# pysteps Reference

***Release 1.0.0***

**PySteps developers**

**Apr 07, 2019**

# Contents

**1 Documentation** **3**

**2 Contents** **5**

**Bibliography** **141**

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.

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 is gives an introductory overview to 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*.

** Under development **

### 2.1.1 Installing pysteps

#### Dependencies

The pysteps package needs the following dependencies

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

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

- dask and toolz (for code parallelisation)

- pyfftw (for faster FFT computation)

Other optional dependencies include:

- cartopy or basemap (for georeferenced visualization)
- h5py (for importing HDF5 data)
- pywavelets (for intensity-scale verification)
- cython (for the variational echo tracking method)

Note that cython also requires a C compiler. See https://cython.readthedocs.io/en/latest/src/quickstart/install.html for instructions.

We recommend that you create a conda environment using the available environment.yml file to install the most important dependencies:

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

This will allow running pysteps with the basic functionality.

### Install from source

**IMPORTANT**: installing from source requires numpy to be installed.

#### 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, so using the default compiler would have disabled multi-threading and you will get 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 lastest 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 circunstances, 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.

#### Installation

The installer needs numpy to compile the Cython extensions. If numpy is not installed you can run in a terminal:

```
pip install numpy
```

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, to install it using setup.py run (global installation):

```
git clone https://github.com/pySTEPS/pysteps
cd pysteps
python setup.py install
```

For user installation:

```
python setup.py install --user
```

If you want to install the package in a specific directory run:

```
python setup.py install --prefix=/path/to/local/dir
```

### Non-anaconda users or minimal anaconda environments

The installation using **setup.py** will try to to install the minimum dependencies needed to run the program correctly. If you are not using the recommended conda environment (defined in environment.yml) or you are working with a minimal python distribution, you may get the following error during the installation:

```
ModuleNotFoundError: No module named 'Cython'
```

This means that Cython is not installed, which is needed to build some of the dependencies of pysteps.

For non-anaconda users, you can install Cython using:

```
pip install Cython
```

Anaconda users can install Cython using:

```
conda install cython
```

### 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 stored in pysteps.rcparams AttrDict, which are loaded from a pystepsrc JSON file located in the system. The configuration parameters can be accessed as attributes or as items in a dictionary. For e.g., to retrieve the default parameters the following ways are equivalent:

```
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
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']
```

(continues on next page)

(continued from previous page)

```
colorscale = rcparams.plot.colorscale

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

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.
- 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.

### Linux and OSX users

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

This are the steps to setup up the configuration file in that directory:

1. Create the directory if it does not exist. Type in a terminal:

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

2. Find the location of the library's pystepsrc file used at the moment. 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"
```

3.Copy the library's default rc file to that directory. In a terminal type:

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

4. Edit the file with the text editor of your preference
5. Check that the location of the library's pystepsrc file used at the moment.:

```
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 place the pystepsrc parameters file in the users folder (defined in the %USERPROFILE% environment variable) in the following path: **%USERPROFILE%/pysteps/pystepsrc**

The following steps are needed to setup up the configuration file in that directory:

1. Create the directory if it does not exist. Type in a terminal:

```
$> mkdir -p %USERPROFILE%/pysteps
```

2. Find the location of the library's pystepsrc file used at the moment. When the pystep is imported, the configuration file loaded is shown:

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

3.Copy the library's default rc file to that directory. In a terminal type:

```
$> cp /path/to/pysteps/library/pystepsrc %USERPROFILE%/pysteps/pystepsrc
```

4. Edit the file with the text editor of your preference
5. Check that the location of the library's pystepsrc file used at the moment:

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

### 2.1.2 pysteps gallery

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

**Note:** Click *here* to download the full example code

#### 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
import numpy as np
from pprint import pprint
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 thing, the sequence of radar composites is imported, converted and transformed into units of dBR.

```
date = datetime.strptime("201505151630", "%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 input files from the archive
fns = io.archive.find_by_date(
    date, root_path, path_fmt, fn_pattern, fn_ext, timestep, num_prev_files=9
)

# Read the radar composites
importer = io.get_method(importer_name, "importer")
```

(continues on next page)

(continued from previous page)

```
R, _, metadata = io.read_timeseries(fns, importer, **importer_kwargs)

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

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

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

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

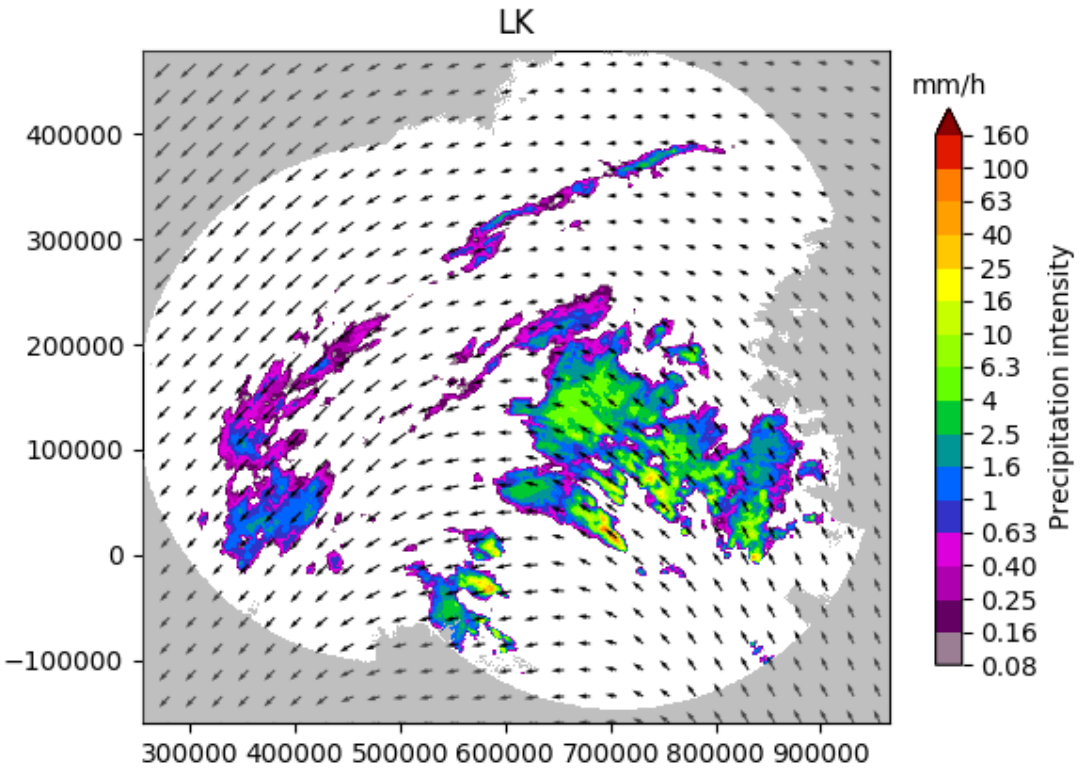
Out:

```
{'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,
 'yorigin': 'upper',
 'ypixelsize': 1000.0,
 'zerovalue': -15.0}
```

### 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
plot_precip_field(R_, geodata=metadata, title="LK")
quiver(V1, geodata=metadata, step=25)
```



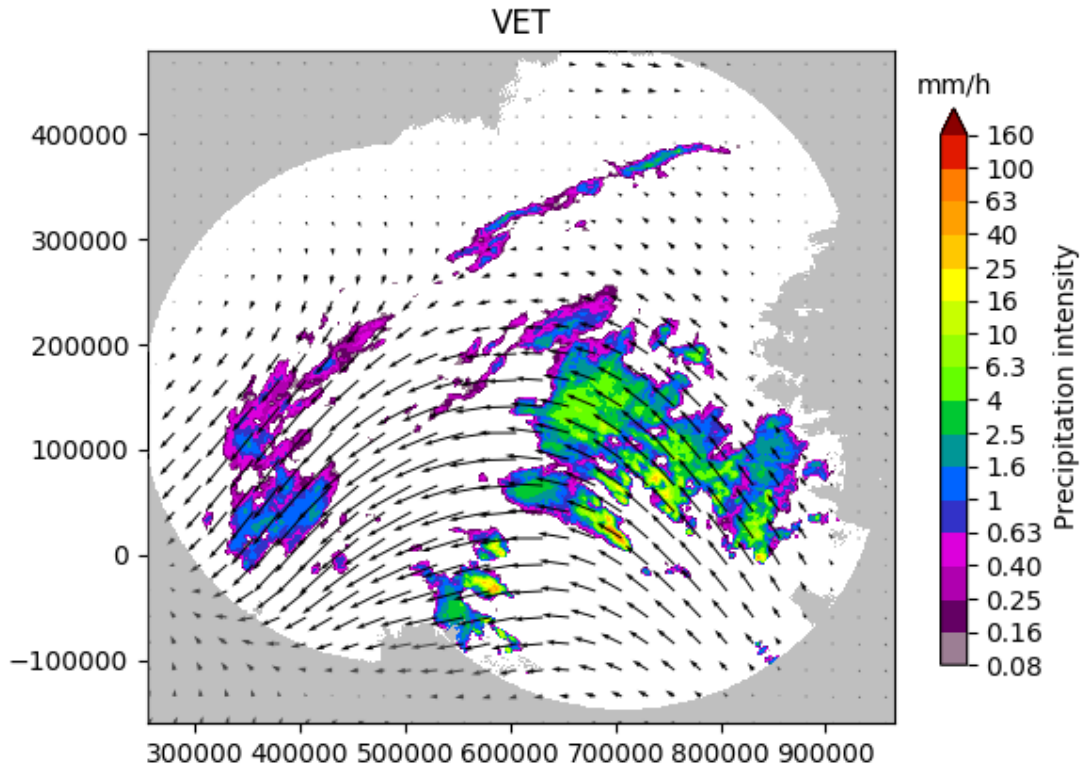

Out:

```
Computing the motion field with the Lucas-Kanade method.
--- LK found 392 sparse vectors ---
--- 158 sparse vectors left for interpolation ---
--- 1.11 seconds ---
```

### 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)
```




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 3860502.2323758407
smoothness_penalty 0.0
original image shape: (3, 640, 712)
padded image shape: (3, 640, 712)
padded template_image image shape: (3, 640, 712)

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

residuals 2852728.474075019
smoothness_penalty 1.2934717610147382
original image shape: (3, 640, 720)
padded image shape: (3, 640, 720)
padded template_image image shape: (3, 640, 720)

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

(continues on next page)

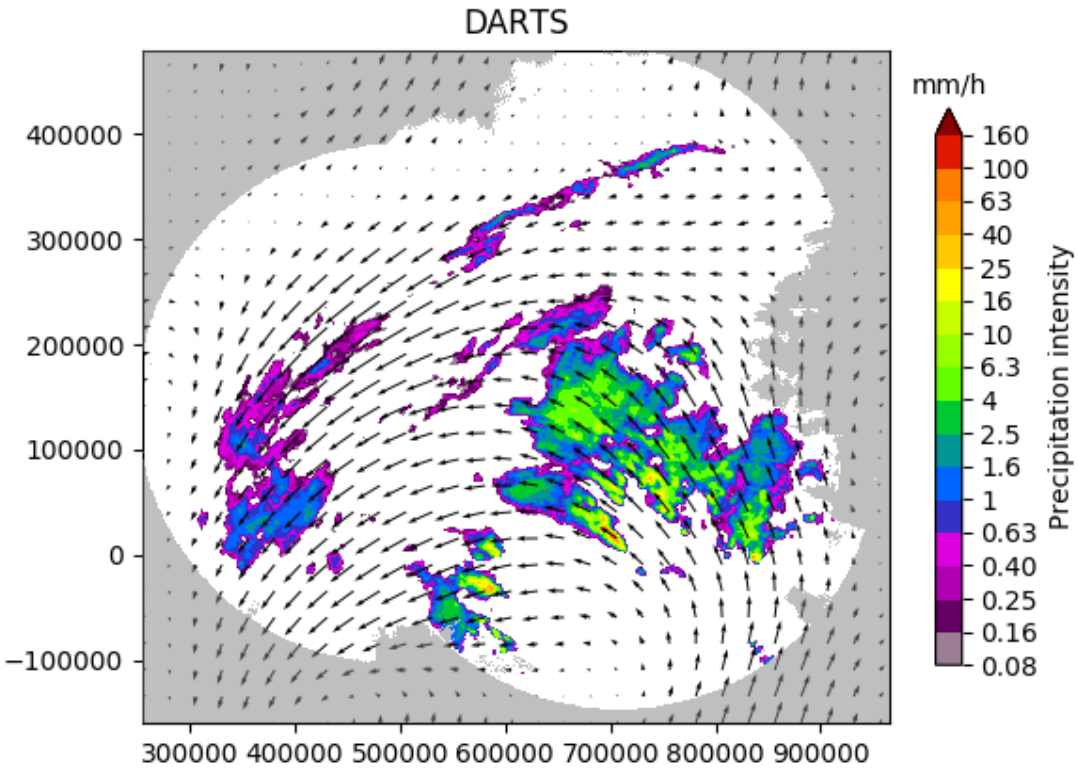
(continued from previous page)

```
residuals 2591853.9497765535
smoothness_penalty 98.75780333213032
original image shape: (3, 640, 736)
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 2615161.846366342
smoothness_penalty 309.47364391248857
```

### 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)

# sphinx_gallery_thumbnail_number = 1
```



Out:

```
Computing the motion field with the DARTS method.
--- 3.309852361679077 seconds ---
```

**Total running time of the script:** ( 0 minutes 40.491 seconds)

**Note:** Click *here* to download the full example code

### Generation of stochastic noise

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

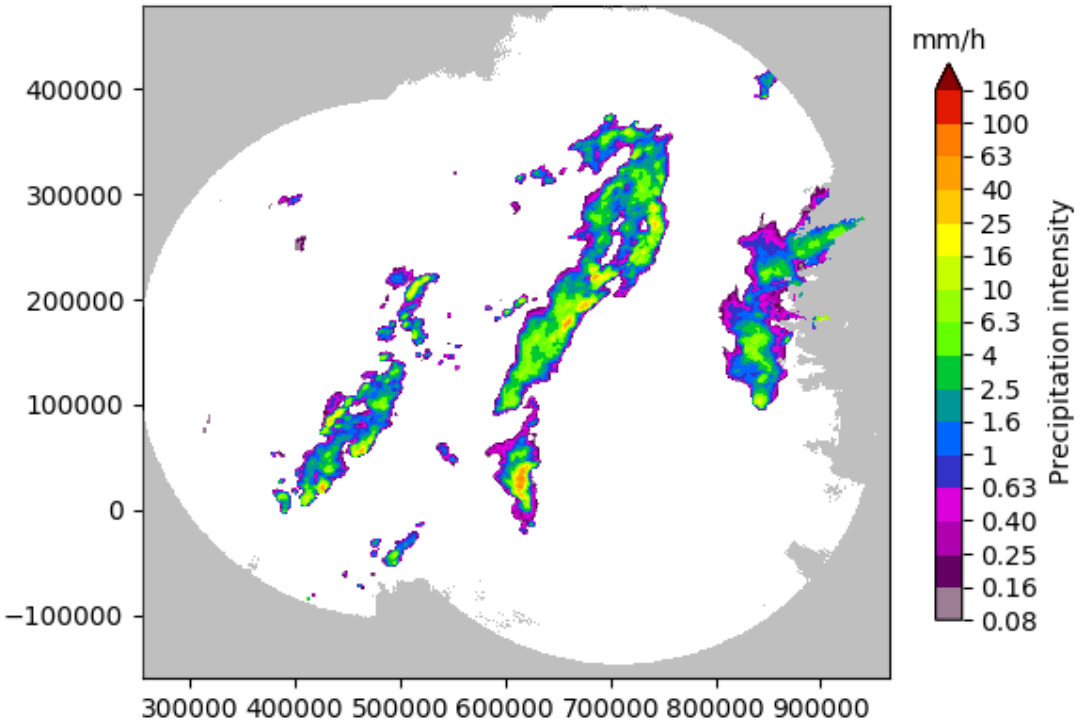
These noise fields are used as perturbation terms during an extrapolation nowcast in order to represent the uncertainty in the evolution of the rainfall field.

