tobac

unknown

Mar 23, 2024

BASIC INFORMATION

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.

Version 1.5 of tobac and the major enhancements that came with that version are currently under review in Geoscientific Model Development:

Sokolowsky, G. A., Freeman, S. W., Jones, W. K., Kukulies, J., Senf, F., Marinescu, P. J., Heikenfeld, M., Brunner, K. N., Bruning, E. C., Collis, S. M., Jackson, R. C., Leung, G. R., Pfeifer, N., Raut, B. A., Saleeby, S. M., Stier, P., and van den Heever, S. C.: tobac v1.5: Introducing Fast 3D Tracking, Splits and Mergers, and Other Enhancements for Identifying and Analysing Meteorological Phenomena, EGUsphere [preprint], https://doi.org/10.5194/egusphere-2023-1722, 2023.

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

CHAPTER

ONE

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/
```

CHAPTER

TWO

DATA INPUT

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 DataArays 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 sorftware 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 a 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

THREE

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

FOUR

PLOTTING

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

CHAPTER

FIVE

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.

CHAPTER

SIX

EXAMPLE GALLERY

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

6.1 Idealized Case 1: Tracking of a Test Blob in 2D

This tutorial shows the most important steps of tracking with `tobac` using an idealized case:

1. *Input Data*
2. *Feature Detection*
3. *Tracking / Trajectory Linking*
4. *Segmentation*
5. *Statistical Analysis*

6.1.1 Import Libraries

We start by importing tobac:

[1]:
```
import tobac
print('using tobac version', str(tobac.__version__))

# we add testing here to create test dataset (typically not needed in standard␣
↪applications)
import tobac.testing
```

```
using tobac version 1.5.3
```

We will also need matplotlib in inline-mode for plotting and numpy:

[2]:
```
import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np
```

For a better readability of the graphs:

[3]:
```
import seaborn as sns

