
tobac

Jul 12, 2023

1	Installation	3
2	Data Input	5
2.1	3D Data	5
2.2	Data Output	5
3	Analysis	7
4	Plotting	9
5	Handling Large Datasets	11
5.1	Split Feature Detection	11
6	Example notebooks	13
7	Refereed Publications	15
8	Feature Detection Basics	17
9	Threshold Feature Detection Parameters	21
9.1	Basic Operating Procedure	21
9.2	Target	21
9.3	Thresholds	21
9.4	Minimum Threshold Number	22
9.5	Feature Position	22
9.6	Filtering Options	22
9.7	Minimum Distance	23
10	Feature Detection Parameter Examples	25
10.1	How multiple thresholds changes the features detected	25
10.2	How <code>n_min_threshold</code> changes what features are detected	31
10.3	Different <code>threshold_position</code> options	41
10.4	<i>tobac</i> Feature Detection Filtering	47
11	Feature Detection Output	53
12	Segmentation	57

13 Watershedding Segmentation Parameters	59
13.1 Basic Operating Procedure	59
13.2 Target	59
13.3 Threshold	60
13.4 Where the 3D seeds are placed for 2D feature detection	60
13.5 Maximum Distance	60
14 Segmentation Output	61
15 Features without segmented areas	63
15.1 Case 1: Segmentation threshold	63
15.2 Case 2: Feature position	64
16 Track on one dataset, segment on another	67
17 Linking	69
18 Tracking Output	73
19 Merge and Split	75
20 tobac package	77
20.1 Submodules	77
20.2 tobac.analysis module	77
20.3 tobac.centerofgravity module	85
20.4 tobac.feature_detection module	86
20.5 tobac.merge_split module	93
20.6 tobac.plotting module	93
20.7 tobac.segmentation module	101
20.8 tobac.testing module	105
20.9 tobac.tracking module	110
20.10 tobac.utils modules	113
20.11 tobac.utils.general modules	113
20.12 tobac.utils.mask modules	116
20.13 tobac.wrapper module	119
20.14 Module contents	119
Python Module Index	121
Index	123

tobac is a Python package to rapidly identify, track and analyze clouds in different types of gridded datasets, such as 3D model output from cloud-resolving model simulations or 2D data from satellite retrievals.

The software is set up in a modular way to include different algorithms for feature identification, tracking, and analyses. **tobac** is also input variable agnostic and doesn't rely on specific input variables, nor a specific grid to work.

Individual features are identified as either maxima or minima in a 2D or 3D time-varying field (see *Feature Detection Basics*). An associated volume can then be determined using these features with a separate (or identical) time-varying 2D or 3D field and a threshold value (see *Segmentation*). The identified objects are linked into consistent trajectories representing the cloud over its lifecycle in the tracking step. Analysis and visualization methods provide a convenient way to use and display the tracking results.

Version 1.2 of **tobac** and some example applications are described in a peer-reviewed article in *Geoscientific Model Development* as:

Heikenfeld, M., Marinescu, P. J., Christensen, M., Watson-Parris, D., Senf, F., van den Heever, S. C., and Stier, P.: **tobac 1.2: towards a flexible framework for tracking and analysis of clouds in diverse datasets**, *Geosci. Model Dev.*, 12, 4551–4570, <https://doi.org/10.5194/gmd-12-4551-2019>, 2019.

The project is currently being extended by several contributors to include additional workflows and algorithms using the same structure, syntax, and data formats.

CHAPTER 1

Installation

tobac works with Python 3 installations.

The easiest way is to install the most recent version of tobac via conda or mamba and the conda-forge channel:

```
conda install -c conda-forge tobac or mamba install -c conda-forge tobac
```

This will take care of all necessary dependencies and should do the job for most users. It also allows for an easy update of the installation by

```
conda update -c conda-forge tobac mamba update -c conda-forge tobac
```

You can also install conda via pip, which is mainly interesting for development purposes or using specific development branches for the Github repository.

The following python packages are required (including dependencies of these packages):

numpy, scipy, scikit-image, pandas, pytables, matplotlib, iris, xarray, cartopy, trackpy

If you are using anaconda, the following command should make sure all dependencies are met and up to date:

```
conda install -c conda-forge -y numpy scipy scikit-image pandas pytables matplotlib_
↪iris xarray cartopy trackpy
```

You can directly install the package directly from github with pip and either of the two following commands:

```
pip install --upgrade git+ssh://git@github.com/tobac-project/tobac.git
```

```
pip install --upgrade git+https://github.com/tobac-project/tobac.git
```

You can also clone the package with any of the two following commands:

```
git clone git@github.com:tobac-project/tobac.git
```

```
git clone https://github.com/tobac-project/tobac.git
```

and install the package from the locally cloned version (The trailing slash is actually necessary):

```
pip install --upgrade tobac/
```


Input data for *tobac* should consist of one or more fields on a common, regular grid with a time dimension and two or three spatial dimensions. The input data can also include latitude and longitude coordinates, either as 1-d or 2-d variables depending on the grid used.

Interoperability with *xarray* is provided by the convenient functions allowing for a transformation between the two data types. *xarray* DataArrays can be easily converted into *iris* cubes using *xarray*'s `to_iris()` method, while the *Iris* cubes produced as output of *tobac* can be turned into *xarray* DataArrays using the `from_iris()` method.

For the future development of the next major version of *tobac* (v2.0), we are moving the basic data structures from *Iris* cubes to *xarray* DataArrays for improved computing performance and interoperability with other open-source software packages, including the Pangeo project (<https://pangeo.io/>).

2.1 3D Data

As of *tobac* version 1.5.0, 3D data are now fully supported for feature detection, tracking, and segmentation. Similar to how *tobac* requires some information on the horizontal grid spacing of the data (e.g., through the `dxy` parameter), some information on the vertical grid spacing is also required. This is documented in detail in the API docs, but briefly, users must specify either `dz`, where the grid has uniform grid spacing, or users must specify `vertical_coord`, where `vertical_coord` is the name of the coordinate representing the vertical, with the same units as `dxy`.

2.2 Data Output

The output of the different analysis steps in *tobac* are output as either *pandas* DataFrames in the case of one-dimensional data, such as lists of identified features or feature tracks or as *Iris* cubes in the case of 2D/3D/4D fields such as feature masks. Note that the dataframe output from tracking is a superset of the features dataframe.

For information on feature detection *output*, see [Feature Detection Output](#). For information on tracking *output*, see [Tracking Output](#).

Note that in future versions of *tobac*, it is planned to combine both output data types into a single hierarchical data structure containing both spatial and object information. Additional information about the planned changes can be

found in the v2.0-dev branch of the main tobac repository (<https://github.com/tobac-project/tobac>), as well as the tobac roadmap (<https://github.com/tobac-project/tobac-roadmap>).

CHAPTER 3

Analysis

tobac provides several analysis functions that allow for the calculation of important quantities based on the tracking results. This includes the calculation of properties such as feature lifetimes and feature areas/volumes, but also allows for a convenient calculation of statistics for arbitrary fields of the same shape as as the input data used for the tracking analysis.

CHAPTER 4

Plotting

tobac provides functions to conveniently visualise the tracking results and analyses.

Handling Large Datasets

Often, one desires to use *tobac* to identify and track features in large datasets (“big data”). This documentation strives to suggest various methods for doing so efficiently. Current versions of *tobac* do not allow for out-of-memory computation, meaning that these strategies may need to be employed for both computational and memory reasons.

5.1 Split Feature Detection

Current versions of threshold feature detection (see *Feature Detection Basics*) are time independent, meaning that one can parallelize feature detection across all times (although not across space). *tobac* provides the `tobac.utils.combine_tobac_feats()` function to combine a list of dataframes produced by a parallelization method (such as `jug` or `multiprocessing.pool`) into a single combined dataframe suitable to perform tracking with.

Example notebooks

tobac is provided with a set of Jupyter notebooks that show examples of the application of tobac for different types of datasets.

The notebooks can be found in the **examples** folder in the repository. The necessary input data for these examples is available on zenodo: [www.zenodo.org/...](http://www.zenodo.org/) and can be downloaded automatically by the Jupyter notebooks.

The examples currently include four different applications of tobac: 1. Tracking of scattered convection based on vertical velocity and condensate mixing ratio for 3D cloud-resolving model output. 2. Tracking of scattered convection based on surface precipitation from the same cloud-resolving model output. 3. Tracking of convective clouds based on outgoing longwave radiation (OLR) for convection-permitting model simulation output. 4. Tracking of convective clouds based on OLR in geostationary satellite retrievals.

The examples are based on the analyses presented in an article describing tobac that has been submitted to the journal GMD (Geophysical model development).

Refereed Publications

List of peer-reviewed publications in which tobac has been used:

Bukowski, J., & van den Heever, S. C. (2021). Direct radiative effects in haboobs. <i>Journal of Geophysical Research: Atmospheres</i> , 126(21), e2021JD034814, doi:10.1029/2021JD034814.
Bukowski, J. (2021). Mineral Dust Lofting and Interactions with Cold Pools (Doctoral dissertation, Colorado State University).
Heikenfeld, M. (2019). Aerosol effects on microphysical processes and deep convective clouds (Doctoral dissertation, University of Oxford).
Kukulies, J., Chen, D., & Curio, J. (2021). The role of mesoscale convective systems in precipitation in the Tibetan Plateau region. <i>Journal of Geophysical Research: Atmospheres</i> , 126(23), e2021JD035279. doi:10.1029/2021JD035279.
Li, Y., Liu, Y., Chen, Y., Chen, B., Zhang, X., Wang, W. & Huo, Z. (2021). Characteristics of Deep Convective Systems and Initiation during Warm Seasons over China and Its Vicinity. <i>Remote Sensing</i> , 13(21), 4289. doi:10.3390/rs13214289.
Marinescu, P. J., Van Den Heever, S. C., Heikenfeld, M., Barrett, A. I., Barthlott, C., Hoose, C., ... & Zhang, Y. (2021). Impacts of varying concentrations of cloud condensation nuclei on deep convective cloud updrafts—a multimodel assessment. <i>Journal of the Atmospheric Sciences</i> , 78(4), 1147-1172, doi: 10.1175/JAS-D-20-0200.1.
Marinescu, P. J. (2020). Observations of Aerosol Particles and Deep Convective Updrafts and the Modeling of Their Interactions (Doctoral dissertation, Colorado State University).
Raut, B. A., Jackson, R., Picel, M., Collis, S. M., Bergemann, M., & Jakob, C. (2021). An Adaptive Tracking Algorithm for Convection in Simulated and Remote Sensing Data. <i>Journal of Applied Meteorology and Climatology</i> , 60(4), 513-526, doi:10.1175/JAMC-D-20-0119.1.
Whitaker, J. W. (2021). An Investigation of an East Pacific Easterly Wave Genesis Pathway and the Impact of the Papagayo and Tehuantepec Wind Jets on the East Pacific Mean State and Easterly Waves (Doctoral dissertation, Colorado State University).
Zhang, X., Yin, Y., Kukulies, J., Li, Y., Kuang, X., He, C., .. & Chen, J. (2021). Revisiting Lightning Activity and Parameterization Using Geostationary Satellite Observations. <i>Remote Sensing</i> , 13(19), 3866, doi: 10.3390/rs13193866.

Have you used tobac in your research?

Please contact us (e.g. by joining our [tobac google group](#)) or submit a pull request containing your reference in our [main repo on GitHub](#)!

Feature Detection Basics

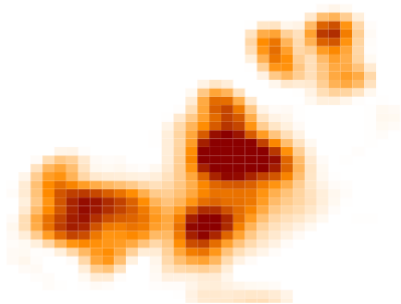
The feature detection is the first step in using **tobac**.

Currently implemented methods:

Multiple thresholds:

Features are identified as regions above or below a sequence of subsequent thresholds (if searching for either maxima or minima in the data). Subsequently more restrictive threshold values are used to further refine the resulting features and allow for separation of features that are connected through a continuous region of less restrictive threshold values.

a) Input data



- + new features
- + old features
- × deleted features
- + final features

b) Threshold 1



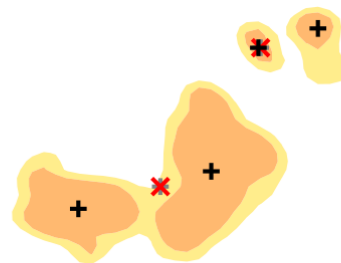
c) Intermediate features 1



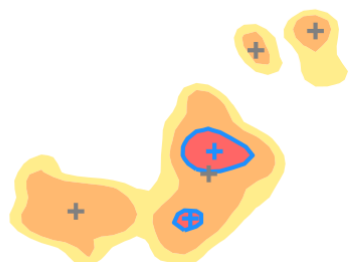
d) Threshold 2



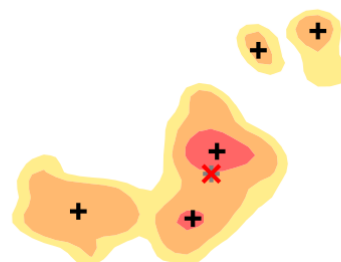
e) Intermediate features 2



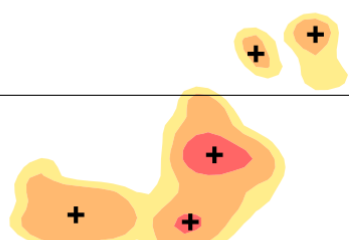
f) Threshold 3



g) Intermediate features 3



h) Final set of features



Current development: We are currently working on additional methods for the identification of cloud features in different types of datasets. Some of these methods are specific to the input data such a combination of different channels from specific satellite imagers. Some of these methods will combine the feature detection and segmentations step in one single algorithm.

Threshold Feature Detection Parameters

The proper selection of parameters used to detect features with the *tobac* multiple threshold feature detection is a critical first step in using *tobac*. This page describes the various parameters available and provides broad comments on the usage of each parameter.

A full list of parameters and descriptions can be found in the API Reference: `tobac.feature_detection.feature_detection_multithreshold()`

9.1 Basic Operating Procedure

The *tobac* multiple threshold algorithm searches the input data (*field_in*) for contiguous regions of data greater than (with *target*='maximum', see *Target*) or less than (with *target*='minimum') the selected thresholds (see *Thresholds*). Contiguous regions (see *Minimum Threshold Number*) are then identified as individual features, with a single point representing their location in the output (see *Position Threshold*). Using this output (see *Feature Detection Output*), segmentation (*Segmentation*) and tracking (*Linking*) can be run.

9.2 Target

First, you must determine whether you want to detect features on maxima or minima in your dataset. For example, if you are trying to detect clouds in IR satellite data, where clouds have relatively lower brightness temperatures than the background, you would set `target='minimum'`. If, instead, you are trying to detect clouds by cloud water in model data, where an increase in mixing ratio indicates the presence of a cloud, you would set `target='maximum'`. The `target` parameter will determine the selection of many of the following parameters.

9.3 Thresholds

You can select to detect features on either one or multiple thresholds. The first threshold (or the single threshold) sets the minimum magnitude (either lowest value for `target='maximum'` or highest value for `target='minimum'`)

that a feature can be detected on. For example, if you have a field made up of values lower than 10, and you set `target='maximum', threshold=[10,]`, *tobac* will detect no features.

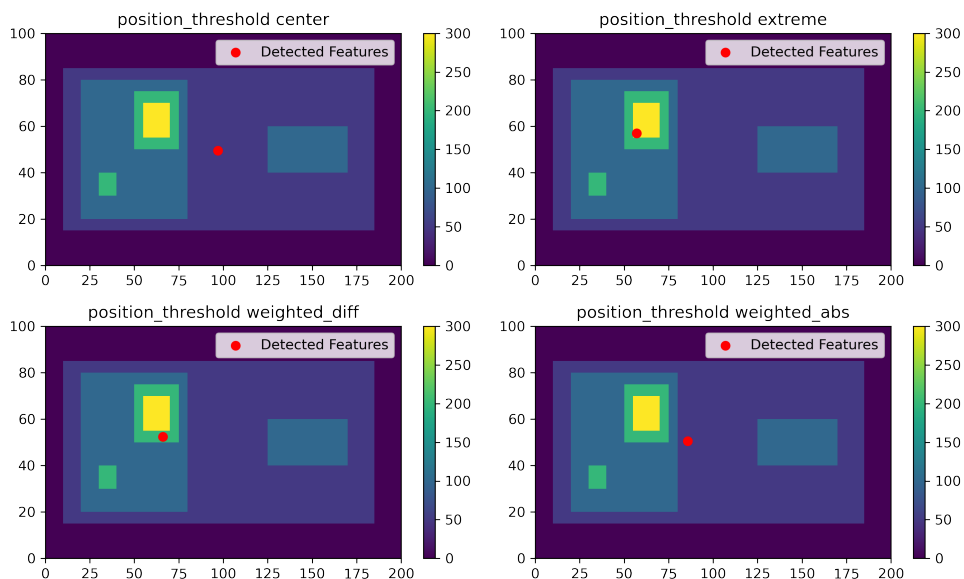
Including *multiple* thresholds will allow *tobac* to refine the detection of features and detect multiple features that are connected through a contiguous region of less restrictive threshold values. You can see a conceptual diagram of that here: [Feature Detection Basics](#). To examine how setting different thresholds can change the number of features detected, see the example in this notebook: [How multiple thresholds changes the features detected](#).

9.4 Minimum Threshold Number

The minimum number of points per threshold, set by `n_min_threshold`, determines how many contiguous pixels are required to be above the threshold for the feature to be detected. Setting this point very low can allow extraneous points to be detected as erroneous features, while setting this value too high will cause some real features to be missed. The default value for this parameter is 0, which will cause any values greater than the threshold after filtering to be identified as a feature. You can see a demonstration of the affect of increasing `n_min_threshold` at: [How n_min_threshold changes what features are detected](#).

9.5 Feature Position

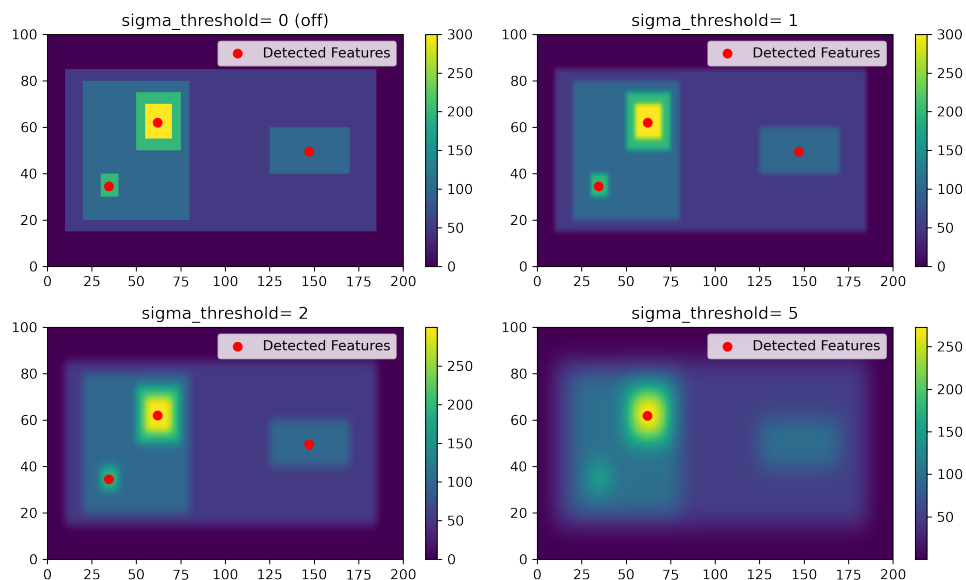
There are four ways of calculating the single point used to represent feature center: arithmetic center, extreme point, difference weighting, and absolute weighting. Generally, difference weighting (`position_threshold='weighted_diff'`) or absolute weighting (`position_threshold='weighted_abs'`) is suggested for most atmospheric applications. An example of these four methods is shown below, and can be further explored in the example notebook: [Different threshold_position options](#).



9.6 Filtering Options

Before *tobac* detects features, two filtering options can optionally be employed. First is a multidimensional Gaussian Filter (`scipy.ndimage.gaussian_filter`), with its standard deviation controlled by the `sigma_threshold` pa-

parameter. It is not required that users use this filter (to turn it off, set `sigma_threshold=0`), but the use of the filter is recommended for most atmospheric datasets that are not otherwise smoothed. An example of varying the `sigma_threshold` parameter can be seen in the below figure, and can be explored in the example notebook: [tobac Feature Detection Filtering](#).



The second filtering option is a binary erosion (`skimage.morphology.binary_erosion`), which reduces the size of features in all directions. The amount of the erosion is controlled by the `n_erosion_threshold` parameter, with larger values resulting in smaller potential features. It is not required to use this feature (to turn it off, set `n_erosion_threshold=0`), and its use should be considered alongside careful selection of `n_min_threshold`. The default value is `n_erosion_threshold=0`.

9.7 Minimum Distance

The parameter `min_distance` sets the minimum distance between two detected features. If two detected features are within `min_distance` of each other, the feature with the larger value is kept, and the feature with the smaller value is discarded.

Feature Detection Parameter Examples

10.1 How multiple thresholds changes the features detected

10.1.1 Imports

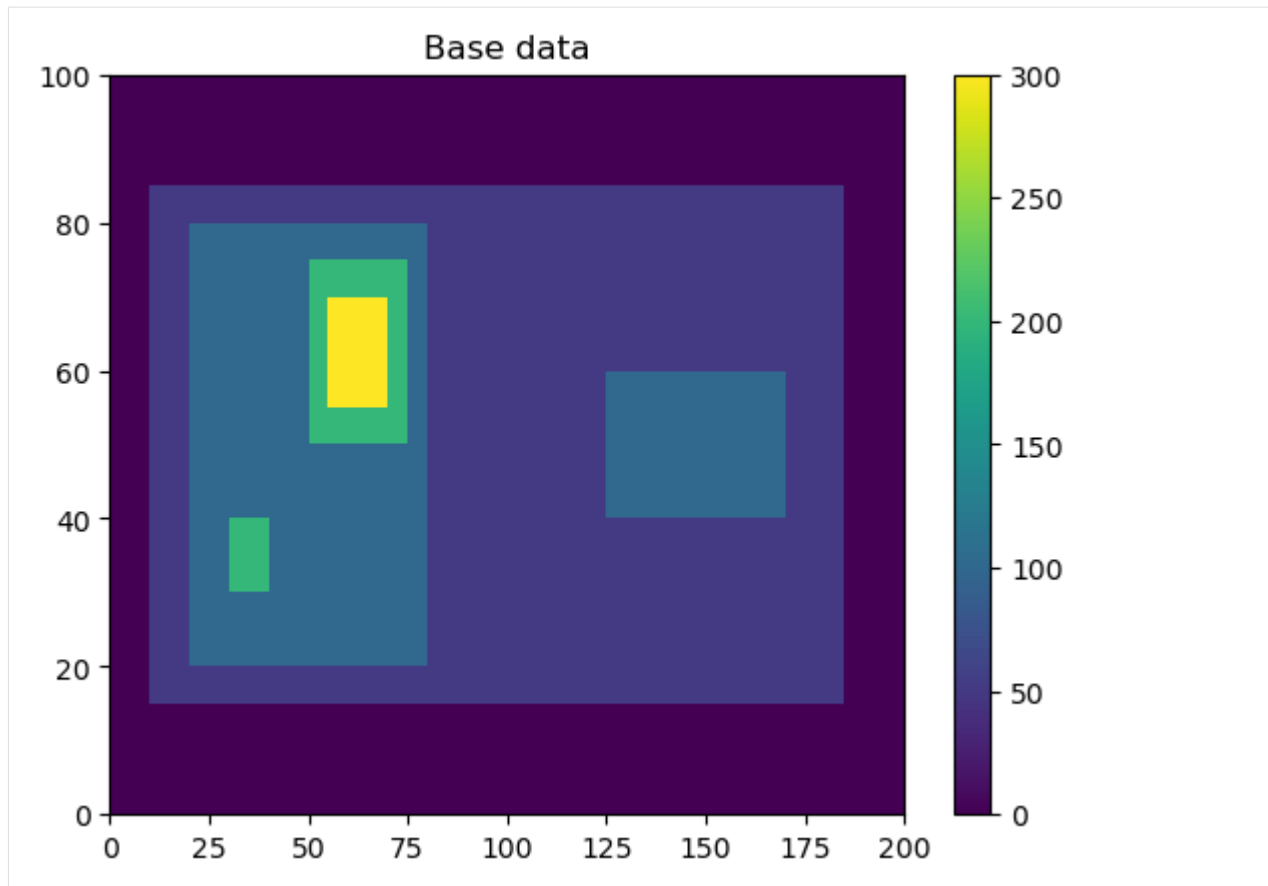
```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tobac
import xarray as xr
```

10.1.2 Generate Feature Data

Here, we will generate some simple feature data where the features that we want to detect are *higher* values than the surrounding (0).

```
[2]: # Dimensions here are time, y, x.
input_field_arr = np.zeros((1,100,200))
input_field_arr[0, 15:85, 10:185]=50
input_field_arr[0, 20:80, 20:80]=100
input_field_arr[0, 40:60, 125:170] = 100
input_field_arr[0, 30:40, 30:40]=200
input_field_arr[0, 50:75, 50:75]=200
input_field_arr[0, 55:70, 55:70]=300

plt.pcolormesh(input_field_arr[0])
plt.colorbar()
plt.title("Base data")
plt.show()
```



We now need to generate an Iris DataCube out of this dataset to run tobac feature detection. One can use xarray to generate a DataArray and then convert it to Iris, as done here. Version 2.0 of tobac (currently in development) will allow the use of xarray directly with tobac.

```
[3]: input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={
    ↳ 'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()
```

10.1.3 Single Threshold

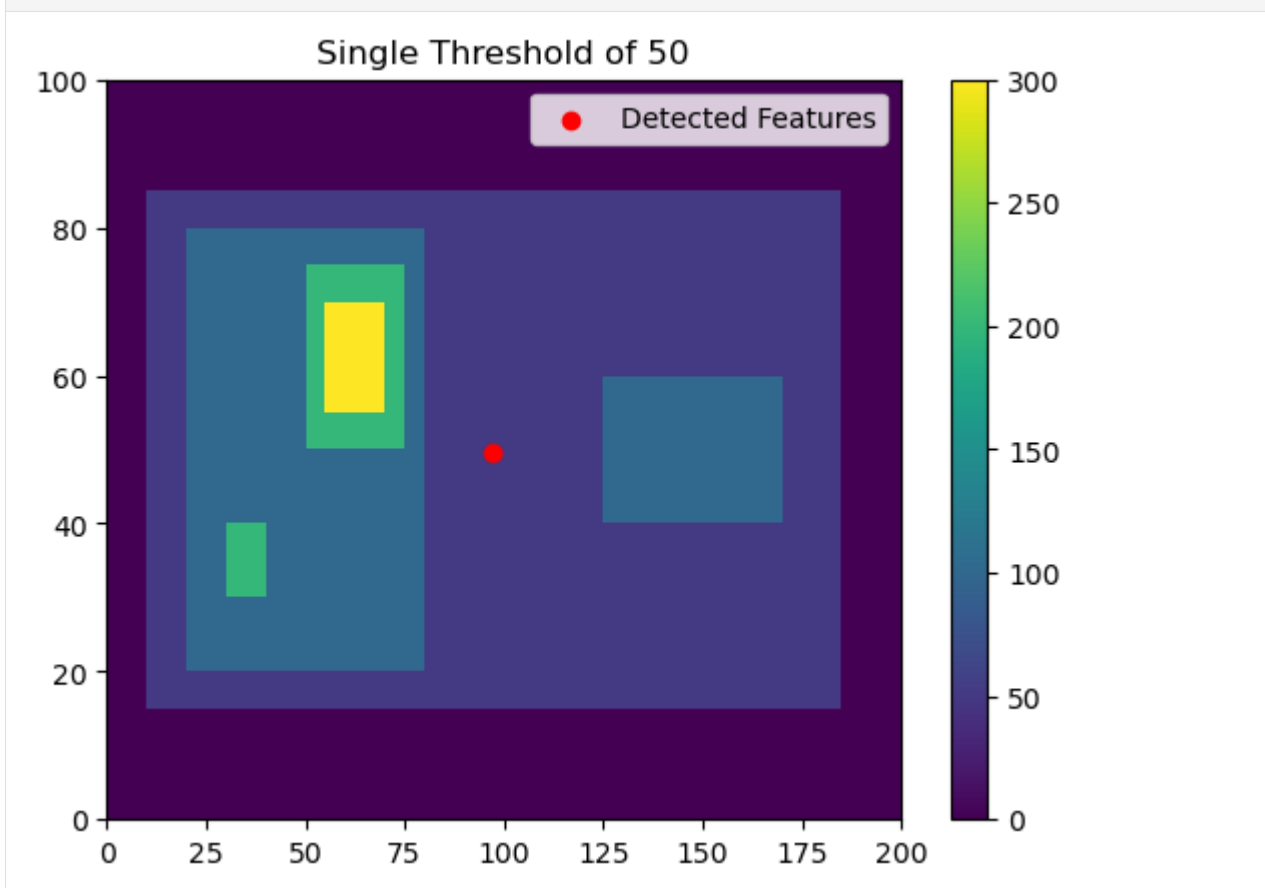
Let's say that you are looking to detect any features above value 50 and don't need to separate out individual cells within the larger feature. For example, if you're interested in tracking a single mesoscale convective system, you may not care about the paths of individual convective cells within the feature.

```
[4]: thresholds = [50,]
    # Using 'center' here outputs the feature location as the arithmetic center of the_
    ↳ detected feature
    single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
    ↳ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_threshold=
    ↳ 'center')
    plt.pcolormesh(input_field_arr[0])
    plt.colorbar()
    # Plot all features detected
    plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
    ↳ 'hdim_1'].values, color='r', label="Detected Features")
```

(continues on next page)

