHub account. Afterward, the core developers review the code and merge it into the master branch. Be aware that pull requests are more than just a notification, they are also an excellent place for discussing the proposed feature. If there is any problem with the changes, the other project collaborators can post feedback and the author of the commit can even fix the problems by pushing follow-up commits to feature branch.

Do not squash your commits after you have submitted a pull request, as this erases context during the review. The commits will be squashed when the pull request is merged.

To keep you forked repository clean, we suggest deleting branches for once the Pull Requests (PRs) are accepted and merged.

Once you've created a pull request, you can push commits from your topic branch to add them to your existing pull request. These commits will appear in chronological order within your pull request and the changes will be visible in the "Files changed" tab.

Other contributors can review your proposed changes, add review comments, contribute to the pull request discussion, and even add commits to the pull request.

Important: each PR should should only address a single objective (e.g. fix a bug, improve documentation, etc). Pushing changes to an open PR that are outside its objective are highly discouraged. Under this circumstances, the recommended way to proceed is creating a new PR for changes, clearly explaining their goal.

Testing your changes

Before committing changes or creating pull requests, check that the all the tests in the pysteps suite pass. See the *Testing pySTEPS* for the instruction to run the tests.

Although it is not strictly needed, we suggest creating minimal tests for new contributions to ensure that it achieves the desired behavior. Pysteps uses the pytest framework, that it is easy to use and also supports complex functional testing for applications and libraries. Check the pytests official documentation for more information.

The tests should be placed under the pysteps.tests module. The file should follow the **test_*.py** naming convention and have a descriptive name.

A quick way to get familiar with the pytest syntax and the testing procedures is checking the python scripts present in the pysteps test module.

Core developer guidelines

Working directly on the master branch is discouraged and is reserved only for small changes and updates that do not compromise the stability of the code. The *master* branch is a production branch that is ready to be deployed (cloned, installed, and ready to use). In consequence, this master branch is meant to be stable.

The pysteps repository uses a Travis CI, a Continuous Integration service that automatically runs a series of tests every time you commit to GitHub. In that way, your modifications along with the entire package is tested.

Pushing untested or work-in-progress changes to the master branch can potentially introduce bugs or brake the stability of the package. Since the tests takes around 10 minutes and the are run after the commit was pushed, any errors introduced in that commit will be noticed after the stable in the master branch was compromised. In addition, other developers start working on a new feature from master, they may start a potentially broken state.

Instead, it is recommended to work on each new feature in its own branch, which can be pushed to the central repository for backup/collaboration. When you're done with the development work on the feature, then you can merge the feature branch into the master or submit a Pull Request. This approach has two main advantages:

- Every commit on the feature branch is tested using Travis CI. If the tests fail, they do not affect the **master** branch.
- Once the new feature, improvement, or bug correction is finished and the all tests passed, the commits history can be squashed into a single commit and then merged into the master branch.

This helps approach helps to keep the commits history clean and allows experimentation in the branch without compromising the stability of the package.

Processing pull requests

Core developers should follow these rules when processing pull requests:

- Always wait for tests to pass before merging PRs.
- Use "Squash and merge" to merge PRs.
- Delete branches for merged PRs (by core devs pushing to the main repo).
- Edit the final commit message before merging to conform to the following style to help having a clean *git log* output:
  - When merging a multi-commit PR make sure that the commit message doesn't contain the local history from the committer and the review history from the PR. Edit the message to only describe the end state of the PR.
  - Make sure there is a *single* newline at the end of the commit message. This way there is a single empty line between commits in *git log* output.
  - Split lines as needed so that the maximum line length of the commit message is under 80 characters, including the subject line.
  - Capitalize the subject and each paragraph.
  - Make sure that the subject of the commit message has no trailing dot.
  - Use the imperative mood in the subject line (e.g. "Fix typo in README").
  - If the PR fixes an issue, make sure something like "Fixes #xxx." occurs in the body of the message (not in the subject).

Preparing a new release

Core developers should follow the steps to prepare a new release (version):

1. Before creating the actual release in GitHub, be sure that every item in the following checklist was followed:
   - In the file setup.py, update the **version="X.X.X"** keyword in the setup function.
   - Update the version in PKG-INFO file.

- If new dependencies were added to pysteps since the last release, add them to the **environment.yml, requirements.txt**, and **requirements_dev.txt** files.