```
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}
```

### Parametric filter

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

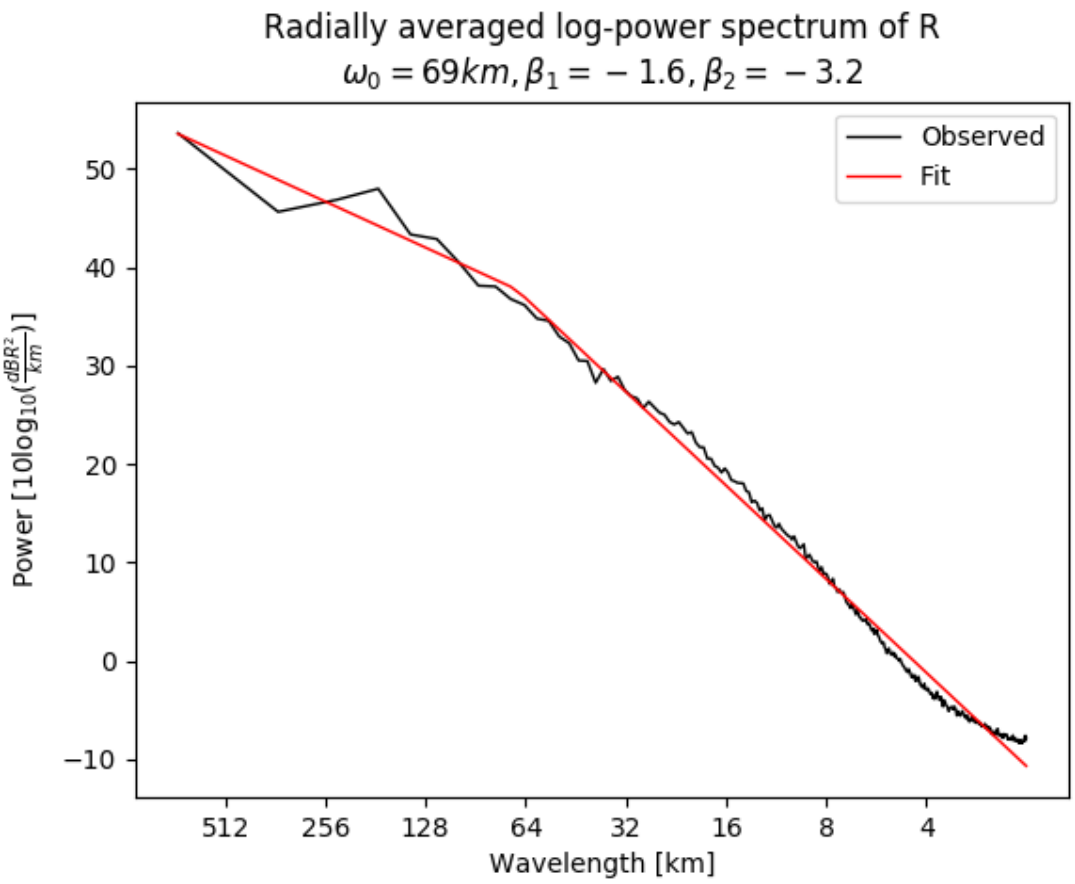
The parametric model uses a piece-wise linear function with two spectral slopes (beta1 and beta2) and one breaking point

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

(continues on next page)

(continued from previous page)

```
# 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)
)
```




### 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

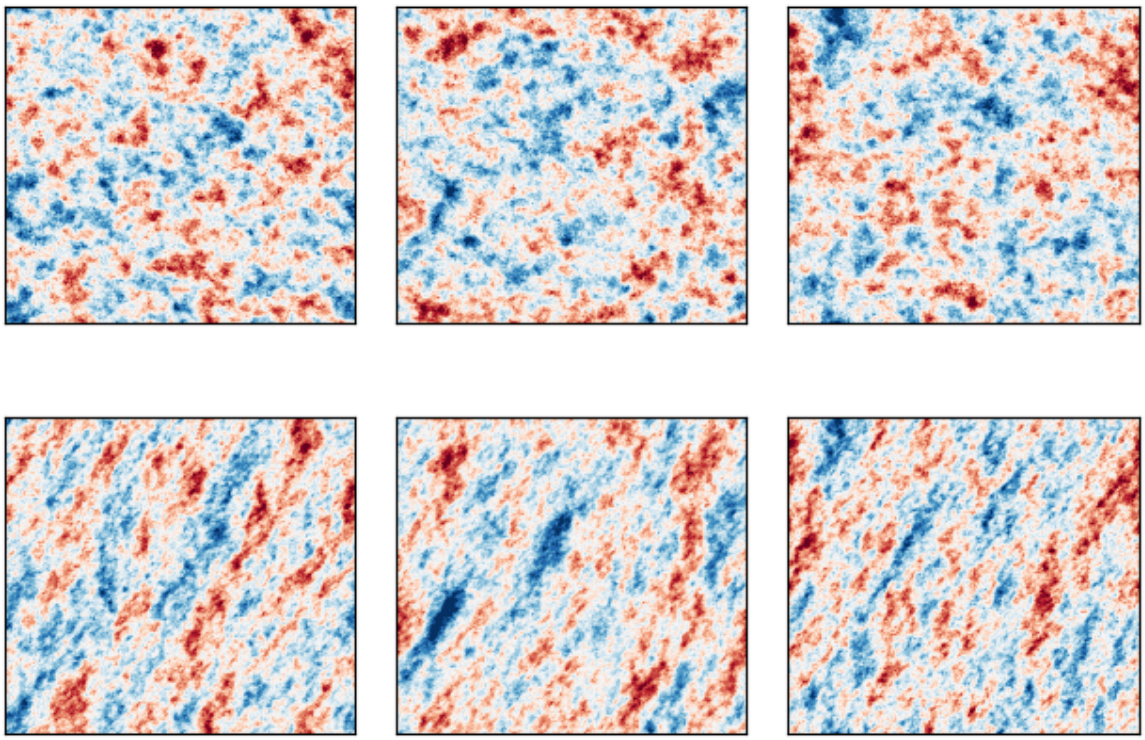
The parametric and nonparametric filters obtained above can now be used to produce N realizations of random fields of prescribed power spectrum, hence with the same correlation structure as the initial rainfall field.

```
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)
```

(continues on next page)

(continued from previous page)

```
# 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 2.153 seconds)

**Note:** Click *here* to download the full example code

### Extrapolation nowcast

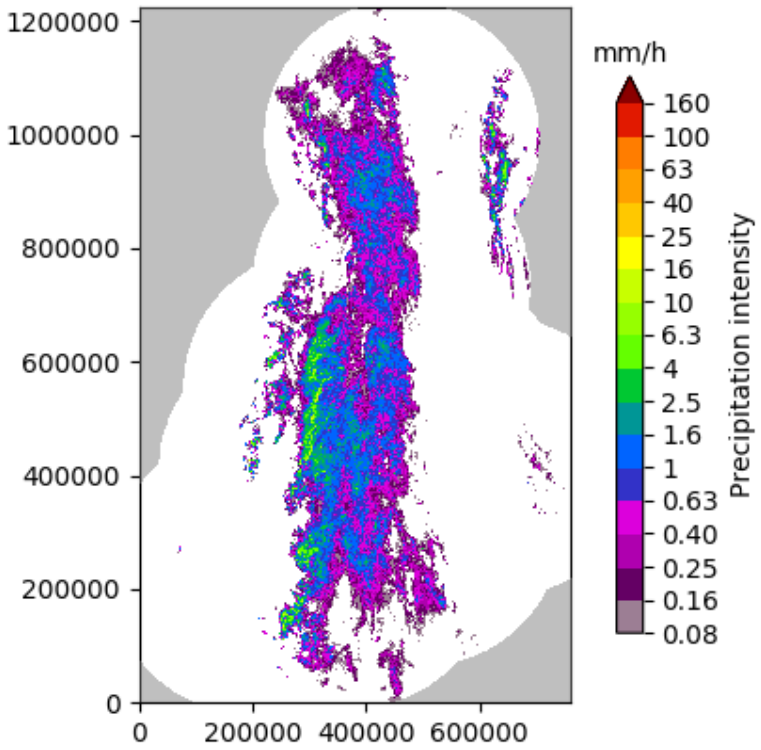
This tutorial shows how to compute and plot an extrapolation nowcast using Finnish radar data.

```
from pylab import *
from datetime import datetime
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 thing, the sequence of radar composites is imported, converted and transformed into units of dBR.

```
date = datetime.strptime("201609281600", "%Y%m%d%H%M")
data_source = "fmi"
n_leadtimes = 12

# 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 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 using the finnish Z-R relationship
R, metadata = conversion.to_rainrate(Z, metadata, 223.0, 1.53)

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

# 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:

```
{'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': -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

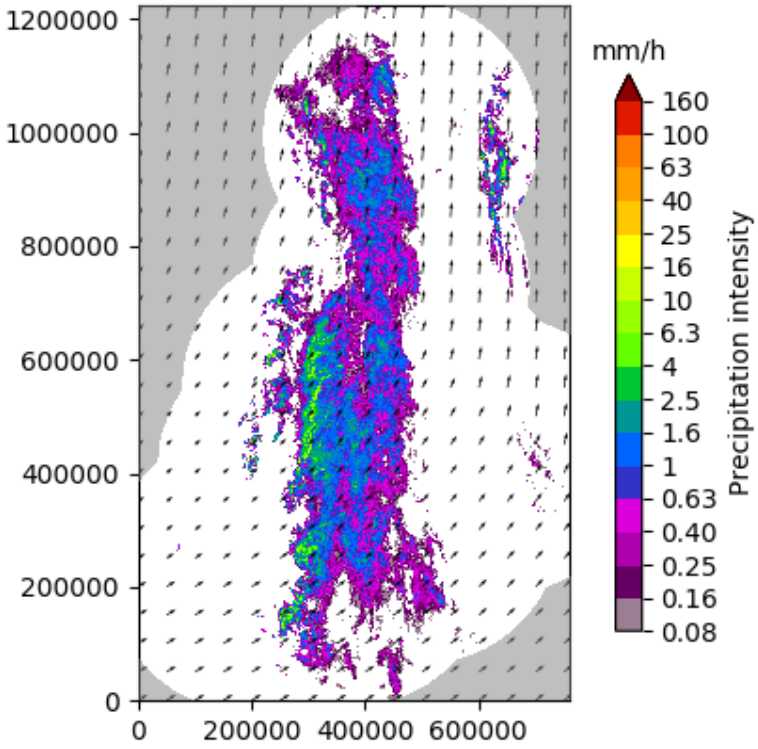
The extrapolation nowcast is based on the estimation of the motion field, which is here performed using a local tacking approach (Lucas-Kanade). The most recent radar rainfall field is then simply advected along this motion field in oder to produce an extrapolation forecast.

```
# Estimate the motion field with Lucas-Kanade
oflow_method = motion.get_method("LK")
```

(continues on next page)

(continued from previous page)

```
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)
```




Out:

```
Computing the motion field with the Lucas-Kanade method.
--- LK found 500 sparse vectors ---
--- 150 sparse vectors left for interpolation ---
--- 2.41 seconds ---
Computing extrapolation nowcast from a 1226x760 input grid...
```

### Verify with FSS

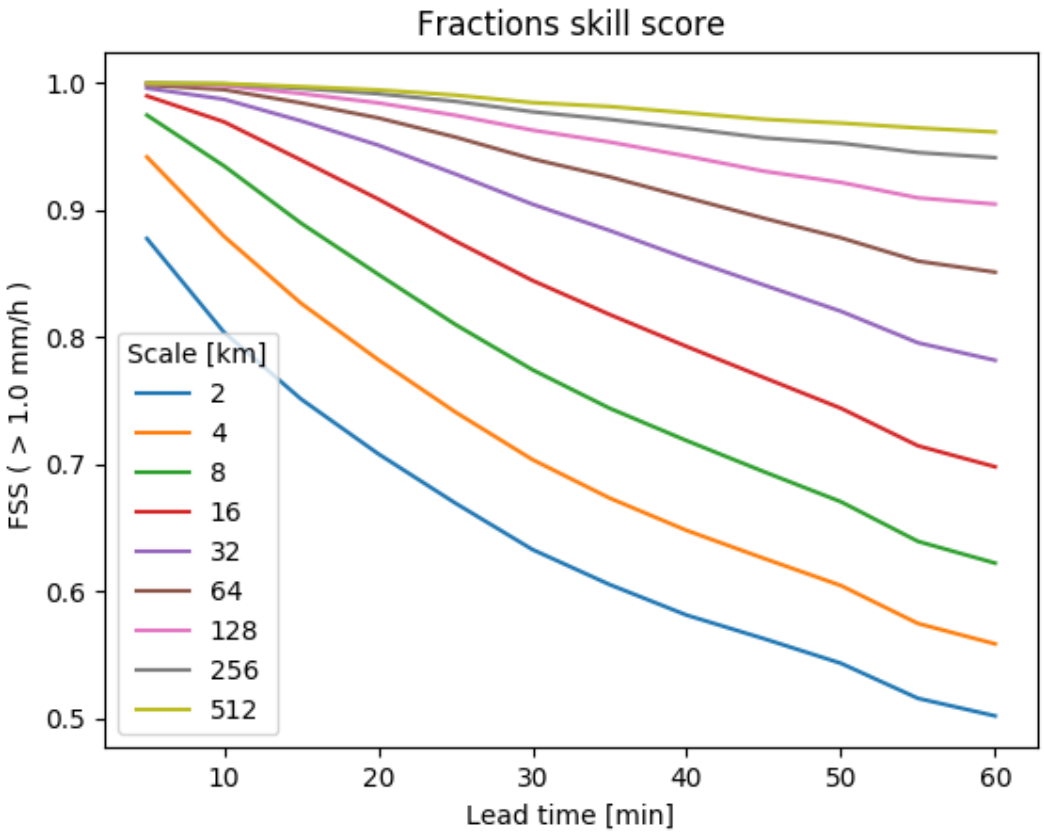
The fractions skill score (FSS) provides an intuitive assessment of the dependency of skill on spatial scale and intensity, which makes it an ideal skill score for high-resolution precipitation forecasts.

```
# Find observations in the data archive
fns = io.archive.find_by_date(
```

(continues on next page)

(continued from previous page)

```
    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_)

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

# sphinx_gallery_thumbnail_number = 3
```




**Total running time of the script:** ( 0 minutes 15.882 seconds)

**Note:** Click *here* to download the full example code

### 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

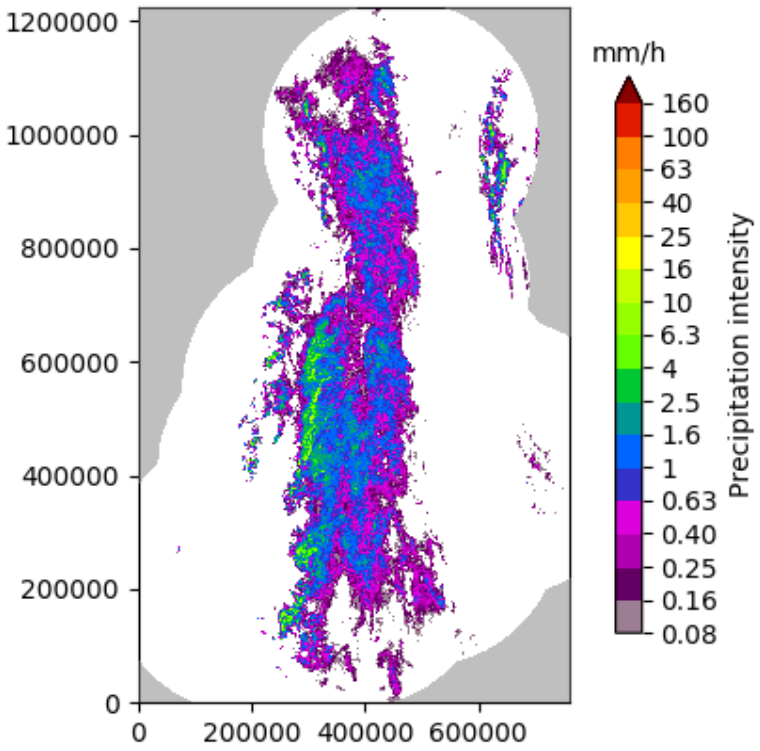
First thing, the radar composite is imported and transformed in units of dB.