sns.set_context("talk")
```

Tobac works with a Python package called `xarray`, which introduces `DataArrays`. In a nutshell these are `numpy`-arrays with labels. For a more extensive description have a look at the xarray Documentation.

6.1.2 1. Input Data

There are several utilities implemented in tobac to create simple examples of such arrays. In this tutorial we will use the function `make_simple_sample_data_2D()` to create a moving test blob in 2D:

[4]:
```
test_data = tobac.testing.make_simple_sample_data_2D(data_type="xarray")
test_data
```

[4]:
```
<xarray.DataArray 'w' (time: 100, y: 50, x: 100)> Size: 4MB
[500000 values with dtype=float64]
Coordinates:
  * time       (time) datetime64[ns] 800B 2000-01-01T12:00:00 ... 2000-01-01T...
  * y          (y) float64 400B 0.0 1e+03 2e+03 ... 4.7e+04 4.8e+04 4.9e+04
  * x          (x) float64 800B 0.0 1e+03 2e+03 ... 9.7e+04 9.8e+04 9.9e+04
    latitude   (y, x) float64 40kB ...
    longitude  (y, x) float64 40kB ...
Attributes:
    units:    m s-1
```

As you can see our generated data describes a field called 'w' with the unit m/s at 100, 50 and 100 datapoints of time, x and y. Additionally, the data contains the latitude and longitude coordinates of the field values. To access the values of 'w' in the first timeframe, we can use

[5]:
```
test_data.data[0]
```

[5]:
```
array([[3.67879441e+00, 4.04541885e+00, 4.40431655e+00, ...,
        2.22319774e-16, 9.26766698e-17, 3.82489752e-17],
       [4.04541885e+00, 4.44858066e+00, 4.84324569e+00, ...,
        2.44475908e-16, 1.01912721e-16, 4.20608242e-17],
       [4.40431655e+00, 4.84324569e+00, 5.27292424e+00, ...,
        2.66165093e-16, 1.10954118e-16, 4.57923372e-17],
       ...,
       [6.45813368e-03, 7.10174389e-03, 7.73178977e-03, ...,
        3.90282972e-19, 1.62694148e-19, 6.71461808e-20],
       [4.43860604e-03, 4.88095244e-03, 5.31397622e-03, ...,
        2.68237303e-19, 1.11817944e-19, 4.61488502e-20],
       [3.02025230e-03, 3.32124719e-03, 3.61589850e-03, ...,
        1.82522243e-19, 7.60865909e-20, 3.14020145e-20]])
```

which is then just an array of numbers of the described shape.

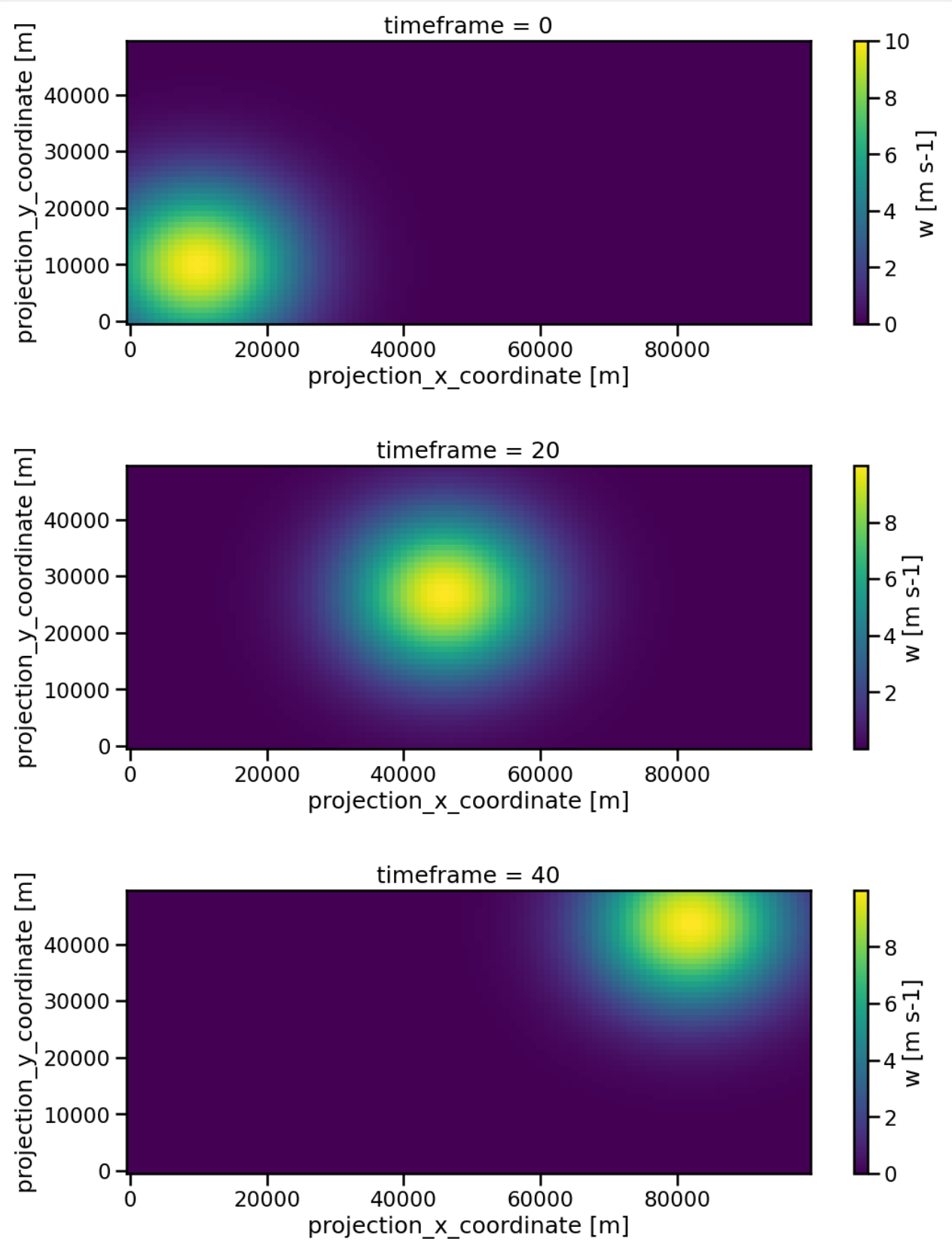
To visualize the data, we can plot individual time frames using matplotlib per imshow:

[6]:
```
fig, axs = plt.subplots(ncols=1, nrows=3, figsize=(12, 16), sharey=True)
plt.subplots_adjust(hspace=0.5)

for i, itime in enumerate([0, 20, 40]):
```

(continues on next page)

(continued from previous page)