2. Create a new release in GitHub following these guidelines. Include a detailed changelog in the release.
3. Generating the source distribution for new pysteps version and upload it to the Python Package Index (PyPI). See *Packaging the pysteps project* for a detailed description of this process.
4. Update the conda-forge pysteps-feedstock following this guidelines: *Updating the conda-forge pysteps-feedstock*

Credits

This documents was based in contributors guides of two Python open source projects:

- Py-Art: Copyright (c) 2013, UChicago Argonne, LLC. License.
- mypy: Copyright (c) 2015-2016 Jukka Lehtosalo and contributors. MIT License.
- Official github documentation (https://help.github.com)

2.3.2 Testing pySTEPS

The pysteps distribution includes a small test suite for some of the modules. To run the tests the pytest package is needed. To install it, in a terminal run:

```
pip install pytest
```

Automatic testing

The simplest way to run the pysteps' test suite is using tox and the tox-conda plugin (conda needed). To install these packages activate your conda development environment and run:

```
conda install -c conda-forge tox tox-conda
```

Then, to run the tests, from the repo's root run:

```
tox             # Run pytests
tox -e install  # Test package installation
tox -e black    # Test for black formatting warnings
```

Manual testing

Example data

The pySTEPS build-in tests require the pySTEPS example data installed. See the installation instructions in the *Installing the Example data* section.

Test an installed package

After the package is installed, you can launch the test suite from any directory by running:

```
pytest --pyargs pysteps
```

Test from sources

Before testing the package directly from the sources, we need to build the extensions in-place. To do that, from the root pysteps folder run:

```
python setup.py build_ext -i
```

Now, the package sources can be tested in-place using the **pytest** command on the root of the pysteps source directory. E.g.:

```
pytest -v --tb=line
```

2.3.3 Building the docs

The pysteps documentations is build using Sphinx, a tool that makes it easy to create intelligent and beautiful documentation

The documentation is located in the **doc** folder in the pysteps repo.

Automatic build

The simplest way to build the documentation is using tox and the tox-conda plugin (conda needed). To install these packages activate your conda development environment and run:

```
conda install -c conda-forge tox tox-conda
```

Then, to build the documentation, from the repo's root run:

```
`tox -e docs`
```

This will create a conda environment will all the necessary dependencies and the data needed to create the examples.

Manual build

To build the docs you need to need to satisfy a few more dependencies related to Sphinx that are specified in the doc/requirements.txt file:

- sphinx
- numpydoc
- sphinxcontrib.bibtex
- sphinx_rtd_theme
- sphinx_gallery

You can install these packages running *pip install -r doc/requirements.txt*.

In addition to this requirements, to build the example gallery in the documentation the example pysteps-data is needed. To download and install this data see the installation instructions in the *Installing the Example data* section.

Once these requirements are met, to build the documentation, in the **doc** folder run:

```
make html
```

This will build the documentation along with the example gallery.

The build documentation (html web page) will be available in **doc/_build/html/**. To correctly visualize the documentation, you need to set up and run a local HTTP server. To do that, in the **doc/_build/html/** directory run:

```
python -m http.server
```

This will set up a local HTTP server on 0.0.0.0 port 8000. To see the built documentation open the following url in the browser: http://0.0.0.0:8000/

2.3.4 Packaging the pysteps project

The Python Package Index (PyPI) is a software repository for the Python programming language. PyPI helps you find and install software developed and shared by the Python community.

The following guide to package pysteps was adapted from the PyPI official documentation.

Generating the source distribution

The first step is to generate a source distribution (sdist) for the pysteps library. These are archives that are uploaded to the Package Index and can be installed by pip.

To create the sdist package we need the **setuptools** package installed.

Then, from the root folder of the pysteps source run:

```
python setup.py sdist
```

Once this command is completed, it should generate a tar.gz (source archive) file the **dist** directory:

```
dist/
  pysteps-a.b.c.tar.gz
```

where a.b.c denote the version number.

Uploading the source distribution to the archive

The last step is to upload your package to the Python Package Index.

Important

Before we actually upload the distribution to the Python Index, we will test it in Test PyPI. Test PyPI is a separate instance of the package index that allows us to try the distribution without affecting the real index (PyPi). Because TestPyPI has a separate database from the actual PyPI, you'll need a separate user account for specifically for TestPyPI. You can register your account in https://test.pypi.org/account/register/.

Once you are registered, you can use twine to upload the distribution packages. Alternatively, the package can be uploaded manually from the **Test PyPI** page.

If Twine is not installed, you can install it by running `pip install twine` or `conda install twine`.

Test PyPI

To upload the recently created source distribution (**dist/pysteps-a.b.c.tar.gz**) under the **dist** directory run:

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/pysteps-a.b.c.tar.
↪gz
```

where a.b.c denote the version number.

You will be prompted for the username and password you registered with Test PyPI. After the command completes, you should see output similar to this:

```
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: [your username]
Enter your password:
Uploading pysteps-a.b.c.tar.gz
100%|| 4.25k/4.25k [00:01<00:00, 3.05kB/s]
```

Once uploaded your package should be viewable on TestPyPI, for example, https://test.pypi.org/project/pysteps

Test the uploaded package

Before uploading the package to the official Python Package Index, test that the package can be installed using pip.

Automatic test

The simplest way to hat the package can be installed using pip is using tox and the tox-conda plugin (conda needed). To install these packages activate your conda development environment and run:

```
conda install -c conda-forge tox tox-conda
```

Then, to test the installation in a minimal and an environment with all the dependencies (full env), run:

```
tox -r -e pypi_test         # Test the installation in a minimal env
tox -r -e pypi_test_full    # Test the installation in an full env
```

Manual test

To manually test the installation on new environment, create a copy of the basic development environment using the environment_dev.yml file in the root folder of the pysteps project:

```
conda env create -f environment_dev.yml -n pysteps_test
```

Then we activate the environment:

```
source activate pysteps_test
```

or:

```
conda activate pysteps_test
```

If the environment pysteps_test was already created, remove any version of pysteps already installed:

```
pip uninstall pysteps
```

Now, install the pysteps package from test.pypi.org. Since not all the dependecies are available in the Test PyPI repository, we need to add the official repo as an extra index to pip. By doing so, pip will look first in the Test PyPI index and then in the official PyPI:

```
pip install --no-cache-dir --index-url https://test.pypi.org/simple/  --extra-
↪index-url=https://pypi.org/simple/ pysteps
```

To test that the installation was successful, from a folder different than the pysteps source, run:

```
pytest --pyargs pysteps
```

If any test didn't pass, check the sources or consider creating a new release fixing those bugs.

Upload package to PyPi

Once the sdist package was tested, we can safely upload it to the Official PyPi repository with:

```
twine upload dist/pysteps-a.b.c.tar.gz
```

Now, **pysteps** can be installed by simply running:

```
pip install pysteps
```

As an extra sanity measure, it is recommended to test the pysteps package installed from the Official PyPi repository (instead of the test PyPi).

Automatic test

Similarly to the *Test the uploaded package* section, to test the installation from PyPI in a clean environment, run:

```
tox -r -e pypi
```

Manual test

Follow test instructions in Test PyPI section.

2.3.5 Updating the conda-forge pysteps-feedstock

Here we will describe the steps to update the pysteps conda-forge feedstock. This tutorial is intended for the core developers listed as maintainers of the conda recipe in the conda-forge/pysteps-feedstock.

Examples for needing to update the pysteps-feedstock are:

- New release
- Fix errors pysteps package errors

The following tutorial was adapted from the official conda-forge.org documentation, released under CC4.0 license

What is a "conda-forge"

Conda-forge is a community effort that provides conda packages for a wide range of software. The conda team from Anaconda packages a multitude of packages and provides them to all users free of charge in their default channel.

conda-forge is a community-led conda channel of installable packages that allows users to share software that is not included in the official Anaconda repository. The main advantages of **conda-forge** are:

- all packages are shared in a single channel named conda-forge
- care is taken that all packages are up-to-date
- common standards ensure that all packages have compatible versions
- by default, packages are built for macOS, linux amd64 and windows amd64

In order to provide high-quality builds, the process has been automated into the conda-forge GitHub organization. The conda-forge organization contains one repository for each of the installable packages. Such a repository is known as a **feedstock**.

The actual pysteps feedstock is https://github.com/conda-forge/pysteps-feedstock

A feedstock is made up of a conda recipe (the instructions on what and how to build the package) and the necessary configurations for automatic building using freely available continuous integration services.

See the official conda-forge documentation for more details.

Maintain pysteps conda-forge package

Pysteps Core developers that are maintainers of the pysteps feedstock.

All pysteps developers listed as maintainers of the pysteps feedstock are given push access to the feedstock repository. This means that a maintainer can create branches in the main repository.

Every time that a new commit is pushed/merged in the feedstock repository, conda-forge runs Continuous Integration (CI) system that run quality checks, builds the pysteps recipe on Windows, OSX, and Linux, and publish the built recipes in the conda-forge channel.

Important

For updates, using a branch in the main repo and a subsequent Pull Request (PR) to the master branch is discouraged because: - CI is run on both the branch and on the Pull Request (if any) associated with that branch. This wastes CI resources. - Branches are automatically published by the CI system. This mean that a for every push, the packages will be published before the PR is actually merged.

For these reasons, to update the feedstock, the maintainers need to fork the feedstock, create a new branch in that fork, push to that branch in the fork, and then open a PR to the conda-forge repo.

Workflow for updating a pysteps-feedstock

The mandatory steps to update the pysteps-feedstock are:

1. Forking the pysteps-feedstock.
   - Clone the forked repository in your computer:

   ```
   git clone https://github.com/<your-github-id>/pysteps-feedstock
   ```

2. Syncing your fork with the pysteps feedstock. This step is only needed if your local repository is not up to date the pysteps-feedstock. If you just cloned the forked pysteps-feedstock, you can ignore this step.
   - Make sure you are on the master branch:

   ```
   git checkout master
   ```

   - Register conda-forge's feedstock with:

   ```
   git remote add upstream https://github.com/conda-forge/pysteps-feedstock
   ```

   - Fetch the latest updates with git fetch upstream:

   ```
   git fetch upstream
   ```

   - Pull in the latest changes into your master branch:

   ```
   git rebase upstream/master
   ```

3. Create a new branch:

   ```
   git checkout -b <branch-name>
   ```

4. Update the recipe and push changes in this new branch
   - See next section "Updating recipes" for more details
   - Push changes:

   ```
   git commit -m <commit message>
   ```

5. Pushing your changes to GitHub:

```
git push origin <branch-name>
```

6. Propose a Pull Request
    - Create a pull request via the web interface

Updating pysteps recipe

The pysteps-feedstock should be updated when:

- We release a new pysteps version
- Need to fix errors in the pysteps package

New release

When a new pysteps version is released, before update the pysteps feedstock, the new version needs to be uploaded to the Python Package Index (PyPI) (see *Packaging the pysteps project* for more details). This step is needed because the conda recipe uses the PyPI to build the pysteps conda package.

Once the new version is available in the PyPI, the conda recipe in pysteps-feedstock/recipe/meta.yaml needs to be updated by:

1. Updating version and hash
2. Checking the dependencies
3. Bumping the build number
    - When the package version changes, reset the build number back to 0.
    - The build number is increased when the source code for the package has not changed but you need to make a new build.
    - In case that the recipe must be updated, increase by 1 the **build_number** in the conda recipe in pysteps-feedstock/recipe/meta.yaml.

      Some examples for needing to increase the build number are:

      – updating the pinned dependencies

      – Fixing wrong dependencies
4. Rerendering feedstocks
    - Rerendering is conda-forge's way to update the files common to all feedstocks (e.g. README, CI configuration, pinned dependencies).
    - When to rerender:

      We need to re-render when there are changes the following parts of the feedstock:

      – the platform configuration (skip sections)

      – the yum_requirements.txt

      – updates in the build matrix due to new versions of Python, NumPy, PERL, R, etc.

      – updates in conda-forge pinning that affect the feedstock

      – build issues that a feedstock configuration update will fix
    - To rerender the feedstock, the first step is to install **conda-smithy** in your root environment:

      ```
      conda install -c conda-forge conda-smithy
      ```
    - Commit all changes and from the root directory of the feedstock, type:

```
conda smithy rerender -c auto
```

Optionally one can commit the changes manually. To do this drop *-c auto* from the command.

More information on https://conda-forge.org/docs/maintainer/updating_pkgs.html#dev-rerender-local

conda-forge autotick bot

The conda-forge autotick bot is now a central part of the conda-forge ecosystem. The conda-forge autotick bot was created to track out-of-date feedstocks and issue pull requests with updated recipes. The bot tracks and updates out-of-date feedstocks in four steps:

- Find the names of all feedstocks on conda-forge.
- Compute the dependency graph of packages on conda-forge found in step 1.
- Find the most recent version of each feedstock's source code.
- Open a PR into each out-of-date feedstock updating the meta.yaml for the most recent upstream release.

These steps are run automatically every six hours.

Hence, when a new pysteps version is upload to PyPI, this bot will automatically update the recipe and submit a PR. If the tests in the PR pass, then it can be merger into the feedstock's master branch.

2.4 Bibliography

Bibliography

[BPS06] N. E. Bowler, C. E. Pierce, and A. W. Seed. STEPS: a probabilistic precipitation forecasting scheme which merges an extrapolation nowcast with downscaled NWP. *Quarterly Journal of the Royal Meteorological Society*, 132(620):2127–2155, 2006. doi:10.1256/qj.04.100.

[BrockerS07] J. Bröcker and L. A. Smith. Increasing the reliability of reliability diagrams. *Weather and Forecasting*, 22(3):651–661, 2007. doi:10.1175/WAF993.1.

[CRS04] B. Casati, G. Ross, and D. B. Stephenson. A new intensity-scale approach for the verification of spatial precipitation forecasts. *Meteorological Applications*, 11(2):141—154, 2004. doi:10.1017/S1350482704001239.

[CP02] A. Clothier and G. Pegram. *Space-time modelling of rainfall using the string of beads model: integration of radar and raingauge data*. WRC Report No. 1010/1/02. Water Research Commission, Durban, South Africa, 2002.

[EWW+13] E. Ebert, L. Wilson, A. Weigel, M. Mittermaier, P. Nurmi, P. Gill, M. Göber, S. Joslyn, B. Brown, T. Fowler, and A. Watkins. Progress and challenges in forecast verification. *Meteorological Applications*, 20(2):130–139, 2013. doi:10.1002/met.1392.

[GZ02] U. Germann and I. Zawadzki. Scale-dependence of the predictability of precipitation from continental radar images. Part I: description of the methodology. *Monthly Weather Review*, 130(12):2859–2873, 2002. doi:10.1175/1520-0493(2002)130<2859:SDOTPO>2.0.CO;2.

[Her00] H. Hersbach. Decomposition of the continuous ranked probability score for ensemble prediction systems. *Weather and Forecasting*, 15(5):559–570, 2000. doi:10.1175/1520-0434(2000)015<0559:DOTCRP>2.0.CO;2.

[NBS+17] D. Nerini, N. Besic, I. Sideris, U. Germann, and L. Foresti. A non-stationary stochastic ensemble generator for radar rainfall fields based on the short-space Fourier transform. *Hydrology and Earth System Sciences*, 21(6):2777–2797, 2017. doi:10.5194/hess-21-2777-2017.

[PvGPO94] M. Proesmans, L. van Gool, E. Pauwels, and A. Oosterlinck. Determination of optical flow and its discontinuities using non-linear diffusion. In J.-O. Eklundh, editor, *Computer Vision — ECCV '94*, volume 801 of Lecture Notes in Computer Science, pages 294–304. Springer Berlin Heidelberg, 1994.

[PCH18a] S. Pulkkinen, V. Chandrasekar, and A.-M. Harri. Fully spectral method for radar-based precipitation nowcasting. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(5):1369–1382, 2018.

[PCH18b] S. Pulkkinen, V. Chandrasekar, and A.-M. Harri. Nowcasting of precipitation in the high-resolution Dallas-Fort Worth (DFW) urban radar remote sensing network. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(8):2773–2787, 2018. doi:10.1109/JSTARS.2018.2840491.

[PCH19] S. Pulkkinen, V. Chandrasekar, and A.-M. Harri. Stochastic spectral method for radar-based probabilistic precipitation nowcasting. *Journal of Atmospheric and Oceanic Technology*, 36(6):971–985, 2019.

[PCvLH20] S. Pulkkinen, V. Chandrasekar, A. von Lerber, and A.-M. Harri. Nowcasting of convective rainfall using volumetric radar observations. *IEEE Transactions on Geoscience and Remote Sensing*, pages 1–15, 2020. doi:10.1109/TGRS.2020.2984594.

[RL08] N. M. Roberts and H. W. Lean. Scale-selective verification of rainfall accumulations from high-resolution forecasts of convective events. *Monthly Weather Review*, 136(1):78–97, 2008. doi:10.1175/2007MWR2123.1.

[RC11] E. Ruzanski and V. Chandrasekar. Scale filtering for improved nowcasting performance in a high-resolution X-band radar network. *IEEE Transactions on Geoscience and Remote Sensing*, 49(6):2296–2307, June 2011.

[RCW11] E. Ruzanski, V. Chandrasekar, and Y. Wang. The CASA nowcasting system. *Journal of Atmospheric and Oceanic Technology*, 28(5):640–655, 2011. doi:10.1175/2011JTECHA1496.1.