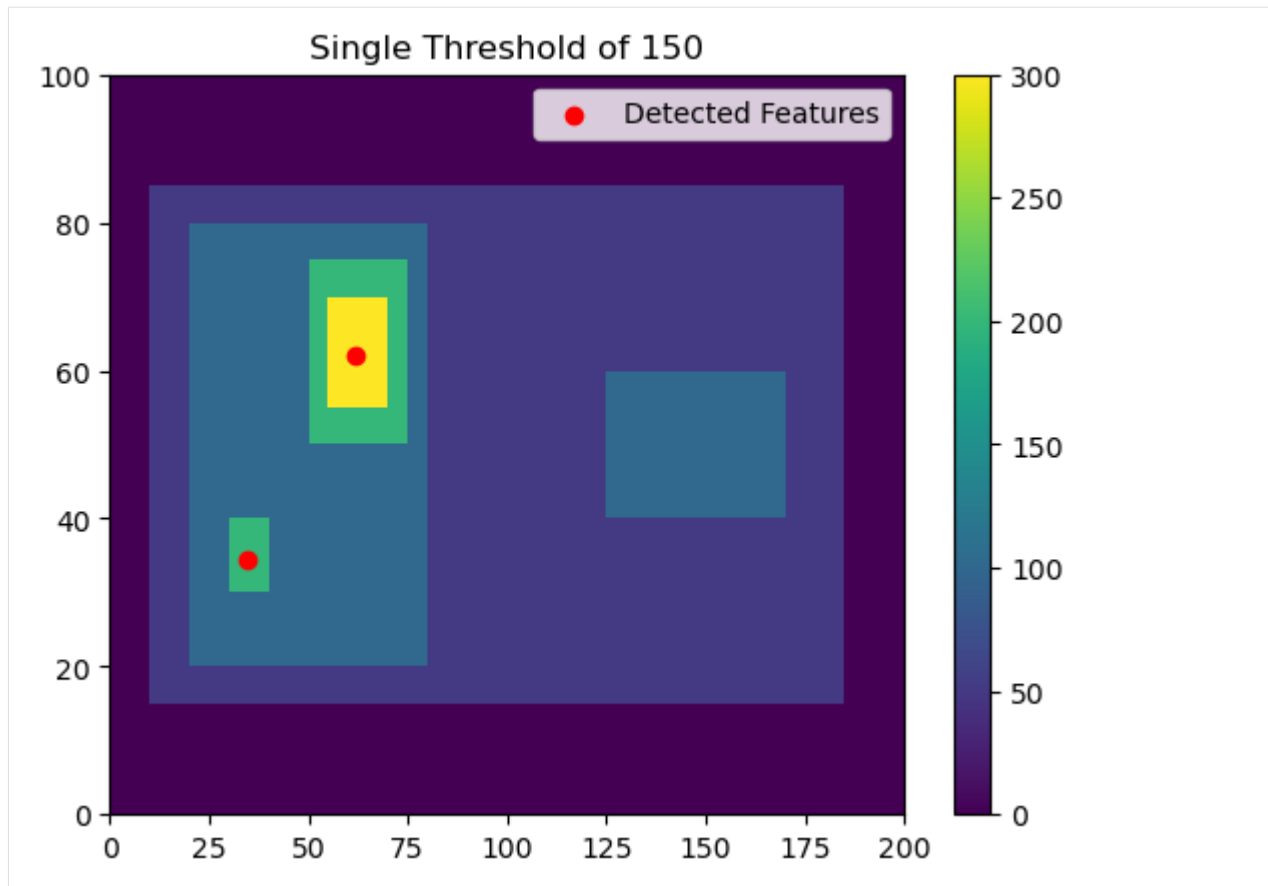
(continued from previous page)

```
plt.legend()
plt.title("Single Threshold of 50")
plt.show()
```



Now, let's try a single threshold of 150, which will give us two features on the left side of the image.

```
[5]: thresholds = [150,]
# Using 'center' here outputs the feature location as the arithmetic center of the_
    ↳ detected feature
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
    ↳ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_threshold=
    ↳ 'center')
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
    ↳ 'hdim_1'].values, color='r', label="Detected Features")
plt.legend()
plt.title("Single Threshold of 150")
plt.show()
```



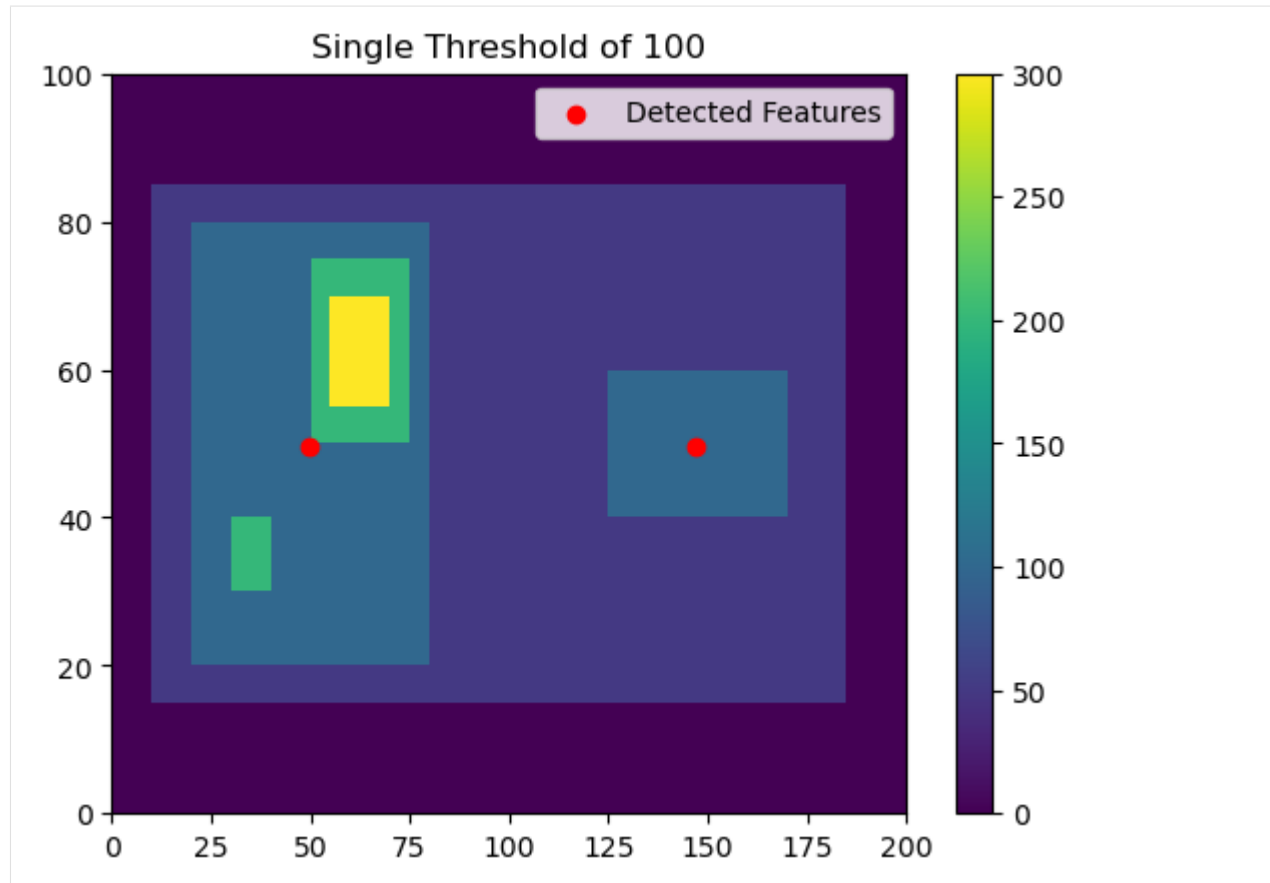
This gives us two detected features with minimum values >150.

10.1.4 Multiple Thresholds

Now let's say that you want to detect all three maxima within this feature. You may want to do this, if, for example, you were trying to detect overshooting tops within a cirrus shield. You could pick a single threshold, but if you pick 100, you won't separate out the two features on the left. For example:

```
[6]: thresholds = [100, ]
# Using 'center' here outputs the feature location as the arithmetic center of the_
# detected feature
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
# field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_threshold=
# 'center')
plt.pcolormesh(input_field_arr[0])
plt.colorbar()

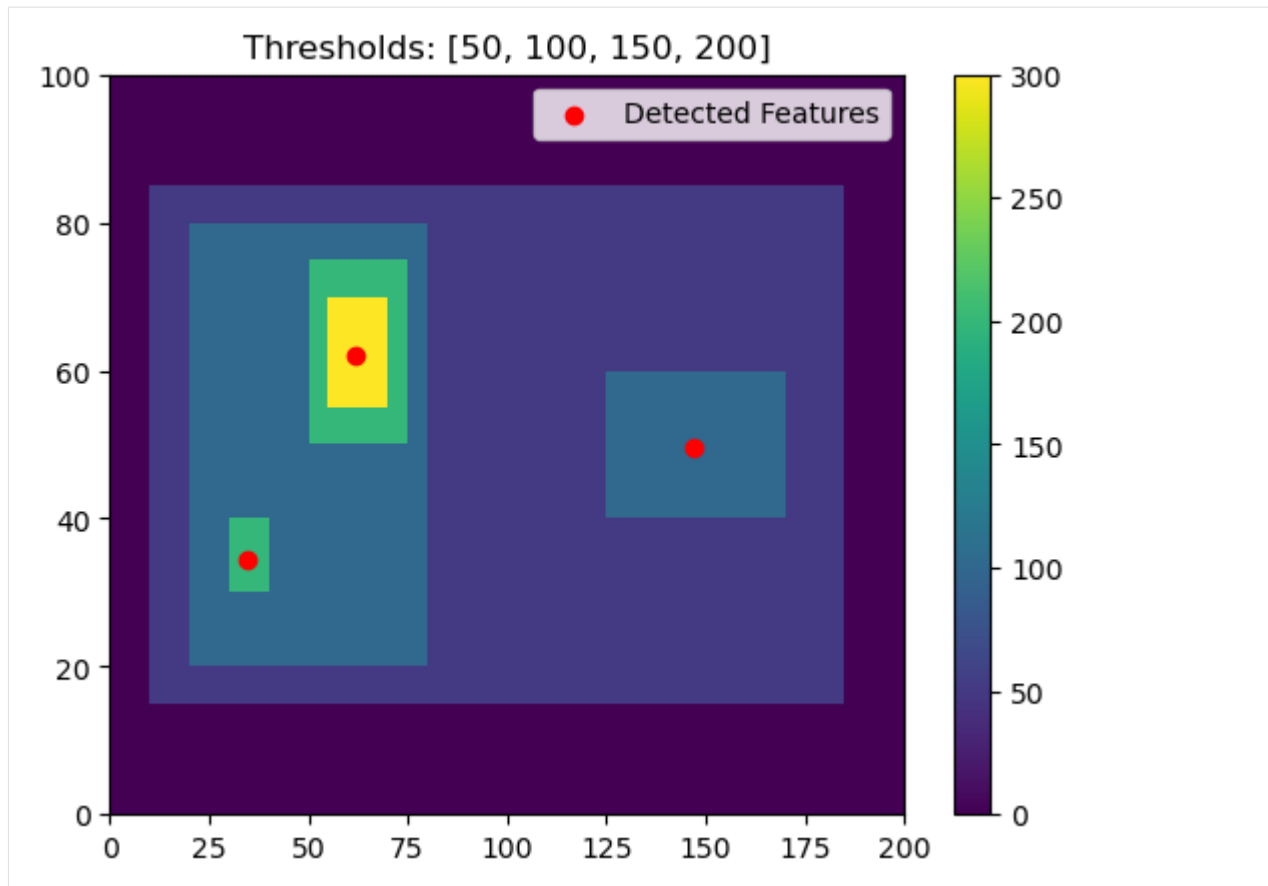
# Plot all features detected
plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
# 'hdim_1'].values, color='r', label="Detected Features")
plt.legend()
plt.title("Single Threshold of 100")
plt.show()
```

This is the power of having multiple thresholds. We can set thresholds of 50, 100, 150, 200 and capture both:

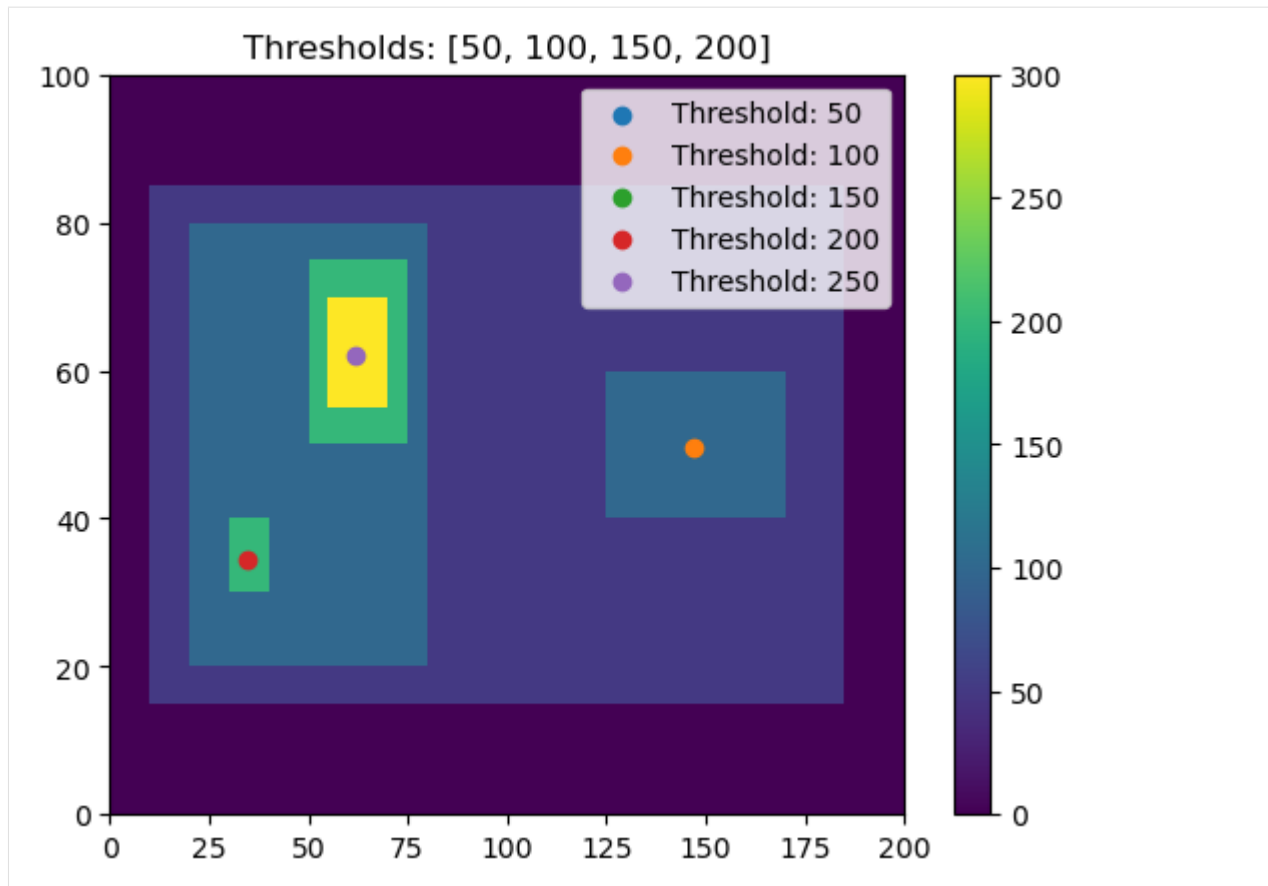
```
[7]: thresholds = [50, 100, 150, 200]
# Using 'center' here outputs the feature location as the arithmetic center of the_
↳ detected feature
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↳ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_threshold=
↳ 'center')
plt.pcolormesh(input_field_arr[0])
plt.colorbar()

# Plot all features detected
plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
↳ 'hdim_1'].values, color='r', label="Detected Features")
plt.legend()
plt.title("Thresholds: [50, 100, 150, 200]")
plt.show()
```



```
[8]: thresholds = [50, 100, 150, 200, 250]
# Using 'center' here outputs the feature location as the arithmetic center of the_
↳ detected feature
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↳ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_threshold=
↳ 'center')
plt.pcolormesh(input_field_arr[0])
plt.colorbar()

# Plot all features detected
for i, threshold in enumerate(thresholds):
    thresholded_points = single_threshold_features[single_threshold_features[
↳ 'threshold_value'] == threshold]
    plt.scatter(x=thresholded_points['hdim_2'].values,
                y=thresholded_points['hdim_1'].values,
                color='C'+str(i),
                label="Threshold: "+str(threshold))
plt.legend()
plt.title("Thresholds: [50, 100, 150, 200]")
plt.show()
```



10.2 How `n_min_threshold` changes what features are detected

10.2.1 Imports

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tobac
import xarray as xr
```

10.2.2 Generate Feature Data

Here, we will generate some simple feature data with a variety of features, large and small.

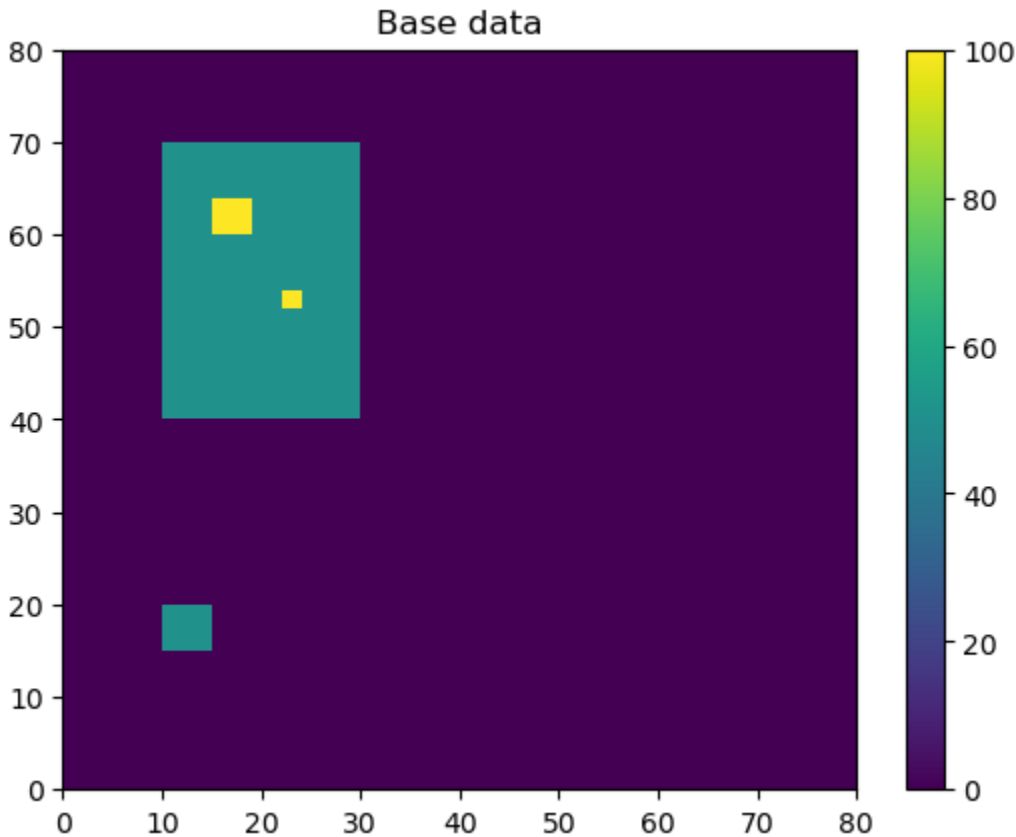
```
[2]: # Dimensions here are time, y, x.
input_field_arr = np.zeros((1, 80, 80))
# small 5x5 feature, area of 25 points
input_field_arr[0, 15:20, 10:15] = 50
# larger 30x30 feature, area of 900
input_field_arr[0, 40:70, 10:30] = 50
# small 2x2 feature within larger 30x30 feature, area of 4 points
input_field_arr[0, 52:54, 22:24] = 100
```

(continues on next page)

(continued from previous page)

```
# small 4x4 feature within larger 30x30 feature, area of 16 points
input_field_arr[0, 60:64, 15:19] = 100

plt.pcolormesh(input_field_arr[0])
plt.colorbar()
plt.title("Base data")
plt.show()
```

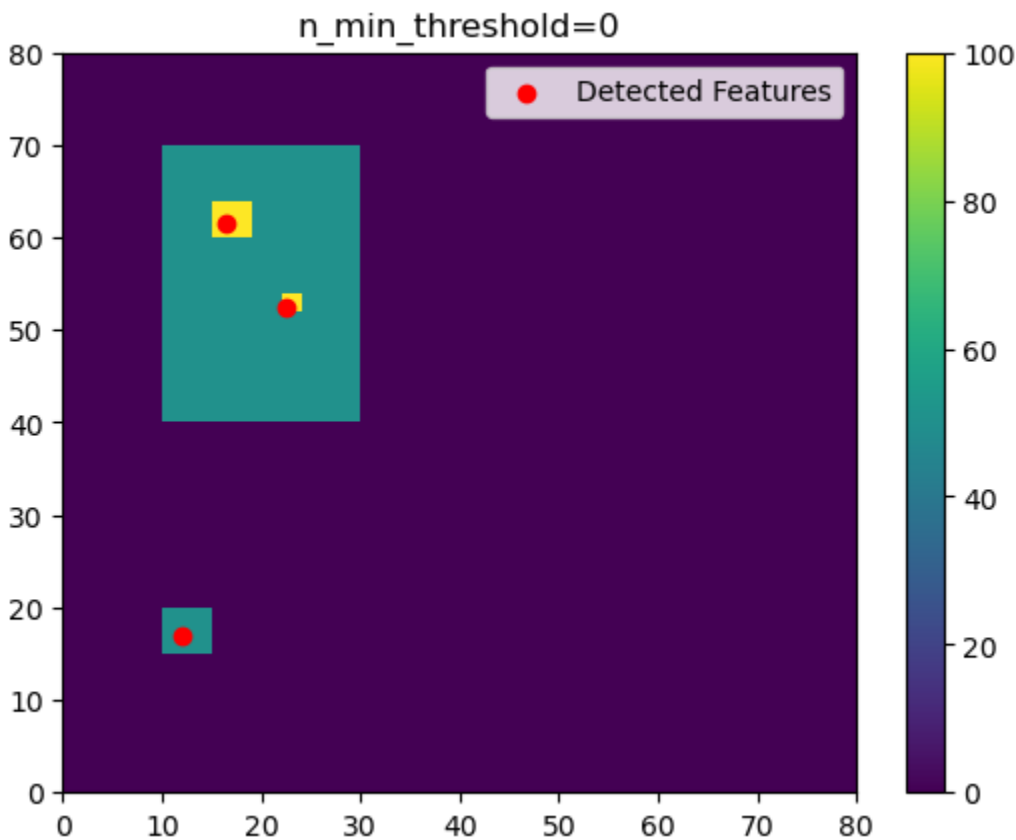


```
[3]: # We now need to generate an Iris DataCube out of this dataset to run tobac feature_
      ↪ detection.
      # One can use xarray to generate a DataArray and then convert it to Iris, as done_
      ↪ here.
input_field_iris = xr.DataArray(
    input_field_arr,
    dims=["time", "Y", "X"],
    coords={"time": [np.datetime64("2019-01-01T00:00:00")]},
).to_iris()
# Version 2.0 of tobac (currently in development) will allow the use of xarray_
↪ directly with tobac.
```

10.2.3 No `n_min_threshold`

If we keep `n_min_threshold` at the default value of 0, all three features will be detected with the appropriate thresholds used.

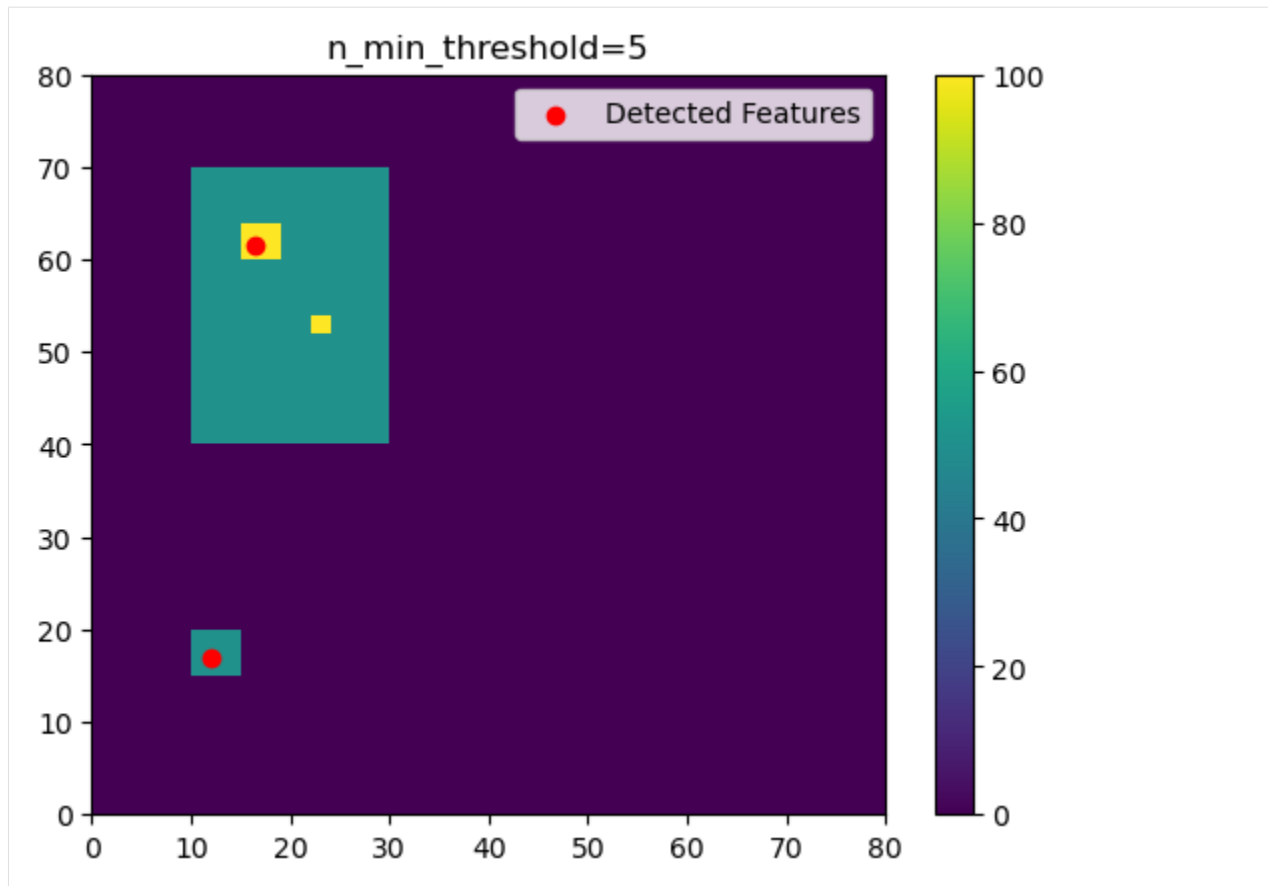
```
[4]: thresholds = [50, 100]
# Using 'center' here outputs the feature location as the arithmetic center of the_
# detected feature.
# All filtering is off in this example, although that is not usually recommended.
single_threshold_features = tobac.feature_detection_multithreshold(
    field_in=input_field_iris,
    dxy=1000,
    threshold=thresholds,
    target="maximum",
    position_threshold="center",
    sigma_threshold=0,
)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(
    x=single_threshold_features["hdim_2"].values,
    y=single_threshold_features["hdim_1"].values,
    color="r",
    label="Detected Features",
)
plt.legend()
plt.title("n_min_threshold=0")
plt.show()
```



10.2.4 Increasing n_min_threshold

As we increase `n_min_threshold`, fewer of these separate features are detected. In this example, if we set `n_min_threshold` to 5, the smallest detected feature goes away.

```
[5]: thresholds = [50, 100]
n_min_threshold = 5
# Using 'center' here outputs the feature location as the arithmetic center of the
# detected feature.
# All filtering is off in this example, although that is not usually recommended.
single_threshold_features = tobac.feature_detection_multithreshold(
    field_in=input_field_iris,
    dxy=1000,
    threshold=thresholds,
    target="maximum",
    position_threshold="center",
    sigma_threshold=0,
    n_min_threshold=n_min_threshold,
)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(
    x=single_threshold_features["hdim_2"].values,
    y=single_threshold_features["hdim_1"].values,
    color="r",
    label="Detected Features",
)
plt.legend()
plt.title("n_min_threshold={0}".format(n_min_threshold))
plt.show()
```



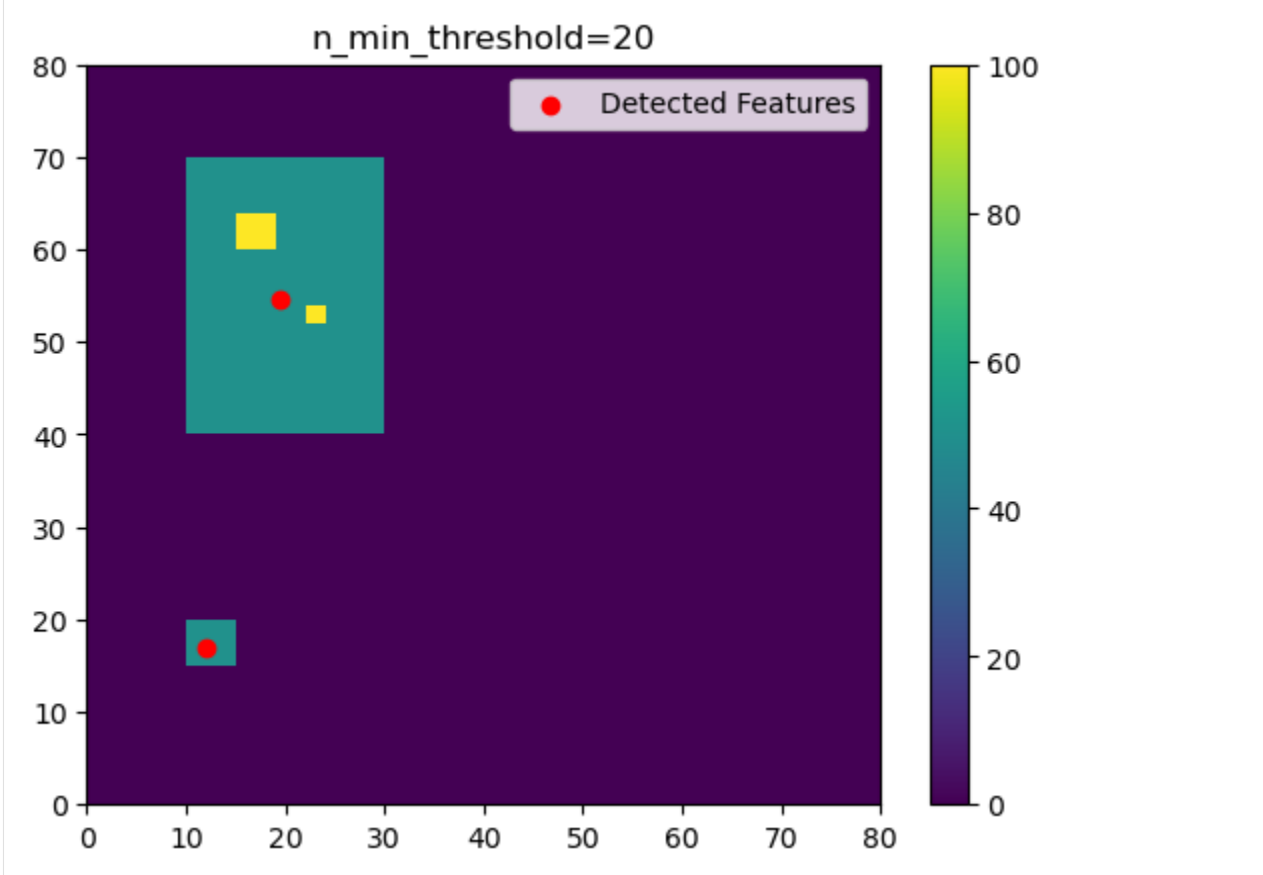
If we increase `n_min_threshold` to 20, only the large 50-valued feature is detected, rather than the two higher-valued squares.