```
# 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"
```

(continues on next page)

(continued from previous page)

```
)
R, _, metadata = io.import_fmi_pgm(filename, gzipped=True)

# Convert to rain rate using the finnish Z-R relationship
R, metadata = conversion.to_rainrate(R, metadata, 223.0, 1.53)

# 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:

```
{'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': 0.0002548805471873859,
 'transform': None,
 'unit': 'mm/h',
 'x1': 0.0049823258887045085,
 'x2': 759752.2852757066,
 'xpixelsize': 999.674053,
 'y1': 0.009731985162943602,
 'y2': 1225544.6588913496,
```

(continues on next page)

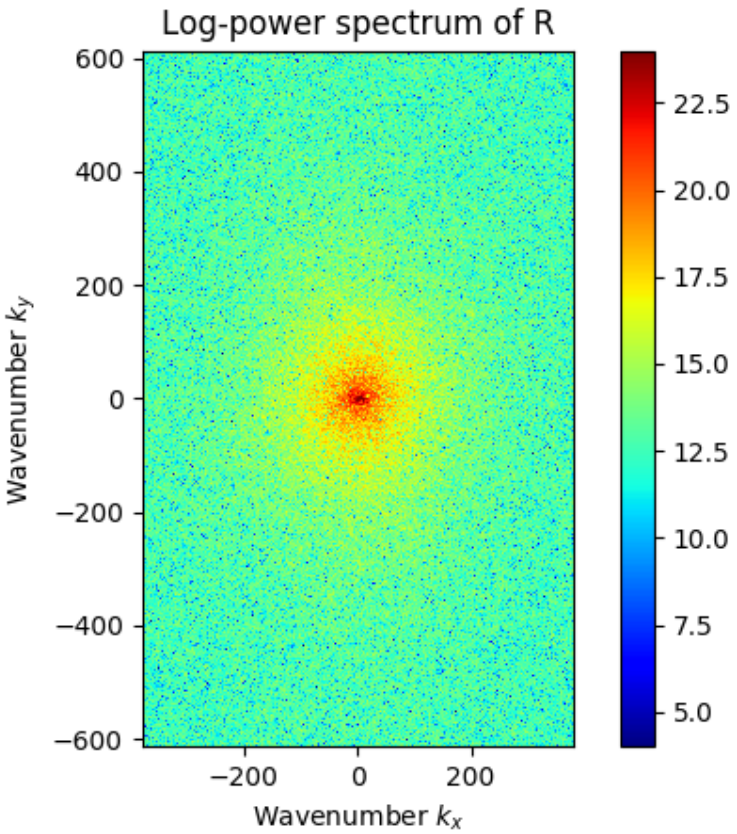
(continued from previous page)

```
 'yorigin': 'upper',
 'ypixelsize': 999.62859,
 'zerovalue': 0.0,
 'zr_a': 223.0,
 'zr_b': 1.53}
```

### 2D Fourier spectrum

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

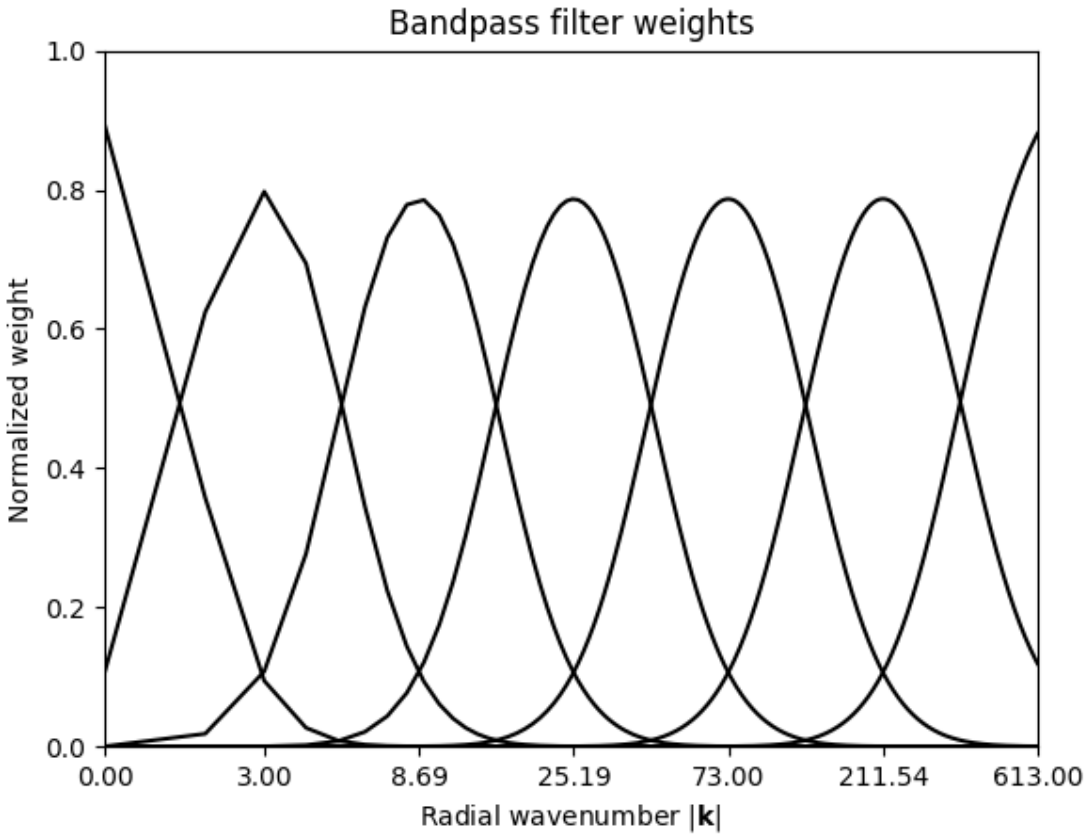
```
# Set Nans as the fill value
R[~np.isfinite(R)] = metadata["zerovalue"]

# 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")
```



### Cascade decomposition

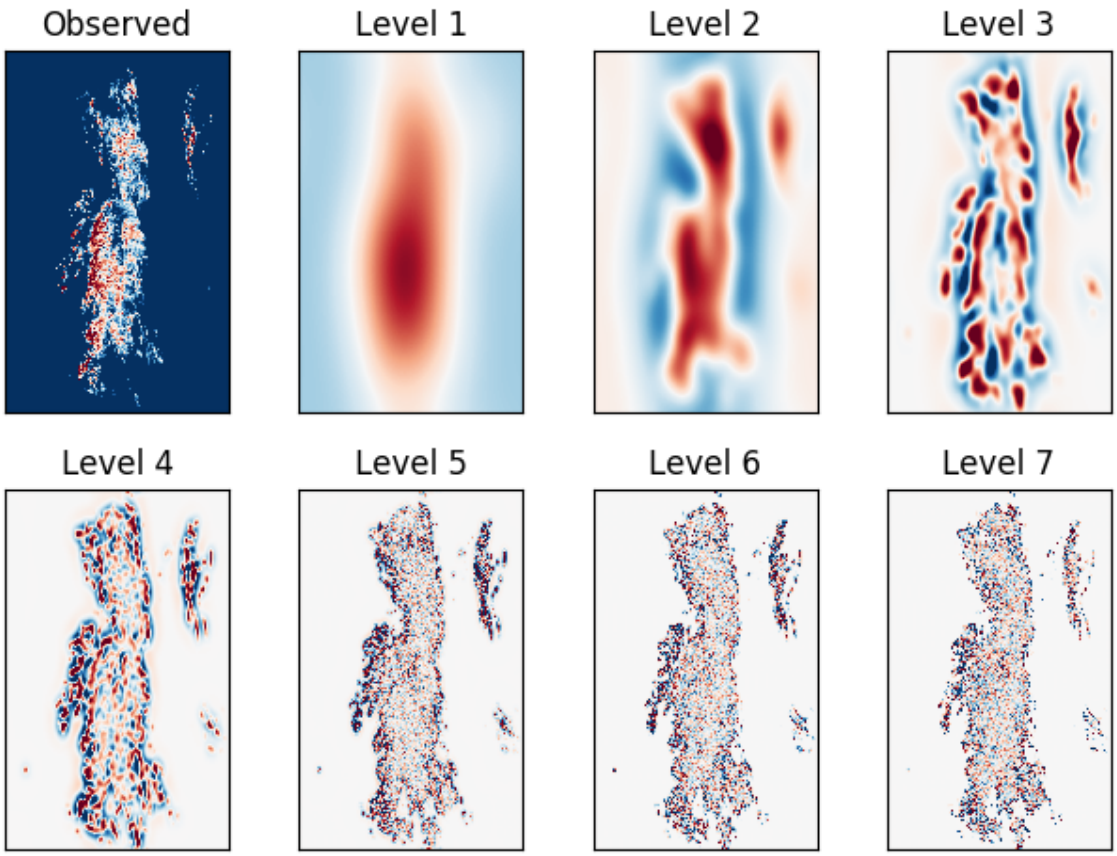
First, construct a set of Gaussian bandpass filters and plot the corresponding 1D filters.

```
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")
```




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)

# 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")
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 5.789 seconds)

**Note:** Click *here* to download the full example code

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

### Read precipitation field

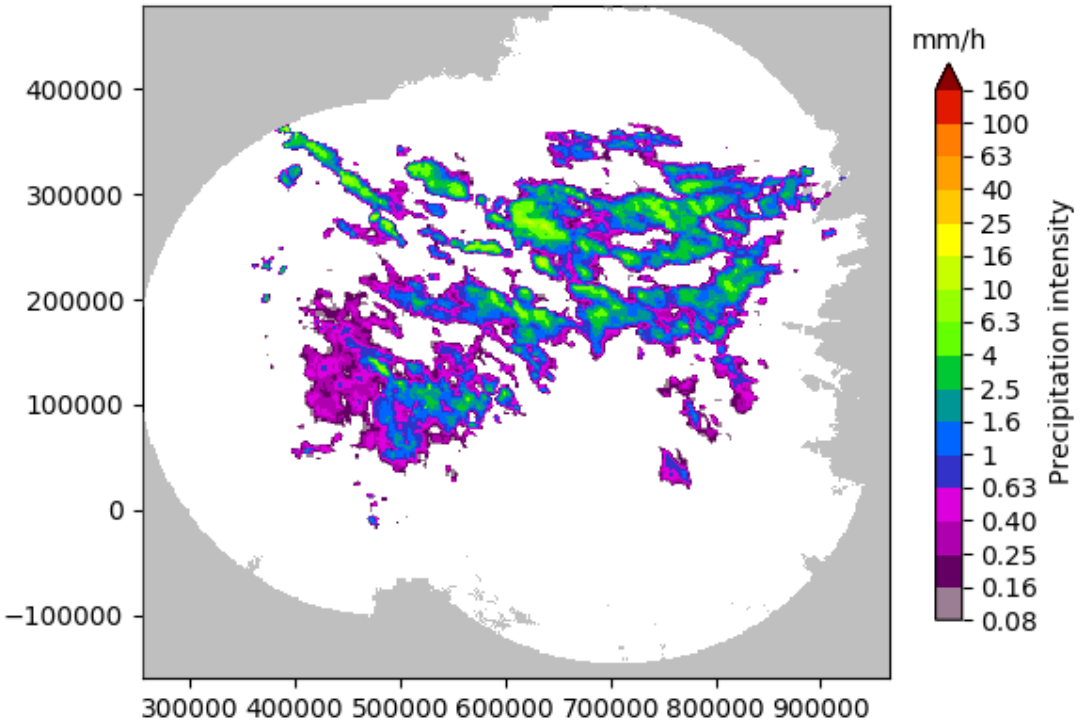
First thing, the sequence of Swiss radar composites is imported, converted and transformed into units of dBR.

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

(continues on next page)

(continued from previous page)

```
# 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:

```
{'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}
```

### Deterministic nowcast with S-PROG

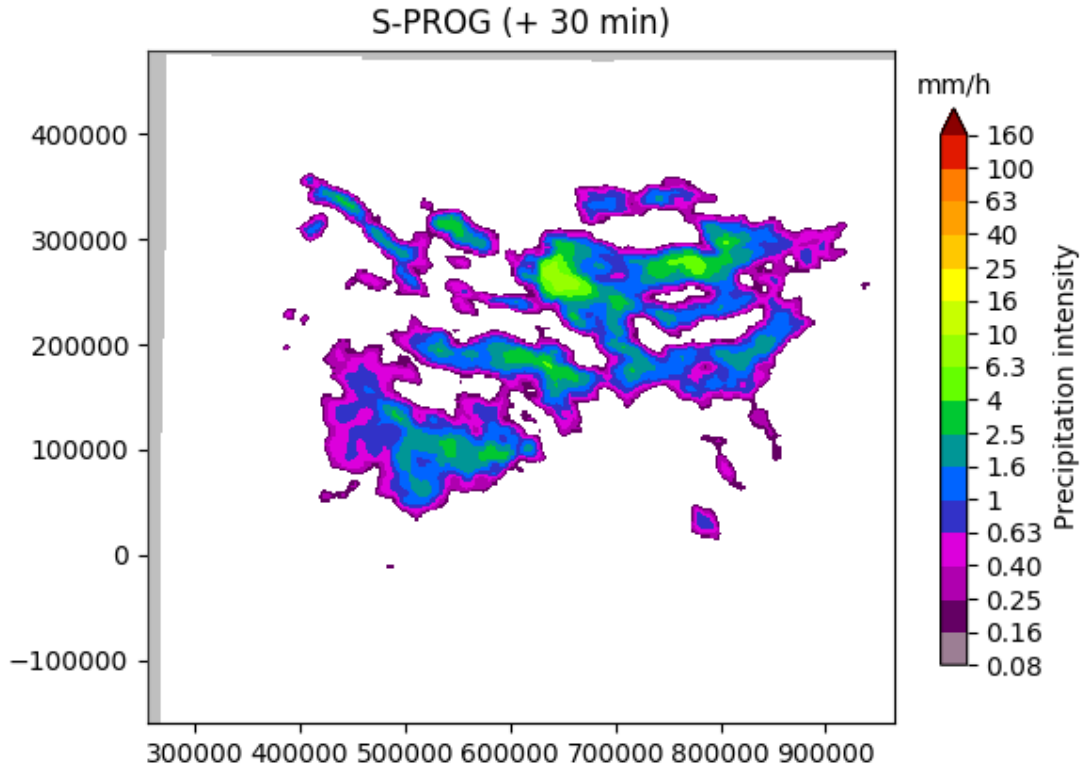
First, the motiong field is estimated using a local tracking approach based on the Lucas-Kanade optical flow. The motion field can then be used to generate a deterministic nowcast with the S-PROG model, which implements a scale filtering appraoch in order to progressively remove the unpredictable spatial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

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

(continues on next page)

(continued from previous page)

```
# 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 the motion field with the Lucas-Kanade method.
--- LK found 353 sparse vectors ---
--- 91 sparse vectors left for interpolation ---
--- 0.30 seconds ---
```

(continues on next page)

(continued from previous page)