[See03] A. W. Seed. A dynamic and spatial scaling approach to advection forecasting. *Journal of Applied Meteorology*, 42(3):381–388, 2003. doi:10.1175/1520-0450(2003)042<0381:ADASSA>2.0.CO;2.

[SPN13] A. W. Seed, C. E. Pierce, and K. Norman. Formulation and evaluation of a scale decomposition-based stochastic precipitation nowcast scheme. *Water Resources Research*, 49(10):6624–6641, 2013. doi:10.1002/wrcr.20536.

[ZR09] P. Zacharov and D. Rezacova. Using the fractions skill score to assess the relationship between an ensemble QPF spread and skill. *Atmospheric Research*, 94(4):684–693, 2009. doi:10.1016/j.atmosres.2009.03.004.

Index

A

adjust_lag2_corrcoef1() (*in module pysteps.timeseries.autoregression*), 146
adjust_lag2_corrcoef2() (*in module pysteps.timeseries.autoregression*), 147
aggregate_fields() (*in module pysteps.utils.dimension*), 162
aggregate_fields_space() (*in module pysteps.utils.dimension*), 163
aggregate_fields_time() (*in module pysteps.utils.dimension*), 163
animate() (*in module pysteps.visualization.animations*), 200
ar_acf() (*in module pysteps.timeseries.autoregression*), 147

B

banddepth() (*in module pysteps.postprocessing.ensemblestats*), 143
binary_mse() (*in module pysteps.verification.spatialscores*), 196
binary_mse_accum() (*in module pysteps.verification.spatialscores*), 197
binary_mse_compute() (*in module pysteps.verification.spatialscores*), 198
binary_mse_init() (*in module pysteps.verification.spatialscores*), 197
binary_mse_merge() (*in module pysteps.verification.spatialscores*), 197
blob_detection() (*in module pysteps.utils.images*), 165
boxcox_transform() (*in module pysteps.utils.transformation*), 172

C

ceil_int() (*in module pysteps.motion.vet*), 120
clip_domain() (*in module pysteps.utils.dimension*), 164
close_forecast_files() (*in module pysteps.io.exporters*), 109
compute_centred_coord_array() (*in module pysteps.utils.arrays*), 158
compute_empirical_cdf() (*in module pysteps.postprocessing.probmatching*), 144
compute_mask_window_function() (*in module pysteps.utils.tapering*), 171
compute_noise_stddev_adjs() (*in module pysteps.noise.utils*), 128
compute_window_function() (*in module pysteps.utils.tapering*), 171
constant() (*in module pysteps.motion.constant*), 111
corrcoef() (*in module pysteps.utils.spectral*), 169
create_default_pystepsrc() (*in module pysteps.datasets*), 92
CRPS() (*in module pysteps.verification.probscores*), 189
CRPS_accum() (*in module pysteps.verification.probscores*), 190
CRPS_compute() (*in module pysteps.verification.probscores*), 190
CRPS_init() (*in module pysteps.verification.probscores*), 189

D

DARTS() (*in module pysteps.motion.darts*), 111
dB_transform() (*in module pysteps.utils.transformation*), 173
decluster() (*in module pysteps.utils.cleansing*), 159
decomposition_fft() (*in module pysteps.cascade.decomposition*), 87
dense_lucaskanade() (*in module pysteps.motion.lucaskanade*), 113
det_cat_fct() (*in module pysteps.verification.detcatscores*), 177
det_cat_fct_accum() (*in module pysteps.verification.detcatscores*), 178
det_cat_fct_compute() (*in module pysteps.verification.detcatscores*), 179
det_cat_fct_init() (*in module pysteps.verification.detcatscores*), 177
det_cat_fct_merge() (*in module pysteps.verification.detcatscores*), 178
det_cont_fct() (*in module pysteps.verification.detcontscores*), 179
det_cont_fct_accum() (*in module pysteps.verification.detcontscores*), 181

det_cont_fct_compute() (*in module pysteps.verification.detcontscores*), 182
det_cont_fct_init() (*in module pysteps.verification.detcontscores*), 181
det_cont_fct_merge() (*in module pysteps.verification.detcontscores*), 182
detect_outliers() (*in module pysteps.utils.cleansing*), 159
download_pysteps_data() (*in module pysteps.datasets*), 91

E

ensemble_skill() (*in module pysteps.verification.ensscores*), 183
ensemble_spread() (*in module pysteps.verification.ensscores*), 184
estimate_ar_params_ols() (*in module pysteps.timeseries.autoregression*), 147
estimate_ar_params_ols_localized() (*in module pysteps.timeseries.autoregression*), 148
estimate_ar_params_yw() (*in module pysteps.timeseries.autoregression*), 149
estimate_ar_params_yw_localized() (*in module pysteps.timeseries.autoregression*), 150
estimate_var_params_ols() (*in module pysteps.timeseries.autoregression*), 151
estimate_var_params_ols_localized() (*in module pysteps.timeseries.autoregression*), 152
estimate_var_params_yw() (*in module pysteps.timeseries.autoregression*), 153
eulerian_persistence() (*in module pysteps.extrapolation.interface*), 89
excprob() (*in module pysteps.postprocessing.ensemblestats*), 142
export_forecast_dataset() (*in module pysteps.io.exporters*), 109
extrapolate() (*in module pysteps.extrapolation.semilagrangian*), 90

F

filter_gaussian() (*in module pysteps.cascade.bandpass_filters*), 86
filter_uniform() (*in module pysteps.cascade.bandpass_filters*), 85
find_by_date() (*in module pysteps.io.archive*), 94
forecast() (*in module pysteps.nowcasts.anvil*), 130