```
[6]: thresholds = [50, 100]
n_min_threshold = 20
# Using 'center' here outputs the feature location as the arithmetic center of the
# detected feature.
# All filtering is off in this example, although that is not usually recommended.
single_threshold_features = tobac.feature_detection_multithreshold(
    field_in=input_field_iris,
    dxy=1000,
    threshold=thresholds,
    target="maximum",
    position_threshold="center",
    sigma_threshold=0,
    n_min_threshold=n_min_threshold,
)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(
    x=single_threshold_features["hdim_2"].values,
    y=single_threshold_features["hdim_1"].values,
    color="r",
    label="Detected Features",
)
plt.legend()
```

(continues on next page)

(continued from previous page)

```
plt.title("n_min_threshold={0}".format(n_min_threshold))
plt.show()
```



If we set `n_min_threshold` to 100, only the largest feature is detected.

```
[7]: thresholds = [50, 100]
n_min_threshold = 100
# Using 'center' here outputs the feature location as the arithmetic center of the_
# detected feature.
# All filtering is off in this example, although that is not usually recommended.
single_threshold_features = tobac.feature_detection_multithreshold(
    field_in=input_field_iris,
    dxy=1000,
    threshold=thresholds,
    target="maximum",
    position_threshold="center",
    sigma_threshold=0,
    n_min_threshold=n_min_threshold,
)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(
    x=single_threshold_features["hdim_2"].values,
    y=single_threshold_features["hdim_1"].values,
    color="r",
```

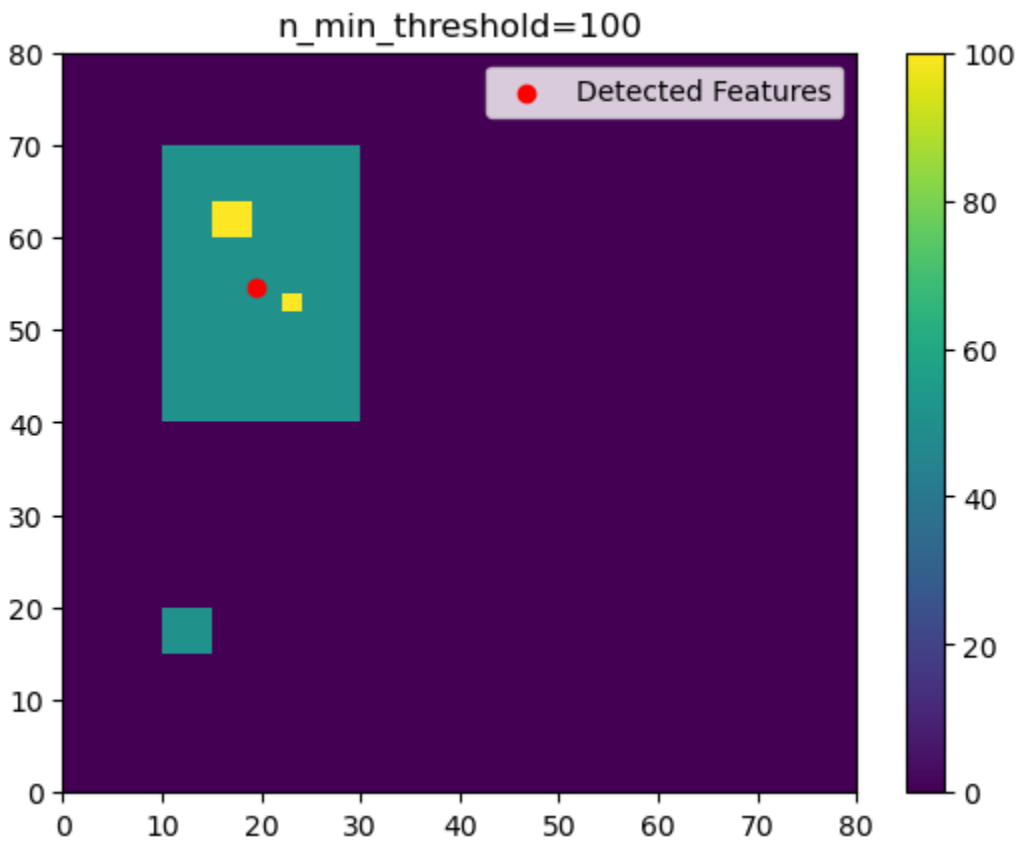
(continues on next page)

(continued from previous page)

```

    label="Detected Features",
)
plt.legend()
plt.title("n_min_threshold={0}".format(n_min_threshold))
plt.show()

```



10.2.5 Different `n_min_threshold` for different threshold values

Another option is to use different `n_min_threshold` values for different threshold values. This can be practical because extreme values are usually not as widespread as less extreme values (think for example of the rain rates in a convective system).

If we set `n_min_threshold` to 100 for the lower threshold and to 5 for the higher threshold, only the larger 50-valued features are detected, but at the same time we make sure that the smaller areas with the 100-valued features are still detected:

```

[8]: thresholds = [50, 100]

# defining multiple n_min_threshold:
n_min_threshold = [100, 5]
# alternatively, these could be given as a dictionary: n_min_threshold = {50:100,100:
↪ 5}

# let's add another larger 30x30 feature, area of 900 to make the example more clear
input_field_iris.data[0, 40:70, 40:60] = 50

```

(continues on next page)

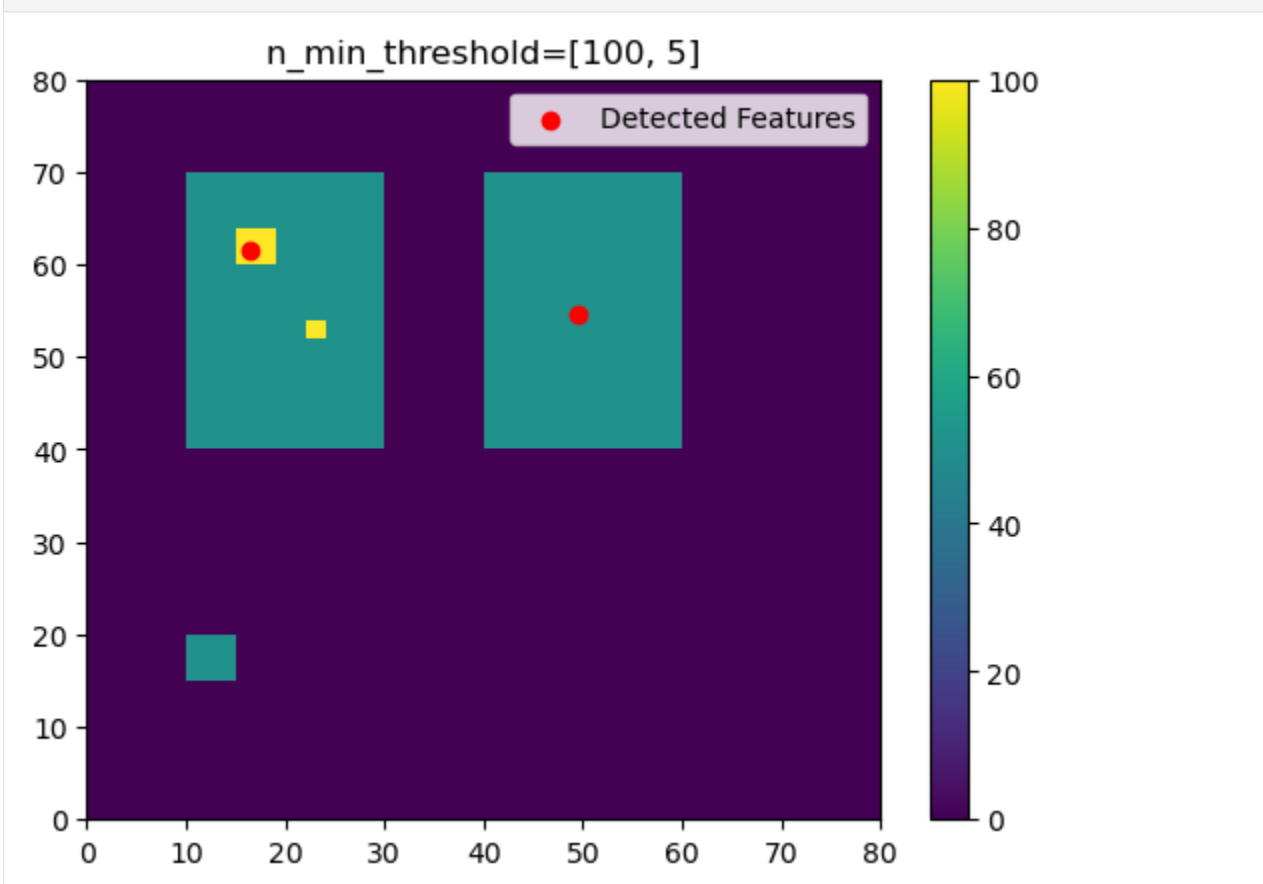
(continued from previous page)

```

# Using 'center' here outputs the feature location as the arithmetic center of the
↳ detected feature.
# All filtering is off in this example, although that is not usually recommended.
single_threshold_features = tobac.feature_detection_multithreshold(
    field_in=input_field_iris,
    dxy=1000,
    threshold=thresholds,
    target="maximum",
    position_threshold="center",
    sigma_threshold=0,
    n_min_threshold=n_min_threshold,
)

plt.pcolormesh(input_field_iris.data[0])
plt.colorbar()
# Plot all features detected
plt.scatter(
    x=single_threshold_features["hdim_2"].values,
    y=single_threshold_features["hdim_1"].values,
    color="r",
    label="Detected Features",
)
plt.legend()
plt.title("n_min_threshold={0}".format(n_min_threshold))
plt.show()

```



10.2.6 Strict Thresholding (`strict_thresholding`)

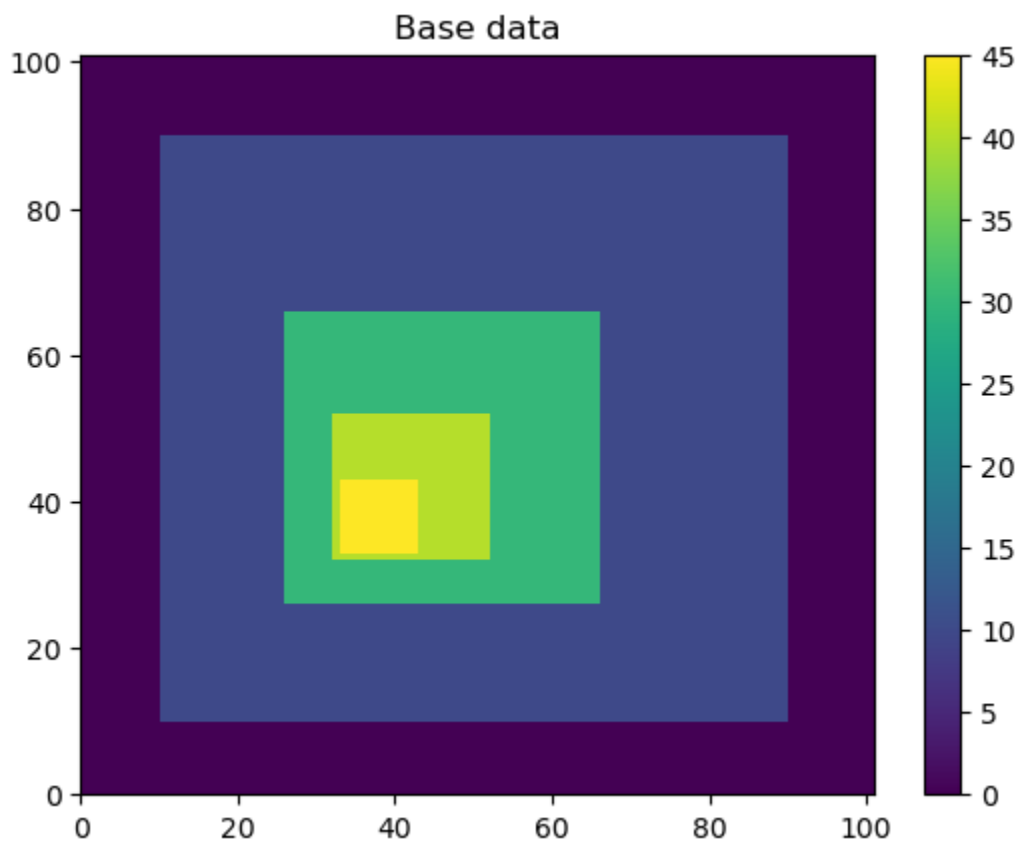
Sometimes it may be desirable to detect only features that satisfy *all* specified `n_min_threshold` thresholds. This can be achieved with the optional argument `strict_thresholding=True`.

```
[9]: # As above, we create test data to demonstrate the effect of strict_thresholding
input_field_arr = np.zeros((1, 101, 101))

for idx, side in enumerate([40, 20, 10, 5]):
    input_field_arr[
        :,
        (50 - side - 4 * idx) : (50 + side - 4 * idx),
        (50 - side - 4 * idx) : (50 + side - 4 * idx),
    ] = (
        50 - side
    )

input_field_iris = xr.DataArray(
    input_field_arr,
    dims=["time", "Y", "X"],
    coords={"time": [np.datetime64("2019-01-01T00:00:00")]},
).to_iris()

plt.pcolormesh(input_field_arr[0])
plt.colorbar()
plt.title("Base data")
plt.show()
```



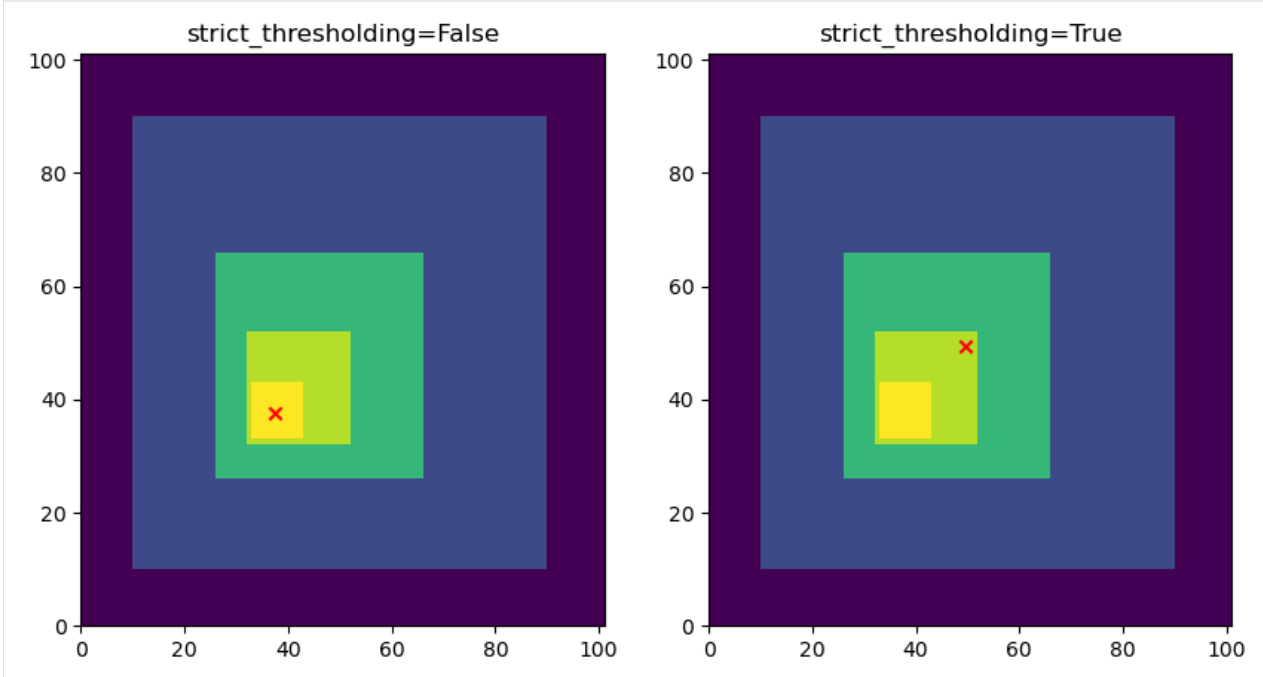
```
[10]: thresholds = [8, 29, 39, 44]

n_min_thresholds = [79**2, input_field_arr.size, 8**2, 3**2]

f = plt.figure(figsize=(10, 5))

# perform feature detection with and without strict thresholding and display the
# results side by side

for idx, strict in enumerate((False, True)):
    features_demo = tobac.feature_detection_multithreshold(
        input_field_iris,
        dxy=1000,
        threshold=thresholds,
        n_min_threshold=n_min_thresholds,
        strict_thresholding=strict,
    )
    ax = f.add_subplot(121 + idx)
    ax.pcolormesh(input_field_iris.data[0])
    ax.set_title(f"strict_thresholding={strict}")
    ax.scatter(
        x=features_demo["hdim_2"].values,
        y=features_demo["hdim_1"].values,
        marker="x",
        color="r",
        label="Detected Features",
    )
```



The effect of `strict_thresholding` can be observed in the plot above: Since the second `n_min_threshold` is not reached, no further features can be detected at higher `threshold` values. In the case of non strict thresholding, the feature with the highest value is still detected even though a previous `n_min_threshold` was not reached.

10.3 Different threshold_position options

10.3.1 Imports

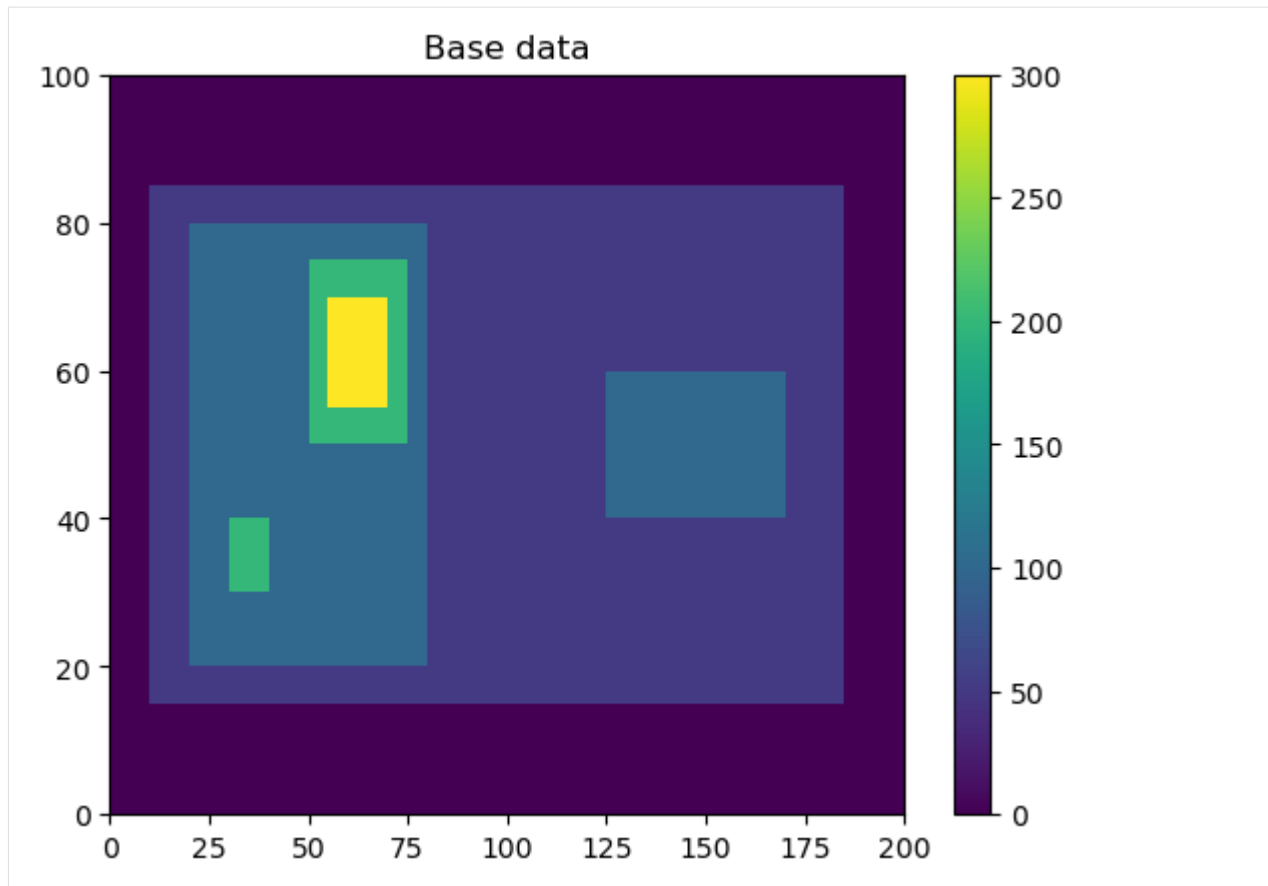
```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tobac
import xarray as xr
```

10.3.2 Generate Feature Data

Here, we will generate some simple feature data where the features that we want to detect are *higher* values than the surrounding (0).

```
[2]: # Dimensions here are time, y, x.
input_field_arr = np.zeros((1,100,200))
input_field_arr[0, 15:85, 10:185]=50
input_field_arr[0, 20:80, 20:80]=100
input_field_arr[0, 40:60, 125:170] = 100
input_field_arr[0, 30:40, 30:40]=200
input_field_arr[0, 50:75, 50:75]=200
input_field_arr[0, 55:70, 55:70]=300

plt.pcolormesh(input_field_arr[0])
plt.colorbar()
plt.title("Base data")
plt.show()
```



```
[3]: # We now need to generate an Iris DataCube out of this dataset to run tobac feature_
      ↪ detection.
      # One can use xarray to generate a DataArray and then convert it to Iris, as done_
      ↪ here.
      input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={
      ↪ 'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()
      # Version 2.0 of tobac (currently in development) will allow the use of xarray_
      ↪ directly with tobac.
```

10.3.3 position_threshold='center'

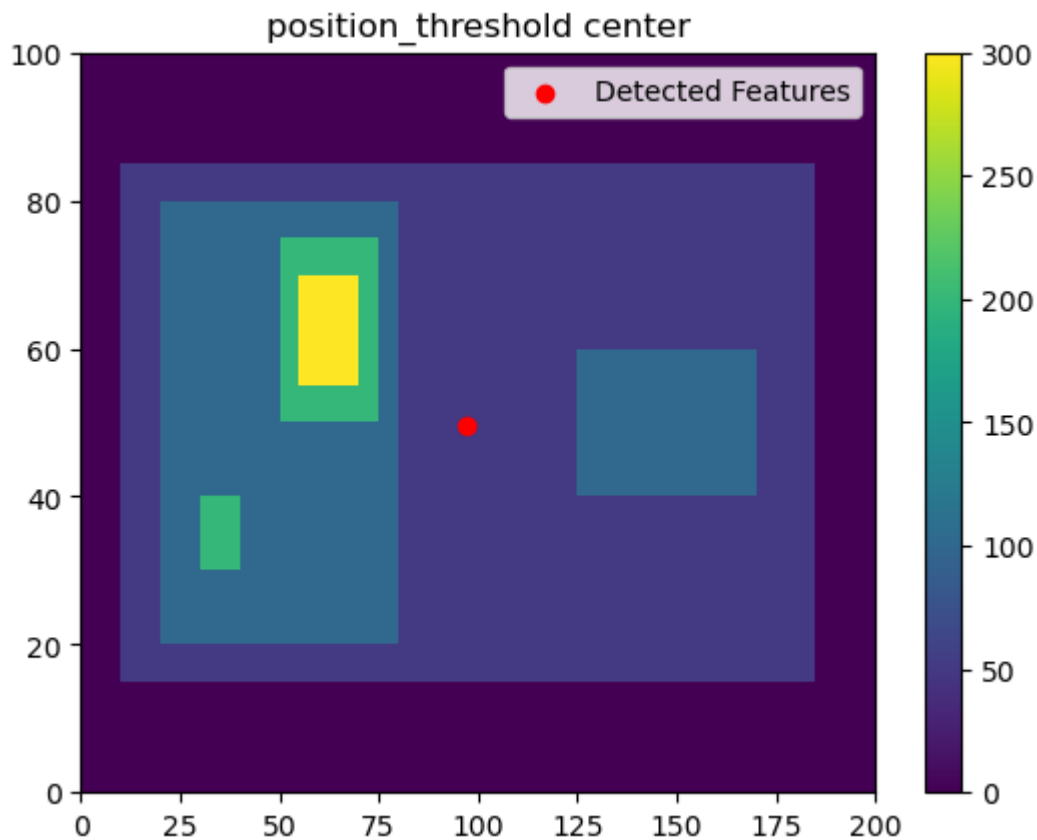
This option will choose the arithmetic center of the area above the threshold. This is typically not recommended for most data.