```
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

Parameters:
-----------
number of time steps:     6
parallel threads:         1
number of cascade levels: 8
order of the AR(p) model: 2
*********************************************
* Correlation coefficients for cascade levels: *
*********************************************
----------------------------------------
| Level |     Lag-1     |     Lag-2     |
----------------------------------------
| 1     | 0.999183      | 0.996831      |
----------------------------------------
| 2     | 0.998209      | 0.991664      |
----------------------------------------
| 3     | 0.994658      | 0.981226      |
----------------------------------------
| 4     | 0.982905      | 0.949207      |
----------------------------------------
| 5     | 0.942153      | 0.842120      |
----------------------------------------
| 6     | 0.822896      | 0.628602      |
----------------------------------------
| 7     | 0.510867      | 0.296569      |
----------------------------------------
| 8     | 0.130524      | 0.038272      |
----------------------------------------
*************************************
* AR(p) parameters for cascade levels: *
*************************************
-------------------------------------------------------
| Level |    Phi-1     |    Phi-2     |    Phi-0     |
-------------------------------------------------------
| 1     | 1.920733     | -0.922304    | 0.015620     |
-------------------------------------------------------
| 2     | 1.883719     | -0.887099    | 0.027615     |
-------------------------------------------------------
| 3     | 1.752557     | -0.761969    | 0.066849     |
-------------------------------------------------------
| 4     | 1.472772     | -0.498387    | 0.159620     |
-------------------------------------------------------
| 5     | 1.323979     | -0.405270    | 0.306424     |
-------------------------------------------------------
| 6     | 0.946662     | -0.150403    | 0.561728     |
```

(continues on next page)

(continued from previous page)

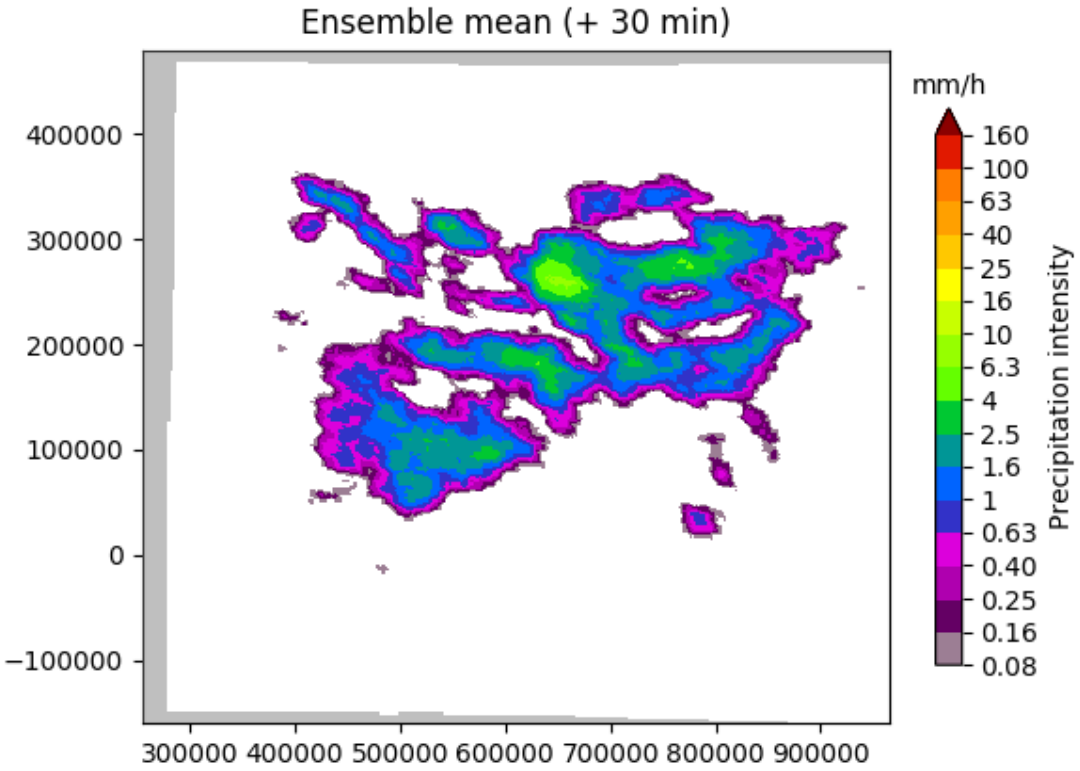
```
-------------------------------------------------------
| 7     | 0.486269     | 0.048149     | 0.858662     |
-------------------------------------------------------
| 8     | 0.127704     | 0.021603     | 0.991214     |
-------------------------------------------------------
Starting nowcast computation.
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.
```

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

Parameters:
-----------
number of time steps:     6
ensemble size:            20
parallel threads:         1
number of cascade levels: 6
order of the AR(p) model: 2
```

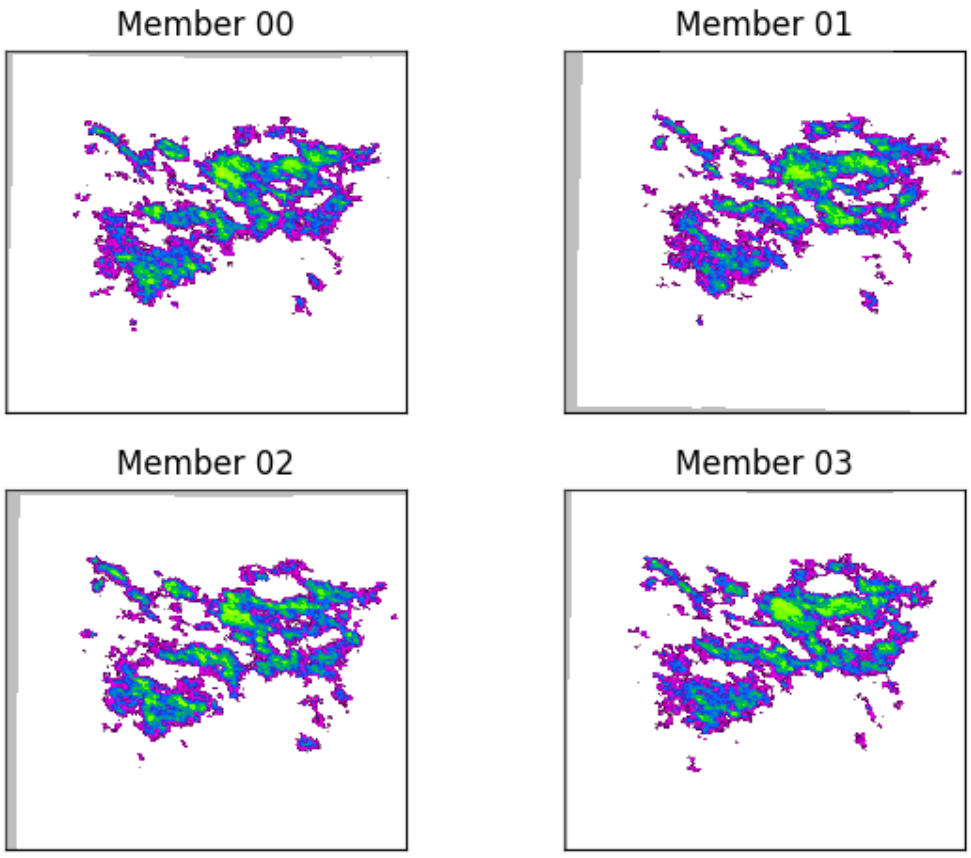
(continues on next page)

(continued from previous page)

```
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     | 0.999213      | 0.996961      |
----------------------------------------
| 2     | 0.997801      | 0.990552      |
----------------------------------------
| 3     | 0.989281      | 0.966368      |
----------------------------------------
| 4     | 0.940789      | 0.844223      |
----------------------------------------
| 5     | 0.724702      | 0.518336      |
----------------------------------------
| 6     | 0.220905      | 0.087744      |
----------------------------------------
**************************************
* AR(p) parameters for cascade levels: *
**************************************
-------------------------------------------------------
| Level |    Phi-1      |    Phi-2      |    Phi-0     |
-------------------------------------------------------
| 1     | 1.922158      | -0.923673     | 0.015204     |
-------------------------------------------------------
| 2     | 1.871560      | -0.875684     | 0.032004     |
-------------------------------------------------------
| 3     | 1.560428      | -0.577335     | 0.119228     |
-------------------------------------------------------
| 4     | 1.275302      | -0.355567     | 0.316840     |
-------------------------------------------------------
| 5     | 0.735167      | -0.014441     | 0.688991     |
-------------------------------------------------------
| 6     | 0.211861      | 0.040943      | 0.974477     |
-------------------------------------------------------
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.
```

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.

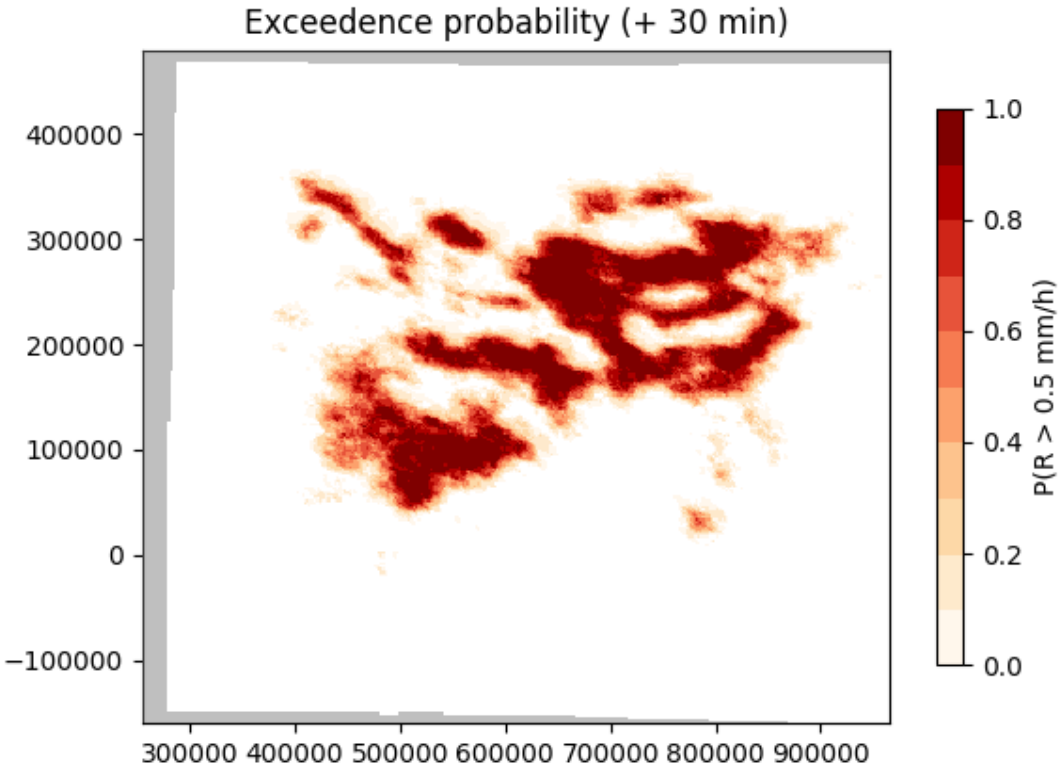
```
# Plot some of the realizations
fig = figure()
for i in range(4):
    ax = fig.add_subplot(221 + i)
    ax.set_title("Member %02d" % i)
    plot_precip_field(R_f[i, -1, :, :], geodata=metadata, colorbar=False, axis="off
↪")
tight_layout()
```




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
```




**Total running time of the script:** ( 0 minutes 24.531 seconds)

**Note:** Click *here* to download the full example code

### Ensemble verification

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

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

# Set nowcast parameters
n_ens_members = 20
n_leadtimes = 6
seed = 24
```

### Read precipitation field

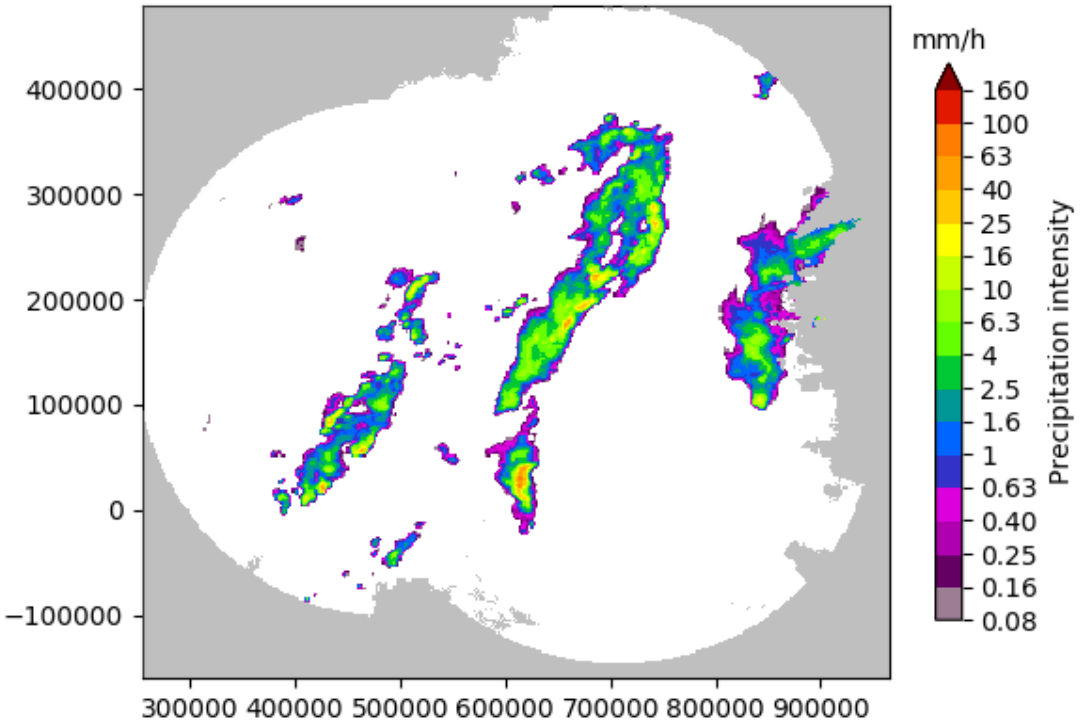
First thing, the sequence of Swiss radar composites is imported, converted and transformed into units of dBR.

```
date = datetime.strptime("201607112100", "%Y%m%d%H%M")
data_source = "mch"
```

(continues on next page)

(continued from previous page)

```
# 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:

```
{'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}
```

### Forecast

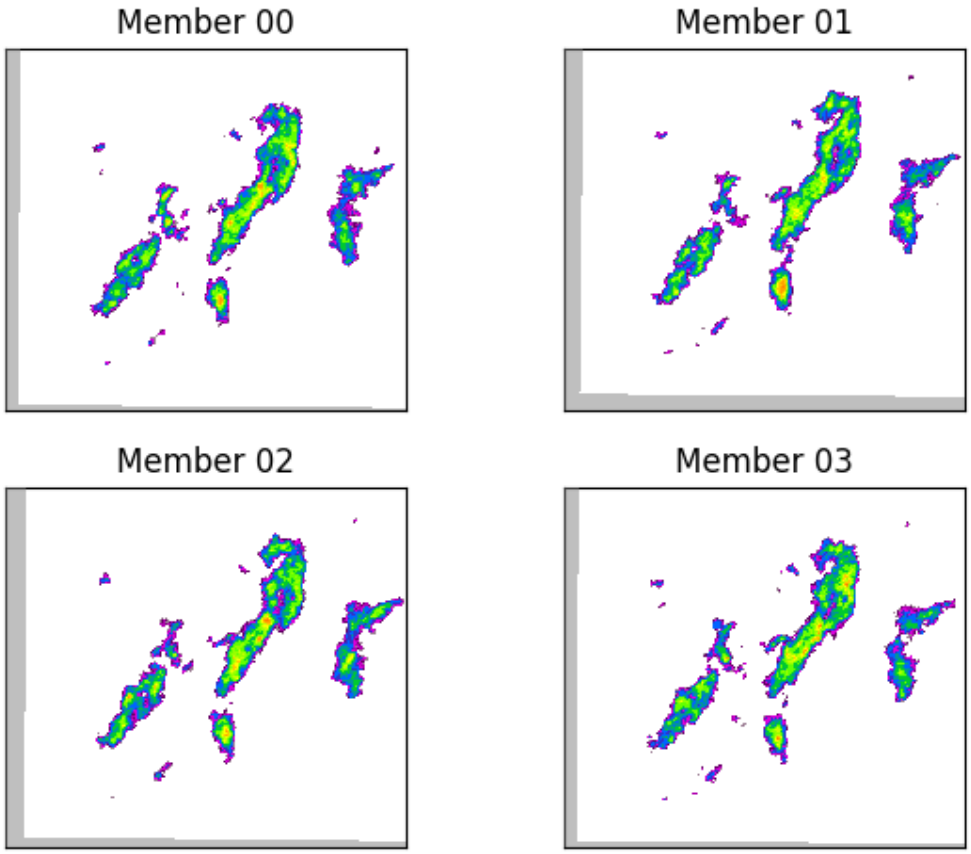
We use the STEPS approach to produce a ensemble nowcast of precipitation fields.

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

# The STEPES nowcast
```