forecast() (*in module pysteps.nowcasts.extrapolation*), 132
forecast() (*in module pysteps.nowcasts.sprog*), 133
forecast() (*in module pysteps.nowcasts.sseps*), 135
forecast() (*in module pysteps.nowcasts.steps*), 138
fss() (*in module pysteps.verification.spatialscores*), 198
fss_accum() (*in module pysteps.verification.spatialscores*), 199
fss_compute() (*in module pysteps.verification.spatialscores*), 199
fss_init() (*in module pysteps.verification.spatialscores*), 198
fss_merge() (*in module pysteps.verification.spatialscores*), 199

G

generate_bps() (*in module pysteps.noise.motion*), 128
generate_noise_2d_fft_filter() (*in module pysteps.noise.fftgenerators*), 125
generate_noise_2d_ssft_filter() (*in module pysteps.noise.fftgenerators*), 126
get_colormap() (*in module pysteps.visualization.precipfields*), 206
get_default_params_bps_par() (*in module pysteps.noise.motion*), 127
get_default_params_bps_perp() (*in module pysteps.noise.motion*), 127
get_method() (*in module pysteps.cascade.interface*), 84
get_method() (*in module pysteps.extrapolation.interface*), 89
get_method() (*in module pysteps.io.interface*), 93
get_method() (*in module pysteps.motion.interface*), 110
get_method() (*in module pysteps.noise.interface*), 121
get_method() (*in module pysteps.nowcasts.interface*), 130
get_method() (*in module pysteps.utils.interface*), 156
get_method() (*in module pysteps.verification.interface*), 175
get_numpy() (*in module pysteps.utils.fft*), 165
get_padding() (*in module pysteps.motion.vet*), 120
get_pyfftw() (*in module pysteps.utils.fft*), 165
get_scipy() (*in module pysteps.utils.fft*), 165

I

import_bom_rf3() (*in module pysteps.io.importers*), 96
import_fmi_geotiff() (*in module pysteps.io.importers*), 97
import_fmi_pgm() (*in module pysteps.io.importers*), 97
import_knmi_hdf5() (*in module pysteps.io.importers*), 98
import_mch_gif() (*in module pysteps.io.importers*), 99
import_mch_hdf5() (*in module pysteps.io.importers*), 99
import_mch_metranet() (*in module pysteps.io.importers*), 100

import_mrms_grib() (*in module pysteps.io.importers*), 101
import_netcdf_pysteps() (*in module pysteps.io.nowcast_importers*), 104
import_opera_hdf5() (*in module pysteps.io.importers*), 102
import_saf_crri() (*in module pysteps.io.importers*), 102
info() (*in module pysteps.datasets*), 92
initialize_bps() (*in module pysteps.noise.motion*), 127
initialize_forecast_exporter_geotiff() (*in module pysteps.io.exporters*), 106
initialize_forecast_exporter_kineros() (*in module pysteps.io.exporters*), 107
initialize_forecast_exporter_netcdf() (*in module pysteps.io.exporters*), 108
initialize_nonparam_2d_fft_filter() (*in module pysteps.noise.fftgenerators*), 123
initialize_nonparam_2d_nested_filter() (*in module pysteps.noise.fftgenerators*), 123
initialize_nonparam_2d_ssft_filter() (*in module pysteps.noise.fftgenerators*), 124
initialize_param_2d_fft_filter() (*in module pysteps.noise.fftgenerators*), 122
intensity_scale() (*in module pysteps.verification.spatialscores*), 194
intensity_scale_accum() (*in module pysteps.verification.spatialscores*), 195
intensity_scale_compute() (*in module pysteps.verification.spatialscores*), 196
intensity_scale_init() (*in module pysteps.verification.spatialscores*), 194
intensity_scale_merge() (*in module pysteps.verification.spatialscores*), 195
iterate_ar_model() (*in module pysteps.timeseries.autoregression*), 153
iterate_var_model() (*in module pysteps.timeseries.autoregression*), 154

L

lifetime() (*in module pysteps.verification.lifetime*), 186
lifetime_accum() (*in module pysteps.verification.lifetime*), 187
lifetime_compute() (*in module pysteps.verification.lifetime*), 187
lifetime_init() (*in module pysteps.verification.lifetime*), 186
load_config_file() (*in module pysteps*), 84
load_dataset() (*in module pysteps.datasets*), 92

M

mean() (*in module pysteps.postprocessing.ensemblestats*), 142
mean() (*in module pysteps.utils.spectral*), 169
morph() (*in module pysteps.motion.vet*), 119
morph_opening() (*in module pysteps.utils.images*), 167

N

nonparam_match_empirical_cdf() (*in module pysteps.postprocessing.probmatching*), 144
NQ_transform() (*in module pysteps.utils.transformation*), 174

P

parse_proj4_string() (*in module pysteps.visualization.utils*), 207
plot_intensityscale() (*in module pysteps.verification.plots*), 187
plot_precip_field() (*in module pysteps.visualization.precipfields*), 204
plot_rankhist() (*in module pysteps.verification.plots*), 188
plot_reldiag() (*in module pysteps.verification.plots*), 188
plot_ROC() (*in module pysteps.verification.plots*), 188
plot_spectrum1d() (*in module pysteps.visualization.spectral*), 206
pmm_compute() (*in module pysteps.postprocessing.probmatching*), 145
pmm_init() (*in module pysteps.postprocessing.probmatching*), 144
print_ar_params() (*in module pysteps.nowcasts.utils*), 141
print_corrcoefs() (*in module pysteps.nowcasts.utils*), 141