```
[4]: thresholds = [50,]
      position_threshold = 'center'
      single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
      ↪ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
      ↪ threshold=position_threshold)
      plt.pcolormesh(input_field_arr[0])
      plt.colorbar()
      # Plot all features detected
      plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
      ↪ 'hdim_1'].values, color='r', label="Detected Features")
```

(continues on next page)

(continued from previous page)

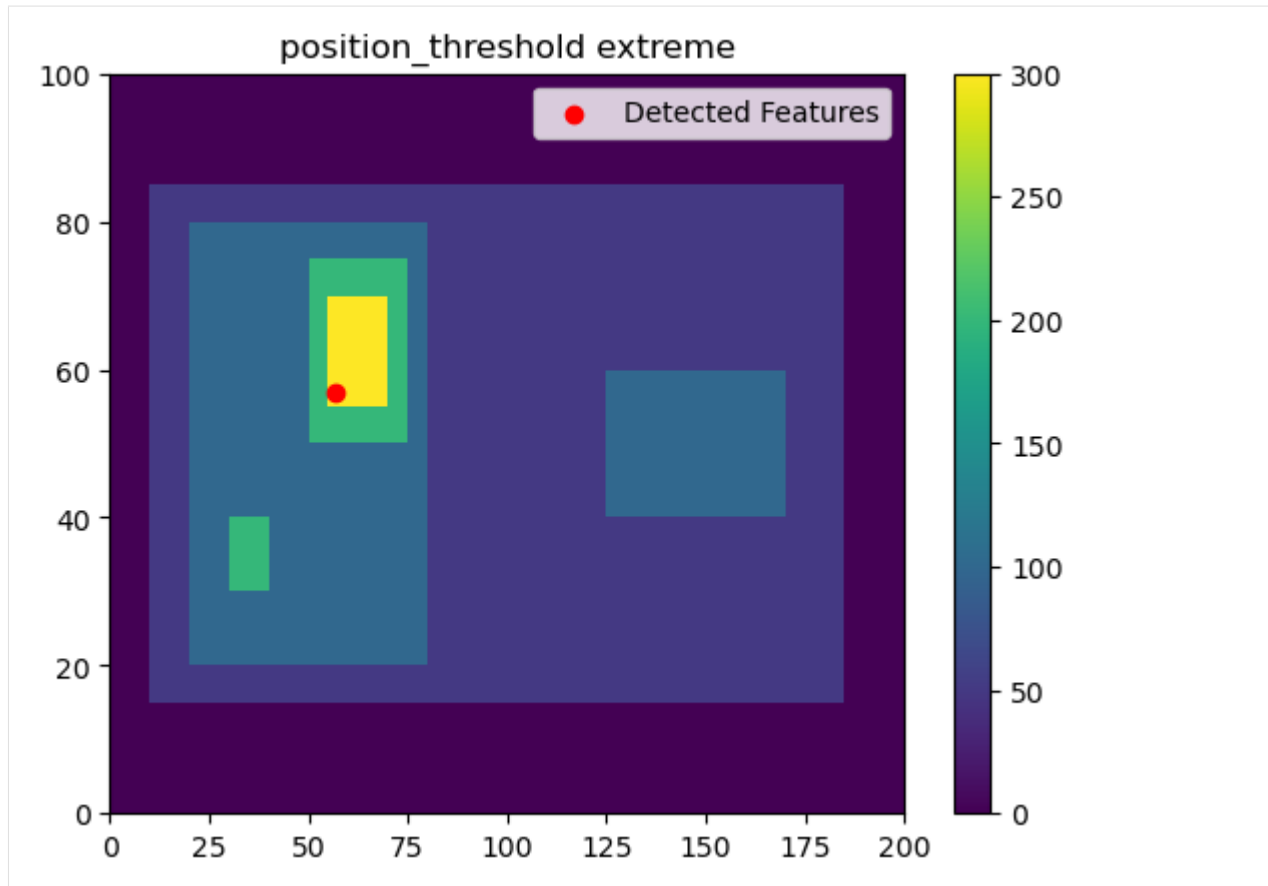
```
plt.legend()
plt.title("position_threshold "+ position_threshold)
plt.show()
```



10.3.4 position_threshold='extreme'

This option will choose the most extreme point of our data. For target='maximum', this will be the largest value in the feature area.

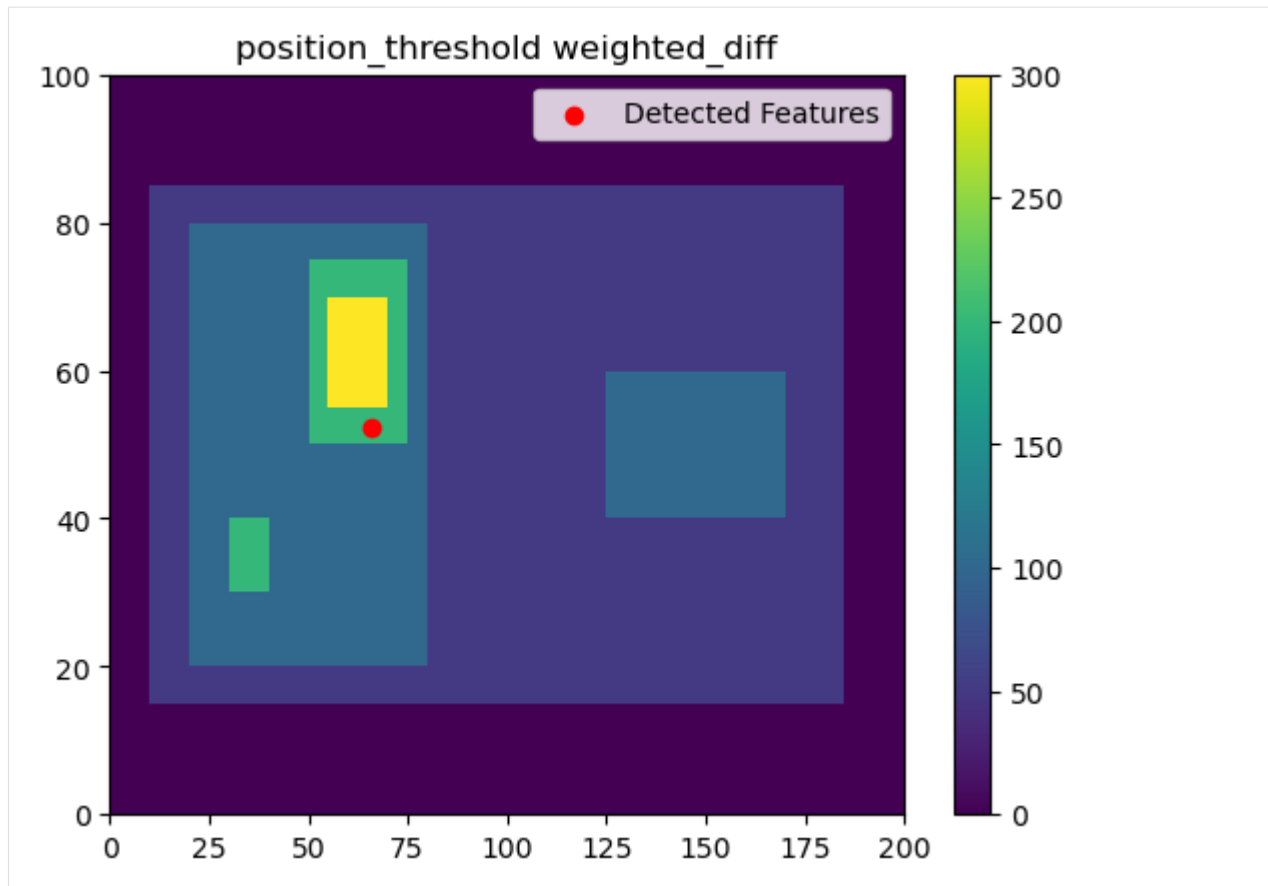
```
[5]: thresholds = [50,]
position_threshold = 'extreme'
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
    ↪ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
    ↪ threshold=position_threshold)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
    ↪ 'hdim_1'].values, color='r', label="Detected Features")
plt.legend()
plt.title("position_threshold "+ position_threshold)
plt.show()
```



10.3.5 position_threshold='weighted_diff'

This option will choose the center of the region weighted by the distance from the threshold value.

```
[6]: thresholds = [50,]
position_threshold = 'weighted_diff'
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
    ↪ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
    ↪ threshold=position_threshold)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
    ↪ 'hdim_1'].values, color='r', label="Detected Features")
plt.legend()
plt.title("position_threshold " + position_threshold)
plt.show()
```

10.3.6 position_threshold='weighted_abs'

This option will choose the center of the region weighted by the absolute values of the field.

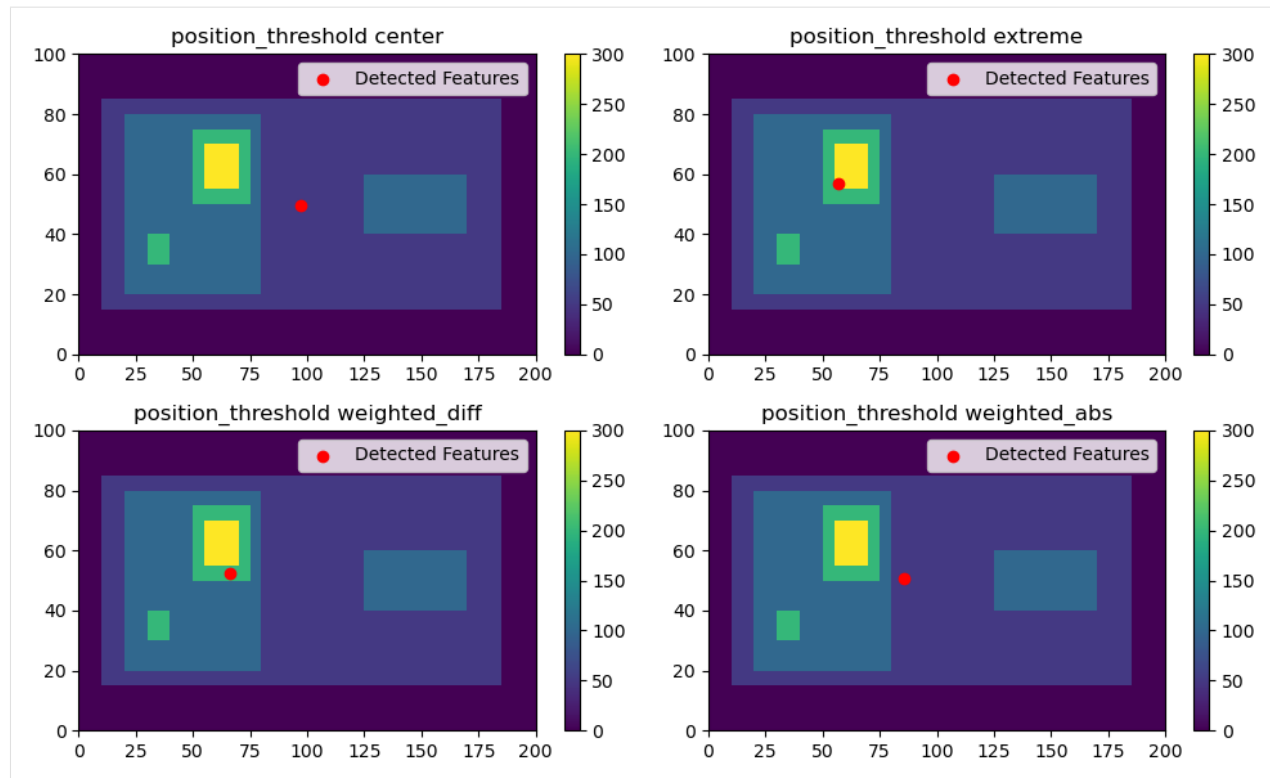
```
[7]: thresholds = [50,]
position_threshold = 'weighted_abs'
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
    ↪ field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
    ↪ threshold=position_threshold)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# Plot all features detected
plt.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_features[
    ↪ 'hdim_1'].values, color='r', label="Detected Features")
plt.legend()
plt.title("position_threshold "+ position_threshold)
plt.show()
```



10.3.7 All four methods together

```
[8]: thresholds = [50,]
fig, axarr = plt.subplots(2,2, figsize=(10,6))
testing_thresholds = ['center', 'extreme', 'weighted_diff', 'weighted_abs']
for position_threshold, ax in zip(testing_thresholds, axarr.flatten()):

    single_threshold_features = tobac.feature_detection_multithreshold(field_in =
    ↪ input_field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
    ↪ threshold=position_threshold)
    color_mesh = ax.pcolormesh(input_field_arr[0])
    plt.colorbar(color_mesh, ax=ax)
    # Plot all features detected
    ax.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_
    ↪ features['hdim_1'].values, color='r', label="Detected Features")
    ax.legend()
    ax.set_title("position_threshold "+ position_threshold)
plt.tight_layout()
plt.show()
```



10.4 *tobac* Feature Detection Filtering

Often, when detecting features with *tobac*, it is advisable to perform some amount of filtering on the data before feature detection is processed to improve the quality of the features detected. This notebook will demonstrate the affects of the various filtering algorithms built into *tobac* feature detection.

10.4.1 Imports

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tobac
import xarray as xr
import scipy.ndimage
import skimage.morphology
```

10.4.2 Generate Feature Data

Here, we will generate some simple feature data where the features that we want to detect are *higher* values than the surrounding (0).

```
[2]: # Dimensions here are time, y, x.
input_field_arr = np.zeros((1,100,200))
input_field_arr[0, 15:85, 10:185]=50
```

(continues on next page)

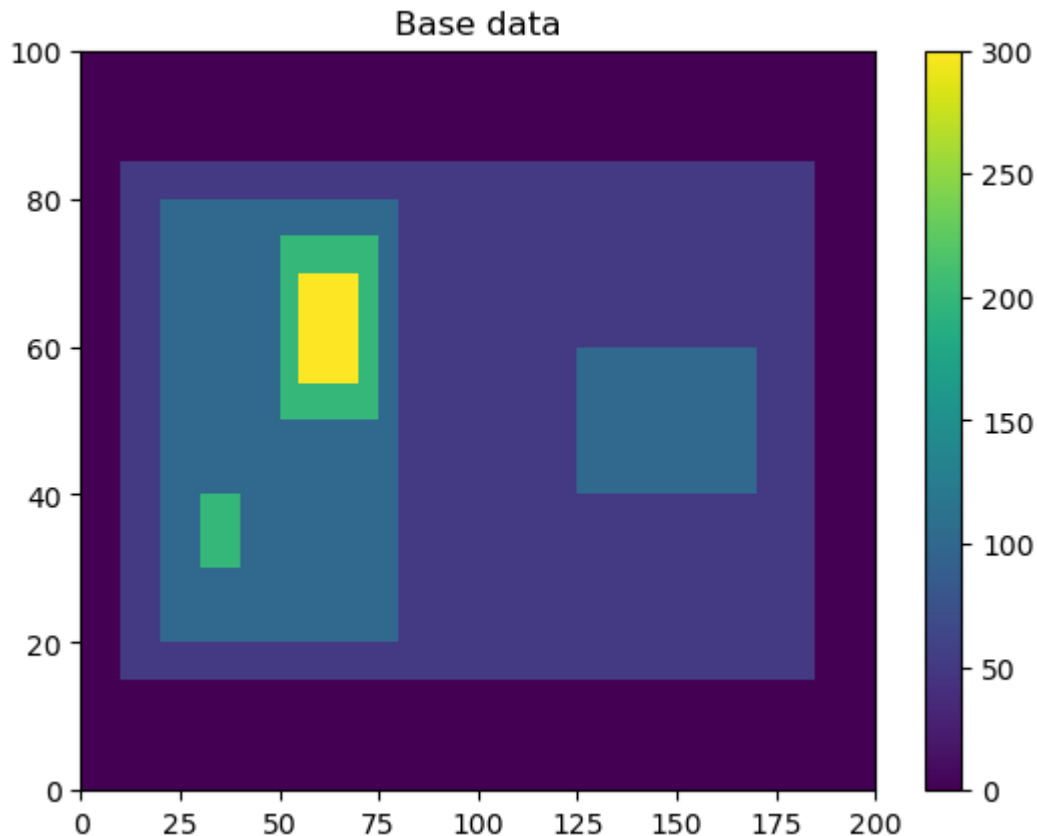
(continued from previous page)

```

input_field_arr[0, 20:80, 20:80]=100
input_field_arr[0, 40:60, 125:170] = 100
input_field_arr[0, 30:40, 30:40]=200
input_field_arr[0, 50:75, 50:75]=200
input_field_arr[0, 55:70, 55:70]=300

plt.pcolormesh(input_field_arr[0])
plt.colorbar()
plt.title("Base data")
plt.show()

```



```

[3]: # We now need to generate an Iris DataCube out of this dataset to run tobac feature_
      ↪ detection.
      # One can use xarray to generate a DataArray and then convert it to Iris, as done_
      ↪ here.
input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={
    ↪ 'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()
      # Version 2.0 of tobac (currently in development) will allow the use of xarray_
      ↪ directly with tobac.

```

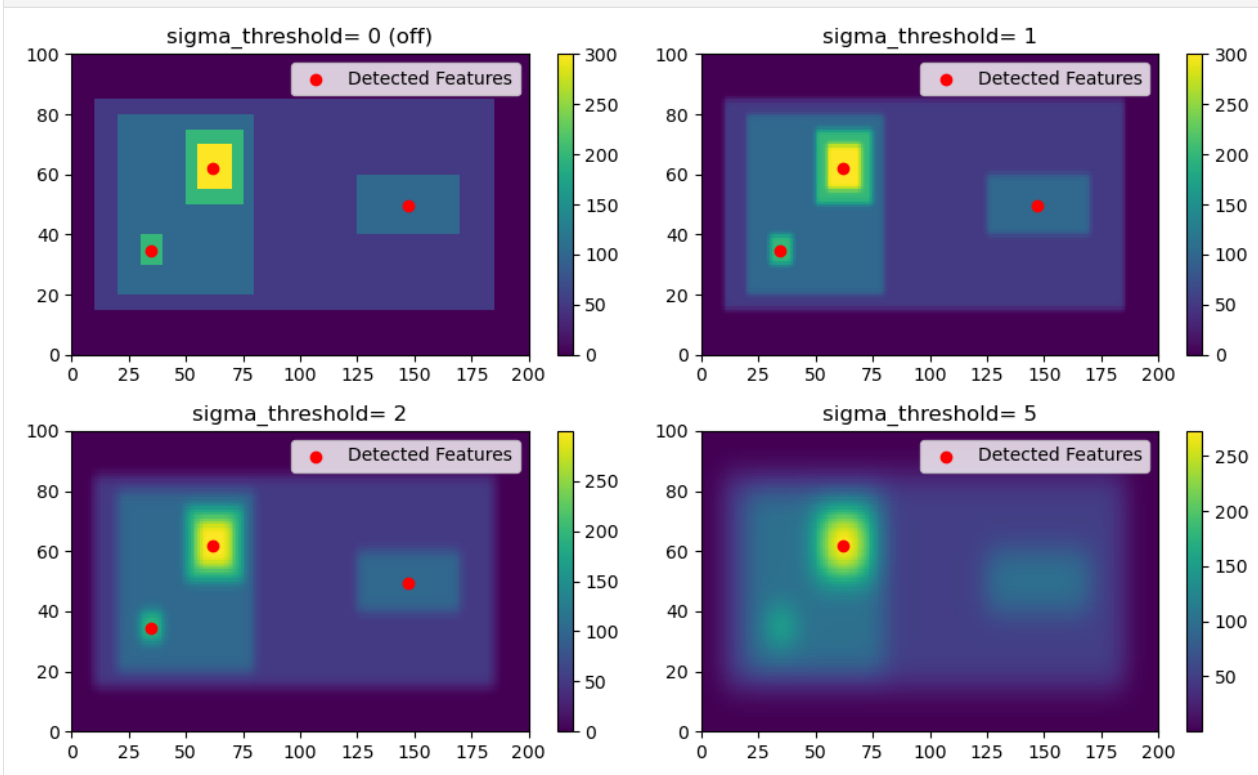
10.4.3 Gaussian Filtering (`sigma_threshold` parameter)

First, we will explore the use of Gaussian Filtering by varying the `sigma_threshold` parameter in *tobac*. Note that when we set the `sigma_threshold` high enough, the right feature isn't detected because it doesn't meet the

higher 100 threshold; instead it is considered part of the larger parent feature that contains the high feature.

```
[4]: thresholds = [50, 100, 150, 200]
fig, axarr = plt.subplots(2,2, figsize=(10,6))
sigma_values = [0, 1, 2, 5]
for sigma_value, ax in zip(sigma_values, axarr.flatten()):
    single_threshold_features = tobac.feature_detection_multithreshold(field_in =
    ↪ input_field_iris, dxy = 1000, threshold=thresholds, target='maximum', sigma_
    ↪ threshold=sigma_value)

    # This is what tobac sees
    filtered_field = scipy.ndimage.gaussian_filter(input_field_arr[0], sigma=sigma_
    ↪ value)
    color_mesh = ax.pcolormesh(filtered_field)
    plt.colorbar(color_mesh, ax=ax)
    # Plot all features detected
    ax.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_
    ↪ features['hdim_1'].values, color='r', label="Detected Features")
    ax.legend()
    if sigma_value == 0:
        sigma_val_str = "0 (off)"
    else:
        sigma_val_str = "{0}".format(sigma_value)
    ax.set_title("sigma_threshold= " + sigma_val_str)
plt.tight_layout()
plt.show()
```



10.4.4 Erosion (n_erosion_threshold parameter)

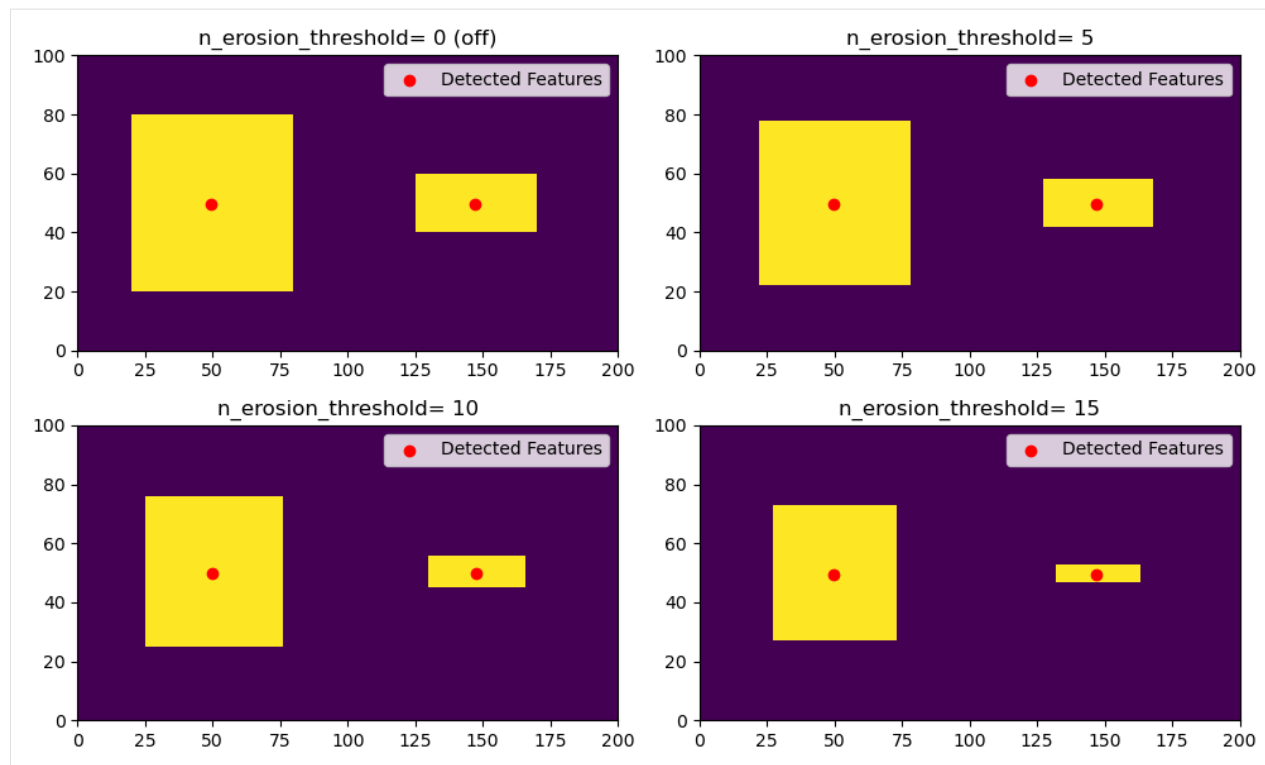
Next, we will explore the use of the erosion filtering by varying the `n_erosion_threshold` parameter in `tobac`. This erosion process only occurs *after* masking the values greater than the threshold, so it's easiest to see this when detecting on a single threshold. As you can see, increasing the `n_erosion_threshold` parameter reduces the size of each of our features.

```
[5]: thresholds = [100]
fig, axarr = plt.subplots(2,2, figsize=(10,6))
erosion_values = [0, 5, 10, 15]
for erosion, ax in zip(erosion_values, axarr.flatten()):
    single_threshold_features = tobac.feature_detection_multithreshold(field_in =
    ↪ input_field_iris, dxy = 1000, threshold=thresholds, target='maximum', n_erosion_
    ↪ threshold=erosion)

    # Create our mask- this is what tobac does internally for each threshold.
    tobac_mask = 1*(input_field_arr[0] >= thresholds[0])

    if erosion > 0:
        # This is the parameter for erosion that gets passed to the scikit-image_
    ↪ library.
        footprint = np.ones((erosion, erosion))
        # This is what tobac sees after erosion.
        filtered_mask = skimage.morphology.binary_erosion(tobac_mask, footprint).
    ↪ astype(np.int64)
    else:
        filtered_mask = tobac_mask

    color_mesh = ax.pcolormesh(filtered_mask)
    # Plot all features detected
    ax.scatter(x=single_threshold_features['hdim_2'].values, y=single_threshold_
    ↪ features['hdim_1'].values, color='r', label="Detected Features")
    ax.legend()
    if erosion == 0:
        sigma_val_str = "0 (off)"
    else:
        sigma_val_str = "{0}".format(erosion)
    ax.set_title("n_erosion_threshold= "+ sigma_val_str)
plt.tight_layout()
plt.show()
```



Feature Detection Output

Feature detection outputs a *pandas* dataframe with several variables. The variables, (with column names listed in the *Variable Name* column), are described below with units. Note that while these variables come initially from the feature detection step, segmentation and tracking also share some of these variables as keys (e.g., the `feature` acts as a universal key between each of these). See [Tracking Output](#) for the additional columns added by tracking.

Variables that are common to all feature detection files:

Table 1: tobac Feature Detection Output Variables

Variable Name	Description	Units	Type
frame	Frame/time/file number; starts from 0 and increments by 1 to N times.	n/a	int64
idx	Feature number within that frame; starts at 1, increments by 1 to the number of features for each frame, and resets to 1 when the frame increments	n/a	int
hdim_1	First horizontal dimension in grid point space (typically, although not always, N/S or y space)	Number of grid points	float
hdim_2	Second horizontal dimension in grid point space (typically, although not always, E/W or x space)	Number of grid points	float
num	Number of grid points that are within the threshold of this feature	Number of grid points	int
threshold	Maximum threshold value reached by the feature	Units of the input feature	int
feature	Unique number of the feature; starts from 1 and increments by 1 to the number of features identified in all frames	n/a	int
time	Time of the feature	Date and time	object/python date-time
timestr	String representation of the feature time	YYYY-MM-DD HH:MM:SS	object/string
y	Grid point y location of the feature (see hdim_1 and hdim_2). Note that this is not necessarily an integer value depending on your selection of position_threshold	Number of grid points	float
x	Grid point x location of the feature (see also y)	Number of grid points	float
projection_y	Y location of the feature in projection coordinates	Projection coordinates (usually m)	float
projection_x	X location of the feature in projection coordinates	Projection coordinates (usually m)	float
lat	Latitude of the feature	Decimal degrees	float

Variables that are included when using 3D feature detection in addition to those above:

Table 2: tobac 3D Feature Detection Output Variables

Variable Name	Description	Units	Type
vdim	vertical dimension in grid point space	Number of grid points	float64
z	grid point z location of the feature (see vdim). Note that this is not necessarily an integer value depending on your selection of position_threshold	Number of grid points	float64
altitude	z location of the feature above ground level	meters	float64

The segmentation step aims at associating cloud areas (2D data) or cloud volumes (3D data) with the identified and tracked features.

Currently implemented methods:

Watershedding in 2D: Markers are set at the position of the individual feature positions identified in the detection step. Then watershedding with a fixed threshold is used to determine the area around each feature above/below that threshold value. This results in a mask with the feature id at all pixels identified as part of the clouds and zeros in all cloud free areas.

Watershedding in 3D: Markers are set in the entire column above the individual feature positions identified in the detection step. Then watershedding with a fixed threshold is used to determine the volume around each feature above/below that threshold value. This results in a mask with the feature id at all voxels identified as part of the clouds and zeros in all cloud free areas.

Watershedding Segmentation Parameters

Appropriate parameters must be chosen to properly use the watershedding segmentation module in *tobac*. This page gives a brief overview of parameters available in watershedding segmentation.

A full list of parameters and descriptions can be found in the API Reference: `tobac.segmentation.segmentation()`.

13.1 Basic Operating Procedure

The *tobac* watershedding segmentation algorithm selects regions of the data field with values greater than `threshold` and associates those regions with the features detected by feature detection (see *Feature Detection Basics*). This algorithm uses a *watershedding* approach, which sets the individual features as initial seed points, and then has identified regions grow from those original seed points. For further information on watershedding segmentation, see the *scikit-image documentation* <https://scikit-image.org/docs/stable/auto_examples/segmentation/plot_watershed.html>.

Note that you can run the watershedding segmentation algorithm on any variable that shares a grid with the variable detected in the feature detection step. It is not required that the variable used in feature detection be the same as the one in segmentation (e.g., you can detect updraft features and then run segmentation on total condensate).

Segmentation can be run on 2D or 3D input data, but segmentation on 3D data using a 2D feature detection field requires careful consideration of where the vertical seeding will occur (see *Level*).

13.2 Target

The `target` parameter works similarly to how it works in feature detection (see *Threshold Feature Detection Parameters*). To segment areas that are greater than `threshold`, use `target='maximum'`. To segment areas that are less than `threshold`, use `target='minimum'`.

13.3 Threshold

Unlike in multiple threshold detection in Feature Detection, Watershedding Segmentation only accepts one threshold. This value will set either the minimum (for `target='maximum'`) or maximum (for `target='minimum'`) value to be segmented.

13.4 Where the 3D seeds are placed for 2D feature detection

When running feature detection on a 2D dataset and then using these detected features to segment data in 3D, there is clearly no information on where to put the seeds in the vertical. This is currently controlled by the `level` parameter. By default, this parameter is `None`, which seeds the full column at every 2D detected feature point. As *tobac* does not run a continuity check, this can result in undesired behavior, such as clouds in multiple layers being detected as one large object.

`level` can also be set to a *slice* <<https://docs.python.org/3/c-api/slice.html>>, which determines where in the vertical dimension (see ‘**Vertical Coordinate**’_) the features are seeded from. Note that `level` operates in *array* coordinates rather than physical coordinates.

13.5 Maximum Distance

tobac’s watershedding segmentation allows you to set a maximum distance away from the feature to classify as a segmented region belonging to that figure. `max_distance` sets this distance in meters away from the detected feature to allow it to be considered part of the point. To turn this feature off, set `max_distance=None`.

Segmentation Output

Segmentation outputs a mask (*iris.cube.Cube* and in the future *xarray.DataArray*) with the same dimensions as the input field, where each segmented area has the same ID as its corresponding feature (see *feature* column in [Feature Detection Output](#)). Note that there are some cases in which a feature is not attributed to a segmented area associated with it (see [Features without segmented areas](#)).

Segmentation also outputs the same *pandas* dataframe as obtained by Feature Detection (see [Feature Detection Basics](#)) but with one additional column:

Table 1: tobac Segmentation Output Variables

Variable Name	Description	Units	Type
ncells	Total number of grid points that belong to the segmented area associated with feature.	n/a	int64

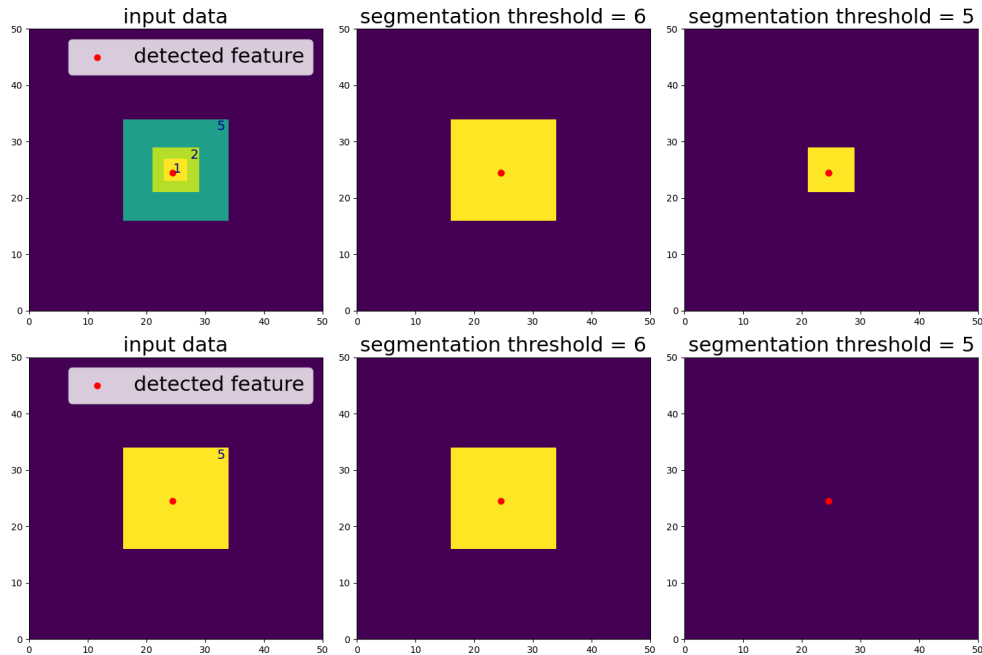
Features without segmented areas

Not all detected features have a segmented area associated with them. Here, we show two cases in which a detected feature might not have a segmented area associated with them (meaning that the mask file does not contain the ID of the feature of interest and *ncells* in the segmentation output dataframe results in 0 grid cells.)