```
    # plot the 2D blob field in colors
    test_data.isel(time=itime).plot(ax=axs[i])

    axs[i].set_title(f"timeframe = {itime}")
```



This tells as that our data is a single moving blob, which is what we are going the analyze with tobac now.

6.1.3 2. Feature Detection

The first step of the general working routine of tobac is the identification of features. This essentially means finding the maxima or minima of the data.

To use the according functions of tobac we need to specify:

- the thresholds below/above the features are detected
- the spacing of our data

The spacing of the temporal and spatial dimension can be extracted from the data using a build-in utility:

[7]:
```
dxy, dt = tobac.get_spacings(test_data)
```

To get an idea of which order of magnitude our thresholds should be, we check the maximum of our data:

[8]:
```
test_data.max()
```

[8]:
```
<xarray.DataArray 'w' ()> Size: 8B
array(10.)
```

Since we know that our data will only have one maximum it is reasoable to choose 9 as our threshold, but keep in mind that we could also add multiple values here if our data would be more complex.

[9]:
```
threshold = 9
```

Now we are ready to apply the feature detection algorithm. Notice that this is a minimal input. The function has several other option we will cover in later tutorials.

[10]:
```
%%capture
features = tobac.feature_detection_multithreshold(test_data, dxy, threshold)
```

Let's inspect the resulting object:

[11]:
```
features
```

[11]:

	frame	idx	hdim_1	hdim_2	num	threshold_value	feature	\
0	0	1	10.000000	10.000000	69	9	1	
1	1	1	10.939394	11.848485	66	9	2	
2	2	1	11.707692	13.661538	65	9	3	
3	3	1	12.569231	15.353846	65	9	4	
4	4	1	13.200000	17.107692	65	9	5	
5	5	1	14.184615	19.000000	65	9	6	
6	6	1	15.060606	20.848485	66	9	7	
7	7	1	15.953125	22.765625	64	9	8	
8	8	1	16.707692	24.338462	65	9	9	
9	9	1	17.676923	26.169231	65	9	10	
10	10	1	18.184615	28.000000	65	9	11	
11	11	1	19.171875	29.828125	64	9	12	
12	12	1	20.046875	31.765625	64	9	13	
13	13	1	20.953125	33.234375	64	9	14	
14	14	1	21.828125	35.171875	64	9	15	
15	15	1	22.815385	37.000000	65	9	16	
16	16	1	23.323077	38.830769	65	9	17	
17	17	1	24.292308	40.661538	65	9	18	
18	18	1	25.046875	42.234375	64	9	19	

(continues on next page)

(continued from previous page)

```
19     19    1  25.939394  44.151515   66                9       20
20     20    1  26.815385  46.000000   65                9       21
21     21    1  27.800000  47.892308   65                9       22
22     22    1  28.430769  49.646154   65                9       23
23     23    1  29.292308  51.338462   65                9       24
24     24    1  30.060606  53.151515   66                9       25
25     25    1  31.000000  55.000000   69                9       26
26     26    1  31.939394  56.848485   66                9       27
27     27    1  32.707692  58.661538   65                9       28
28     28    1  33.569231  60.353846   65                9       29
29     29    1  34.200000  62.107692   65                9       30
30     30    1  35.184615  64.000000   65                9       31
31     31    1  36.060606  65.848485   66                9       32
32     32    1  36.953125  67.765625   64                9       33
33     33    1  37.707692  69.338462   65                9       34
34     34    1  38.676923  71.169231   65                9       35
35     35    1  39.184615  73.000000   65                9       36
36     36    1  40.171875  74.828125   64                9       37
37     37    1  41.046875  76.765625   64                9       38
38     38    1  41.953125  78.234375   64                9       39
39     39    1  42.828125  80.171875   64                9       40
40     40    1  43.815385  82.000000   65                9       41
41     41    1  44.462687  83.820896   67                9       42
42     42    1  45.292308  85.661538   65                9       43
43     43    1  45.836066  87.278689   61                9       44
44     44    1  46.254545  89.145455   55                9       45
45     45    1  46.770833  91.000000   48                9       46
46     46    1  47.256410  93.000000   39                9       47
47     47    1  47.500000  94.764706   34                9       48
48     48    1  47.782609  96.130435   23                9       49
49     49    1  48.153846  97.230769   13                9       50
50     50    1  48.600000  98.200000    5                9       51