(continues on next page)

(continued from previous page)

```
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 = 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
↪")
tight_layout()
```



Out:

```
Computing the motion field with the Lucas-Kanade method.
--- LK found 143 sparse vectors ---
--- 57 sparse vectors left for interpolation ---
--- 0.28 seconds ---
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

Parameters:
-----------
number of time steps:     6
ensemble size:            20
parallel threads:         1
number of cascade levels: 6
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     | 0.998969      | 0.995469      |
----------------------------------------
| 2     | 0.997921      | 0.991668      |
----------------------------------------
| 3     | 0.993385      | 0.978192      |
----------------------------------------
| 4     | 0.955325      | 0.865544      |
----------------------------------------
| 5     | 0.759267      | 0.519884      |
----------------------------------------
| 6     | 0.216036      | 0.058508      |
----------------------------------------
**************************************
* AR(p) parameters for cascade levels: *
**************************************
------------------------------------------------------
| Level |    Phi-1     |    Phi-2     |    Phi-0     |
------------------------------------------------------
| 1     | 1.911179     | -0.913151    | 0.018504     |
```

(continues on next page)

(continued from previous page)

```
-------------------------------------------------------
| 2     | 1.874990     | -0.878897    | 0.030745     |
-------------------------------------------------------
| 3     | 1.642916     | -0.653857    | 0.086883     |
-------------------------------------------------------
| 4     | 1.470428     | -0.539192    | 0.248914     |
-------------------------------------------------------
| 5     | 0.860742     | -0.133649    | 0.644941     |
-------------------------------------------------------
| 6     | 0.213353     | 0.012416     | 0.976310     |
-------------------------------------------------------
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.
```

### Verification

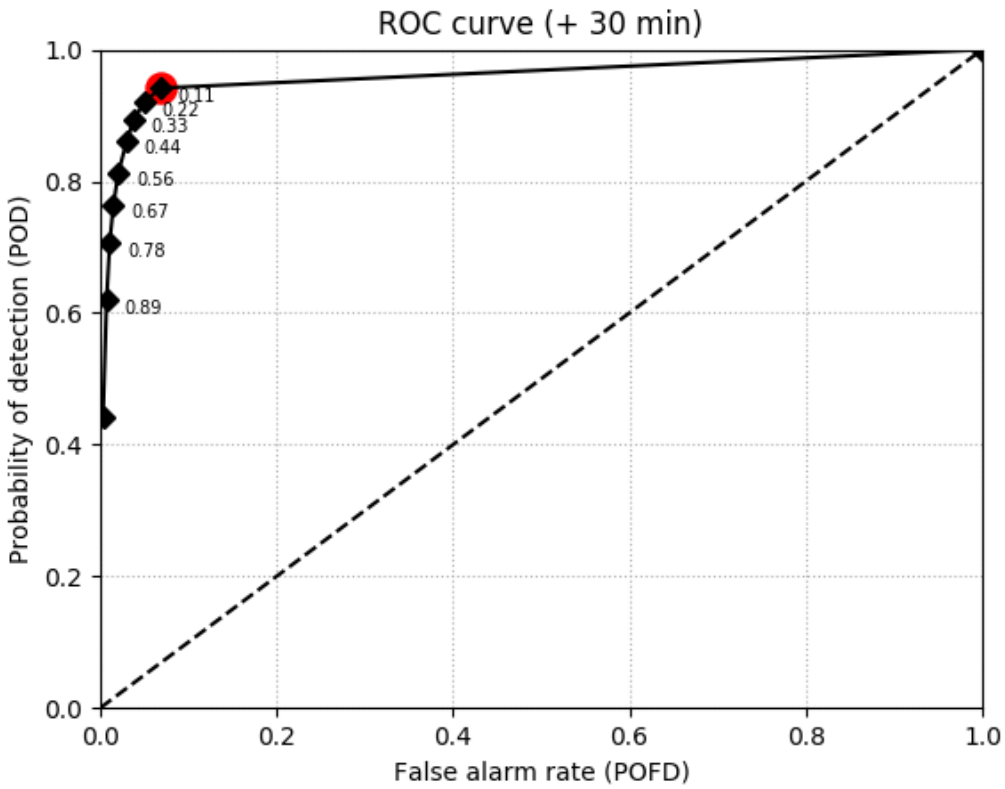
Pysteps includes a number of verification metrics to help users to analyze the general characteristics of the nowcasts in terms of consistency and quality (or goodness). Here, we will verify our probabilistic forecasts using the ROC curve, reliability diagrams, and rank histograms, as implemented in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

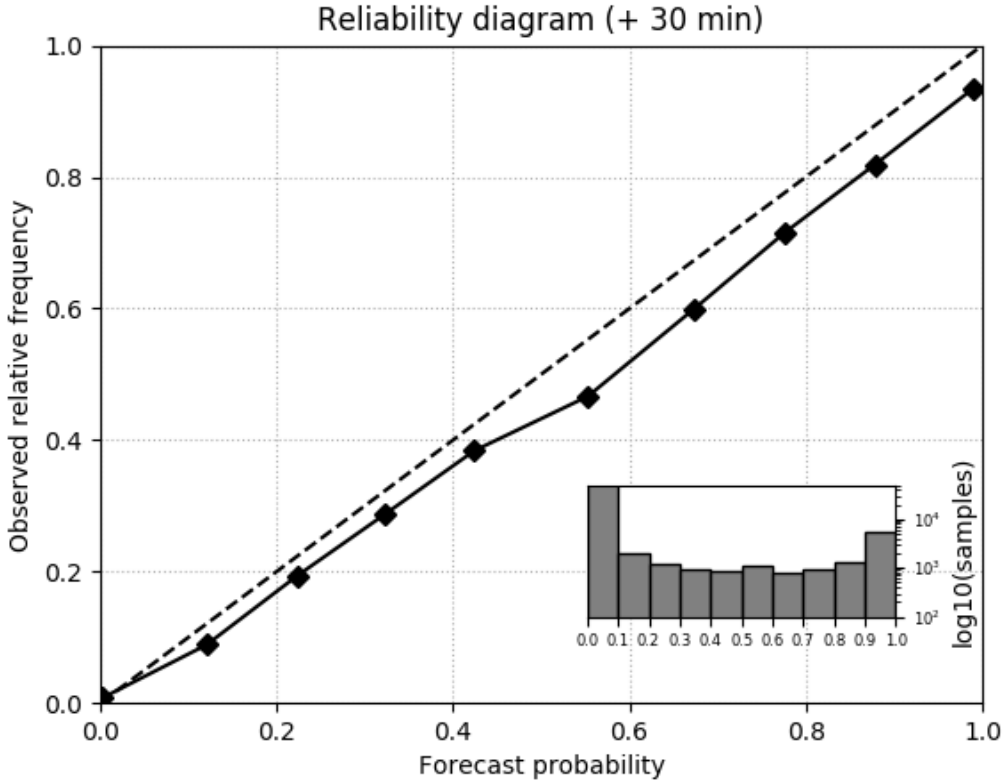
```
# Find the files containing the verifying observations
fns = io.archive.find_by_date(
    date,
    root_path,
    path_fmt,
    fn_pattern,
    fn_ext,
    timestep,
    0,
    num_next_files=n_leadtimes,
)

# 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)

# compute and plot the 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 = subplots()
verification.plot_ROC(roc, ax, opt_prob_thr=True)
ax.set_title("ROC curve (+ %i min)" % (n_leadtimes * timestep))

# compute and plot the reliability diagram
reldiag = verification.reldiag_init(0.1)
verification.reldiag_accum(reldiag, P_f, R_o[-1, :, :])
```

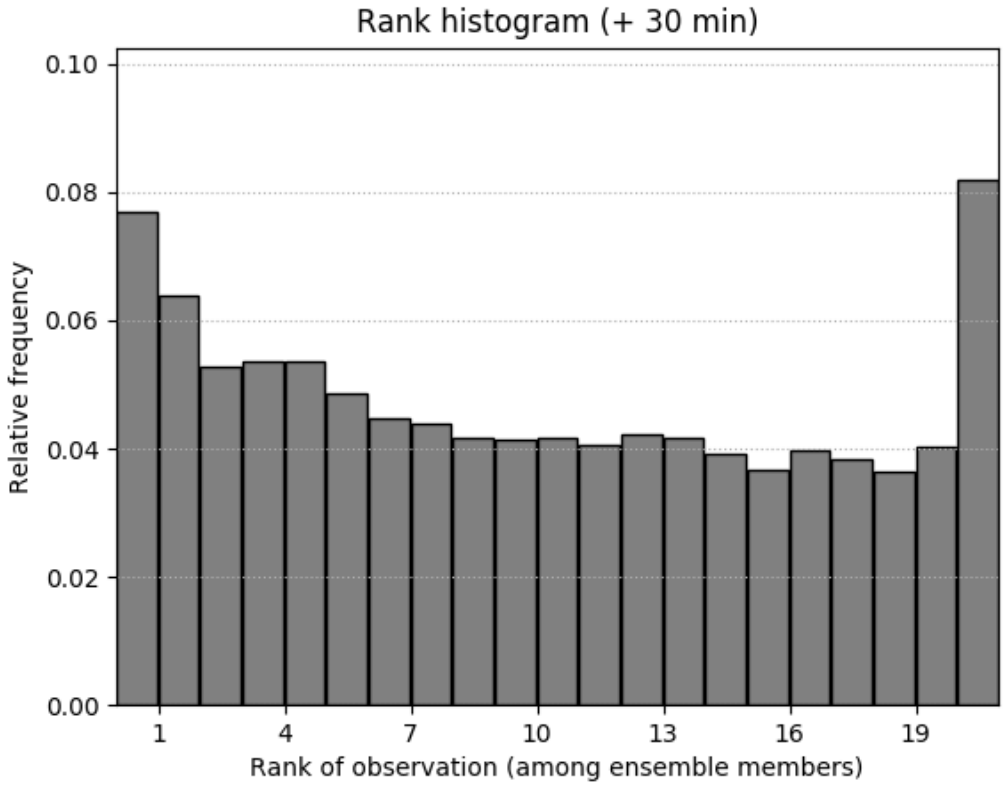
(continues on next page)

(continued from previous page)

```
fig, ax = subplots()
verification.plot_reldiag(reldiag, ax)
ax.set_title("Reliability diagram (+ %i min)" % (n_leadtimes * timestep))

# compute and plot the rank histogram
rankhist = verification.rankhist_init(R_f.shape[0], 0.1)
verification.rankhist_accum(rankhist, R_f[:, -1, :, :], R_o[-1, :, :])
fig, ax = subplots()
verification.plot_rankhist(rankhist, ax)
ax.set_title("Rank histogram (+ %i min)" % (n_leadtimes * timestep))

# sphinx_gallery_thumbnail_number = 5
```

- 

- 


- 


**Total running time of the script:** ( 0 minutes 23.097 seconds)

## 2.2 pySTEPS reference

**Release**
1.0.0

**Date**
Apr 07, 2019

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

### 2.2.1 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 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 decomposition method corresponding to the given name.

Filter methods:

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

Decomposition methods:

| Name | Description |
|---|---|
| fft | decomposition based on Fast Fourier Transform (FFT) and a bandpass filter |

#### 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 |

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

[PCH18]

## pysteps.cascade.decomposition

Methods for decomposing two-dimensional images into multiple spatial scales.

The methods in this module implement the following interface:

```
decomposition_xxx(X, filter, **kwargs)
```

where X is the input field and filter is a dictionary returned by a filter method implemented in *pysteps.cascade.bandpass_filters*. 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) |
| means | list of mean values for each cascade level |
| stds | list of standard deviations for each cascade level |

### Available methods

| | |
|---|---|
| *decomposition_fft*(X, filter, **kwargs) | Decompose a 2d input field into multiple spatial scales by using the Fast Fourier Transform (FFT) and a bandpass filter. |

## pysteps.cascade.decomposition.decomposition_fft

pysteps.cascade.decomposition.**decomposition_fft**(*X*, *filter*, ***kwargs*)
Decompose a 2d input field into multiple spatial scales by using the Fast Fourier Transform (FFT) and a bandpass filter.

**Parameters**

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

**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 number 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".

**MASK**
[array_like] Optional mask to use for computing the statistics for the cascade levels. Pixels with MASK==False are excluded from the computations.

### 2.2.2 pysteps.extrapolation

Extrapolation module functions and interfaces.