15.1 Case 1: Segmentation threshold

If the segmentation threshold is lower (assuming *target='minimum'*) than the highest threshold specified in the Feature Detection (see *Threshold Feature Detection Parameters*) this could leave some features without a segmented area, simply because there are no values to be segmented.

Consider for example the following data with 5 being the highest threshold specified for the Feature Detection (see *Feature Detection Basics*):

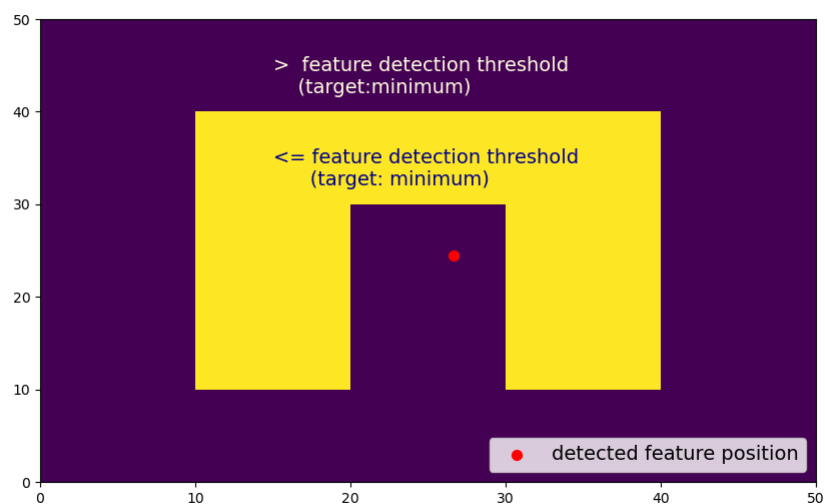


If the segmentation threshold is larger than 5 (e.g. *threshold* = 6), the segmented area contains all values ≤ 5 (still assuming *target* = 'minimum'), no matter if the detected feature has a threshold lower than 5 (upper panels) or if it is exactly equal to 5 and does not contain any features with lower thresholds inside (lower panels).

If the segmentation threshold is lower than or equal to the highest feature detection threshold (e.g. *threshold* = 5), features with threshold values lower than 5 still get a segmented area associated with them (upper panels). However, features that are exactly equal to 5 and do not contain any features with lower thresholds inside will not get any segmented area associated with them (lower panels) which results in no values in the mask for this feature and *ncells*=0.

15.2 Case 2: Feature position

Another reason for features that do not have a segmented area associated with them is the rare but possible case when the feature position is located outside of the threshold area:

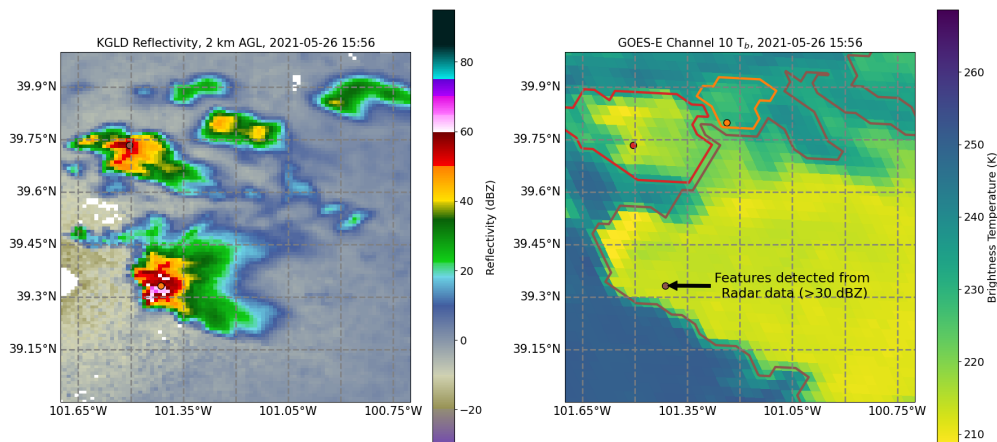


In this case, it may help to change the *position_threshold* (see *Threshold Feature Detection Parameters*) to *extreme* instead of *center*:



Track on one dataset, segment on another

tobac also has the capability to combine datasets through *Segmentation*, which includes the ability to track on one dataset (e.g., gridded radar data) and run segmentation on a different dataset *on a different grid* (e.g., satellite data).

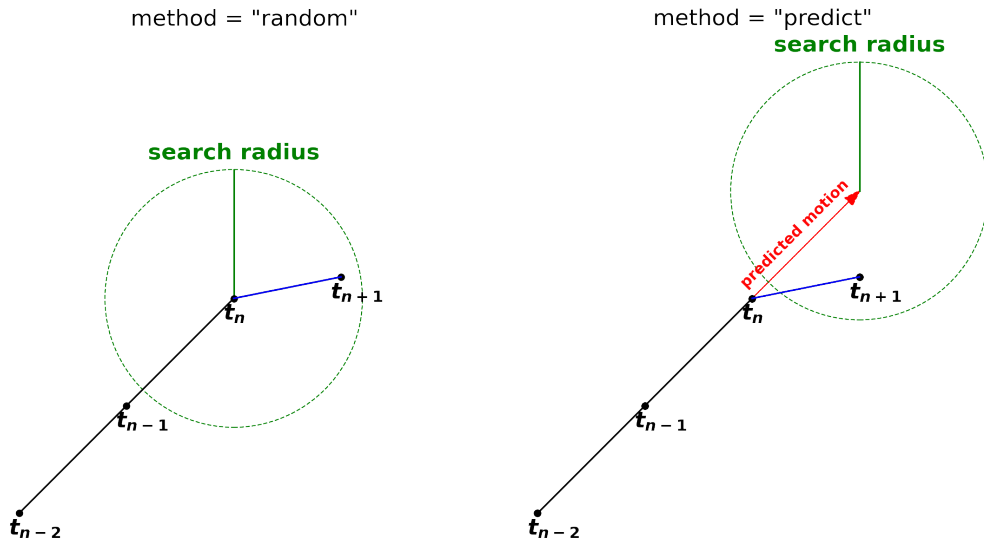


To do this, users should first run *Feature Detection Basics* with a dataset that contains latitude and longitude coordinates, such that they appear in the output dataframe from Feature Detection. Next, use `tobac.utils.transform_feature_points()` to transform the feature dataframe into the new coordinate system. Finally, use the output from `tobac.utils.transform_feature_points()` to run segmentation with the new data. This can be done with both 2D and 3D feature detection and segmentation.

Currently implemented options for linking detected features into tracks:

Trackpy:

This method uses the trackpy library (<http://soft-matter.github.io/trackpy>). This approach only takes the point-like position of the feature, e.g. determined as the weighted mean, into account. Features to link with are looked for in a search radius defined by the parameters `v_max` or `d_max`. The position of the center of this search radius is determined by the method keyword. `method="random"` uses the position of the current feature (t_i), while `method="predict"` makes use of the information from the linked feature in the previous timestep (t_{i-1}) to predict the next position. For a simple case the search radii of the two methods look like this:

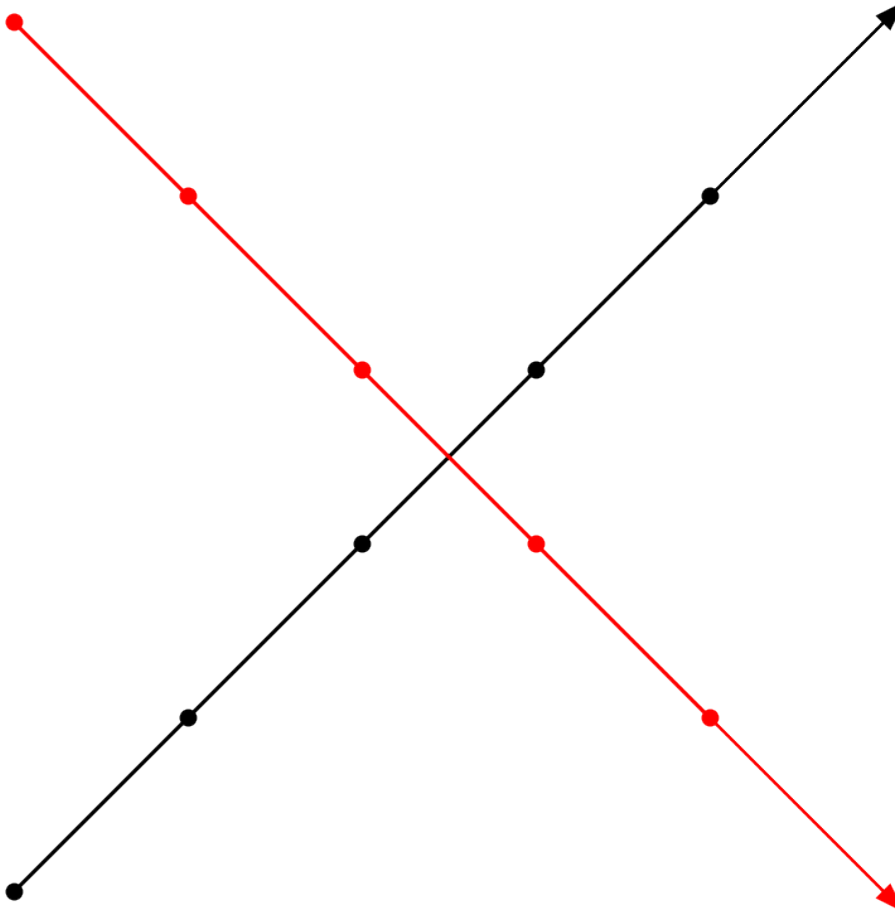


If there is only one feature in the search radius, the linking can happen immediately. If there are none, the track ends at this timestep. If there are more options, trackpy performs a decision process. Assume there are N features in the current and also N in the next timeframe and they are all within each search radius. This means there are $N!$ options for linking. Each of these options means that N distances between the center of the search radius of a current feature and a feature from the next time frame $\delta_n, n = 1, 2, \dots, N$ are traveled by the features. Trackpy will calculate the sum

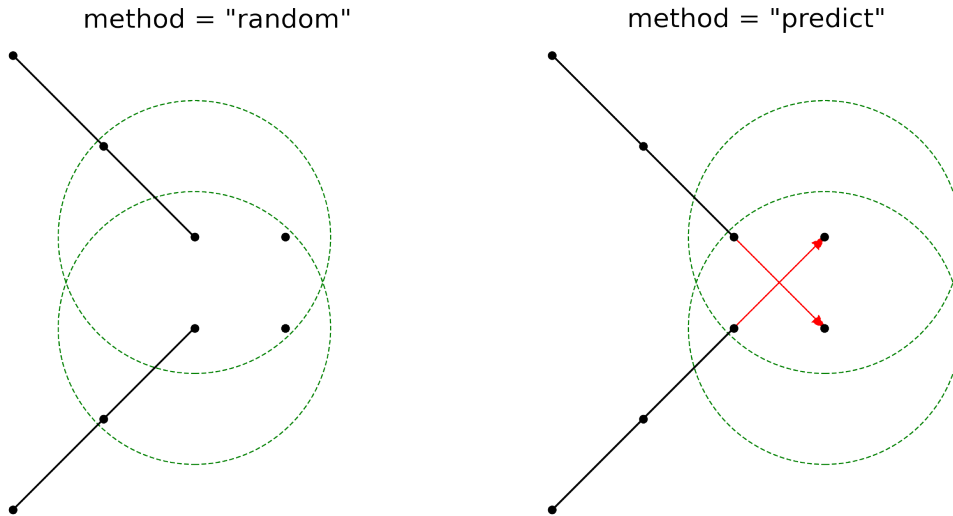
of the squared distances

$$\sum_{n=1}^N \delta_n^2.$$

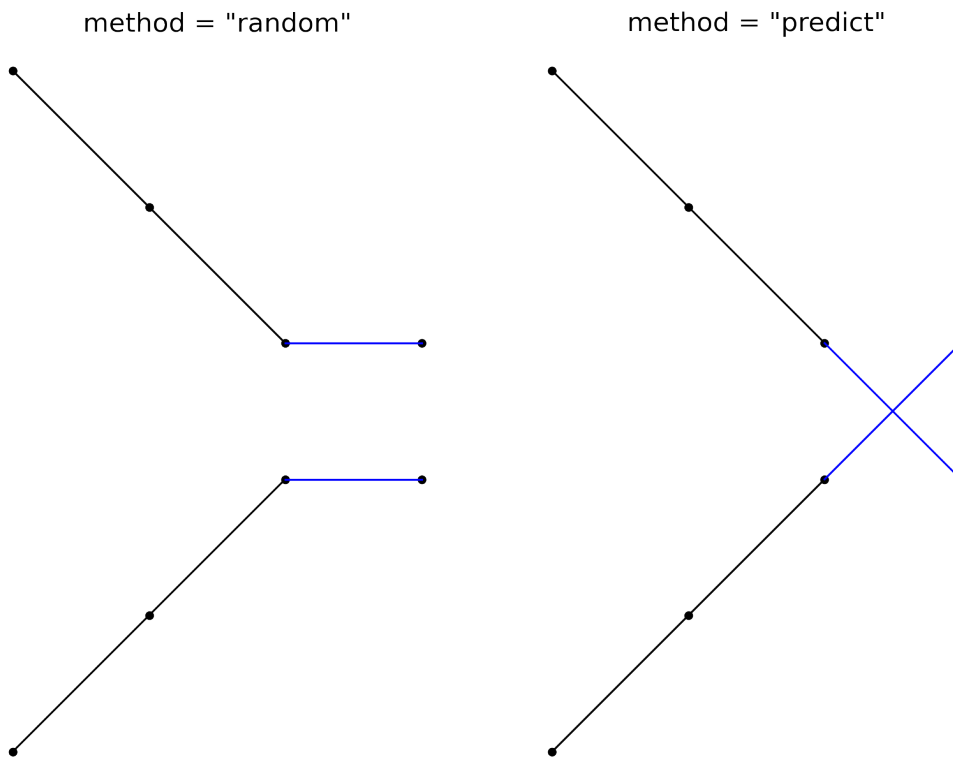
For every option the lowest value of this sum is used for linking. As an example, consider these two crossing features:



If the search radius is chosen large enough, each will contain two options, a horizontal and a diagonal feature:



The linking will look differently for both methods in the end. The horizontal features are closer to the current position than the diagonal ones. This is why these are linked by the “random”-method. On the other hand, the diagonal features lie exactly on the guessed positions for the “predict”-method. This means, that the sum of squared distances is 0 and they are the optimal decision for linking in this case:



CHAPTER 18

Tracking Output

Tracking outputs a *pandas* dataframe with variables in addition to the variables output by Feature Detection (see [Feature Detection Output](#)). While this is a separate dataframe than the one output by Feature Detection, it is identical except for the addition of the columns listed below. The additional variables added by tracking, with column names listed in the *Variable Name* column, are described below with units.

Variables that are common to all tracking files:

Table 1: tobac Tracking Output Variables

Variable Name	Description	Units	Type
cell	Tracked cell number; generally starts from 1. Untracked cell value can be set; but by default is -1.	n/a	int
time_cell	Time since cell was first detected.	minutes	object/python timedelta

CHAPTER 19

Merge and Split

This submodule is a post processing step to address tracked cells which merge/split. The first iteration of this module is to combine the cells which are merging but have received a new cell id (and are considered a new cell) once merged. This module uses a minimum euclidian spanning tree to combine merging cells, thus the postfix for the function is MEST. This submodule will label merged/split cells with a TRACK number in addition to its CELL number.

Features, cells, and tracks are combined using parent/child nomenclature. (quick note on terms; “feature” is a detected object at a single time step (see *Feature Detection Basics*). “cell” is a series of features linked together over multiple timesteps (see *Linking*). “track” may be an individual cell or series of cells which have merged and/or split.)

Overview of the output dataframe from merge_split

d : *xarray.core.dataset.Dataset*

xarray dataset of tobac merge/split cells with parent and child designations.

Parent/child variables include:

- `cell_parent_track_id`: The associated track id for each cell. All cells that have merged or split will have the same parent track id. If a cell never merges/splits, only one cell will have a particular track id.
- `feature_parent_cell_id`: The associated parent cell id for each feature. All feature in a given cell will have the same cell id.
- `feature_parent_track_id`: The associated parent track id for each feature. This is not the same as the cell id number.
- `track_child_cell_count`: The total number of features belonging to all child cells of a given track id.
- `cell_child_feature_count`: The total number of features for each cell.

Example usage:

```
d = merge_split_MEST(Track)
```

`merge_split` outputs an *xarray* dataset with several variables. The variables, (with column names listed in the *Variable Name* column), are described below with units. Coordinates and dataset dimensions are Feature, Cell, and Track.

Variables that are common to all feature detection files:

Table 1: tobac Merge_Split Track Output Variables

Variable Name	Description	Units	Type
feature	Unique number of the feature; starts from 1 and increments by 1 to the number of features identified in all frames	n/a	int64
cell	Tracked cell number; generally starts from 1. Untracked cell value is -1.	n/a	int64
track	Unique number of the track; starts from 0 and increments by 1 to the number of tracks identified. Untracked cells and features have a track id of -1.	n/a	int64
cell_parent	The track id of the parent cell. All cells that have merged or split will have the same parent track id. If a cell never merges/splits, only one cell will have a particular track id.	n/a	int64
feature_cell	The cell id of the parent cell for each feature. All feature in a given cell will have the same cell id.	n/a	int64
feature_track	The track id of the parent track for each feature. This is not the same as the cell id number.	n/a	int64
track_child	The cell id of features belonging to all child cells of a given track id.	n/a	int64
cell_child	The number of features for each cell.	n/a	int64

20.1 Submodules

20.2 `tobac.analysis` module

Provide tools to analyse and visualize the tracked objects. This module provides a set of routines that enables performing analyses and deriving statistics for individual tracks, such as the time series of integrated properties and vertical profiles. It also provides routines to calculate summary statistics of the entire population of tracked features in the field like histograms of areas/volumes or mass and a detailed cell lifetime analysis. These analysis routines are all built in a modular manner. Thus, users can reuse the most basic methods for interacting with the data structure of the package in their own analysis procedures in Python. This includes functions performing simple tasks like looping over all identified objects or trajectories and masking arrays for the analysis of individual features. Plotting routines include both visualizations for individual convective cells and their properties. [\[1\]](#)

References

Notes

`tobac.analysis.area_histogram` (*features, mask, bin_edges=array([0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000, 9500, 10000, 10500, 11000, 11500, 12000, 12500, 13000, 13500, 14000, 14500, 15000, 15500, 16000, 16500, 17000, 17500, 18000, 18500, 19000, 19500, 20000, 20500, 21000, 21500, 22000, 22500, 23000, 23500, 24000, 24500, 25000, 25500, 26000, 26500, 27000, 27500, 28000, 28500, 29000, 29500])*, *density=False, method_area=None, return_values=False, representative_area=False*)

Create an area histogram of the features. If the DataFrame does not contain an area column, the areas are calculated.

Parameters

- **features** (*pandas.DataFrame*) – DataFrame of the features.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int for tracked volumes 0 everywhere else). Needs to contain either `projection_x_coordinate` and `projection_y_coordinate` or latitude and longitude coordinates. The output of a segmentation should be used here.
- **bin_edges** (*int or ndarray, optional*) – If `bin_edges` is an int, it defines the number of equal-width bins in the given range. If `bins` is a ndarray, it defines a monotonically increasing array of bin edges, including the rightmost edge. Default is `np.arange(0, 30000, 500)`.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- **return_values** (*bool, optional*) – Bool determining whether the areas of the features are returned from this function. Default is False.
- **representative_area** (*bool, optional*) – If False, no weights will be associated to the values. If True, the weights for each area will be the areas itself, i.e. each bin count will have the value of the sum of all areas within the edges of the bin. Default is False.

Returns

- **hist** (*ndarray*) – The values of the histogram.
- **bin_edges** (*ndarray*) – The edges of the histogram.
- **bin_centers** (*ndarray*) – The centers of the histogram intervals.
- **areas** (*ndarray, optional*) – A numpy array approximating the area of each feature.

`tobac.analysis.calculate_area(features, mask, method_area=None)`

Calculate the area of the segments for each feature.

Parameters

- **features** (*pandas.DataFrame*) – DataFrame of the features whose area is to be calculated.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int for tracked volumes 0 everywhere else). Needs to contain either `projection_x_coordinate` and `projection_y_coordinate` or latitude and longitude coordinates.
- **method_area** (*{None, 'xy', 'latlon'}, optional*) – Flag determining how the area is calculated. 'xy' uses the areas of the individual pixels, 'latlon' uses the `area_weights` method of `iris.analysis.cartography`, None checks whether the required coordinates are present and starts with 'xy'. Default is None.

Returns features – DataFrame of the features with a new column 'area', containing the calculated areas.

Return type `pandas.DataFrame`

Raises `ValueError` – If neither latitude/longitude nor `projection_x_coordinate/projection_y_coordinate` are present in `mask_coors`.

If latitude/longitude coordinates are 2D.

If latitude/longitude shapes are not supported.

If method is undefined, i.e. method is neither None, 'xy' nor 'latlon'.

`tobac.analysis.calculate_areas_2Dlatlon(_2Dlat_coord, _2Dlon_coord)`

Calculate an array of cell areas when given two 2D arrays of latitude and longitude values

NOTE: This currently assumes that the lat/lon grid is orthogonal, which is not strictly true! It's close enough for most cases, but should be updated in future to use the cross product of the distances to the neighbouring cells. This will require the use of a more advanced calculation. I would advise using pyproj at some point in the future to solve this issue and replace haversine distance.

Parameters

- **_2Dlat_coord** (*AuxCoord*) – Iris auxiliary coordinate containing a 2d grid of latitudes for each point.
- **_2Dlon_coord** (*AuxCoord*) – Iris auxiliary coordinate containing a 2d grid of longitudes for each point.

Returns **area** – A numpy array approximating the area of each cell.

Return type ndarray

`tobac.analysis.calculate_distance(feature_1, feature_2, method_distance=None)`

Compute the distance between two features. It is based on either lat/lon coordinates or x/y coordinates.

Parameters

- **feature_2** (*feature_1*,) – Dataframes containing multiple features or pandas.Series of one feature. Need to contain either projection_x_coordinate and projection_y_coordinate or latitude and longitude coordinates.
- **method_distance** (*{None, 'xy', 'latlon'}*, *optional*) – Method of distance calculation. 'xy' uses the length of the vector between the two features, 'latlon' uses the haversine distance. None checks whether the required coordinates are present and starts with 'xy'. Default is None.

Returns **distance** – Float with the distance between the two features in meters if the input are two pandas.Series containing one feature, pandas.Series of the distances if one of the inputs contains multiple features.

Return type float or pandas.Series

`tobac.analysis.calculate_nearestneighbordistance(features, method_distance=None)`

Calculate the distance between a feature and the nearest other feature in the same timeframe.

Parameters

- **features** (*pandas.DataFrame*) – DataFrame of the features whose nearest neighbor distance is to be calculated. Needs to contain either projection_x_coordinate and projection_y_coordinate or latitude and longitude coordinates.
- **method_distance** (*{None, 'xy', 'latlon'}*, *optional*) – Method of distance calculation. 'xy' uses the length of the vector between the two features, 'latlon' uses the haversine distance. None checks whether the required coordinates are present and starts with 'xy'. Default is None.

Returns **features** – DataFrame of the features with a new column 'min_distance', containing the calculated minimal distance to other features.

Return type pandas.DataFrame

`tobac.analysis.calculate_overlap(track_1, track_2, min_sum_inv_distance=None, min_mean_inv_distance=None)`

Count the number of time frames in which the individual cells of two tracks are present together and calculate their mean and summed inverse distance.

Parameters

- **track_2** (*track_1*,) – The tracks containing the cells to analyze.
- **min_sum_inv_distance** (*float, optional*) – Minimum of the inverse net distance for two cells to be counted as overlapping. Default is None.
- **min_mean_inv_distance** (*float, optional*) – Minimum of the inverse mean distance for two cells to be counted as overlapping. Default is None.

Returns **overlap** – DataFrame containing the columns `cell_1` and `cell_2` with the index of the cells from the tracks, `n_overlap` with the number of frames both cells are present in, `mean_inv_distance` with the mean inverse distance and `sum_inv_distance` with the summed inverse distance of the cells.

Return type `pandas.DataFrame`

`tobac.analysis.calculate_velocity(track, method_distance=None)`

Calculate the velocities of a set of linked features.

Parameters

- **track** (*pandas.DataFrame*) –
Dataframe of linked features, containing the columns ‘cell’, ‘time’ and either ‘projection_x_coordinate’ and ‘projection_y_coordinate’ or ‘latitude’ and ‘longitude’.
- **method_distance** (*{None, ‘xy’, ‘latlon’}, optional*) – Method of distance calculation, used to calculate the velocity. ‘xy’ uses the length of the vector between the two features, ‘latlon’ uses the haversine distance. None checks whether the required coordinates are present and starts with ‘xy’. Default is None.

Returns **track** – DataFrame from the input, with an additional column ‘v’, contain the value of the velocity for every feature at every possible timestep

Return type `pandas.DataFrame`

`tobac.analysis.calculate_velocity_individual(feature_old, feature_new, method_distance=None)`

Calculate the mean velocity of a feature between two timeframes.

Parameters

- **feature_old** (*pandas.Series*) – pandas.Series of a feature at a certain time-frame. Needs to contain a ‘time’ column and either `projection_x_coordinate` and `projection_y_coordinate` or latitude and longitude coordinates.
- **feature_new** (*pandas.Series*) – pandas.Series of the same feature at a later time-frame. Needs to contain a ‘time’ column and either `projection_x_coordinate` and `projection_y_coordinate` or latitude and longitude coordinates.
- **method_distance** (*{None, ‘xy’, ‘latlon’}, optional*) – Method of distance calculation, used to calculate the velocity. ‘xy’ uses the length of the vector between the two features, ‘latlon’ uses the haversine distance. None checks whether the required coordinates are present and starts with ‘xy’. Default is None.

Returns **velocity** – Value of the approximate velocity.

Return type `float`

`tobac.analysis.cell_statistics(input_cubes, track, mask, aggregators, cell, output_path='.', output_name='Profiles', width=10000, z_coord='model_level_number', dimensions=['x', 'y'], **kwargs)`

Parameters

- **input_cubes** (*iris.cube.Cube*) –
- **track** (*dask.dataframe.DataFrame*) –
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tracked volumes 0 everywhere else).
- **list** (*aggregators*) – list of *iris.analysis.Aggregator* instances
- **cell** (*int*) – Integer id of cell to create masked cube for output.
- **output_path** (*str, optional*) – Default is `'./'`.
- **output_name** (*str, optional*) – Default is `'Profiles'`.
- **width** (*int, optional*) – Default is 10000.
- **z_coord** (*str, optional*) – Name of the vertical coordinate in the cube. Default is `'model_level_number'`.
- **dimensions** (*list of str, optional*) – Default is `['x', 'y']`.
- ****kwargs** –

Returns

Return type None

```
tobac.analysis.cell_statistics_all(input_cubes, track, mask, aggregators, output_path='./',
                                  cell_selection=None, output_name='Profiles',
                                  width=10000, z_coord='model_level_number',
                                  dimensions=['x', 'y'], **kwargs)
```

Parameters

- **input_cubes** (*iris.cube.Cube*) –
- **track** (*dask.dataframe.DataFrame*) –
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tracked volumes 0 everywhere else).
- **aggregators** (*list*) – list of *iris.analysis.Aggregator* instances
- **output_path** (*str, optional*) – Default is `'./'`.
- **cell_selection** (*optional*) – Default is None.
- **output_name** (*str, optional*) – Default is `'Profiles'`.
- **width** (*int, optional*) – Default is 10000.
- **z_coord** (*str, optional*) – Name of the vertical coordinate in the cube. Default is `'model_level_number'`.
- **dimensions** (*list of str, optional*) – Default is `['x', 'y']`.
- ****kwargs** –

Returns

Return type None

```
tobac.analysis.cog_cell(cell, Tracks=None, M_total=None, M_liquid=None, M_frozen=None,
                        Mask=None, savedir=None)
```

Parameters

- **cell** (*int*) – Integer id of cell to create masked cube for output.
- **Tracks** (*optional*) – Default is None.
- **M_total** (*subset of cube, optional*) – Default is None.
- **M_liquid** (*subset of cube, optional*) – Default is None.
- **M_frozen** (*subset of cube, optional*) – Default is None.
- **savedir** (*str*) – Default is None.

Returns

Return type None

`tobac.analysis.haversine(lat1, lon1, lat2, lon2)`

Computes the Haversine distance in kilometers.

Calculates the Haversine distance between two points (based on implementation CIS <https://github.com/cedadev/cis>).

Parameters

- **lon1** (*lat1,*) – First point or points as array in degrees.
- **lon2** (*lat2,*) – Second point or points as array in degrees.

Returns `arclen * RADIUS_EARTH` – Array of Distance(s) between the two points(-arrays) in kilometers.

Return type array

`tobac.analysis.histogram_cellwise(Track, variable=None, bin_edges=None, quantity='max', density=False)`

Create a histogram of the maximum, minimum or mean of a variable for the cells (series of features linked together over multiple timesteps) of a track. Essentially a wrapper of the `numpy.histogram()` method.

Parameters

- **Track** (*pandas.DataFrame*) – The track containing the variable to create the histogram from.
- **variable** (*string, optional*) – Column of the DataFrame with the variable on which the histogram is to be based on. Default is None.
- **bin_edges** (*int or ndarray, optional*) – If `bin_edges` is an int, it defines the number of equal-width bins in the given range. If `bins` is a ndarray, it defines a monotonically increasing array of bin edges, including the rightmost edge.
- **quantity** (*{ 'max', 'min', 'mean' }, optional*) – Flag determining whether to use maximum, minimum or mean of a variable from all timeframes the cell covers. Default is 'max'.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.