                   time              timestr  projection_y_coordinate  \
0   2000-01-01 12:00:00  2000-01-01 12:00:00             10000.000000
1   2000-01-01 12:01:00  2000-01-01 12:01:00             10939.393939
2   2000-01-01 12:02:00  2000-01-01 12:02:00             11707.692308
3   2000-01-01 12:03:00  2000-01-01 12:03:00             12569.230769
4   2000-01-01 12:04:00  2000-01-01 12:04:00             13200.000000
5   2000-01-01 12:05:00  2000-01-01 12:05:00             14184.615385
6   2000-01-01 12:06:00  2000-01-01 12:06:00             15060.606061
7   2000-01-01 12:07:00  2000-01-01 12:07:00             15953.125000
8   2000-01-01 12:08:00  2000-01-01 12:08:00             16707.692308
9   2000-01-01 12:09:00  2000-01-01 12:09:00             17676.923077
10  2000-01-01 12:10:00  2000-01-01 12:10:00             18184.615385
11  2000-01-01 12:11:00  2000-01-01 12:11:00             19171.875000
12  2000-01-01 12:12:00  2000-01-01 12:12:00             20046.875000
13  2000-01-01 12:13:00  2000-01-01 12:13:00             20953.125000
14  2000-01-01 12:14:00  2000-01-01 12:14:00             21828.125000
15  2000-01-01 12:15:00  2000-01-01 12:15:00             22815.384615
16  2000-01-01 12:16:00  2000-01-01 12:16:00             23323.076923
17  2000-01-01 12:17:00  2000-01-01 12:17:00             24292.307692
```

(continues on next page)

(continued from previous page)

```
18  2000-01-01 12:18:00  2000-01-01 12:18:00             25046.875000
19  2000-01-01 12:19:00  2000-01-01 12:19:00             25939.393939
20  2000-01-01 12:20:00  2000-01-01 12:20:00             26815.384615
21  2000-01-01 12:21:00  2000-01-01 12:21:00             27800.000000
22  2000-01-01 12:22:00  2000-01-01 12:22:00             28430.769231
23  2000-01-01 12:23:00  2000-01-01 12:23:00             29292.307692
24  2000-01-01 12:24:00  2000-01-01 12:24:00             30060.606061
25  2000-01-01 12:25:00  2000-01-01 12:25:00             31000.000000
26  2000-01-01 12:26:00  2000-01-01 12:26:00             31939.393939
27  2000-01-01 12:27:00  2000-01-01 12:27:00             32707.692308
28  2000-01-01 12:28:00  2000-01-01 12:28:00             33569.230769
29  2000-01-01 12:29:00  2000-01-01 12:29:00             34200.000000
30  2000-01-01 12:30:00  2000-01-01 12:30:00             35184.615385
31  2000-01-01 12:31:00  2000-01-01 12:31:00             36060.606061
32  2000-01-01 12:32:00  2000-01-01 12:32:00             36953.125000
33  2000-01-01 12:33:00  2000-01-01 12:33:00             37707.692308
34  2000-01-01 12:34:00  2000-01-01 12:34:00             38676.923077
35  2000-01-01 12:35:00  2000-01-01 12:35:00             39184.615385
36  2000-01-01 12:36:00  2000-01-01 12:36:00             40171.875000
37  2000-01-01 12:37:00  2000-01-01 12:37:00             41046.875000
38  2000-01-01 12:38:00  2000-01-01 12:38:00             41953.125000
39  2000-01-01 12:39:00  2000-01-01 12:39:00             42828.125000
40  2000-01-01 12:40:00  2000-01-01 12:40:00             43815.384615
41  2000-01-01 12:41:00  2000-01-01 12:41:00             44462.686567
42  2000-01-01 12:42:00  2000-01-01 12:42:00             45292.307692
43  2000-01-01 12:43:00  2000-01-01 12:43:00             45836.065574
44  2000-01-01 12:44:00  2000-01-01 12:44:00             46254.545455
45  2000-01-01 12:45:00  2000-01-01 12:45:00             46770.833333
46  2000-01-01 12:46:00  2000-01-01 12:46:00             47256.410256
47  2000-01-01 12:47:00  2000-01-01 12:47:00             47500.000000
48  2000-01-01 12:48:00  2000-01-01 12:48:00             47782.608696
49  2000-01-01 12:49:00  2000-01-01 12:49:00             48153.846154
50  2000-01-01 12:50:00  2000-01-01 12:50:00             48600.000000