#### pysteps.extrapolation.interface

The functions in the extrapolation module implement the following interface:

```
extrapolate(extrap, precip, velocity, num_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. num_timesteps is an integer specifying the number of time steps to extrapolate. 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, ...) | 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*, *num_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.

**num_timesteps**
[int] Number 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, num_timesteps) | Apply semi-Lagrangian extrapolation to a two-dimensional precipitation field. |

## pysteps.extrapolation.semilagrangian.extrapolate

pysteps.extrapolation.semilagrangian.**extrapolate**(*precip*, *velocity*, *num_timesteps*, *outval=nan*, *xy_coords=None*, ***kwargs*)

Apply semi-Lagrangian 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.

**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.

**num_timesteps**
[int] Number of time steps to extrapolate.

**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.

**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. Default : 3

**inverse**
[bool] If True, the extrapolation trajectory is computed backward along the flow (default), forward otherwise. Default : True

**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)

### 2.2.3 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_pgm | PGM files used in the Finnish Meteorological Institute (FMI) archive, containing reflectivity composites (dBZ). |
| 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. |
| odim_hdf5 | ODIM HDF5 file format used by Eumetnet/OPERA. |
| knmi_hdf5 | HDF5 file format used by KNMI. |

Exporters:

| Name | Description |
|---|---|
| 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**
[str] Type of the method. The available options are 'importer' and 'exporter'.

## 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*)

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.

**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 2d precipitation fields.

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 precipitation field, 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 mandatory key-value pairs:

| Key | Value |
|---|---|
| projection | PROJ.4-compatible projection definition |
| x1 | x-coordinate of the lower-left corner of the data raster (meters) |
| y1 | y-coordinate of the lower-left corner of the data raster (meters) |
| x2 | x-coordinate of the upper-right corner of the data raster (meters) |
| y2 | y-coordinate of the upper-right corner of the data raster (meters) |
| xpixelsize | grid resolution in x-direction (meters) |
| ypixelsize | grid resolution in y-direction (meters) |
| 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 | the value assigned to the no rain pixels with the same unit, transformation and accutime of the data. |

### Available Importers

| | |
|---|---|
| *import_bom_rf3*(filename, **kwargs) | Import a NetCDF radar rainfall product from the BoM Rainfields3. |
| *import_fmi_pgm*(filename, **kwargs) | Import a 8-bit PGM radar reflectivity composite from the FMI archive. |
| *import_mch_gif*(filename, product, unit, accutime) | Import a 8-bit gif radar reflectivity composite from the MeteoSwiss archive. |
| *import_mch_hdf5*(filename, **kwargs) | Import a precipitation field (and optionally the quality field) from a 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_odim_hdf5*(filename, **kwargs) | Import a precipitation field (and optionally the quality field) from a HDF5 file conforming to the ODIM specification. |
| *import_knmi_hdf5*(filename, **kwargs) | Import a precipitation field (and optionally the quality field) from a HDF5 file conforming to the KNMI Data Centre specification. |

### 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.

### pysteps.io.importers.import_fmi_pgm

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

Import a 8-bit PGM radar reflectivity composite from the FMI archive.

**Parameters**

**filename**
[str] Name of the file to import.

**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**

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

### pysteps.io.importers.import_mch_gif

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

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

**Parameters**

**filename**
[str] Name of the file to import.

**product**
[{"AQC", "CPC", "RZC", "AZC"}] The name of the MeteoSwiss QPE product.

Currently supported prducts:

| Name | Product |
|---|---|
| AQC | Acquire |
| CPC | CombiPrecip |
| RZC | Precip |
| AZC | RZC accumulation |

**unit**
[{"mm/h", "mm", "dBZ"}] 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.

### pysteps.io.importers.import_mch_hdf5

pysteps.io.importers.**import_mch_hdf5**(*filename*, ***kwargs*)

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

**Parameters**

**filename**
[str] Name of the file to import.

**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**

**qty**
[{'RATE', 'ACRR', 'DBZH'}] 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'.

### 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**
[{"AQC", "CPC", "RZC", "AZC"}] The name of the MeteoSwiss QPE product.

Currently supported prducts:

| Name | Product |
|---|---|
| AQC | Acquire |
| CPC | CombiPrecip |
| RZC | Precip |
| AZC | RZC accumulation |

**unit**
[{"mm/h", "mm", "dBZ"}] 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.

### pysteps.io.importers.import_odim_hdf5

pysteps.io.importers.**import_odim_hdf5**(*filename*, ***kwargs*)

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

**Parameters**

**filename**
[str] Name of the file to import.

**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**

**qty**
[{'RATE', 'ACRR', 'DBZH'}] 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'.

### pysteps.io.importers.import_knmi_hdf5

pysteps.io.importers.**import_knmi_hdf5**(*filename*, ***kwargs*)

Import a precipitation 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.

**Returns**

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

**Other Parameters**

**accutime**
[float] The accumulation time of the dataset in minutes.

**pixelsize: float**
The pixelsize of a raster cell in meters.

### 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 (meters) |
| y1 | y-coordinate of the lower-left corner of the data raster (meters) |
| x2 | x-coordinate of the upper-right corner of the data raster (meters) |
| y2 | y-coordinate of the upper-right corner of the data raster (meters) |
| xpixelsize | grid resolution in x-direction (meters) |
| ypixelsize | grid resolution in y-direction (meters) |
| 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, **kwargs) | Read a nowcast or a nowcast ensemble 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*, ***kwargs*)
Read a nowcast or a nowcast ensemble from a NetCDF file conforming to the CF 1.7 specification.

### pysteps.io.exporter

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(filename, startdate, timestep,
                                 num_timesteps, shape, num_ens_members,
                                 metadata, incremental=None)
```

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

This function creates the file and writes the metadata. The datasets are written by calling *pysteps.io.exporters.export_forecast_dataset()*, and the file is closed by calling *pysteps.io.exporters.close_forecast_file()*.

The arguments in the above are defined as follows:

| Argument | Type/values | Description |
|---|---|---|
| filename | str | name of the output file |
| 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 | two-element tuple defining the shape (height,width) of the forecast grids |
| n_ens_members | int | number of ensemble members in the forecast. This argument is ignored if incremental is set to 'member' |
| metadata | dict | metadata dictionary containing the projection,x1,x2,y1,y2 and unit attributes described in the documentation of pysteps.io.importers |
| incremental | {None, 'timestep', 'member'} | Allow incremental writing of datasets into the netCDF file 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 |

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

### Available Exporters

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

### pysteps.io.exporters.initialize_forecast_exporter_kineros

pysteps.io.exporters.**initialize_forecast_exporter_kineros**(*filename*, *startdate*, *timestep*, *n_timesteps*, *shape*, *n_ens_members*, *metadata*, *incremental=None*)

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 memeber.

**Parameters**

**filename**
[str] Name of the output file.

**startdate**
[datetime.datetime] Start date of the forecast as datetime object.

**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.

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

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

**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**(*filename*, *startdate*, *timestep*, *n_timesteps*, *shape*, *n_ens_members*, *metadata*, *incremental=None*)

Initialize a netCDF forecast exporter.

**Parameters**

**filename**
[str] Name of the output file.

**startdate**
[datetime.datetime] Start date of the forecast as datetime object.

**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.

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

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

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

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*(F, exporter) | Write a forecast array into a file. |
| *close_forecast_file*(exporter) | Close the file associated with a forecast exporter. |

### pysteps.io.exporters.export_forecast_dataset

pysteps.io.exporters.**export_forecast_dataset**(*F*, *exporter*)
Write a forecast array into a file.

The written dataset has dimensions (num_ens_members,num_timesteps,shape[0],shape[1]), where shape refers to the shape of the two-dimensional forecast grids. 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*.

**F**
[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]) |

### pysteps.io.exporters.close_forecast_file

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

Close the file associated with a forecast exporter.

Finish writing forecasts and close the file associated with 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.

## 2.2.4 pysteps.motion

Implementations of optical flow methods.

### pysteps.motion.darts

Implementation of the DARTS algorithm.

| | |
|---|---|
| *DARTS*(Z, **kwargs) | Compute the advection field from a sequence of input images by using the DARTS method. |

### pysteps.motion.darts.DARTS

pysteps.motion.darts.**DARTS**(*Z*, ***kwargs*)

Compute the advection field from a sequence of input images by using the DARTS method. [RCW11]

**Parameters**

**Z**
[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.

**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'.

**print_info**
[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

OpenCV implementation of the Lucas-Kanade method with interpolated motion vectors for areas with no precipitation.

| |
|---|
| *dense_lucaskanade*(R, **kwargs) |

### pysteps.motion.lucaskanade.dense_lucaskanade

pysteps.motion.lucaskanade.**dense_lucaskanade**(*R*, ***kwargs*)

OpenCV implementation of the local Lucas-Kanade method with interpolation of the sparse motion vectors to fill the whole grid.

**Parameters**

**R**
[ndarray or MaskedArray] Array of shape (T,m,n) containing a sequence of T two-dimensional input images of shape (m,n).

In case of an ndarray, invalid values (Nans or infs) are masked. The mask in the MaskedArray defines a region where velocity vectors are not computed.

**Returns**

**out**
[ndarray] Three-dimensional array (2,m,n) containing the dense x- and y-components of the motion field. Return an empty array when no motion vectors are found.

**Other Parameters**

**buffer_mask**
[int, optional] A mask buffer width in pixels. This extends the input mask (if any) to help avoiding the erroneous interpretation of velocities near the maximum range of the radars (0 by default).

**max_corners_ST**
[int, optional] The maxCorners parameter in the Shi-Tomasi corner detection method. It represents the maximum number of points to be tracked (corners), by default this is 500. If set to zero, all detected corners are used.

**quality_level_ST**
[float, optional] The qualityLevel parameter in the Shi-Tomasi corner detection method. It represents the minimal accepted quality for the points to be tracked (corners), by default this is set to 0.1. Higher quality thresholds can lead to no detection at all.

**min_distance_ST**
[int, optional] The minDistance parameter in the Shi-Tomasi corner detection method. It represents minimum possible Euclidean distance in pixels between corners, by default this is set to 3 pixels.

**block_size_ST**
[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 neighborhood, by default this is set to 15 pixels.

**winsize_LK**
: [tuple of int, optional] The winSize parameter in the Lucas-Kanade optical flow method. It represents the size of the search window that it is used at each pyramid level, by default this is set to (50, 50) pixels.

**nr_levels_LK**
: [int, optional] The maxLevel parameter in the Lucas-Kanade optical flow method. It represents the 0-based maximal pyramid level number, by default this is set to 3.

**nr_IQR_outlier**
: [int, optional] Maximum acceptable deviation from the median velocity value in terms of number of inter quantile ranges (IQR). Any velocity that is larger than this value is flagged as outlier and excldued from the interpolation. By default this is set to 3.

**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. By default this is set to 3. If set to zero, the fitlering is not perfomed.

**decl_grid**
: [int, optional] The cell size in pixels of the declustering grid that is used to filter out outliers in a sparse motion field and get more representative data points before the interpolation. This simply computes new sparse vectors over a coarser grid by taking the median of all vectors within one cell. By default this is set to 20 pixels. If set to less than 2 pixels, the declustering is not perfomed.

**min_nr_samples**
: [int, optional] The minimum number of samples necessary for computing the median vector within given declustering cell, otherwise all sparse vectors in that cell are discarded. By default this is set to 2.

**rbfunction**
: [string, optional] The name of the radial basis function used for the interpolation of the sparse vectors. This is based on the Euclidian norm d. By default this is set to "inverse" and the available names are "nearest", "inverse", "gaussian".

**k**
: [int, optional] The number of nearest neighbors used for fast interpolation, by default this is set to 20. If set equal to zero, it employs all the neighbors.

**epsilon**
: [float, optional] The adjustable constant used in the gaussian and inverse radial basis functions. by default this is computed as the median distance between the sparse vectors.

**nchunks**
: [int, optional] Split the grid points in n chunks to limit the memory usage during the interpolation. By default this is set to 5, if set to 1 the interpolation is computed with the whole grid.

**extra_vectors**
: [ndarray, optional] Additional sparse motion vectors as 2d array (columns: x,y,u,v; rows: nbr. of vectors) to be integrated with the sparse vectors from the Lucas-Kanade local tracking. x and y must be in pixel coordinates, with (0,0) being the upper-left corner of the field R. u and v must be in pixel units. By default this is set to None.

**verbose**
: [bool, optional] If set to True, it prints information about the program (True by default).

### 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.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, ...) | |
| *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 sum of the residuals of the squared differences of the images pixels and the contribution of a smoothness constrain.

In order to find the minimum an scaling guess procedure is applied, from larger scales to a finer scale. This reduces the changes that the minimization procedure converges to a local minimum. The scaling guess is defined by the scaling sectors (see **sectors** keyword).

The smoothness of the returned displacement field is controlled by the smoothness constrain gain (**smooth_gain** keyword).

If a first guess is not given, zero displacements are used as first guess.

To minimize the cost function, the scipy minimization function is used with the 'CG' method. This method proved to give the best results under any 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] The number of sectors for each dimension used in the scaling procedure. If dimension is 1, the same sectors will be used both image dimensions (x and y). If is 2D, the each row determines the sectors of the each dimension.

**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).

**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.

### 2.2.5 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(X, **kwargs)
```

or:

```
initialize_nonparam_2d_xxx_filter(X, **kwargs)
```

where X 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 F 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(F, randstate=np.random, seed=None, **kwargs)
```

where F (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*(X, **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*(X, **kwargs) | Takes one ore more 2d input fields and produces one non-paramtric, global and anasotropic fourier filter. |
| *initialize_nonparam_2d_nested_filter*(X[, ...]) | Function to compute the local Fourier filters using a nested approach. |
| *initialize_nonparam_2d_ssft_filter*(X, **kwargs) | 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**(*X*, ***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**

**X**

[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 a dictionary containing the keys F, 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_type**
[{'hanning', 'flat-hanning' or None}] Optional tapering function to be applied to X, generated with `pysteps.noise.fftgenerators.build_2D_tapering_function()` (default None).

**model**
[{'power-law'}] The name of the parametric model to be used to fit the power spectrum of X (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 `pysteps.utils.fft.get_method()`). Defaults to "numpy".

### pysteps.noise.fftgenerators.initialize_nonparam_2d_fft_filter

`pysteps.noise.fftgenerators.`**`initialize_nonparam_2d_fft_filter`**(*X*, ***kwargs*)
Takes one ore more 2d input fields and produces one non-paramtric, global and anasotropic fourier filter.

**Parameters**

**X**
[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 F 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_type**
[{'hanning', 'flat-hanning'}] Optional tapering function to be applied to X, generated with `pysteps.noise.fftgenerators.build_2D_tapering_function()` (default 'flat-hanning').

**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 `pysteps.utils.fft.get_method()`). Defaults to "numpy".

### pysteps.noise.fftgenerators.initialize_nonparam_2d_nested_filter

pysteps.noise.fftgenerators.**initialize_nonparam_2d_nested_filter**(*X*, *gridres=1.0*, ***kwargs*)

Function to compute the local Fourier filters using a nested approach.

**Parameters**

**X**
[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**

**F**
[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_type**
[{'hanning', 'flat-hanning'}] Optional tapering function to be applied to X, generated with `pysteps.noise.fftgenerators.build_2D_tapering_function()` (default 'flat-hanning').

**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 `pysteps.utils.fft.get_method()`). Defaults to "numpy".

### pysteps.noise.fftgenerators.initialize_nonparam_2d_ssft_filter

pysteps.noise.fftgenerators.**initialize_nonparam_2d_ssft_filter**(*X*, ***kwargs*)

Function to compute the local Fourier filters using the Short-Space Fourier filtering approach.

**Parameters**

**X**
[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**

**F**
[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_type**
[{'hanning', 'flat-hanning'}] Optional tapering function to be applied to X, generated with `pysteps.noise.fftgenerators.build_2D_tapering_function()` (default 'flat-hanning').

**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 `pysteps.utils.fft.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*)

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 `pysteps.utils.fft.get_method()`). Defaults to "numpy".

**Returns**

**N**
[array-like] A two-dimensional numpy array of stationary correlated noise.

### 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_type**
[{'hanning', 'flat-hanning'}] Optional tapering function to be applied to X, generated with `pysteps.noise.fftgenerators.build_2D_tapering_function()` (default 'flat-hanning').

**fft_method**
[str or tuple] A string or a (function,kwargs) tuple defining the FFT method to use (see `pysteps.utils.fft.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.

## 2.2.6 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 |
|---|---|
| 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.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'*, *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, optional] Specifies the threshold value for minimum observable precipitation intensity. Required if mask_method is not None or conditional is True.

**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'.

**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]

## 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*, *minwet=50*, *extrap_method='semilagrangian'*, *decomp_method='fft'*, *bandpass_filter_method='gaussian'*, *noise_method='nonparametric'*, *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={}*, *filter_kwargs={}*, *noise_kwargs={}*, *vel_pert_kwargs={}*, *mask_kwargs={}*, *measure_time=True*)

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*.

**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.

**minwet**
[int] Minimum number of wet pixels accepted 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'*, *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'.

**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]

## 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[, donorm]) | Stack the given cascades into a larger array. |
| *recompose_cascade*(R, mu, sigma) | Recompose a cascade by inverting the normalization and summing the cascade levels. |

## 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*, *donorm=True*)
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] Number of cascade levels.

**donorm**
[bool] If True, normalize the cascade levels before stacking.

**Returns**

**out**
[tuple] A three-element tuple containing a four-dimensional array of stacked cascade levels and lists of mean values and standard deviations for each cascade level (taken from the last cascade).

#### pysteps.nowcasts.utils.recompose_cascade

`pysteps.nowcasts.utils.`**`recompose_cascade`**(*R*, *mu*, *sigma*)
Recompose a cascade by inverting the normalization and summing the cascade levels.

**Parameters**

**R**
[array_like]

### 2.2.7 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. |

#### 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 (n_members,m,n) containing an 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.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. |
| *nonparam_match_empirical_cdf*(R, R_trg) | 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**(*R*, *R_trg*)

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**

**R**
[array_like] The initial array whose CDF is to be matched with the target.

**R_trg**
[array_like] The target array.

**Returns**

**out**
[array_like] The matched 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.

### 2.2.8 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. |

Continued on next page

Table 28 – continued from previous page

| | |
|---|---|
| *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_yw*(gamma) | Estimate the parameters of an AR(p) model from the Yule-Walker equations using the given set of autocorrelation coefficients. |
| *iterate_ar_model*(X, phi[, EPS]) | Apply an AR(p) model to a time-series of two-dimensional fields. |

### 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_yw

pysteps.timeseries.autoregression.**estimate_ar_params_yw**(*gamma*)
Estimate the parameters of an AR(p) model from the Yule-Walker equations using the given set of 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.