Returns

- **hist** (*ndarray*) – The values of the histogram
- **bin_edges** (*ndarray*) – The edges of the histogram
- **bin_centers** (*ndarray*) – The centers of the histogram intervals

Raises `ValueError` – If `quantity` is not 'max', 'min' or 'mean'.

`tobac.analysis.histogram_featurewise` (*Track*, *variable=None*, *bin_edges=None*, *density=False*)

Create a histogram of a variable from the features (detected objects at a single time step) of a track. Essentially a wrapper of the `numpy.histogram()` method.

Parameters

- **Track** (*pandas.DataFrame*) – The track containing the variable to create the histogram from.
- **variable** (*string, optional*) – Column of the DataFrame with the variable on which the histogram is to be based on. Default is None.
- **bin_edges** (*int or ndarray, optional*) – If `bin_edges` is an int, it defines the number of equal-width bins in the given range. If bins is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.

Returns

- **hist** (*ndarray*) – The values of the histogram
- **bin_edges** (*ndarray*) – The edges of the histogram
- **bin_centers** (*ndarray*) – The centers of the histogram intervals

`tobac.analysis.lifetime_histogram` (*Track*, *bin_edges=array([0, 20, 40, 60, 80, 100, 120, 140, 160, 180])*, *density=False*, *return_values=False*)

Compute the lifetime histogram of linked features.

Parameters

- **Track** (*pandas.DataFrame*) – Dataframe of linked features, containing the columns 'cell' and 'time_cell'.
- **bin_edges** (*int or ndarray, optional*) – If `bin_edges` is an int, it defines the number of equal-width bins in the given range. If bins is a ndarray, it defines a monotonically increasing array of bin edges, including the rightmost edge. The unit is minutes. Default is `np.arange(0, 200, 20)`.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- **return_values** (*bool, optional*) – Bool determining whether the lifetimes of the features are returned from this function. Default is False.

Returns

- **hist** (*ndarray*) – The values of the histogram.
- **bin_edges** (*ndarray*) – The edges of the histogram.
- **bin_centers** (*ndarray*) – The centers of the histogram intervals.
- **minutes, optional** (*ndarray*) – Numpy.array of the lifetime of each feature in minutes. Returned if `return_values` is True.

```
tobac.analysis.nearestneighbordistance_histogram(features, bin_edges=array([ 0, 500,
1000, 1500, 2000, 2500, 3000,
3500, 4000, 4500, 5000, 5500,
6000, 6500, 7000, 7500, 8000,
8500, 9000, 9500, 10000, 10500,
11000, 11500, 12000, 12500,
13000, 13500, 14000, 14500, 15000,
15500, 16000, 16500, 17000, 17500,
18000, 18500, 19000, 19500, 20000,
20500, 21000, 21500, 22000, 22500,
23000, 23500, 24000, 24500, 25000,
25500, 26000, 26500, 27000, 27500,
28000, 28500, 29000, 29500]), den-
sity=False, method_distance=None, re-
turn_values=False)
```

Create an nearest neighbor distance histogram of the features. If the DataFrame does not contain a 'min_distance' column, the distances are calculated.

features

bin_edges [int or ndarray, optional] If bin_edges is an int, it defines the number of equal-width bins in the given range. If bins is a ndarray, it defines a monotonically increasing array of bin edges, including the rightmost edge. Default is np.arange(0, 30000, 500).

density [bool, optional] If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.

method_distance [{None, 'xy', 'latlon'}, optional] Method of distance calculation. 'xy' uses the length of the vector between the two features, 'latlon' uses the haversine distance. None checks whether the required coordinates are present and starts with 'xy'. Default is None.

return_values [bool, optional] Bool determining whether the nearest neighbor distance of the features are returned from this function. Default is False.

Returns

- **hist** (ndarray) – The values of the histogram.
- **bin_edges** (ndarray) – The edges of the histogram.
- **distances, optional** (ndarray) – A numpy array with the nearest neighbor distances of each feature.

```
tobac.analysis.velocity_histogram(track, bin_edges=array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29]), density=False, method_distance=None, re-
turn_values=False)
```

Create an velocity histogram of the features. If the DataFrame does not contain a velocity column, the velocities are calculated.

Parameters

- **track** (pandas.DataFrame) – DataFrame of the linked features, containing the columns 'cell', 'time' and either 'projection_x_coordinate' and 'projection_y_coordinate' or 'latitude' and 'longitude'.
- **bin_edges** (int or ndarray, optional) – If bin_edges is an int, it defines the number of equal-width bins in the given range. If bins is a ndarray, it defines a monotonically

increasing array of bin edges, including the rightmost edge. Default is `np.arange(0, 30000, 500)`.

- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- **methods_distance** (*{None, 'xy', 'latlon'}, optional*) – Method of distance calculation, used to calculate the velocity. 'xy' uses the length of the vector between the two features, 'latlon' uses the haversine distance. None checks whether the required coordinates are present and starts with 'xy'. Default is None.
- **return_values** (*bool, optional*) – Bool determining whether the velocities of the features are returned from this function. Default is False.

Returns

- **hist** (*ndarray*) – The values of the histogram.
- **bin_edges** (*ndarray*) – The edges of the histogram.
- **velocities, optional** (*ndarray*) – Numpy array with the velocities of each feature.

20.3 tobac.centerofgravity module

Identify center of gravity and mass for analysis.

`tobac.centerofgravity.calculate_cog(tracks, mass, mask)`

Calculate center of gravity and mass for each tracked cell.

Parameters

- **tracks** (*pandas.DataFrame*) – DataFrame containing trajectories of cell centers.
- **mass** (*iris.cube.Cube*) – Cube of quantity (need coordinates 'time', 'geopotential_height', 'projection_x_coordinate' and 'projection_y_coordinate').
- **mask** (*iris.cube.Cube*) – Cube containing mask (int > where belonging to area/volume of feature, 0 else).

Returns tracks_out – Dataframe containing t, x, y, z positions of center of gravity and total mass of each tracked cell at each timestep.

Return type `pandas.DataFrame`

`tobac.centerofgravity.calculate_cog_domain(mass)`

Calculate center of gravity and mass for entire domain.

Parameters mass (*iris.cube.Cube*) – Cube of quantity (need coordinates 'time', 'geopotential_height', 'projection_x_coordinate' and 'projection_y_coordinate').

Returns tracks_out – Dataframe containing t, x, y, z positions of center of gravity and total mass of the entire domain.

Return type `pandas.DataFrame`

`tobac.centerofgravity.calculate_cog_untracked(mass, mask)`

Calculate center of gravity and mass for untracked domain parts.

Parameters

- **mass** (*iris.cube.Cube*) – Cube of quantity (need coordinates 'time', 'geopotential_height', 'projection_x_coordinate' and 'projection_y_coordinate').

- **mask** (*iris.cube.Cube*) – Cube containing mask (int > where belonging to area/volume of feature, 0 else).

Returns **tracks_out** – Dataframe containing t, x, y, z positions of center of gravity and total mass for untracked part of the domain.

Return type pandas.DataFrame

`tobac.centerofgravity.center_of_gravity(cube_in)`

Calculate center of gravity and sum of quantity.

Parameters **cube_in** (*iris.cube.Cube*) – Cube (potentially masked) of quantity (need coordinates ‘geopotential_height’, ‘projection_x_coordinate’ and ‘projection_y_coordinate’).

Returns

- **x** (*float*) – X position of center of gravity.
- **y** (*float*) – Y position of center of gravity.
- **z** (*float*) – Z position of center of gravity.
- **variable_sum** (*float*) – Sum of quantity of over unmasked part of the cube.

20.4 tobac.feature_detection module

Provide feature detection.

This module can work with any two-dimensional field. To identify the features, contiguous regions above or below a threshold are determined and labelled individually. To describe the specific location of the feature at a specific point in time, different spatial properties are used to describe the identified region. [2]

References

`tobac.feature_detection.feature_detection_multithreshold` (*field_in*, *dxy=None*, *threshold=None*, *min_num=0*, *target='maximum'*, *position_threshold='center'*, *sigma_threshold=0.5*, *n_erosion_threshold=0*, *n_min_threshold=0*, *min_distance=0*, *feature_number_start=1*, *PBC_flag='none'*, *vertical_coord=None*, *vertical_axis=None*, *detect_subset=None*, *wavelength_filtering=None*, *dz=None*, *strict_thresholding=False*)

Perform feature detection based on contiguous regions.

The regions are above/below a threshold.

Parameters

- **field_in** (*iris.cube.Cube*) – 2D field to perform the tracking on (needs to have coordinate ‘time’ along one of its dimensions),
- **dx** (*float*) – Grid spacing of the input data (in meter).
- **thresholds** (*list of floats, optional*) – Threshold values used to select target regions to track. Default is None.
- **target** ({‘maximum’, ‘minimum’}, *optional*) – Flag to determine if tracking is targetting minima or maxima in the data. Default is ‘maximum’.
- **position_threshold** ({‘center’, ‘extreme’, ‘weighted_diff’, ‘weighted_abs’}, *optional*) – Flag choosing method used for the position of the tracked feature. Default is ‘center’.
- **coord_interp_kind** (*str, optional*) – The kind of interpolation for coordinates. Default is ‘linear’. For 1d interp, {‘linear’, ‘nearest’, ‘nearest-up’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘previous’, ‘next’}.
For 2d interp, {‘linear’, ‘cubic’, ‘quintic’}.
- **sigma_threshold** (*float, optional*) – Standard deviation for intial filtering step. Default is 0.5.
- **n_erosion_threshold** (*int, optional*) – Number of pixel by which to erode the identified features. Default is 0.
- **n_min_threshold** (*int, optional*) – Minimum number of identified features. Default is 0.
- **min_distance** (*float, optional*) – Minimum distance between detected features (in meter). Default is 0.
- **feature_number_start** (*int, optional*) – Feature id to start with. Default is 1.
- **PBC_flag** (*str(‘none’, ‘hdim_1’, ‘hdim_2’, ‘both’)*) – Sets whether to use periodic boundaries, and if so in which directions. ‘none’ means that we do not have periodic boundaries ‘hdim_1’ means that we are periodic along hdim1 ‘hdim_2’ means that we are periodic along hdim2 ‘both’ means that we are periodic along both horizontal dimensions
- **vertical_coord** (*str*) – Name of the vertical coordinate. If None, tries to auto-detect. It looks for the coordinate or the dimension name corresponding to the string.
- **vertical_axis** (*int or None.*) – The vertical axis number of the data. If None, uses vertical_coord to determine axis. This must be >=0.
- **detect_subset** (*dict-like or None*) – Whether to run feature detection on only a subset of the data. If this is not None, it will subset the grid that we run feature detection on to the range specified for each axis specified. The format of this dict is: {axis-number: (start, end)}, where axis-number is the number of the axis to subset, start is inclusive, and end is exclusive. For example, if your data are oriented as (time, z, y, x) and you want to only detect on values between z levels 10 and 29, you would set: {1: (10, 30)}.
- **wavelength_filtering** (*tuple, optional*) – Minimum and maximum wavelength for horizontal spectral filtering in meter. Default is None.
- **dz** (*float*) – Constant vertical grid spacing (m), optional. If not specified and the input is 3D, this function requires that *altitude* is available in the *features* input. If you specify a value here, this function assumes that it is the constant z spacing between points, even if ‘z_coordinate_name’ is specified.

- **strict_thresholding** (*Bool, optional*) – If True, a feature can only be detected if all previous thresholds have been met. Default is False.

Returns features – Detected features. The structure of this dataframe is explained [here](#)

Return type pandas.DataFrame

```
tobac.feature_detection.feature_detection_multithreshold_timestep(data_i,
                                                                i_time,
                                                                thresh-
                                                                old=None,
                                                                min_num=0,
                                                                tar-
                                                                get='maximum',
                                                                posi-
                                                                tion_threshold='center',
                                                                sigma_threshold=0.5,
                                                                n_erosion_threshold=0,
                                                                n_min_threshold=0,
                                                                min_distance=0,
                                                                fea-
                                                                ture_number_start=1,
                                                                PBC_flag='none',
                                                                verti-
                                                                cal_axis=None,
                                                                dxy=-1,
                                                                wave-
                                                                length_filtering=None,
                                                                strict_thresholding=False)
```

Find features in each timestep.

Based on iteratively finding regions above/below a set of thresholds. Smoothing the input data with the Gaussian filter makes output less sensitive to noisiness of input data.

Parameters

- **data_i** (*iris.cube.Cube*) – 2D field to perform the feature detection (single timestep) on.
- **threshold** (*float, optional*) – Threshold value used to select target regions to track. Default is None.
- **min_num** (*int, optional*) – This parameter is not used in the function. Default is 0.
- **target** (*{'maximum', 'minimum'}, optional*) – Flag to determine if tracking is targetting minima or maxima in the data. Default is 'maximum'.
- **position_threshold** (*{'center', 'extreme', 'weighted_diff', 'weighted_abs'}*, optional) – Flag choosing method used for the position of the tracked feature. Default is 'center'.
- **sigma_threshold** (*float, optional*) – Standard deviation for initial filtering step. Default is 0.5.
- **n_erosion_threshold** (*int, optional*) – Number of pixel by which to erode the identified features. Default is 0.
- **n_min_threshold** (*int, optional*) – Minimum number of identified features. Default is 0.

- **min_distance** (*float, optional*) – Minimum distance between detected features (in meter). Default is 0.
- **feature_number_start** (*int, optional*) – Feature id to start with. Default is 1.
- **PBC_flag** (*str('none', 'hdim_1', 'hdim_2', 'both')*) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions
- **vertical_axis** (*int*) – The vertical axis number of the data.
- **dxy** (*float*) – Grid spacing in meter.
- **wavelength_filtering** (*tuple, optional*) – Minimum and maximum wavelength for spectral filtering in meter. Default is None.
- **strict_thresholding** (*Bool, optional*) – If True, a feature can only be detected if all previous thresholds have been met. Default is False.

Returns **features_threshold** – Detected features for individual timestep.

Return type pandas DataFrame

```
tobac.feature_detection.feature_detection_threshold(data_i, i_time, threshold=None,
                                                  min_num=0, target='maximum',
                                                  position_threshold='center',
                                                  sigma_threshold=0.5,
                                                  n_erosion_threshold=0,
                                                  n_min_threshold=0,
                                                  min_distance=0, idx_start=0,
                                                  PBC_flag='none', vertical_axis=0)
```

Find features based on individual threshold value.

Parameters

- **data_i** (*iris.cube.Cube*) – 2D field to perform the feature detection (single timestep) on.
- **i_time** (*int*) – Number of the current timestep.
- **threshold** (*float, optional*) –
Threshold value used to select target regions to track. Default is None.
- **target** (*{'maximum', 'minimum'}, optional*) – Flag to determine if tracking is targetting minima or maxima in the data. Default is 'maximum'.
- **position_threshold** (*{'center', 'extreme', 'weighted_diff', 'weighted_abs'}*) – optional Flag choosing method used for the position of the tracked feature. Default is 'center'.
- **sigma_threshold** (*float, optional*) – Standard deviation for intial filtering step. Default is 0.5.
- **n_erosion_threshold** (*int, optional*) – Number of pixel by which to erode the identified features. Default is 0.
- **n_min_threshold** (*int, optional*) – Minimum number of identified features. Default is 0.

- **min_distance** (*float, optional*) – Minimum distance between detected features (in meter). Default is 0.
- **idx_start** (*int, optional*) – Feature id to start with. Default is 0.
- **PBC_flag** (*{'none', 'hdim_1', 'hdim_2', 'both'}*) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions
- **vertical_axis** (*int*) – The vertical axis number of the data.

Returns

- **features_threshold** (*pandas DataFrame*) – Detected features for individual threshold.
- **regions** (*dict*) – Dictionary containing the regions above/below threshold used for each feature (feature ids as keys).

```
tobac.feature_detection.feature_position(hdim1_indices, hdim2_indices,
                                         vdim_indices=None, region_small=None,
                                         region_bbox=None, track_data=None, thresh-
                                         old_i=None, position_threshold='center',
                                         target=None, PBC_flag='none', hdim1_min=0,
                                         hdim1_max=0, hdim2_min=0, hdim2_max=0)
```

Determine feature position with regard to the horizontal dimensions in pixels from the identified region above threshold values

Parameters

- **hdim1_indices** (*list*) – indices of pixels in region along first horizontal dimension
- **hdim2_indices** (*list*) – indices of pixels in region along second horizontal dimension
- **vdim_indices** (*list, optional*) – List of indices of feature along optional vdim (typically `z`)
- **region_small** (*2D or 3D array-like*) – A true/false array containing True where the threshold is met and false where the threshold isn't met. This array should be the size specified by region_bbox, and can be a subset of the overall input array (i.e., `track_data`).
- **region_bbox** (*list or tuple with length of 4 or 6*) – The coordinates that region_small occupies within the total track_data array. This is in the order that the coordinates come from the `get_label_props_in_dict` function. For 2D data, this should be: (hdim1 start, hdim 2 start, hdim 1 end, hdim 2 end). For 3D data, this is: (vdim start, hdim1 start, hdim 2 start, vdim end, hdim 1 end, hdim 2 end).
- **track_data** (*2D or 3D array-like*) – 2D or 3D array containing the data
- **threshold_i** (*float*) – The threshold value that we are testing against
- **position_threshold** (*{'center', 'extreme', 'weighted_diff', 'weighted_abs'}*) – How to select the single point position from our data. 'center' picks the geometrical centre of the region, and is typically not recommended. 'extreme' picks the maximum or minimum value inside the region (max/min set by `target`) 'weighted_diff' picks the centre of the region weighted by the distance from the threshold value
'weighted_abs' picks the centre of the region weighted by the absolute values of the field
- **target** (*{'maximum', 'minimum'}*) – Used only when position_threshold is set to 'extreme', this sets whether it is looking for maxima or minima.

- **PBC_flag** (`{'none', 'hdim_1', 'hdim_2', 'both'}`) – Sets whether to use periodic boundaries, and if so in which directions. ‘none’ means that we do not have periodic boundaries ‘hdim_1’ means that we are periodic along hdim1 ‘hdim_2’ means that we are periodic along hdim2 ‘both’ means that we are periodic along both horizontal dimensions
- **hdim1_min** (*int*) – Minimum real array index of the first horizontal dimension (for PBCs)
- **hdim1_max** (*int*) – Maximum real array index of the first horizontal dimension (for PBCs) Note that this coordinate is INCLUSIVE, meaning that this is the maximum coordinate value, and it is not a length.
- **hdim2_min** (*int*) – Minimum real array index of the first horizontal dimension (for PBCs)
- **hdim2_max** (*int*) – Maximum real array index of the first horizontal dimension (for PBCs) Note that this coordinate is INCLUSIVE, meaning that this is the maximum coordinate value, and it is not a length.

Returns If input data is 2D, this will be a 2-element tuple of floats, where the first element is the feature position along the first horizontal dimension and the second element is the feature position along the second horizontal dimension. If input data is 3D, this will be a 3-element tuple of floats, where the first element is the feature position along the vertical dimension and the second two elements are the feature position on the first and second horizontal dimensions. Note for PBCs: this point *can* be >hdim1_max or hdim2_max if the point is between hdim1_max and hdim1_min. For example, if a feature lies exactly between hdim1_max and hdim1_min, the output could be between hdim1_max and hdim1_max+1. While a value between hdim1_min-1 and hdim1_min would also be valid, we choose to overflow on the max side of things.

Return type 2-element or 3-element tuple of floats

```
tobac.feature_detection.filter_min_distance(features, dxy=None,
                                             dz=None, min_distance=None,
                                             x_coordinate_name=None,
                                             y_coordinate_name=None,
                                             z_coordinate_name=None, tar-
                                             get='maximum', PBC_flag='none',
                                             min_h1=0, max_h1=0, min_h2=0,
                                             max_h2=0)
```

Function to remove features that are too close together. If two features are closer than *min_distance*, it keeps the larger feature.

Parameters

- **features** (*pandas DataFrame*) – features
- **dxy** (*float*) – Constant horizontal grid spacing (m).
- **dz** (*float*) – Constant vertical grid spacing (m), optional. If not specified and the input is 3D, this function requires that *z_coordinate_name* is available in the *features* input. If you specify a value here, this function assumes that it is the constant z spacing between points, even if ‘*z_coordinate_name*’ is specified.
- **min_distance** (*float*) – minimum distance between detected features (m)
- **x_coordinate_name** (*str*) – The name of the x coordinate to calculate distance based on in meters. This is typically *projection_x_coordinate*. Currently unused.
- **y_coordinate_name** (*str*) – The name of the y coordinate to calculate distance based on in meters. This is typically *projection_y_coordinate*. Currently unused.

- **z_coordinate_name** (*str or None*) – The name of the z coordinate to calculate distance based on in meters. This is typically *altitude*. If *None*, tries to auto-detect.
- **target** (*{'maximum', 'minimum'}, optional*) – Flag to determine if tracking is targetting minima or maxima in the data. Default is 'maximum'.
- **PBC_flag** (*str('none', 'hdim_1', 'hdim_2', 'both')*) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions
- **min_h1** (*int, optional*) – Minimum real point in hdim_1, for use with periodic boundaries.
- **max_h1** (*int, optional*) – Maximum point in hdim_1, exclusive. max_h1-min_h1 should be the size.
- **min_h2** (*int, optional*) – Minimum real point in hdim_2, for use with periodic boundaries.
- **max_h2** (*int, optional*) – Maximum point in hdim_2, exclusive. max_h2-min_h2 should be the size.

Returns features after filtering

Return type pandas DataFrame

`tobac.feature_detection.remove_parents(features_thresholds, regions_i, regions_old)`

Remove parents of newly detected feature regions.

Remove features where its regions surround newly detected feature regions.

Parameters

- **features_thresholds** (*pandas.DataFrame*) – Dataframe containing detected features.
- **regions_i** (*dict*) – Dictionary containing the regions above/below threshold for the newly detected feature (feature ids as keys).
- **regions_old** (*dict*) – Dictionary containing the regions above/below threshold from previous threshold (feature ids as keys).

Returns **features_thresholds** – Dataframe containing detected features excluding those that are superseded by newly detected ones.

Return type pandas.DataFrame

`tobac.feature_detection.test_overlap(region_inner, region_outer)`

Test for overlap between two regions

Parameters

- **region_1** (*list*) – list of 2-element tuples defining the indices of all cell in the region
- **region_2** (*list*) – list of 2-element tuples defining the indices of all cell in the region

Returns **overlap** – True if there are any shared points between the two regions

Return type bool

20.5 tobac.merge_split module

Tobac merge and split This submodule is a post processing step to address tracked cells which merge/split. The first iteration of this module is to combine the cells which are merging but have received a new cell id (and are considered a new cell) once merged. In general this submodule will label merged/split cells with a TRACK number in addition to its CELL number.

`tobac.merge_split.merge_split_MEST (TRACK, dxy, distance=None, frame_len=5)`

function to postprocess tobac track data for merge/split cells using a minimum euclidian spanning tree

Parameters

- **TRACK** (*pandas.core.frame.DataFrame*) – Pandas dataframe of tobac Track information
- **dxy** (*float, mandatory*) – The x/y grid spacing of the data. Should be in meters.

distance [float, optional] Distance threshold determining how close two features must be in order to consider merge/splitting. Default is 25x the x/y grid spacing of the data, given in dxy. The distance should be in units of meters.

frame_len [float, optional] Threshold for the maximum number of frames that can separate the end of cell and the start of a related cell. Default is five (5) frames.

Returns

d –

xarray dataset of tobac merge/split cells with parent and child designations.

Parent/child variables include:

- **cell_parent_track_id**: The associated track id for each cell. All cells that have merged or split will have the same parent track id. If a cell never merges/splits, only one cell will have a particular track id.
- **feature_parent_cell_id**: The associated parent cell id for each feature. All features in a given cell will have the same cell id. This is the original TRACK cell_id.
- **feature_parent_track_id**: The associated parent track id for each feature. This is not the same as the cell id number.
- **track_child_cell_count**: The total number of features belonging to all child cells of a given track id.
- **cell_child_feature_count**: The total number of features for each cell.

Return type xarray.core.dataset.Dataset

Example usage: `d = merge_split_MEST(Track) ds = tobac.utils.standardize_track_dataset(Track, refl_mask)`
`both_ds = xr.merge([ds, d], compat = 'override')` `both_ds = tobac.utils.compress_all(both_ds)`
`both_ds.to_netcdf(os.path.join(savedir, 'Track_features_merges.nc'))`

20.6 tobac.plotting module

Provide methods for plotting analyzed data.

Plotting routines including both visualizations for the entire dataset including all tracks, and detailed visualizations for individual cells and their properties.

References

`tobac.plotting.animation_mask_field(track, features, field, mask, interval=500, figsize=(10, 10), **kwargs)`

Create animation of field, features and segments of all timeframes.

Parameters

- **track** (*pandas.DataFrame*) – Output of `linking_trackpy`.
- **features** (*pandas.DataFrame*) – Output of the feature detection.
- **field** (*iris.cube.Cube*) – Original input data.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tacked volumes 0 everywhere else), output of the segmentation step.
- **interval** (*int, optional*) – Delay between frames in milliseconds. Default is 500.
- **figsize** (*tuple of float, optional*) – Width, height of the plot in inches. Default is (10, 10).
- ****kwargs** –

Returns **animation** – Created animation as object.

Return type `matplotlib.animation.FuncAnimation`

`tobac.plotting.make_map(axes)`

Configure the parameters of cartopy for plotting.

Parameters **axes** (*cartopy.mpl.geoaxes.GeoAxesSubplot*) – `GeoAxesSubplot` to configure.

Returns **axes** – Cartopy axes to configure

Return type `cartopy.mpl.geoaxes.GeoAxesSubplot`

`tobac.plotting.map_tracks(track, axis_extent=None, figsize=None, axes=None, untracked_cell_value=-1)`

Plot the trajectories of the cells on a map.

Parameters

- **track** (*pandas.DataFrame*) – Dataframe containing the linked features with a column 'cell'.
- **axis_extent** (*matplotlib.axes, optional*) – Array containing the bounds of the longitude and latitude values. The structure is [long_min, long_max, lat_min, lat_max]. Default is None.
- **figsize** (*tuple of floats, optional*) – Width, height of the plot in inches. Default is (10, 10).
- **axes** (*cartopy.mpl.geoaxes.GeoAxesSubplot, optional*) – `GeoAxesSubplot` to use for plotting. Default is None.
- **untracked_cell_value** (*int or np.nan, optional*) – Value of untracked cells in `track['cell']`. Default is -1.

Returns **axes** – Axes with the plotted trajectories.

Return type `cartopy.mpl.geoaxes.GeoAxesSubplot`

Raises `ValueError` – If no axes is passed.

`tobac.plotting.plot_histogram_cellwise` (*track*, *bin_edges*, *variable*, *quantity*, *axes=None*, *density=False*, ***kwargs*)

Plot the histogram of a variable based on the cells.

Parameters

- **track** (*pandas.DataFrame*) – DataFrame of the features containing the variable as column and a column ‘cell’.
- **bin_edges** (*int or ndarray*) – If *bin_edges* is an int, it defines the number of equal-width bins in the given range. If *bins* is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge.
- **variable** (*string*) – Column of the DataFrame with the variable on which the histogram is to be based on. Default is None.
- **quantity** (*{‘max’, ‘min’, ‘mean’}, optional*) – Flag determining whether to use maximum, minimum or mean of a variable from all timeframes the cell covers. Default is ‘max’.
- **axes** (*matplotlib.axes.Axes, optional*) – Matplotlib axes to plot on. Default is None.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- ****kwargs** –

Returns `plot_hist` – List containing the `matplotlib.lines.Line2D` instance of the histogram

Return type list

`tobac.plotting.plot_histogram_featurewise` (*Track*, *bin_edges*, *variable*, *axes=None*, *density=False*, ***kwargs*)

Plot the histogram of a variable based on the features.

Parameters

- **Track** (*pandas.DataFrame*) – DataFrame of the features containing the variable as column.
- **bin_edges** (*int or ndarray*) – If *bin_edges* is an int, it defines the number of equal-width bins in the given range. If *bins* is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge.
- **variable** (*str*) – Column of the DataFrame with the variable on which the histogram is to be based on.
- **axes** (*matplotlib.axes.Axes, optional*) – Matplotlib axes to plot on. Default is None.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- ****kwargs** –

Returns `plot_hist` – List containing the `matplotlib.lines.Line2D` instance of the histogram

Return type list

```
tobac.plotting.plot_lifetime_histogram(track, axes=None, bin_edges=array([ 0, 20, 40,
60, 80, 100, 120, 140, 160, 180]), density=False,
**kwargs)
```

Plot the lifetime histogram of the cells.

Parameters

- **track** (*pandas.DataFrame*) – DataFrame of the features containing the columns ‘cell’ and ‘time_cell’.
- **axes** (*matplotlib.axes.Axes, optional*) – Matplotlib axes to plot on. Default is None.
- **bin_edges** (*int or ndarray, optional*) – If bin_edges is an int, it defines the number of equal-width bins in the given range. If bins is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge. Default is np.arange(0, 200, 20).
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- ****kwargs** –

Returns **plot_hist** – List containing the matplotlib.lines.Line2D instance of the histogram

Return type list

```
tobac.plotting.plot_lifetime_histogram_bar(track, axes=None, bin_edges=array([ 0,
20, 40, 60, 80, 100, 120, 140, 160, 180]),
density=False, width_bar=1, shift=0.5,
**kwargs)
```

Plot the lifetime histogram of the cells as bar plot.