    projection_x_coordinate   latitude   longitude
0              10000.000000  24.100000  150.100000
1              11848.484848  24.118485  150.109394
2              13661.538462  24.136615  150.117077
3              15353.846154  24.153538  150.125692
4              17107.692308  24.171077  150.132000
5              19000.000000  24.190000  150.141846
6              20848.484848  24.208485  150.150606
7              22765.625000  24.227656  150.159531
8              24338.461538  24.243385  150.167077
9              26169.230769  24.261692  150.176769
10             28000.000000  24.280000  150.181846
11             29828.125000  24.298281  150.191719
12             31765.625000  24.317656  150.200469
13             33234.375000  24.332344  150.209531
14             35171.875000  24.351719  150.218281
15             37000.000000  24.370000  150.228154
16             38830.769231  24.388308  150.233231
```

(continues on next page)

(continued from previous page)

```
17              40661.538462  24.406615  150.242923
18              42234.375000  24.422344  150.250469
19              44151.515152  24.441515  150.259394
20              46000.000000  24.460000  150.268154
21              47892.307692  24.478923  150.278000
22              49646.153846  24.496462  150.284308
23              51338.461538  24.513385  150.292923
24              53151.515152  24.531515  150.300606
25              55000.000000  24.550000  150.310000
26              56848.484848  24.568485  150.319394
27              58661.538462  24.586615  150.327077
28              60353.846154  24.603538  150.335692
29              62107.692308  24.621077  150.342000