**Returns**

**out**
[ndarray] An array of shape (n,p+1) containing the AR(p) parameters for for the lag-p terms for each cascade level, and also the standard deviation of the innovation term.

### pysteps.timeseries.autoregression.iterate_ar_model

pysteps.timeseries.autoregression.**iterate_ar_model**(*X*, *phi*, *EPS=None*)
Apply an AR(p) model to a time-series of two-dimensional fields.

**Parameters**

**X**
[array_like] Three-dimensional array of shape (p,w,h) containing a time series of p two-dimensional fields of shape (w,h). The fields are assumed to be in ascending order by time, and the timesteps are assumed to be regular.

**phi**
[array_like] 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 perturbation field for the AR(p) process. 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[, MASK]) | Compute lag-l autocorrelation coefficients gamma_l, l=1,2,...,n-1, for a time series of n two-dimensional input fields. |

#### pysteps.timeseries.correlation.temporal_autocorrelation

pysteps.timeseries.correlation.**temporal_autocorrelation**(*X*, *MASK=None*)

Compute lag-l autocorrelation coefficients gamma_l, l=1,2,...,n-1, for a time series of n two-dimensional input fields.

**Parameters**

**X**
[array_like] Two-dimensional array of shape (n, w, h) containing a time series of n two-dimensional fields of shape (w, h). The input fields are assumed to be in increasing order with respect to time, and the time step is assumed to be regular (i.e. no missing data). X is required to have finite values.

**MASK**
[array_like] Optional mask to use for computing the correlation coefficients. Pixels with MASK==False are excluded from the computations.

**Returns**

**out**
[ndarray] Array of length n-1 containing the temporal autocorrelation coefficients for time lags l=1,2,...,n-1.

### 2.2.9 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 |

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] |

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 |

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 |

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 |

## 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.conversion

Methods for converting physical units.

| | |
|---|---|
| *to_rainrate*(R, metadata[, a, b]) | Convert to rain rate [mm/h]. |
| *to_raindepth*(R, metadata[, a, b]) | Convert to rain depth [mm]. |
| *to_reflectivity*(R, metadata[, a, b]) | Convert to reflectivity [dBZ]. |

## pysteps.utils.conversion.to_rainrate

pysteps.utils.conversion.**to_rainrate**(*R*, *metadata*, *a=None*, *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 a=b=None.

**a,b**
[float, optional] The a and b coefficients of the Z-R relationship.

**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*, *a=None*, *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 a=b=None.

**a,b**
[float, optional] The a and b coefficients of the Z-R relationship.

**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*, *a=None*, *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 a=b=None.

**a,b**
[float, optional] The a and b coefficients of the Z-R relationship.

**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_time*(R, metadata, ...[, ...]) | Aggregate fields in time. |
| *aggregate_fields_space*(R, metadata, ...[, ...]) | Upscale fields in space. |
| *aggregate_fields*(R, window_size[, axis, method]) | Aggregate fields. |
| *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_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_m*, *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_m**
[float or None] The length in meters of the space window that is used to upscale the fields. The space spanned by the m and n dimensions of R must be a multiple of space_window_m. 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*delta/space_window_m and j = n*delta/space_window_m; delta is the grid size.

**metadata**
[dict] The metadata with updated attributes.

**See also:**

*pysteps.utils.dimension.aggregate_fields_time*, *pysteps.utils.dimension.aggregate_fields*

### pysteps.utils.dimension.aggregate_fields

pysteps.utils.dimension.**aggregate_fields**(*R*, *window_size*, *axis=0*, *method='mean'*)
Aggregate fields. It attemps 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.

**Parameters**

**R**
[array-like] Array of any shape containing the input fields.

**window_size**
[int] The length of the window that is used to aggregate the fields.

**axis**
[int, optional] The axis where to perform the aggregation.

**method**
[string, optional] Optional argument that specifies the operation to use to aggregate the values within the window. Default to mean operator.

**Returns**

**outputarray**
[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.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.spectral

Utility methods for processing and analyzing precipitation fields in the Fourier domain.

| | |
|---|---|
| *rapsd*(Z[, fft_method, return_freq, d]) | 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. |

### pysteps.utils.spectral.rapsd

pysteps.utils.spectral.**rapsd**(*Z*, *fft_method=None*, *return_freq=False*, *d=1.0*, ***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'.

**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.transformation

Methods for transforming data values.

| | |
|---|---|
| *boxcox_transform*(R[, metadata, Lambda, ...]) | The one-parameter Box-Cox transformation. |
| *boxcox_transform_test_lambdas*(R[, Lambdas, ...]) | Test and plot various lambdas for the 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. |
| *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. Default parameters will produce a log transform (i.e. Lambda=0).

**Parameters**

**R**
[array-like] Array of any shape to be transformed.

**metadata**
[dict] Metadata dictionary containing the transform, zerovalue and threshold attributes as described in the documentation of *pysteps.io.importers*.

**Lambda**
[float] Parameter lambda of the Box-Cox transformation. Default : 0

**threshold**
[float] Optional value that is used for thresholding with the same units as R. If None, the threshold contained in metadata is used.

**zerovalue**
[float] Optional value to be assigned to no rain pixels as defined by the threshold.

**inverse**
[bool] Optional, if set to True, it performs the inverse transform

**Returns**

**R**
[array-like] Array of any shape containing the (back-)transformed units.

**metadata**
[dict] The metadata with updated attributes.

### pysteps.utils.transformation.boxcox_transform_test_lambdas

pysteps.utils.transformation.**boxcox_transform_test_lambdas**(*R*, *Lambdas=None*, *threshold=0.1*)

Test and plot various lambdas for the Box-Cox transformation.

### 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] Metadata dictionary containing the transform, zerovalue and threshold attributes as described in the documentation of *pysteps.io.importers*.

**threshold**
[float] Optional value that is used for thresholding with the same units as R. If None, the threshold contained in metadata is used.

**zerovalue**
[float] Optional value to be assigned to no rain pixels as defined by the threshold.

**inverse**
[bool] Optional, if set to True, it performs the inverse transform

**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. Zero rain vales are set to zero in norm space.

**Parameters**

**R**
[array-like] Array of any shape to be transformed.

**metadata**
[dict] 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

**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; typically in (0,1). Default : 0., i.e. it spaces the points evenly in the uniform distribution

#### 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] 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

**Returns**

**R**
[array-like] Array of any shape containing the (back-)transformed units.

**metadata**
[dict] The metadata with updated attributes.

### 2.2.10 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) |
| FA | false alarm rate (prob. of false detection) |
| FAR | false alarm ratio |
| GSS | Gilbert skill score (equitable threat score) |
| HK | Hanssen-Kuipers discriminant (Pierce skill score) |
| HSS | Heidke skill score |
| POD | probability of detection (hit rate) |
| SEDI | symmetric extremal dependency index |
| beta | linear regression slope (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 |
| 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.: 10log10(pred/obs).

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 update using _init, _accum and _compute methods of this module.

Score names denoted by * can only be computed offline.

### 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

## 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_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) |
| FA | false alarm rate (prob. of false detection) |
| FAR | false alarm ratio |
| GSS | Gilbert skill score (equitable threat score) |
| HK | Hanssen-Kuipers discriminant (Pierce skill score) |
| HSS | Heidke skill score |
| POD | probability of detection (hit 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_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) |
| FA | false alarm rate (prob. of false detection) |
| FAR | false alarm ratio |
| GSS | Gilbert skill score (equitable threat score) |
| HK | Hanssen-Kuipers discriminant (Pierce skill score) |
| HSS | Heidke skill score |
| POD | probability of detection (hit 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]) | Initialize a verification error object. |
| *det_cont_fct_accum*(err, pred, obs) | Accumulate the forecast error in the verification error object. |
| *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*)

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 |
|---|---|
| beta | linear regression slope (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 |
| 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 on zeros used for the verification. The default, conditioning=None, includes zero pairs. With conditioning='single', only pairs with either pred or obs > 0 are included. With conditioning='double', only pairs with both pred and obs > 0 are included.

**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.: 10log10(pred/obs).

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 update 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

### pysteps.verification.detcontscores.det_cont_fct_init

pysteps.verification.detcontscores.**det_cont_fct_init**(*axis=None*, *conditioning=None*)

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 on zeros used for the verification. The default, conditioning=None, includes zero pairs. With conditioning='single', only pairs with either pred or obs > 0 are included. With conditioning='double', only pairs with both pred and obs > 0 are included.

**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_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 |
|---|---|
| beta | linear regression slope (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 |
| 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, num_ens_members[, X_min, ...]) | 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).

**Other Parameters**

**thr**
[float] Intensity threshold for categorical scores.

**scale**
[int] The spatial scale to verify in px. In practice it represents the size of the moving window that it is used to compute the fraction of pixels above the threshold for 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 skill metric to be used, the list includes:

**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).

**Other Parameters**

**thr**
[float] Intensity threshold for categorical scores.

**scale**
[int] The spatial scale to verify in px. In practice it represents the size of the moving window that it is used to compute the fraction of pixels above the threshold for the FSS.

### References

[ZR09]

### pysteps.verification.ensscores.rankhist

pysteps.verification.ensscores.**rankhist**(*X_f*, *X_o*, *num_ens_members*, *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.

### 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. |
| *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*(iss[, fig, vmin, vmax, ...]) | 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**(*iss*, *fig=None*, *vmin=-2*, *vmax=1*, *kmperpixel=None*, *unit=None*)

Plot a intensity-scale verification table with a color bar and axis labels.

**Parameters**

**iss**
[dict] An intensity-scale verification results dictionary returned by pysteps.verification.spatialscores.intensity_scale.

**fig**
[matplotlib.figure.Figure, optional] The figure object to use for plotting. If not supplied, a new figure is created.

**vmin**
[float, optional] The minimum value for the intensity-scale skill score in the plot. Defaults to -2.

**vmax**
[float, optional] The maximum value for the intensity-scale skill score in the plot. Defaults to 1.

**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 ROC curve 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). |
| *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 intensty-scale verification object. |
| *intensity_scale_accum*(intscale, X_f, X_o) | Compute and update the intensity-scale verification scores. |
| *intensity_scale_compute*(intscale) | Return the intensity scale matrix. |
| *binary_mse*(X_f, X_o, thr[, wavelet]) | Compute an intensity-scale verification as the MSE of the binary error. |
| *fss*(X_f, X_o, thr, scale) | Compute the fractions skill score (FSS) for a deterministic forecast field and the corresponding observation. |

### 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 (n,m) containing the forecast field.

**X_o**
[array_like] Array of shape (n,m) 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**
[sequence] A sequence of intensity thresholds for which to compute the verification.

**scales**
[sequence, optional] A sequence of spatial scales in pixels to be used in the 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.

### pysteps.verification.spatialscores.intensity_scale_init

pysteps.verification.spatialscores.**intensity_scale_init**(*name*, *thrs*, *scales=None*, *wavelet='Haar'*)

Initialize an intensty-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**
[sequence] A sequence of intensity thresholds for which to compute the verification.

**scales**
[sequence, optional] A sequence of spatial scales in pixels to be used in the 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**
[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.

**X_f**
[array_like] Array of shape (n,m) containing the forecast field.

**X_o**
[array_like] Array of shape (n,m) containing the verification observation field.

### 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.

**Returns**

**out**
[array_like] The two-dimensional array containing the intensity-scale skill scores for each given spatial scale and intensity threshold.

### pysteps.verification.spatialscores.binary_mse

pysteps.verification.spatialscores.**binary_mse**(*X_f*, *X_o*, *thr*, *wavelet='haar'*)
Compute an intensity-scale verification as the MSE of the binary error.

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 (n,m) containing the forecast field.

**X_o**
[array_like] Array of shape (n,m) 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.

**Returns**

**SS**
[array] One-dimensional array containing the binary MSE for each spatial scale.

**spatial_scale**
[list]

#### References

[CRS04]

### 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.

**Parameters**

**X_f**
[array_like] Array of shape (n,m) containing the forecast field.

**X_o**
[array_like] Array of shape (n,m) containing the reference field (observation).

**thr**
[float] Intensity threshold.

**scale**
[int] The spatial scale in px. In practice they represent 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]

### 2.2.11 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 (meters) |
| y1 | y-coordinate of the lower-left corner of the data raster (meters) |
| x2 | x-coordinate of the upper-right corner of the data raster (meters) |
| y2 | y-coordinate of the upper-right corner of the data raster (meters) |
| 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:

**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.

| Key | Value |
|---|---|
| projection | PROJ.4-compatible projection definition |
| x1 | x-coordinate of the lower-left corner of the data raster (meters) |
| y1 | y-coordinate of the lower-left corner of the data raster (meters) |
| x2 | x-coordinate of the upper-right corner of the data raster (meters) |
| y2 | y-coordinate of the upper-right corner of the data raster (meters) |
| 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 |

**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:

**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.

| Key | Value |
|---|---|
| projection | PROJ.4-compatible projection definition |
| x1 | x-coordinate of the lower-left corner of the data raster (meters) |
| y1 | y-coordinate of the lower-left corner of the data raster (meters) |
| x2 | x-coordinate of the upper-right corner of the data raster (meters) |
| y2 | y-coordinate of the upper-right corner of the data raster (meters) |
| 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 |

**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'*, *colorscale='pysteps'*, *probthr=None*, *title=None*, *colorbar=True*, *drawlonlatlines=False*, *basemap_resolution='l'*, *basemap_scale_args=None*, *cartopy_scale='50m'*, *lw=0.5*, *cartopy_subplot=(1, 1, 1)*, *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 exceedance probability map.

**type**
[{'intensity', 'depth', 'prob'}, optional] Type of the map to plot: 'intensity' = pre-

cipitation 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. 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 (meters) |
| y1 | y-coordinate of the lower-left corner of the data raster (meters) |
| x2 | x-coordinate of the upper-right corner of the data raster (meters) |
| y2 | y-coordinate of the upper-right corner of the data raster (meters) |
| 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.

**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'.

**basemap_resolution**
[str, optional] The resolution of the basemap, see the documentation of mpl_toolkits.basemap. Applicable if map is 'basemap'.

**basemap_scale_args**
[list, optional] If not None, a map scale bar is drawn with basemap_scale_args supplied to mpl_toolkits.basemap.Basemap.drawmapscale.

**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.

**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.

### 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 by the function 'rapsd' in utils/spectral.py

**fft_power: array-like**
1d array containing the radially averaged Fourier power spectrum computed by the function 'rapsd' in utils/spectral.py

**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.

#### Getting started, building, and testing

If you haven't already, take a look at the project's README.rst file. and the pysteps documentation. There you will find all the necessary information to install pysteps.

#### Code Style

Although it is not strictly enforced yet, we strongly suggest to follow the pep8 coding standards. Two popular modules used to check pep8 are pycodestyle and pylint.

You can install them using pip:

```
pip install pylint
pip install pycodestyle
```

or using anaconda:

```
conda install pylint
conda install pycodestyle
```

For further instructions please refer to their official documentation.

- https://pycodestyle.readthedocs.io/en/latest/
- https://www.pylint.org/

#### Discussion

You are welcome to start a discussion in the project's GitHub issue tracker if you have run into behavior in pysteps that you don't understand or you have found a bug or would like make a feature request.

#### Contributions workflow

#### Submitting Changes

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.

### Fork the repository

The first 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

### Preparing Changes

**IMPORTANT**

If your changes will take a significant amount of work, we highly recommend opening an issue first, explaining what do you want to do and why. It is better to start the discussions early in case that 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 completed a 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 commits 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.

### Core developer guidelines

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).