proesmans() (*in module pysteps.motion.proesmans*), 116
proj4_to_basemap() (*in module pysteps.visualization.utils*), 208
proj4_to_cartopy() (*in module pysteps.visualization.utils*), 208
pysteps.cascade.bandpass_filters (*module*), 85
pysteps.cascade.decomposition (*module*), 86
pysteps.cascade.interface (*module*), 84
pysteps.datasets (*module*), 91
pysteps.extrapolation.interface (*module*), 88
pysteps.extrapolation.semilagrangian (*module*), 90
pysteps.io.archive (*module*), 94
pysteps.io.exporters (*module*), 105
pysteps.io.importers (*module*), 95
pysteps.io.interface (*module*), 93
pysteps.io.nowcast_importers (*module*), 103
pysteps.io.readers (*module*), 109
pysteps.motion.constant (*module*), 111
pysteps.motion.darts (*module*), 111

pysteps.motion.interface (*module*), 110
pysteps.motion.lucaskanade (*module*), 112
pysteps.motion.proesmans (*module*), 115
pysteps.motion.vet (*module*), 116
pysteps.noise.fftgenerators (*module*), 121
pysteps.noise.interface (*module*), 121
pysteps.noise.motion (*module*), 126
pysteps.noise.utils (*module*), 128
pysteps.nowcasts.anvil (*module*), 130
pysteps.nowcasts.extrapolation (*module*), 132
pysteps.nowcasts.interface (*module*), 129
pysteps.nowcasts.sprog (*module*), 133
pysteps.nowcasts.sseps (*module*), 134
pysteps.nowcasts.steps (*module*), 137
pysteps.nowcasts.utils (*module*), 141
pysteps.postprocessing.ensemblestats (*module*), 142
pysteps.postprocessing.probmatching (*module*), 143
pysteps.timeseries.autoregression (*module*), 146
pysteps.timeseries.correlation (*module*), 154
pysteps.utils.arrays (*module*), 158
pysteps.utils.cleansing (*module*), 158
pysteps.utils.conversion (*module*), 160
pysteps.utils.dimension (*module*), 161
pysteps.utils.fft (*module*), 165
pysteps.utils.images (*module*), 165
pysteps.utils.interface (*module*), 156
pysteps.utils.interpolate (*module*), 167
pysteps.utils.spectral (*module*), 168
pysteps.utils.tapering (*module*), 171
pysteps.utils.transformation (*module*), 172
pysteps.verification.detcatscores (*module*), 176
pysteps.verification.detcontscores (*module*), 179
pysteps.verification.ensscores (*module*), 183
pysteps.verification.interface (*module*), 175
pysteps.verification.lifetime (*module*), 185
pysteps.verification.plots (*module*), 187
pysteps.verification.probscores (*module*), 188
pysteps.verification.spatialscores (*module*), 193
pysteps.visualization.animations (*module*), 200
pysteps.visualization.motionfields (*module*), 201
pysteps.visualization.precipfields (*module*), 204
pysteps.visualization.spectral (*module*), 206
pysteps.visualization.utils (*module*), 207

Q

quiver() (*in module pysteps.visualization.motionfields*), 202

R

rankhist() (*in module pysteps.verification.ensscores*), 184
rankhist_accum() (*in module pysteps.verification.ensscores*), 185
rankhist_compute() (*in module pysteps.verification.ensscores*), 185
rankhist_init() (*in module pysteps.verification.ensscores*), 184
rapsd() (*in module pysteps.utils.spectral*), 170
rbfinterp2d() (*in module pysteps.utils.interpolate*), 167
read_timeseries() (*in module pysteps.io.readers*), 109
recompose_fft() (*in module pysteps.cascade.decomposition*), 88
reldiag() (*in module pysteps.verification.probscores*), 190
reldiag_accum() (*in module pysteps.verification.probscores*), 191
reldiag_compute() (*in module pysteps.verification.probscores*), 191
reldiag_init() (*in module pysteps.verification.probscores*), 191
remove_rain_norain_discontinuity() (*in module pysteps.utils.spectral*), 170
reproject_geodata() (*in module pysteps.visualization.utils*), 208
ROC_curve() (*in module pysteps.verification.probscores*), 192
ROC_curve_accum() (*in module pysteps.verification.probscores*), 192
ROC_curve_compute() (*in module pysteps.verification.probscores*), 193
ROC_curve_init() (*in module pysteps.verification.probscores*), 192
round_int() (*in module pysteps.motion.vet*), 120

S

shift_scale() (*in module pysteps.postprocessing.probmatching*), 145
shitomasi_detection() (*in module pysteps.utils.images*), 166
sqrt_transform() (*in module pysteps.utils.transformation*), 174
square_domain() (*in module pysteps.utils.dimension*), 164
stack_cascades() (*in module pysteps.nowcasts.utils*), 141
std() (*in module pysteps.utils.spectral*), 170

streamplot() (*in module pysteps.visualization.motionfields*), 203

T

temporal_autocorrelation() (*in module pysteps.timeseries.correlation*), 154
temporal_autocorrelation_multivariate() (*in module pysteps.timeseries.correlation*), 155
to_raindepth() (*in module pysteps.utils.conversion*), 161
to_rainrate() (*in module pysteps.utils.conversion*), 160
to_reflectivity() (*in module pysteps.utils.conversion*), 161
track_features() (*in module pysteps.motion.lucaskanade*), 114

V

vet() (*in module pysteps.motion.vet*), 117
vet_cost_function() (*in module pysteps.motion.vet*), 118
vet_cost_function_gradient() (*in module pysteps.motion.vet*), 119