30              64000.000000  24.640000  150.351846
31              65848.484848  24.658485  150.360606
32              67765.625000  24.677656  150.369531
33              69338.461538  24.693385  150.377077
34              71169.230769  24.711692  150.386769
35              73000.000000  24.730000  150.391846
36              74828.125000  24.748281  150.401719
37              76765.625000  24.767656  150.410469
38              78234.375000  24.782344  150.419531
39              80171.875000  24.801719  150.428281
40              82000.000000  24.820000  150.438154
41              83820.895522  24.838209  150.444627
42              85661.538462  24.856615  150.452923
43              87278.688525  24.872787  150.458361
44              89145.454545  24.891455  150.462545
45              91000.000000  24.910000  150.467708
46              93000.000000  24.930000  150.472564
47              94764.705882  24.947647  150.475000
48              96130.434783  24.961304  150.477826
49              97230.769231  24.972308  150.481538
50              98200.000000  24.982000  150.486000
```

The ouputs tells us that features were found in 51 **frame**s (index 0 to 50) of our data. The variable **idx** is 1 for every frames, which means that only 1 feature was found in every time step, as we expected. **hdim_1** and **hdim_2** are the position of this feature with respect to the y and x-indices.

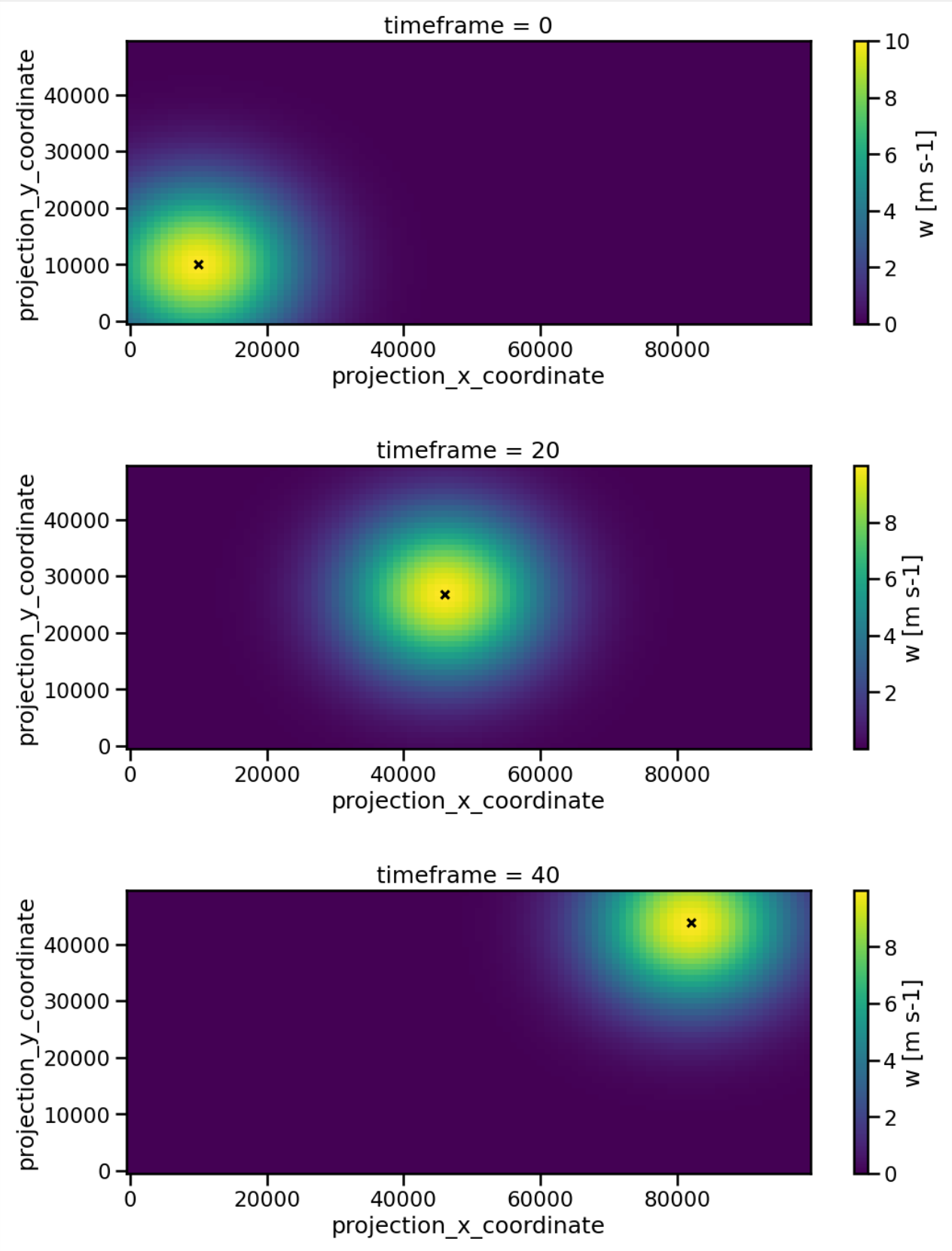
We can plot the detected feature positions on top of the colored blob.

[12]:
```
fig, axs = plt.subplots(ncols=1, nrows=3, figsize=(12, 16), sharey=True)
plt.subplots_adjust(hspace=0.5)