#### Testing your changes

Before committing changes or creating pull requests, check that the build-in tests passed. See the Test wiki 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.

#### 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.

### 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
```

#### 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 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 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-1.0.0.tar.gz
```

#### 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`.

To upload the recently created source distribution (**dist/pysteps-1.0.0.tar.gz**) under the **dist** directory run:

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/pysteps-1.0.0.tar.
↪gz
```

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-1.0.0.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 by running:

```
pip install --index-url https://test.pypi.org/simple/ pysteps
```

To test that the installation was successful, from a folder different than the pysteps source, run the pysteps's test suite:

```
pytest --pyargs pysteps
```

#### Uploaded package to the Official PyPi

Once the sdist package was tested, we can safely upload it to the Official PyPi repository with:

```
twine upload dist/pysteps-1.0.0.tar.gz
```

Now, **pysteps** can be installed by simply running:

```
pip install pysteps
```

### 2.3.4 Creating a pySTEPS conda package

Here we will describe the steps to create an anaconda package for pySTEPS.

#### What is a "conda package"?

A conda package is a compressed tarball file that may contain:

- python or other modules
- executables
- other important files

The package format is the same across platforms and their advantage is that Conda understands how to install these packages into a conda environment (regardless the platform where the package is installed) so that it may be available when that environment is active.

For more information about conda package, check the conda-build documentation

#### Building the pysteps conda package

Building a conda package requires installing conda-build and creating a conda recipe.

Before you start, make sure you have installed:

- conda
- conda-build
- any compilers you want.

#### Conda recipe

Building a conda package requires a recipe. A conda build recipe is a flat directory that contains the following files:

- **meta.yaml**: A file that contains all the metadata in the recipe.
- **build.sh**: The script that installs the files for the package on macOS and Linux. It is executed using the bash command.

- **bld.bat**: The build script that installs the files for the package on Windows. It is executed using **cmd**.
- **run_test.[py,pl,sh,bat]**: An optional Python test file, a test script that runs automatically if it is part of the recipe.
- **Optional patches** that are applied to the source.
- **Other resources** that are not included in the source and cannot be generated by the build scripts. Examples are icon files, readme files and build notes.

### Conda build process

The conda package is build using the *conda-build* command.

*conda-build* performs the following steps:

1. Reads the metadata.
2. Downloads the source into a cache.
3. Extracts the source into the source directory.
4. Applies any patches.
5. Re-evaluates the metadata, if source need to fill any metadata values.
6. Creates a build environment, and then installs the build dependencies there.
7. Runs the build script. The current working directory is the source directory with environment variables set. The build script installs into the build environment.
8. Performs some necessary post-processing steps, such as shebang and rpath.
9. Creates a conda package containing all the files in the build environment that are new from step 5, along with the necessary conda package metadata.
10. Tests the new conda package if the recipe includes tests:
    1. Deletes the build environment.
    2. Creates a test environment with the package and its dependencies.
    3. Runs the test scripts.

### Build pysteps's recipe

The conda recipe for pysteps is defined in the **pysteps/conda_recipe/meta.yaml** file. Before we actually build the package, let's specify the build root folder where the build packages will be stored. This folder should be located outside the pysteps root:

```
export CONDA_BLD_PATH=$HOME/localStorage/conda_builds/
```

On a linux distribution, from the pystep root directory we can now build the conda package for two python versions (or more if you like):

```
conda build --python=3.6 conda_recipe/

conda build --python=3.7 conda_recipe/
```

The anaconda packages for each python version are stored in the **$CONDA_BLD_PATH** folder.

### Upload the packages to the anaconda cloud

The final step is to upload the package to the anaconda cloud using:

```
anaconda upload <path_to_file_package>
```

### Credits

The description of the conda build process was adapted from the official conda-build documentation (Copyright 2012 Continuum Analytics, Inc.)

## 2.4 Blibliography

# 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.

[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.

[PCH18] 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.

[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*), 91
adjust_lag2_corrcoef2() (*in module pysteps.timeseries.autoregression*), 91
aggregate_fields() (*in module pysteps.utils.dimension*), 98
aggregate_fields_space() (*in module pysteps.utils.dimension*), 97
aggregate_fields_time() (*in module pysteps.utils.dimension*), 97
animate() (*in module pysteps.visualization.animations*), 124
ar_acf() (*in module pysteps.timeseries.autoregression*), 91

## B

binary_mse() (*in module pysteps.verification.spatialscores*), 123
boxcox_transform() (*in module pysteps.utils.transformation*), 101
boxcox_transform_test_lambdas() (*in module pysteps.utils.transformation*), 101

## C

ceil_int() (*in module pysteps.motion.vet*), 67
clip_domain() (*in module pysteps.utils.dimension*), 98
close_forecast_file() (*in module pysteps.io.exporters*), 60
compute_centred_coord_array() (*in module pysteps.utils.arrays*), 94
compute_empirical_cdf() (*in module pysteps.postprocessing.probmatching*), 88
compute_noise_stddev_adjs() (*in module pysteps.noise.utils*), 75
CRPS() (*in module pysteps.verification.probscores*), 116
CRPS_accum() (*in module pysteps.verification.probscores*), 117
CRPS_compute() (*in module pysteps.verification.probscores*), 117
CRPS_init() (*in module pysteps.verification.probscores*), 117

## D

DARTS() (*in module pysteps.motion.darts*), 61
dB_transform() (*in module pysteps.utils.transformation*), 102
decomposition_fft() (*in module pysteps.cascade.decomposition*), 47
dense_lucaskanade() (*in module pysteps.motion.lucaskanade*), 62
det_cat_fct() (*in module pysteps.verification.detcatscores*), 105
det_cat_fct_accum() (*in module pysteps.verification.detcatscores*), 106
det_cat_fct_compute() (*in module pysteps.verification.detcatscores*), 106
det_cat_fct_init() (*in module pysteps.verification.detcatscores*), 106
det_cont_fct() (*in module pysteps.verification.detcontscores*), 107
det_cont_fct_accum() (*in module pysteps.verification.detcontscores*), 109
det_cont_fct_compute() (*in module pysteps.verification.detcontscores*), 109
det_cont_fct_init() (*in module pysteps.verification.detcontscores*), 109

## E

ensemble_skill() (*in module pysteps.verification.ensscores*), 110
ensemble_spread() (*in module pysteps.verification.ensscores*), 111
estimate_ar_params_yw() (*in module pysteps.timeseries.autoregression*), 92
eulerian_persistence() (*in module pysteps.extrapolation.interface*), 48
excprob() (*in module pysteps.postprocessing.ensemblestats*), 88
export_forecast_dataset() (*in module pysteps.io.exporters*), 59
extrapolate() (*in module pysteps.extrapolation.semilagrangian*), 49

## F

filter_gaussian() (*in module pysteps.cascade.bandpass_filters*), 46

filter_uniform() (*in module pysteps.cascade.bandpass_filters*), 46
find_by_date() (*in module pysteps.io.archive*), 51
forecast() (*in module pysteps.nowcasts.extrapolation*), 77
forecast() (*in module pysteps.nowcasts.sprog*), 78
forecast() (*in module pysteps.nowcasts.sseps*), 80
forecast() (*in module pysteps.nowcasts.steps*), 83
fss() (*in module pysteps.verification.spatialscores*), 123

## G

generate_bps() (*in module pysteps.noise.motion*), 75
generate_noise_2d_fft_filter() (*in module pysteps.noise.fftgenerators*), 72
generate_noise_2d_ssft_filter() (*in module pysteps.noise.fftgenerators*), 73
get_colormap() (*in module pysteps.visualization.precipfields*), 130
get_default_params_bps_par() (*in module pysteps.noise.motion*), 74
get_default_params_bps_perp() (*in module pysteps.noise.motion*), 74
get_method() (*in module pysteps.cascade.interface*), 45
get_method() (*in module pysteps.extrapolation.interface*), 48
get_method() (*in module pysteps.io.interface*), 50
get_method() (*in module pysteps.noise.interface*), 68
get_method() (*in module pysteps.nowcasts.interface*), 77
get_method() (*in module pysteps.utils.interface*), 93
get_method() (*in module pysteps.verification.interface*), 103
get_numpy() (*in module pysteps.utils.fft*), 99
get_padding() (*in module pysteps.motion.vet*), 68
get_pyfftw() (*in module pysteps.utils.fft*), 100
get_scipy() (*in module pysteps.utils.fft*), 99

## I

import_bom_rf3() (*in module pysteps.io.importers*), 53
import_fmi_pgm() (*in module pysteps.io.importers*), 53
import_knmi_hdf5() (*in module pysteps.io.importers*), 56
import_mch_gif() (*in module pysteps.io.importers*), 54
import_mch_hdf5() (*in module pysteps.io.importers*), 54
import_mch_metranet() (*in module pysteps.io.importers*), 55
import_netcdf_pysteps() (*in module pysteps.io.nowcast_importers*), 57
import_odim_hdf5() (*in module pysteps.io.importers*), 55
initialize_bps() (*in module pysteps.noise.motion*), 74
initialize_forecast_exporter_kineros() (*in module pysteps.io.exporters*), 58
initialize_forecast_exporter_netcdf() (*in module pysteps.io.exporters*), 58
initialize_nonparam_2d_fft_filter() (*in module pysteps.noise.fftgenerators*), 70
initialize_nonparam_2d_nested_filter() (*in module pysteps.noise.fftgenerators*), 71
initialize_nonparam_2d_ssft_filter() (*in module pysteps.noise.fftgenerators*), 71
initialize_param_2d_fft_filter() (*in module pysteps.noise.fftgenerators*), 69
intensity_scale() (*in module pysteps.verification.spatialscores*), 121
intensity_scale_accum() (*in module pysteps.verification.spatialscores*), 122
intensity_scale_compute() (*in module pysteps.verification.spatialscores*), 122
intensity_scale_init() (*in module pysteps.verification.spatialscores*), 122
iterate_ar_model() (*in module pysteps.timeseries.autoregression*), 92

## L

lifetime() (*in module pysteps.verification.lifetime*), 113
lifetime_accum() (*in module pysteps.verification.lifetime*), 114
lifetime_compute() (*in module pysteps.verification.lifetime*), 114
lifetime_init() (*in module pysteps.verification.lifetime*), 114

## M

mean() (*in module pysteps.postprocessing.ensemblestats*), 87
morph() (*in module pysteps.motion.vet*), 67

## N

nonparam_match_empirical_cdf() (*in module pysteps.postprocessing.probmatching*), 89
NQ_transform() (*in module pysteps.utils.transformation*), 102

## P

parse_proj4_string() (*in module pysteps.visualization.utils*), 131
plot_intensityscale() (*in module pysteps.verification.plots*), 115
plot_precip_field() (*in module pysteps.visualization.precipfields*), 128
plot_rankhist() (*in module pysteps.verification.plots*), 115

plot_reldiag() (*in module pysteps.verification.plots*), 115
plot_ROC() (*in module pysteps.verification.plots*), 116
plot_spectrum1d() (*in module pysteps.visualization.spectral*), 130
pmm_compute() (*in module pysteps.postprocessing.probmatching*), 89
pmm_init() (*in module pysteps.postprocessing.probmatching*), 89
print_ar_params() (*in module pysteps.nowcasts.utils*), 86
print_corrcoefs() (*in module pysteps.nowcasts.utils*), 86
proj4_to_basemap() (*in module pysteps.visualization.utils*), 132
proj4_to_cartopy() (*in module pysteps.visualization.utils*), 132
pysteps.cascade.bandpass_filters (*module*), 45
pysteps.cascade.decomposition (*module*), 47
pysteps.cascade.interface (*module*), 45
pysteps.extrapolation.interface (*module*), 48
pysteps.extrapolation.semilagrangian (*module*), 49
pysteps.io.archive (*module*), 51
pysteps.io.exporters (*module*), 57
pysteps.io.importers (*module*), 52
pysteps.io.interface (*module*), 50
pysteps.io.nowcast_importers (*module*), 56
pysteps.io.readers (*module*), 60
pysteps.motion.darts (*module*), 60
pysteps.motion.interface (*module*), 60
pysteps.motion.lucaskanade (*module*), 62
pysteps.motion.vet (*module*), 64
pysteps.noise.fftgenerators (*module*), 68
pysteps.noise.interface (*module*), 68
pysteps.noise.motion (*module*), 73
pysteps.noise.utils (*module*), 75
pysteps.nowcasts.extrapolation (*module*), 77
pysteps.nowcasts.interface (*module*), 76
pysteps.nowcasts.sprog (*module*), 78
pysteps.nowcasts.sseps (*module*), 79
pysteps.nowcasts.steps (*module*), 82
pysteps.nowcasts.utils (*module*), 86
pysteps.postprocessing.ensemblestats (*module*), 87
pysteps.postprocessing.probmatching (*module*), 88
pysteps.timeseries.autoregression (*module*), 90
pysteps.timeseries.correlation (*module*), 92
pysteps.utils.arrays (*module*), 94
pysteps.utils.conversion (*module*), 95
pysteps.utils.dimension (*module*), 96
pysteps.utils.fft (*module*), 99
pysteps.utils.interface (*module*), 93
pysteps.utils.spectral (*module*), 100
pysteps.utils.transformation (*module*), 101
pysteps.verification.detcatscores (*module*), 105
pysteps.verification.detcontscores (*module*), 107
pysteps.verification.ensscores (*module*), 110
pysteps.verification.interface (*module*), 103
pysteps.verification.lifetime (*module*), 113
pysteps.verification.plots (*module*), 114
pysteps.verification.probscores (*module*), 116
pysteps.verification.spatialscores (*module*), 120
pysteps.visualization.animations (*module*), 124
pysteps.visualization.motionfields (*module*), 125
pysteps.visualization.precipfields (*module*), 128
pysteps.visualization.spectral (*module*), 130
pysteps.visualization.utils (*module*), 131

## Q

quiver() (*in module pysteps.visualization.motionfields*), 126

## R

rankhist() (*in module pysteps.verification.ensscores*), 112
rankhist_accum() (*in module pysteps.verification.ensscores*), 112
rankhist_compute() (*in module pysteps.verification.ensscores*), 112
rankhist_init() (*in module pysteps.verification.ensscores*), 112
rapsd() (*in module pysteps.utils.spectral*), 100
read_timeseries() (*in module pysteps.io.readers*), 60
recompose_cascade() (*in module pysteps.nowcasts.utils*), 87
reldiag() (*in module pysteps.verification.probscores*), 118
reldiag_accum() (*in module pysteps.verification.probscores*), 119
reldiag_compute() (*in module pysteps.verification.probscores*), 119
reldiag_init() (*in module pysteps.verification.probscores*), 118

remove_rain_norain_discontinuity() (*in module pysteps.utils.spectral*), 100
reproject_geodata() (*in module pysteps.visualization.utils*), 132
ROC_curve() (*in module pysteps.verification.probscores*), 119
ROC_curve_accum() (*in module pysteps.verification.probscores*), 120
ROC_curve_compute() (*in module pysteps.verification.probscores*), 120
ROC_curve_init() (*in module pysteps.verification.probscores*), 120
round_int() (*in module pysteps.motion.vet*), 67

## S

shift_scale() (*in module pysteps.postprocessing.probmatching*), 89
sqrt_transform() (*in module pysteps.utils.transformation*), 103
square_domain() (*in module pysteps.utils.dimension*), 99
stack_cascades() (*in module pysteps.nowcasts.utils*), 86
streamplot() (*in module pysteps.visualization.motionfields*), 127

## T

temporal_autocorrelation() (*in module pysteps.timeseries.correlation*), 93
to_raindepth() (*in module pysteps.utils.conversion*), 95
to_rainrate() (*in module pysteps.utils.conversion*), 95
to_reflectivity() (*in module pysteps.utils.conversion*), 96

## V

vet() (*in module pysteps.motion.vet*), 64
vet_cost_function() (*in module pysteps.motion.vet*), 66
vet_cost_function_gradient() (*in module pysteps.motion.vet*), 67