Parameters

- **track** (*pandas.DataFrame*) – DataFrame of the features containing the columns ‘cell’ and ‘time_cell’.
- **axes** (*matplotlib.axes.Axes, optional*) – Matplotlib axes to plot on. Default is None.
- **bin_edges** (*int or ndarray, optional*) – If bin_edges is an int, it defines the number of equal-width bins in the given range. If bins is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge.
- **density** (*bool, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Default is False.
- **width_bar** (*float*) – Width of the bars. Default is 1.
- **shift** (*float*) – Value to shift the bin centers to the right. Default is 0.5.
- ****kwargs** –

Returns **plot_hist** – matplotlib.container.BarContainer instance of the histogram

Return type matplotlib.container.BarContainer

```
tobac.plotting.plot_mask_cell_individual_3Dstatic(cell_i, track, cog, features,
                                                mask_total, field_contour,
                                                field_filled, axes=None,
                                                xlim=None, ylim=None, label_field_contour=None,
                                                cmap_field_contour='Blues',
                                                norm_field_contour=None,
                                                linewidths_contour=0.8,
                                                contour_labels=False,
                                                vmin_field_contour=0,
                                                vmax_field_contour=50, levels_field_contour=None,
                                                nlevels_field_contour=10,
                                                label_field_filled=None,
                                                cmap_field_filled='summer',
                                                norm_field_filled=None,
                                                vmin_field_filled=0,
                                                vmax_field_filled=100,
                                                levels_field_filled=None,
                                                nlevels_field_filled=10, title=None,
                                                feature_number=False, ele=10.0,
                                                azim=210.0)
```

Make plots for cell in fixed frame and with one background field as filling and one background field as contours

Input: Output:

```
tobac.plotting.plot_mask_cell_individual_follow(cell_i, track, cog, features,
                                                mask_total, field_contour, field_filled,
                                                axes=None, width=10000,
                                                label_field_contour=None,
                                                cmap_field_contour='Blues',
                                                norm_field_contour=None,
                                                linewidths_contour=0.8,
                                                contour_labels=False,
                                                vmin_field_contour=0,
                                                vmax_field_contour=50, levels_field_contour=None,
                                                nlevels_field_contour=10,
                                                label_field_filled=None,
                                                cmap_field_filled='summer',
                                                norm_field_filled=None,
                                                vmin_field_filled=0,
                                                vmax_field_filled=100,
                                                levels_field_filled=None,
                                                nlevels_field_filled=10, title=None)
```

Make individual plot for cell centred around cell and with one background field as filling and one background field as contours Input: Output:

```
tobac.plotting.plot_mask_cell_individual_static(cell_i, track, cog, features,
                                              mask_total, field_contour, field_filled,
                                              axes=None, xlim=None, ylim=None,
                                              label_field_contour=None,
                                              cmap_field_contour='Blues',
                                              norm_field_contour=None,
                                              linewidths_contour=0.8,
                                              contour_labels=False,
                                              vmin_field_contour=0,
                                              vmax_field_contour=50,          lev-
                                              els_field_contour=None,
                                              nlevels_field_contour=10,
                                              label_field_filled=None,
                                              cmap_field_filled='summer',
                                              norm_field_filled=None,
                                              vmin_field_filled=0,
                                              vmax_field_filled=100,
                                              levels_field_filled=None,
                                              nlevels_field_filled=10,    title=None,
                                              feature_number=False)
```

Make plots for cell in fixed frame and with one background field as filling and one background field as contours

Input: Output:

```
tobac.plotting.plot_mask_cell_track_2D3Dstatic(cell, track, cog, features, mask_total,
                                              field_contour, field_filled, width=10000,
                                              n_extend=1, name='test', plot-
                                              dir='./, file_format=['png'],
                                              figsize=(3.937007874015748,
                                              3.937007874015748), dpi=300, ele=10,
                                              azim=30, **kwargs)
```

Make plots for all cells with fixed frame including entire development of the cell and with one background field as filling and one background field as contours Input: Output:

```
tobac.plotting.plot_mask_cell_track_3Dstatic(cell, track, cog, features, mask_total,
                                              field_contour, field_filled, width=10000,
                                              n_extend=1, name='test', plot-
                                              dir='./, file_format=['png'],
                                              figsize=(3.937007874015748,
                                              3.937007874015748), dpi=300, **kwargs)
```

Make plots for all cells with fixed frame including entire development of the cell and with one background field as filling and one background field as contours Input: Output:

```
tobac.plotting.plot_mask_cell_track_follow(cell, track, cog, features, mask_total,
                                           field_contour, field_filled, width=10000,
                                           name='test', plotdir='./, file_format=['png'],
                                           figsize=(3.937007874015748,
                                           3.937007874015748), dpi=300, **kwargs)
```

Make plots for all cells centred around cell and with one background field as filling and one background field as contours Input: Output:

```
tobac.plotting.plot_mask_cell_track_static(cell, track, cog, features, mask_total,
                                           field_contour, field_filled, width=10000,
                                           n_extend=1, name='test', plot-
                                           dir='./, file_format=['png'],
                                           figsize=(3.937007874015748,
                                           3.937007874015748), dpi=300, **kwargs)
```

Make plots for all cells with fixed frame including entire development of the cell and with one background field

as filling and one background field as contours Input: Output:

```
tobac.plotting.plot_mask_cell_track_static_timeseries (cell, track, cog, features, mask_total, field_contour, field_filled, track_variable=None, variable=None, variable_ylabel=None, variable_label=[None], variable_legend=False, variable_color=None, width=10000, n_extend=1, name='test', plotdir='./', file_format=['png'], figsize=(7.874015748031496, 3.937007874015748), dpi=300, **kwargs)
```

Make plots for all cells with fixed frame including entire development of the cell and with one background field as filling and one background field as contours Input: Output:

```
tobac.plotting.plot_tracks_mask_field (track, field, mask, features, axes=None, axis_extent=None, plot_outline=True, plot_marker=True, marker_track='x', markersize_track=4, plot_number=True, plot_features=False, marker_feature=None, markersize_feature=None, title=None, title_str=None, vmin=None, vmax=None, n_levels=50, cmap='viridis', extend='neither', orientation_colorbar='horizontal', pad_colorbar=0.05, label_colorbar=None, fraction_colorbar=0.046, rasterized=True, linewidth_contour=1)
```

Plot field, features and segments of a timeframe and on a map projection. It is required to pass vmin, vmax, axes and axis_extent as keyword arguments.

Parameters

- **track** (*pandas.DataFrame*) – One or more timeframes of a dataframe generated by linking_trackpy.
- **field** (*iris.cube.Cube*) – One frame/time step of the original input data.
- **mask** (*iris.cube.Cube*) – One frame/time step of the Cube containing mask (int id for tracked volumes 0 everywhere else), output of the segmentation step.
- **features** (*pandas.DataFrame*) – Output of the feature detection, one or more frames/time steps.
- **axes** (*cartopy.mpl.geoaxes.GeoAxesSubplot*) – GeoAxesSubplot to use for plotting. Default is None.
- **axis_extent** (*ndarray*) – Array containing the bounds of the longitude and latitude values. The structure is [long_min, long_max, lat_min, lat_max]. Default is None.
- **plot_outline** (*bool, optional*) – Boolean defining whether the outlines of the segments are plotted. Default is True.
- **plot_marker** (*bool, optional*) – Boolean defining whether the positions of the features from the track dataframe are plotted. Default is True.

- **marker_track** (*str*, *optional*) – String defining the shape of the marker for the feature positions from the track dataframe. Default is 'x'.
- **markersize_track** (*int*, *optional*) – Int defining the size of the marker for the feature positions from the track dataframe. Default is 4.
- **plot_number** (*bool*, *optional*) – Boolean defining whether the index of the cells is plotted next to the individual feature position. Default is True.
- **plot_features** (*bool*, *optional*) – Boolean defining whether the positions of the features from the features dataframe are plotted. Default is True.
- **marker_feature** (*optional*) – String defining the shape of the marker for the feature positions from the features dataframe. Default is None.
- **markersize_feature** (*optional*) – Int defining the size of the marker for the feature positions from the features dataframe. Default is None.
- **title** (*str*, *optional*) – Flag determining the title of the plot. 'datestr' uses date and time of the field. None sets no title. Default is None.
- **title_str** (*str*, *optional*) – Additional string added to the beginning of the title. Default is None.
- **vmin** (*float*) – Lower bound of the colorbar. Default is None.
- **vmax** (*float*) – Upper bound of the colorbar. Default is None.
- **n_levels** (*int*, *optional*) – Number of levels of the contour plot of the field. Default is 50.
- **cmap** ({'viridis',...}, *optional*) – Colormap of the contour plot of the field. matplotlib.colors. Default is 'viridis'.
- **extend** (*str*, *optional*) – Determines the coloring of values that are outside the levels range. If 'neither', values outside the levels range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the levels range. Values below min(levels) and above max(levels) are mapped to the under/over values of the Colormap. Default is 'neither'.
- **orientation_colorbar** (*str*, *optional*) – Orientation of the colorbar, 'horizontal' or 'vertical' Default is 'horizontal'.
- **pad_colorbar** (*float*, *optional*) – Fraction of original axes between colorbar and new image axes. Default is 0.05.
- **label_colorbar** (*str*, *optional*) – Label of the colorbar. If none, name and unit of the field are used. Default is None.
- **fraction_colorbar** (*float*, *optional*) – Fraction of original axes to use for colorbar. Default is 0.046.
- **rasterized** (*bool*, *optional*) – True enables, False disables rasterization. Default is True.
- **linewidth_contour** (*int*, *optional*) – Linewidth of the contour plot of the segments. Default is 1.

Returns *axes* – Axes with the plot.

Return type cartopy.mpl.geoaxes.GeoAxesSubplot

Raises *ValueError* – If axes are not cartopy.mpl.geoaxes.GeoAxesSubplot.

If mask.ndim is neither 2 nor 3.


```
tobac.plotting.plot_tracks_mask_field_loop(track, field, mask, features,
                                           axes=None, name=None, plot_dir='.',
                                           figsize=(3.937007874015748,
                                           3.937007874015748), dpi=300, mar-
                                           gin_left=0.05, margin_right=0.05, mar-
                                           gin_bottom=0.05, margin_top=0.05,
                                           **kwargs)
```

Plot field, feature positions and segments onto individual maps for all timeframes and save them as pngs.

Parameters

- **track** (*pandas.DataFrame*) – Output of linking_trackpy.
- **field** (*iris.cube.Cube*) – Original input data.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tacked volumes, 0 everywhere else). Output of the segmentation step.
- **features** (*pandas.DataFrame*) – Output of the feature detection.
- **axes** (*cartopy.mpl.geoaxes.GeoAxesSubplot, optional*) – Not used. Default is None.
- **name** (*str, optional*) – Filename without file extension. Same for all pngs. If None, the name of the field is used. Default is None.
- **plot_dir** (*str, optional*) – Path where the plots will be saved. Default is '.'.
- **figsize** (*tuple of floats, optional*) – Width, height of the plot in inches. Default is (10/2.54, 10/2.54).
- **dpi** (*int, optional*) – Plot resolution. Default is 300.
- **margin_left** (*float, optional*) – The position of the left edge of the axes, as a fraction of the figure width. Default is 0.05.
- **margin_right** (*float, optional*) – The position of the right edge of the axes, as a fraction of the figure width. Default is 0.05.
- **margin_bottom** (*float, optional*) – The position of the bottom edge of the axes, as a fraction of the figure width. Default is 0.05.
- **margin_top** (*float, optional*) – The position of the top edge of the axes, as a fraction of the figure width. Default is 0.05.
- ****kwargs** –

Returns

Return type None

20.7 tobac.segmentation module

Provide segmentation techniques.

Segmentation techniques are used to associate areas or volumes to each identified feature. The segmentation is implemented using watershedding techniques from the field of image processing with a fixed threshold value. This value has to be set specifically for every type of input data and application. The segmentation can be performed for both two-dimensional and three-dimensional data. At each timestep, a marker is set at the position (weighted mean center) of each feature identified in the detection step in an array otherwise filled with zeros. In case of the three-dimensional watershedding, all cells in the column above the weighted mean center position of the identified features fulfilling the

threshold condition are set to the respective marker. The algorithm then fills the area (2D) or volume (3D) based on the input field starting from these markers until reaching the threshold. If two or more features are directly connected, the border runs along the watershed line between the two regions. This procedure creates a mask that has the same form as the input data, with the corresponding integer number at all grid points that belong to a feature, else with zero. This mask can be conveniently and efficiently used to select the volume of each feature at a specific time step for further analysis or visualization.

References

`tobac.segmentation.add_markers` (*features*, *marker_arr*, *seed_3D_flag*, *seed_3D_size*=5, *level*=None, *PBC_flag*='none')

Adds markers for watershedding using the *features* dataframe to the *marker_arr*.

Parameters

- **features** (*pandas.DataFrame*) – Features for one point in time to add as markers.
- **marker_arr** (2D or 3D array-like) – Array to add the markers to. Assumes a (z, y, x) configuration.
- **seed_3D_flag** (*str*('column', 'box')) –
Seed 3D field at feature positions with either the full column or a box of user-set size
- **seed_3D_size** (int or tuple (dimensions equal to dimensions of *field*)) – This sets the size of the seed box when *seed_3D_flag* is 'box'. If it's an integer, the seed box is identical in all dimensions. If it's a tuple, it specifies the seed area for each dimension separately. Note: we recommend the use of odd numbers for this. If you give an even number, your seed box will be biased and not centered around the feature. Note: if two seed boxes overlap, the feature that is seeded will be the closer feature.
- **level** (*slice* or *None*) – If *seed_3D_flag* is 'column', the levels at which to seed the cells for the watershedding algorithm. If None, seeds all levels.
- **PBC_flag** ({'none', 'hdim_1', 'hdim_2', 'both'}) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions

Returns The marker array

Return type 2D or 3D array like (same type as *marker_arr*)

`tobac.segmentation.check_add_unseeded_across_bdrys` (*dim_to_run*: *str*, *segmentation_mask*: *numpy.array*, *unseeded_labels*: *numpy.array*, *border_min*: *int*, *border_max*: *int*, *markers_arr*: *numpy.array*, *inplace*: *bool* = True)

Add new markers to unseeded but eligible regions when they are bordering an appropriate boundary.

Parameters

- **dim_to_run** ({'hdim_1', 'hdim_2'}) – what dimension to run
- **segmentation_mask** (*np.array*) – the incoming segmentation mask
- **unseeded_labels** (*np.array*) – The list of labels that are unseeded
- **border_min** (*int*) – minimum real point in the dimension we are running on
- **border_max** (*int*) – maximum real point in the dimension we are running on (inclusive)

- **markers_arr** (*np.array*) – The array of markers to re-run segmentation with
- **inplace** (*bool*) – whether or not to modify markers_arr in place

Returns

Return type markers_arr with new markers added

```
tobac.segmentation.segmentation(features, field, dxy, threshold=0.003, target='maximum',
                                level=None, method='watershed', max_distance=None, vertical_coord=None, PBC_flag='none', seed_3D_flag='column',
                                seed_3D_size=5, segment_number_below_threshold=0, segment_number_unassigned=0)
```

Use watershedding to determine region above a threshold value around initial seeding position for all time steps of the input data. Works both in 2D (based on single seeding point) and 3D and returns a mask with zeros everywhere around the identified regions and the feature id inside the regions.

Calls segmentation_timestep at each individual timestep of the input data.

features [pandas.DataFrame] Output from trackpy/maketrack.

field [iris.cube.Cube] Containing the field to perform the watershedding on.

dxy [float] Grid spacing of the input data.

Output: segmentation_out: iris.cube.Cube

Cloud mask, 0 outside and integer numbers according to track inside the cloud

threshold [float, optional] Threshold for the watershedding field to be used for the mask. Default is 3e-3.

target [{ 'maximum', 'minimum' }, optional] Flag to determine if tracking is targetting minima or maxima in the data. Default is 'maximum'.

level [slice of iris.cube.Cube, optional] Levels at which to seed the cells for the watershedding algorithm. Default is None.

method [{ 'watershed' }, optional] Flag determining the algorithm to use (currently watershedding implemented). 'random_walk' could be uncommented.

max_distance [float, optional] Maximum distance from a marker allowed to be classified as belonging to that cell. Default is None.

vertical_coord [{ 'auto', 'z', 'model_level_number', 'altitude', 'geopotential_height' }, optional]

Name of the vertical coordinate for use in 3D segmentation case

PBC_flag [{ 'none', 'hdim_1', 'hdim_2', 'both' }] Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions

seed_3D_flag: str('column', 'box')

Seed 3D field at feature positions with either the full column (default) or a box of user-set size

seed_3D_size: int or tuple (dimensions equal to dimensions of *field*) This sets the size of the seed box when *seed_3D_flag* is 'box'. If it's an integer, the seed box is identical in all dimensions. If it's a tuple, it specifies the seed area for each dimension separately. Note: we

recommend the use of odd numbers for this. If you give an even number, your seed box will be biased and not centered around the feature.

segment_number_below_threshold: int the marker to use to indicate a segmentation point is below the threshold.

segment_number_unassigned: int the marker to use to indicate a segmentation point is above the threshold but unsegmented.

segmentation_out [iris.cube.Cube] Mask, 0 outside and integer numbers according to track inside the area/volume of the feature.

features_out [pandas.DataFrame] Feature dataframe including the number of cells (2D or 3D) in the segmented area/volume of the feature at the timestep.

ValueError If field_in.ndim is neither 3 nor 4 and 'time' is not included in coords.

```
tobac.segmentation.segmentation_2D(features, field, dxy, threshold=0.003, target='maximum',
                                   level=None, method='watershed', max_distance=None,
                                   PBC_flag='none', seed_3D_flag='column')
```

Wrapper for the segmentation()-function.

```
tobac.segmentation.segmentation_3D(features, field, dxy, threshold=0.003, target='maximum',
                                   level=None, method='watershed', max_distance=None,
                                   PBC_flag='none', seed_3D_flag='column')
```

Wrapper for the segmentation()-function.

```
tobac.segmentation.segmentation_timestep(field_in, features_in, dxy, threshold=0.003, target='maximum', level=None,
                                          method='watershed', max_distance=None,
                                          vertical_coord=None, PBC_flag='none',
                                          seed_3D_flag='column', seed_3D_size=5,
                                          segment_number_below_threshold=0,
                                          segment_number_unassigned=0)
```

Perform watershedding for an individual time step of the data. Works for both 2D and 3D data

Parameters

- **field_in** (*iris.cube.Cube*) – Input field to perform the watershedding on (2D or 3D for one specific point in time).
- **features_in** (*pandas.DataFrame*) – Features for one specific point in time.
- **dxy** (*float*) – Grid spacing of the input data in metres
- **threshold** (*float, optional*) – Threshold for the watershedding field to be used for the mask. Default is 3e-3.
- **target** (*{'maximum', 'minimum'}, optional*) – Flag to determine if tracking is targetting minima or maxima in the data to determine from which direction to approach the threshold value. Default is 'maximum'.
- **level** (*slice of iris.cube.Cube, optional*) – Levels at which to seed the cells for the watershedding algorithm. Default is None.
- **method** (*{'watershed'}, optional*) – Flag determining the algorithm to use (currently watershedding implemented). 'random_walk' could be uncommented.
- **max_distance** (*float, optional*) – Maximum distance from a marker allowed to be classified as belonging to that cell. Default is None.

- **vertical_coord**(*str*, *optional*) – Vertical coordinate in 3D input data. If None, input is checked for one of {'z', 'model_level_number', 'altitude', 'geopotential_height'} as a likely coordinate name
- **PBC_flag**({'none', 'hdim_1', 'hdim_2', 'both'}) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions
- **seed_3D_flag**(*str*('column', 'box')) –
Seed 3D field at feature positions with either the full column (default) or a box of user-set size
- **seed_3D_size**(int or tuple (dimensions equal to dimensions of *field*)) – This sets the size of the seed box when *seed_3D_flag* is 'box'. If it's an integer, the seed box is identical in all dimensions. If it's a tuple, it specifies the seed area for each dimension separately. Note: we recommend the use of odd numbers for this. If you give an even number, your seed box will be biased and not centered around the feature.
- **segment_number_below_threshold**(*int*) – the marker to use to indicate a segmentation point is below the threshold.
- **segment_number_unassigned**(*int*) – the marker to use to indicate a segmentation point is above the threshold but unsegmented.

Returns

- **segmentation_out**(*iris.cube.Cube*) – Mask, 0 outside and integer numbers according to track inside the objects.
- **features_out**(*pandas.DataFrame*) – Feature dataframe including the number of cells (2D or 3D) in the segmented area/volume of the feature at the timestep.

Raises *ValueError* – If target is neither 'maximum' nor 'minimum'.

If **vertical_coord** is not in {'auto', 'z', 'model_level_number', 'altitude', 'geopotential_height'}.

If there is more than one coordinate name.

If the spatial dimension is neither 2 nor 3.

If method is not 'watershed'.

`tobac.segmentation.watershedding_2D`(*track*, *field_in*, ***kwargs*)
Wrapper for the `segmentation()`-function.

`tobac.segmentation.watershedding_3D`(*track*, *field_in*, ***kwargs*)
Wrapper for the `segmentation()`-function.

20.8 tobac.testing module

Containing methods to make simple sample data for testing.

`tobac.testing.generate_grid_coords`(*min_max_coords*, *lengths*)
Generates a grid of coordinates, such as fake lat/lons for testing.

Parameters

- **min_max_coords** (array-like, either length 2, length 4, or length 6.) – The minimum and maximum values in each dimension as: (min_dim1, max_dim1, min_dim2, max_dim2, min_dim3, max_dim3) to use all 3 dimensions. You can omit any dimensions that you aren't using.
- **lengths** (array-like, either length 1, 2, or 3.) – The lengths of values in each dimension. Length must equal 1/2 the length of min_max_coords.

Returns array-like of grid coordinates in the number of dimensions requested and with the number of arrays specified (meshed coordinates)

Return type 1, 2, or 3 array-likes

```
tobac.testing.generate_single_feature(start_h1, start_h2, start_v=None, spd_h1=1,
                                     spd_h2=1, spd_v=1, min_h1=0, max_h1=None,
                                     min_h2=0, max_h2=None, num_frames=1,
                                     dt=datetime.timedelta(seconds=300),
                                     start_date=datetime.datetime(2022, 1, 1, 0, 0),
                                     PBC_flag='none', frame_start=0, feature_num=1,
                                     feature_size=None, threshold_val=None)
```

Function to generate a dummy feature dataframe to test the tracking functionality

Parameters

- **start_h1** (float) – Starting point of the feature in hdim_1 space
- **start_h2** (float) – Starting point of the feature in hdim_2 space
- **start_v** (float, optional) – Starting point of the feature in vdim space (if 3D). For 2D, set to None. Default is None
- **spd_h1** (float, optional) – Speed (per frame) of the feature in hdim_1 Default is 1
- **spd_h2** (float, optional) – Speed (per frame) of the feature in hdim_2 Default is 1
- **spd_v** (float, optional) – Speed (per frame) of the feature in vdim Default is 1
- **min_h1** (int, optional) – Minimum value of hdim_1 allowed. If PBC_flag is not 'none', then this will be used to know when to wrap around periodic boundaries. If PBC_flag is 'none', features will disappear if they are above/below these bounds. Default is 0
- **max_h1** (int, optional) – Similar to min_h1, but the max value of hdim_1 allowed. Default is 1000
- **min_h2** (int, optional) – Similar to min_h1, but the minimum value of hdim_2 allowed. Default is 0
- **max_h2** (int, optional) – Similar to min_h1, but the maximum value of hdim_2 allowed. Default is 1000
- **num_frames** (int, optional) – Number of frames to generate Default is 1
- **dt** (datetime.timedelta, optional) – Difference in time between each frame Default is datetime.timedelta(minutes=5)
- **start_date** (datetime.datetime, optional) – Start datetime Default is datetime.datetime(2022, 1, 1, 0)
- **PBC_flag** (str('none', 'hdim_1', 'hdim_2', 'both')) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions

- **frame_start** (*int*) – Number to start the frame at Default is 1
- **feature_num** (*int, optional*) – What number to start the feature at Default is 1
- **feature_size** (*int or None*) – ‘num’ column in output; feature size If None, doesn’t set this column
- **threshold_val** (*float or None*) – Threshold value of this feature

`tobac.testing.get_single_pbc_coordinate(h1_min, h1_max, h2_min, h2_max, h1_coord, h2_coord, PBC_flag='none')`

Function to get the PBC-adjusted coordinate for an original non-PBC adjusted coordinate.

Parameters

- **h1_min** (*int*) – Minimum point in hdim_1
- **h1_max** (*int*) – Maximum point in hdim_1
- **h2_min** (*int*) – Minimum point in hdim_2
- **h2_max** (*int*) – Maximum point in hdim_2
- **h1_coord** (*int*) – hdim_1 query coordinate
- **h2_coord** (*int*) – hdim_2 query coordinate
- **PBC_flag** (*str('none', 'hdim_1', 'hdim_2', 'both')*) – Sets whether to use periodic boundaries, and if so in which directions. ‘none’ means that we do not have periodic boundaries ‘hdim_1’ means that we are periodic along hdim1 ‘hdim_2’ means that we are periodic along hdim2 ‘both’ means that we are periodic along both horizontal dimensions

Returns Returns a tuple of (hdim_1, hdim_2).

Return type tuple

Raises `ValueError` – Raises a `ValueError` if the point is invalid (e.g., `h1_coord < h1_min` when `PBC_flag = 'none'`)

`tobac.testing.get_start_end_of_feat(center_point, size, axis_min, axis_max, is_pbc=False)`

Gets the start and ending points for a feature given a size and PBC conditions

Parameters

- **center_point** (*float*) – The center point of the feature
- **size** (*float*) – The size of the feature in this dimension
- **axis_min** (*int*) – Minimum point on the axis (usually 0)
- **axis_max** (*int*) – Maximum point on the axis (exclusive). This is 1 after the last real point on the axis, such that `axis_max - axis_min` is the size of the axis
- **is_pbc** (*bool*) – True if we should give wrap around points, false if we shouldn’t.

Returns

- *tuple (start_point, end_point)*
- *Note that if is_pbc is True, start_point can be less than axis_min and*
- *end_point can be greater than or equal to axis_max. This is designed to be used with*
- *`get_pbc_coordinates`*

```
tobac.testing.lists_equal_without_order(a, b)
```

This will make sure the inner list contain the same, but doesn't account for duplicate groups. from: <https://stackoverflow.com/questions/31501909/assert-list-of-list-equality-without-order-in-python/31502000>

```
tobac.testing.make_dataset_from_arr(in_arr, data_type='xarray', time_dim_num=None,
                                   z_dim_num=None, z_dim_name='altitude',
                                   y_dim_num=0, x_dim_num=1)
```

Makes a dataset (xarray or iris) for feature detection/segmentation from a raw numpy/dask/etc. array.

Parameters

- **in_arr** (*array-like*) – The input array to convert to iris/xarray
- **data_type** (*str('xarray' or 'iris'), optional*) – Type of the dataset to return Default is 'xarray'
- **time_dim_num** (*int or None, optional*) – What axis is the time dimension on, None for a single timestep Default is None
- **z_dim_num** (*int or None, optional*) – What axis is the z dimension on, None for a 2D array
- **z_dim_name** (*str*) – What the z dimension name is named
- **y_dim_num** (*int*) – What axis is the y dimension on, typically 0 for a 2D array Default is 0
- **x_dim_num** (*int, optional*) – What axis is the x dimension on, typically 1 for a 2D array Default is 1

Returns

Return type Iris or xarray dataset with everything we need for feature detection/tracking.

```
tobac.testing.make_feature_blob(in_arr, h1_loc, h2_loc, v_loc=None, h1_size=1,
                               h2_size=1, v_size=1, shape='rectangle', amplitude=1,
                               PBC_flag='none')
```

Function to make a defined “blob” in location (zloc, yloc, xloc) with user-specified shape and amplitude. Note that this function will round the size and locations to the nearest point within the array.

Parameters

- **in_arr** (*array-like*) – input array to add the “blob” to
- **h1_loc** (*float*) – Center hdim_1 location of the blob, required
- **h2_loc** (*float*) – Center hdim_2 location of the blob, required
- **v_loc** (*float, optional*) – Center vdim location of the blob, optional. If this is None, we assume that the dataset is 2D. Default is None
- **h1_size** (*float, optional*) – Size of the bubble in array coordinates in hdim_1 Default is 1
- **h2_size** (*float, optional*) – Size of the bubble in array coordinates in hdim_2 Default is 1
- **v_size** (*float, optional*) – Size of the bubble in array coordinates in vdim Default is 1
- **shape** (*str('rectangle'), optional*) – The shape of the blob that is added. For now, this is just rectangle ‘rectangle’ adds a rectangular/rectangular prism bubble with constant amplitude *amplitude*. Default is “rectangle”
- **amplitude** (*float, optional*) – Maximum amplitude of the blob Default is 1

- **PBC_flag** (*str('none', 'hdim_1', 'hdim_2', 'both')*) – Sets whether to use periodic boundaries, and if so in which directions. ‘none’ means that we do not have periodic boundaries ‘hdim_1’ means that we are periodic along hdim1 ‘hdim_2’ means that we are periodic along hdim2 ‘both’ means that we are periodic along both horizontal dimensions

Returns An array with the same type as *in_arr* that has the blob added.

Return type array-like

`tobac.testing.make_sample_data_2D_3blobs (data_type='iris')`

Create a simple dataset to use in tests.

The grid has a grid spacing of 1km in both horizontal directions and 100 grid cells in x direction and 200 in y direction. Time resolution is 1 minute and the total length of the dataset is 100 minutes around a arbitrary date (2000-01-01 12:00). The longitude and latitude coordinates are added as 2D aux coordinates and arbitrary, but in realistic range. The data contains three individual blobs travelling on a linear trajectory through the dataset for part of the time.

Parameters `data_type` (*{'iris', 'xarray'}, optional*) – Choose type of the dataset that will be produced. Default is ‘iris’

Returns `sample_data`

Return type `iris.cube.Cube` or `xarray.DataArray`

`tobac.testing.make_sample_data_2D_3blobs_inv (data_type='iris')`

Create a version of the dataset with switched coordinates.

Create a version of the dataset created in the function `make_sample_cube_2D`, but with switched coordinate order for the horizontal coordinates for tests to ensure that this does not affect the results.

Parameters `data_type` (*{'iris', 'xarray'}, optional*) – Choose type of the dataset that will be produced. Default is ‘iris’

Returns `sample_data`

Return type `iris.cube.Cube` or `xarray.DataArray`

`tobac.testing.make_sample_data_3D_3blobs (data_type='iris', invert_xy=False)`

Create a simple dataset to use in tests.

The grid has a grid spacing of 1km in both horizontal directions and 100 grid cells in x direction and 200 in y direction. Time resolution is 1 minute and the total length of the dataset is 100 minutes around a arbitrary date (2000-01-01 12:00). The longitude and latitude coordinates are added as 2D aux coordinates and arbitrary, but in realistic range. The data contains three individual blobs travelling on a linear trajectory through the dataset for part of the time.

Parameters

- **data_type** (*{'iris', 'xarray'}, optional*) – Choose type of the dataset that will be produced. Default is ‘iris’
- **invert_xy** (*bool, optional*) – Flag to determine whether to switch x and y coordinates Default is False

Returns `sample_data`

Return type `iris.cube.Cube` or `xarray.DataArray`

`tobac.testing.make_simple_sample_data_2D (data_type='iris')`

Create a simple dataset to use in tests.

The grid has a grid spacing of 1km in both horizontal directions and 100 grid cells in x direction and 500 in y direction. Time resolution is 1 minute and the total length of the dataset is 100 minutes around a arbitrary date (2000-01-01 12:00). The longitude and latitude coordinates are added as 2D aux coordinates and arbitrary, but in realistic range. The data contains a single blob travelling on a linear trajectory through the dataset for part of the time.

Parameters `data_type` (`{'iris', 'xarray'}`, *optional*) – Choose type of the dataset that will be produced. Default is 'iris'

Returns `sample_data`

Return type `iris.cube.Cube` or `xarray.DataArray`

`tobac.testing.set_arr_2D_3D` (*in_arr*, *value*, *start_h1*, *end_h1*, *start_h2*, *end_h2*, *start_v=None*, *end_v=None*)

Function to set part of *in_arr* for either 2D or 3D points to *value*. If *start_v* and *end_v* are not none, we assume that the array is 3D. If they are none, we will set the array as if it is a 2D array.

Parameters

- **in_arr** (*array-like*) – Array of values to set
- **value** (*int, float, or array-like of size (end_v-start_v, end_h1-start_h1, end_h2-start_h2)*) – The value to assign to *in_arr*. This will work to assign an array, but the array must have the same dimensions as the size specified in the function.
- **start_h1** (*int*) – Start index to set for *hdim_1*
- **end_h1** (*int*) – End index to set for *hdim_1* (exclusive, so it acts like `[start_h1:end_h1]`)
- **start_h2** (*int*) – Start index to set for *hdim_2*
- **end_h2** (*int*) – End index to set for *hdim_2*
- **start_v** (*int, optional*) – Start index to set for *vdim* Default is None
- **end_v** (*int, optional*) – End index to set for *vdim* Default is None

Returns *in_arr* with the new values set.

Return type *array-like*

20.9 tobac.tracking module

Provide tracking methods.

The individual features and associated area/volumes identified in each timestep have to be linked into trajectories to analyse the time evolution of their properties for a better understanding of the underlying physical processes. The implementations are structured in a way that allows for the future addition of more complex tracking methods recording a more complex network of relationships between features at different points in time.

References

`tobac.tracking.add_cell_time` (*t*: *<sphinx.ext.autodoc.importer.MockObject object at 0x7f560ab68190>*, *cell_number_unassigned: int*)
add cell time as time since the initiation of each cell

Parameters

- **t** (*pandas.DataFrame*) – trajectories with added coordinates

- **cell_number_unassigned**(*int*) – unassigned cell value

Returns *t* – trajectories with added cell time

Return type pandas.DataFrame

`tobac.tracking.build_distance_function(min_h1, max_h1, min_h2, max_h2, PBC_flag)`

Function to build a partial `calc_distance_coords_pbc` function suitable for use with trackpy

Parameters

- **min_h1**(*int*) – Minimum point in hdim_1
- **max_h1**(*int*) – Maximum point in hdim_1
- **min_h2**(*int*) – Minimum point in hdim_2
- **max_h2**(*int*) – Maximum point in hdim_2
- **PBC_flag**(*str*('none', 'hdim_1', 'hdim_2', 'both')) – Sets whether to use periodic boundaries, and if so in which directions. 'none' means that we do not have periodic boundaries 'hdim_1' means that we are periodic along hdim1 'hdim_2' means that we are periodic along hdim2 'both' means that we are periodic along both horizontal dimensions

Returns A version of `calc_distance_coords_pbc` suitable to be called by just `f(coords_1, coords_2)`

Return type function object

`tobac.tracking.fill_gaps(t, order=1, extrapolate=0, frame_max=None, hdim_1_max=None, hdim_2_max=None)`

Add cell time as time since the initiation of each cell.

Parameters

- **t**(*pandas.DataFrame*) – Trajectories from trackpy.
- **order**(*int, optional*) – Order of polynomial used to extrapolate trajectory into gaps and beyond start and end point. Default is 1.
- **extrapolate**(*int, optional*) – Number of timesteps to extrapolate trajectories. Default is 0.
- **frame_max**(*int, optional*) – Size of input data along time axis. Default is None.
- **hdim2_max**(*hdim_1_max,*) – Size of input data along first and second horizontal axis. Default is None.

Returns *t* – Trajectories from trackpy with with filled gaps and potentially extrapolated.

Return type pandas.DataFrame

`tobac.tracking.linking_trackpy(features, field_in, dt, dxy, dz=None, v_max=None, d_max=None, d_min=None, subnetwork_size=None, memory=0, stubs=1, time_cell_min=None, order=1, extrapolate=0, method_linking='random', adaptive_step=None, adaptive_stop=None, cell_number_start=1, cell_number_unassigned=-1, vertical_coord='auto', min_h1=None, max_h1=None, min_h2=None, max_h2=None, PBC_flag='none')`

Perform Linking of features in trajectories.

The linking determines which of the features detected in a specific timestep is most likely identical to an existing feature in the previous timestep. For each existing feature, the movement within a time step is extrapolated based on the velocities in a number previous time steps. The algorithm then breaks the search process down to a few

candidate features by restricting the search to a circular search region centered around the predicted position of the feature in the next time step. For newly initialized trajectories, where no velocity from previous time steps is available, the algorithm resorts to the average velocity of the nearest tracked objects. `v_max` and `d_min` are given as physical quantities and then converted into pixel-based values used in `trackpy`. This allows for tracking that is controlled by physically-based parameters that are independent of the temporal and spatial resolution of the input data. The algorithm creates a continuous track for the feature that is the most probable based on the previous cell path.

Parameters

- **features** (*pandas.DataFrame*) – Detected features to be linked.
- **field_in** (*xarray.DataArray*) – Input field to perform the watershedding on (2D or 3D for one specific point in time).
- **dt** (*float*) – Time resolution of tracked features.
- **dxy** (*float*) – Horizontal grid spacing of the input data.
- **dz** (*float*) – Constant vertical grid spacing (m), optional. If not specified and the input is 3D, this function requires that `vertical_coord` is available in the `features` input. If you specify a value here, this function assumes that it is the constant z spacing between points, even if `vertical_coord` is specified.
- **d_max** (*float, optional*) – Maximum search range Default is None.
- **d_min** (*float, optional*) – Variations in the shape of the regions used to determine the positions of the features can lead to quasi-instantaneous shifts of the position of the feature by one or two grid cells even for a very high temporal resolution of the input data, potentially jeopardising the tracking procedure. To prevent this, `tobac` uses an additional minimum radius of the search range. Default is None.
- **subnetwork_size** (*int, optional*) – Maximum size of subnetwork for linking. This parameter should be adjusted when using adaptive search. Usually a lower value is desired in that case. For a more in depth explanation have look [here](#) If None, 30 is used for regular search and 15 for adaptive search. Default is None.
- **v_max** (*float, optional*) – Speed at which features are allowed to move. Default is None.
- **memory** (*int, optional*) – Number of output timesteps features allowed to vanish for to be still considered tracked. Default is 0. .. warning :: This parameter should be used with caution, as it
 can lead to erroneous trajectory linking, espacially for data with low time resolution.
- **stubs** (*int, optional*) – Minimum number of timesteps of a tracked cell to be reported Default is 1
- **time_cell_min** (*float, optional*) – Minimum length in time of tracked cell to be reported in minutes Default is None.
- **order** (*int, optional*) – Order of polynomial used to extrapolate trajectory into gaps and on start and end point. Default is 1.
- **extrapolate** (*int, optional*) – Number of timesteps to extrapolate trajectories. Default is 0.
- **method_linking** (*{'random', 'predict'}, optional*) – Flag choosing method used for trajectory linking. Default is 'random'.
- **adaptive_step** (*float, optional*) – Reduce search range by multiplying it by this factor. Needs to be used in combination with `adaptive_stop`. Default is None.

- **adaptive_stop** (*float, optional*) – If not None, when encountering an oversize subnet, retry by progressively reducing search_range by multiplying with adaptive_step until the subnet is solvable. If search_range becomes \leq adaptive_stop, give up and raise a SubnetOversizeException. Needs to be used in combination with adaptive_step. Default is None.
- **cell_number_start** (*int, optional*) – Cell number for first tracked cell. Default is 1
- **cell_number_unassigned** (*int*) – Number to set the unassigned/non-tracked cells to. Note that if you set this to *np.nan*, the data type of ‘cell’ will change to float. Default is -1
- **vertical_coord** (*str*) – Name of the vertical coordinate. The vertical coordinate used must be meters. If None, tries to auto-detect. It looks for the coordinate or the dimension name corresponding to the string. To use *dz*, set this to *None*.
- **min_h1** (*int*) – Minimum hdim_1 value, required when PBC_flag is ‘hdim_1’ or ‘both’
- **max_h1** (*int*) – Maximum hdim_1 value, required when PBC_flag is ‘hdim_1’ or ‘both’
- **min_h2** (*int*) – Minimum hdim_2 value, required when PBC_flag is ‘hdim_2’ or ‘both’
- **max_h2** (*int*) – Maximum hdim_2 value, required when PBC_flag is ‘hdim_2’ or ‘both’
- **PBC_flag** (*str('none', 'hdim_1', 'hdim_2', 'both')*) – Sets whether to use periodic boundaries, and if so in which directions. ‘none’ means that we do not have periodic boundaries ‘hdim_1’ means that we are periodic along hdim1 ‘hdim_2’ means that we are periodic along hdim2 ‘both’ means that we are periodic along both horizontal dimensions

Returns **trajectories_final** – Dataframe of the linked features, containing the variable ‘cell’, with integers indicating the affiliation of a feature to a specific track, and the variable ‘time_cell’ with the time the cell has already existed.

Return type pandas.DataFrame

Raises ValueError – If method_linking is neither ‘random’ nor ‘predict’.

tobac.tracking.**remap_particle_to_cell_nv** (*particle_cell_map, input_particle*)

Remaps the particles to new cells given an input map and the current particle. Helper function that is designed to be vectorized with np.vectorize

Parameters

- **particle_cell_map** (*dict-like*) – The dictionary mapping particle number to cell number
- **input_particle** (*key for particle_cell_map*) – The particle number to remap

20.10 tobac.utils modules

20.11 tobac.utils.general modules

General tobac utilities

tobac.utils.general.**add_coordinates** (*t, variable_cube*)

Add coordinates from the input cube of the feature detection to the trajectories/features.

Parameters

- **t** (*pandas.DataFrame*) – Trajectories/features from feature detection or linking step.
- **variable_cube** (*iris.cube.Cube*) – Input data used for the tracking with coordinate information to transfer to the resulting DataFrame. Needs to contain the coordinate ‘time’.

Returns **t** – Trajectories with added coordinates.

Return type *pandas.DataFrame*

```
tobac.utils.general.add_coordinates_3D(t, variable_cube, vertical_axis=None, assume_coords_fixed_in_time=True)
```

Function adding coordinates from the tracking cube to the trajectories for the 3D case: time, longitude&latitude, x&y dimensions, and altitude

Parameters

- **t** (*pandas.DataFrame*) – trajectories/features
- **variable_cube** (*iris.cube.Cube*) – Cube (usually the one you are tracking on) at least containing the dimension of ‘time’. Typically, ‘longitude’, ‘latitude’, ‘x_projection_coordinate’, ‘y_projection_coordinate’, and ‘altitude’ (if 3D) are the coordinates that we expect, although this function will happily interpolate along any dimension coordinates you give.
- **vertical_coord** (*str or int*) – Name or axis number of the vertical coordinate. If None, tries to auto-detect. If it is a string, it looks for the coordinate or the dimension name corresponding to the string. If it is an int, it assumes that it is the vertical axis. Note that if you only have a 2D or 3D coordinate for altitude, you must pass in an int.
- **vertical_axis** (*int or None*) – Axis number of the vertical.
- **assume_coords_fixed_in_time** (*bool*) – If true, it assumes that the coordinates are fixed in time, even if the coordinates say they vary in time. This is, by default, True, to preserve legacy functionality. If False, it assumes that if a coordinate says it varies in time, it takes the coordinate at its word.

Returns trajectories with added coordinates

Return type *pandas.DataFrame*

```
tobac.utils.general.combine_feature_dataframes(feature_df_list, renumber_features=True, old_feature_column_name=None, sort_features_by=None)
```

Function to combine a list of tobac feature detection dataframes into one combined dataframe that can be used for tracking or segmentation. :param feature_df_list: A list of dataframes (generated, for example, by running feature detection on multiple nodes).

Parameters

- **renumber_features** (*bool, optional (default: True)*) – If true, features are renumber with contiguous integers. If false, the old feature numbers will be retained, but an exception will be raised if there are any non-unique feature numbers. If you have non-unique feature numbers and want to preserve them, use the `old_feature_column_name` to save the old feature numbers to under a different column name.

- **old_feature_column_name** (*str or None, optional (default: None)*) – The column name to preserve old feature numbers in. If None, these old numbers will be deleted. Users may want to enable this feature if they have run segmentation with the separate dataframes and therefore old feature numbers.
- **sort_features_by** (*list, str or None, optional (default: None)*) – The sorting order to pass to `Dataframe.sort_values` for the merged dataframe. If None, will default to ["frame", "idx"] if `renumber_features` is True, or "feature" if `renumber_features` is False.

Returns One combined DataFrame.

Return type `pd.DataFrame`

`tobac.utils.general.combine_tobac_feats(list_of_feats, preserve_old_feat_nums=None)`

WARNING: This function has been deprecated and will be removed in a future release, please use 'combine_feature_dataframes' instead

Function to combine a list of tobac feature detection dataframes into one combined dataframe that can be used for tracking or segmentation. :param list_of_feats: A list of dataframes (generated, for example, by running feature detection on multiple nodes).

Parameters **preserve_old_feat_nums** (*str or None*) – The column name to preserve old feature numbers in. If None, these old numbers will be deleted. Users may want to enable this feature if they have run segmentation with the separate dataframes and therefore old feature numbers.

Returns One combined DataFrame.

Return type `pd.DataFrame`

`tobac.utils.general.get_bounding_box(x, buffer=1)`

Finds the bounding box of a ndarray, i.e. the smallest bounding rectangle for nonzero values as explained here: <https://stackoverflow.com/questions/31400769/bounding-box-of-numpy-array> :param x: Array for which the bounding box is to be determined. :type x: `numpy.ndarray` :param buffer: Number to set a buffer between the nonzero values and

the edges of the box. Default is 1.

Returns **bbox** – Dimensionwise list of the indices representing the edges of the bounding box.

Return type `list`

`tobac.utils.general.get_spacings(field_in, grid_spacing=None, time_spacing=None)`

Determine spatial and temporal grid spacing of the input data.

Parameters

- **field_in** (*iris.cube.Cube*) – Input field where to get spacings.
- **grid_spacing** (*float, optional*) – Manually sets the grid spacing if specified. Default is None.
- **time_spacing** (*float, optional*) – Manually sets the time spacing if specified. Default is None.

Returns

- **dxy** (*float*) – Grid spacing in metres.
- **dt** (*float*) – Time resolution in seconds.

Raises `ValueError` – If `input_cube` does not contain `projection_x_coord` and `projection_y_coord` or keyword argument `grid_spacing`.

`tobac.utils.general.spectral_filtering`(*dxy*, *field_in*, *lambda_min*, *lambda_max*, *return_transfer_function=False*)

This function creates and applies a 2D transfer function that can be used as a bandpass filter to remove certain wavelengths of an atmospheric input field (e.g. vorticity, IVT, etc).

dxy [float] Grid spacing in m.

field_in: `numpy.array` 2D field with input data.

lambda_min: `float` Minimum wavelength in m.

lambda_max: `float` Maximum wavelength in m.

return_transfer_function: `boolean`, optional default: `False`. If set to `True`, then the 2D transfer function and the corresponding wavelengths are returned.

filtered_field: `numpy.array` Spectrally filtered 2D field of data (with same shape as input data).

transfer_function: `tuple` Two 2D fields, where the first one corresponds to the wavelengths in the spectral space of the domain and the second one to the 2D transfer function of the bandpass filter. Only returned, if `return_transfer_function` is `True`.

20.12 tobac.utils.mask modules

Provide essential methods for masking

`tobac.utils.mask.column_mask_from2D`(*mask_2D*, *cube*, *z_coord='model_level_number'*)

Turn 2D watershedding mask into a 3D mask of selected columns.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube.
- **mask_2D** (*iris.cube.Cube*) – 2D cube containing mask (int id for tacked volumes 0 everywhere else).
- **z_coord** (*str*) – Name of the vertical coordinate in the cube.

Returns **mask_2D** – 3D cube containing columns of 2D mask (int id for tracked volumes, 0 everywhere else).

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_all_surface`(*mask*, *masked=False*, *z_coord='model_level_number'*)

Create surface projection of 3d-mask for all features by collapsing one coordinate.

Parameters

- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tacked volumes 0 everywhere else).
- **masked** (*bool*, *optional*) – Bool determining whether to mask the mask for the cell where it is 0. Default is `False`
- **z_coord** (*str*, *optional*) – Name of the coordinate to collapse. Default is `'model_level_number'`.

Returns **mask_i_surface** – Collapsed Masked cube for the features with the maximum value along the collapsed coordinate.

Return type `iris.cube.Cube` (2D)

`tobac.utils.mask.mask_cell(mask, cell, track, masked=False)`

Create mask for specific cell.

Parameters

- **mask** (`iris.cube.Cube`) – Cube containing mask (int id for tracked volumes 0 everywhere else).
- **cell** (`int`) – Integer id of cell to create masked cube for.
- **track** (`pandas.DataFrame`) – Output of the linking.
- **masked** (`bool, optional`) – Bool determining whether to mask the mask for the cell where it is 0. Default is False.

Returns `mask_i` – Mask for a specific cell.

Return type `numpy.ndarray`

`tobac.utils.mask.mask_cell_surface(mask, cell, track, masked=False, z_coord='model_level_number')`

Create surface projection of 3d-mask for individual cell by collapsing one coordinate.

Parameters

- **mask** (`iris.cube.Cube`) – Cube containing mask (int id for tracked volumes, 0 everywhere else).
- **cell** (`int`) – Integer id of cell to create masked cube for.
- **track** (`pandas.DataFrame`) – Output of the linking.
- **masked** (`bool, optional`) – Bool determining whether to mask the mask for the cell where it is 0. Default is False.
- **z_coord** (`str, optional`) – Name of the coordinate to collapse. Default is 'model_level_number'.

Returns `mask_i_surface` – Collapsed Masked cube for the cell with the maximum value along the collapsed coordinate.

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_cube(cube_in, mask)`

Mask cube where mask is not zero.

Parameters

- **cube_in** (`iris.cube.Cube`) – Unmasked data cube.
- **mask** (`iris.cube.Cube`) – Mask to use for masking, >0 where cube is supposed to be masked.

Returns `variable_cube_out` – Masked cube.

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_cube_all(variable_cube, mask)`

Mask cube (`iris.cube`) for tracked volume.

Parameters

- **variable_cube** (`iris.cube.Cube`) – Unmasked data cube.
- **mask** (`iris.cube.Cube`) – Cube containing mask (int id for tracked volumes 0 everywhere else).

Returns `variable_cube_out` – Masked cube for untracked volume.

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_cube_cell` (*variable_cube*, *mask*, *cell*, *track*)

Mask cube for tracked volume of an individual cell.

Parameters

- **variable_cube** (*iris.cube.Cube*) – Unmasked data cube.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tracked volumes, 0 everywhere else).
- **cell** (*int*) – Integer id of cell to create masked cube for.
- **track** (*pandas.DataFrame*) – Output of the linking.

Returns `variable_cube_out` – Masked cube with data for respective cell.

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_cube_features` (*variable_cube*, *mask*, *feature_ids*)

Mask cube for tracked volume of one or more specific features.

Parameters

- **variable_cube** (*iris.cube.Cube*) – Unmasked data cube.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tracked volumes, 0 everywhere else).
- **feature_ids** (*int or list of ints*) – Integer ids of features to create masked cube for.

Returns `variable_cube_out` – Masked cube with data for respective features.

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_cube_untracked` (*variable_cube*, *mask*)

Mask cube (`iris.cube`) for untracked volume.

Parameters

- **variable_cube** (*iris.cube.Cube*) – Unmasked data cube.
- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tracked volumes 0 everywhere else).

Returns `variable_cube_out` – Masked cube for untracked volume.

Return type `iris.cube.Cube`

`tobac.utils.mask.mask_features` (*mask*, *feature_ids*, *masked=False*)

Create mask for specific features.

Parameters

- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tracked volumes 0 everywhere else).
- **feature_ids** (*int or list of ints*) – Integer ids of the features to create the masked cube for.
- **masked** (*bool, optional*) – Bool determining whether to mask the mask for the cell where it is 0. Default is False.

Returns `mask_i` – Masked cube for specific features.

Return type numpy.ndarray

`tobac.utils.mask.mask_features_surface(mask, feature_ids, masked=False, z_coord='model_level_number')`

Create surface projection of 3d-mask for specific features by collapsing one coordinate.

Parameters

- **mask** (*iris.cube.Cube*) – Cube containing mask (int id for tacked volumes 0 everywhere else).
- **feature_ids** (*int or list of ints*) – Integer ids of the features to create the masked cube for.
- **masked** (*bool, optional*) – Bool determining whether to mask the mask for the cell where it is 0. Default is False
- **z_coord** (*str, optional*) – Name of the coordinate to collapse. Default is 'model_level_number'.

Returns **mask_i_surface** – Collapsed Masked cube for the features with the maximum value along the collapsed coordinate.

Return type *iris.cube.Cube*

20.13 tobac.wrapper module

`tobac.wrapper.maketrack(field_in, grid_spacing=None, time_spacing=None, target='maximum', v_max=None, d_max=None, memory=0, stubs=5, order=1, extrapolate=0, method_detection='threshold', position_threshold='center', sigma_threshold=0.5, n_erosion_threshold=0, threshold=1, min_num=0, min_distance=0, method_linking='random', cell_number_start=1, subnetwork_size=None, adaptive_stop=None, adaptive_step=None, return_intermediate=False)`

`tobac.wrapper.tracking_wrapper(field_in_features, field_in_segmentation, time_spacing=None, grid_spacing=None, parameters_features=None, parameters_tracking=None, parameters_segmentation=None)`

20.14 Module contents

t

- `tobac`, [119](#)
- `tobac.analysis`, [77](#)
- `tobac.centerofgravity`, [85](#)
- `tobac.feature_detection`, [86](#)
- `tobac.merge_split`, [93](#)
- `tobac.plotting`, [93](#)
- `tobac.segmentation`, [101](#)
- `tobac.testing`, [105](#)
- `tobac.tracking`, [110](#)
- `tobac.utils.general`, [113](#)
- `tobac.utils.mask`, [116](#)
- `tobac.wrapper`, [119](#)

A

[add_cell_time\(\)](#) (in module *tobac.tracking*), 110
[add_coordinates\(\)](#) (in module *tobac.utils.general*), 113
[add_coordinates_3D\(\)](#) (in module *tobac.utils.general*), 114
[add_markers\(\)](#) (in module *tobac.segmentation*), 102
[animation_mask_field\(\)](#) (in module *tobac.plotting*), 94
[area_histogram\(\)](#) (in module *tobac.analysis*), 77

B

[build_distance_function\(\)](#) (in module *tobac.tracking*), 111

C

[calculate_area\(\)](#) (in module *tobac.analysis*), 78
[calculate_areas_2Dlatlon\(\)](#) (in module *tobac.analysis*), 78
[calculate_cog\(\)](#) (in module *tobac.centerofgravity*), 85
[calculate_cog_domain\(\)](#) (in module *tobac.centerofgravity*), 85
[calculate_cog_untracked\(\)](#) (in module *tobac.centerofgravity*), 85
[calculate_distance\(\)](#) (in module *tobac.analysis*), 79
[calculate_nearestneighbordistance\(\)](#) (in module *tobac.analysis*), 79
[calculate_overlap\(\)](#) (in module *tobac.analysis*), 79
[calculate_velocity\(\)](#) (in module *tobac.analysis*), 80
[calculate_velocity_individual\(\)](#) (in module *tobac.analysis*), 80
[cell_statistics\(\)](#) (in module *tobac.analysis*), 80
[cell_statistics_all\(\)](#) (in module *tobac.analysis*), 81

[center_of_gravity\(\)](#) (in module *tobac.centerofgravity*), 86
[check_add_unseeded_across_bdrys\(\)](#) (in module *tobac.segmentation*), 102
[cog_cell\(\)](#) (in module *tobac.analysis*), 81
[column_mask_from2D\(\)](#) (in module *tobac.utils.mask*), 116
[combine_feature_dataframes\(\)](#) (in module *tobac.utils.general*), 114
[combine_tobac_feats\(\)](#) (in module *tobac.utils.general*), 115

F

[feature_detection_multithreshold\(\)](#) (in module *tobac.feature_detection*), 86
[feature_detection_multithreshold_timestep\(\)](#) (in module *tobac.feature_detection*), 88
[feature_detection_threshold\(\)](#) (in module *tobac.feature_detection*), 89
[feature_position\(\)](#) (in module *tobac.feature_detection*), 90
[fill_gaps\(\)](#) (in module *tobac.tracking*), 111
[filter_min_distance\(\)](#) (in module *tobac.feature_detection*), 91

G

[generate_grid_coords\(\)](#) (in module *tobac.testing*), 105
[generate_single_feature\(\)](#) (in module *tobac.testing*), 106
[get_bounding_box\(\)](#) (in module *tobac.utils.general*), 115
[get_single_pbc_coordinate\(\)](#) (in module *tobac.testing*), 107
[get_spacings\(\)](#) (in module *tobac.utils.general*), 115
[get_start_end_of_feat\(\)](#) (in module *tobac.testing*), 107

H

[haversine\(\)](#) (in module *tobac.analysis*), 82

histogram_cellwise() (in module *tobac.analysis*), 82

histogram_featurewise() (in module *tobac.analysis*), 82

L

lifetime_histogram() (in module *tobac.analysis*), 83

linking_trackpy() (in module *tobac.tracking*), 111

lists_equal_without_order() (in module *tobac.testing*), 107

M

make_dataset_from_arr() (in module *tobac.testing*), 108

make_feature_blob() (in module *tobac.testing*), 108

make_map() (in module *tobac.plotting*), 94

make_sample_data_2D_3blobs() (in module *tobac.testing*), 109

make_sample_data_2D_3blobs_inv() (in module *tobac.testing*), 109

make_sample_data_3D_3blobs() (in module *tobac.testing*), 109

make_simple_sample_data_2D() (in module *tobac.testing*), 109

maketrack() (in module *tobac.wrapper*), 119

map_tracks() (in module *tobac.plotting*), 94

mask_all_surface() (in module *tobac.utils.mask*), 116

mask_cell() (in module *tobac.utils.mask*), 117

mask_cell_surface() (in module *tobac.utils.mask*), 117

mask_cube() (in module *tobac.utils.mask*), 117

mask_cube_all() (in module *tobac.utils.mask*), 117

mask_cube_cell() (in module *tobac.utils.mask*), 118

mask_cube_features() (in module *tobac.utils.mask*), 118

mask_cube_untracked() (in module *tobac.utils.mask*), 118

mask_features() (in module *tobac.utils.mask*), 118

mask_features_surface() (in module *tobac.utils.mask*), 119

merge_split_MEST() (in module *tobac.merge_split*), 93

N

nearestneighbordistance_histogram() (in module *tobac.analysis*), 83

P

plot_histogram_cellwise() (in module *tobac.plotting*), 95

plot_histogram_featurewise() (in module *tobac.plotting*), 95

plot_lifetime_histogram() (in module *tobac.plotting*), 95

plot_lifetime_histogram_bar() (in module *tobac.plotting*), 96

plot_mask_cell_individual_3Dstatic() (in module *tobac.plotting*), 96

plot_mask_cell_individual_follow() (in module *tobac.plotting*), 97

plot_mask_cell_individual_static() (in module *tobac.plotting*), 97

plot_mask_cell_track_2D3Dstatic() (in module *tobac.plotting*), 98

plot_mask_cell_track_3Dstatic() (in module *tobac.plotting*), 98

plot_mask_cell_track_follow() (in module *tobac.plotting*), 98

plot_mask_cell_track_static() (in module *tobac.plotting*), 98

plot_mask_cell_track_static_timeseries() (in module *tobac.plotting*), 99

plot_tracks_mask_field() (in module *tobac.plotting*), 99

plot_tracks_mask_field_loop() (in module *tobac.plotting*), 100

R

remap_particle_to_cell_nv() (in module *tobac.tracking*), 113

remove_parents() (in module *tobac.feature_detection*), 92

S

segmentation() (in module *tobac.segmentation*), 103

segmentation_2D() (in module *tobac.segmentation*), 104

segmentation_3D() (in module *tobac.segmentation*), 104

segmentation_timestep() (in module *tobac.segmentation*), 104

set_arr_2D_3D() (in module *tobac.testing*), 110

spectral_filtering() (in module *tobac.utils.general*), 116

T

test_overlap() (in module *tobac.feature_detection*), 92

tobac (module), 119

tobac.analysis (module), 77

tobac.centerofgravity (module), 85

tobac.feature_detection (module), 86

tobac.merge_split (module), 93

tobac.plotting (*module*), 93
tobac.segmentation (*module*), 101
tobac.testing (*module*), 105
tobac.tracking (*module*), 110
tobac.utils.general (*module*), 113
tobac.utils.mask (*module*), 116
tobac.wrapper (*module*), 119
tracking_wrapper() (*in module tobac.wrapper*),
119

V

velocity_histogram() (*in module tobac.analysis*),
84

W

watershedding_2D() (*in module tobac.segmentation*), 105
watershedding_3D() (*in module tobac.segmentation*), 105