for i, itime in enumerate([0, 20, 40]):
    # plot the 2D blob field in colors
    test_data.isel(time=itime).plot(ax=axs[i])

    # plot the detected feature as black cross
    f = features.loc[[itime]]
    f.plot.scatter(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
```

(continues on next page)

(continued from previous page)

```
        s=40,
        ax=axs[i],
        color="black",
        marker="x",
    )

    axs[i].set_title(f"timeframe = {itime}")
```



The function has succesfully detected the maximum of our data in the individual timeframes.

6.1.4 3. Trajectory Linking

After we are done finding the features and associated segments for each frame it is necessary for further analysis to keep track of those elements troughout time. Linking is the tool for that. It connects the features of the timesteps which belong together. We are going to use the *linking_trackpy()* function here. The required inputs are the features, the two spacings and a maximum velocity of the features.

[13]:
```
trajectories = tobac.linking_trackpy(features, test_data, dt=dt, dxy=dxy, v_max=100)
```

```
Frame 50: 1 trajectories present.
```

fails without v_max

Unsurprisingly, one trajectory was found. The returned object is another Dataset:

[14]:
```
trajectories
```

[14]:

	frame	idx	hdim_1	hdim_2	num	threshold_value	feature	\
0	0	1	10.000000	10.000000	69	9	1	
1	1	1	10.939394	11.848485	66	9	2	
2	2	1	11.707692	13.661538	65	9	3	
3	3	1	12.569231	15.353846	65	9	4	
4	4	1	13.200000	17.107692	65	9	5	
5	5	1	14.184615	19.000000	65	9	6	
6	6	1	15.060606	20.848485	66	9	7	
7	7	1	15.953125	22.765625	64	9	8	
8	8	1	16.707692	24.338462	65	9	9	
9	9	1	17.676923	26.169231	65	9	10	
10	10	1	18.184615	28.000000	65	9	11	
11	11	1	19.171875	29.828125	64	9	12	
12	12	1	20.046875	31.765625	64	9	13	
13	13	1	20.953125	33.234375	64	9	14	
14	14	1	21.828125	35.171875	64	9	15	
15	15	1	22.815385	37.000000	65	9	16	
16	16	1	23.323077	38.830769	65	9	17	
17	17	1	24.292308	40.661538	65	9	18	
18	18	1	25.046875	42.234375	64	9	19	
19	19	1	25.939394	44.151515	66	9	20	
20	20	1	26.815385	46.000000	65	9	21	
21	21	1	27.800000	47.892308	65	9	22	
22	22	1	28.430769	49.646154	65	9	23	
23	23	1	29.292308	51.338462	65	9	24	
24	24	1	30.060606	53.151515	66	9	25	
25	25	1	31.000000	55.000000	69	9	26	
26	26	1	31.939394	56.848485	66	9	27	
27	27	1	32.707692	58.661538	65	9	28	
28	28	1	33.569231	60.353846	65	9	29	
29	29	1	34.200000	62.107692	65	9	30	
30	30	1	35.184615	64.000000	65	9	31	
31	31	1	36.060606	65.848485	66	9	32	
32	32	1	36.953125	67.765625	64	9	33	
33	33	1	37.707692	69.338462	65	9	34	
34	34	1	38.676923	71.169231	65	9	35	
35	35	1	39.184615	73.000000	65	9	36	
36	36	1	40.171875	74.828125	64	9	37	

(continues on next page)

(continued from previous page)

```
37     37    1  41.046875  76.765625   64                9       38
38     38    1  41.953125  78.234375   64                9       39
39     39    1  42.828125  80.171875   64                9       40
40     40    1  43.815385  82.000000   65                9       41
41     41    1  44.462687  83.820896   67                9       42
42     42    1  45.292308  85.661538   65                9       43
43     43    1  45.836066  87.278689   61                9       44
44     44    1  46.254545  89.145455   55                9       45
45     45    1  46.770833  91.000000   48                9       46
46     46    1  47.256410  93.000000   39                9       47
47     47    1  47.500000  94.764706   34                9       48
48     48    1  47.782609  96.130435   23                9       49
49     49    1  48.153846  97.230769   13                9       50
50     50    1  48.600000  98.200000    5                9       51

                   time              timestr  projection_y_coordinate  \
0   2000-01-01 12:00:00  2000-01-01 12:00:00              10000.000000
1   2000-01-01 12:01:00  2000-01-01 12:01:00              10939.393939
2   2000-01-01 12:02:00  2000-01-01 12:02:00              11707.692308
3   2000-01-01 12:03:00  2000-01-01 12:03:00              12569.230769
4   2000-01-01 12:04:00  2000-01-01 12:04:00              13200.000000
5   2000-01-01 12:05:00  2000-01-01 12:05:00              14184.615385
6   2000-01-01 12:06:00  2000-01-01 12:06:00              15060.606061
7   2000-01-01 12:07:00  2000-01-01 12:07:00              15953.125000
8   2000-01-01 12:08:00  2000-01-01 12:08:00              16707.692308
9   2000-01-01 12:09:00  2000-01-01 12:09:00              17676.923077
10  2000-01-01 12:10:00  2000-01-01 12:10:00              18184.615385
11  2000-01-01 12:11:00  2000-01-01 12:11:00              19171.875000
12  2000-01-01 12:12:00  2000-01-01 12:12:00              20046.875000
13  2000-01-01 12:13:00  2000-01-01 12:13:00              20953.125000
14  2000-01-01 12:14:00  2000-01-01 12:14:00              21828.125000
15  2000-01-01 12:15:00  2000-01-01 12:15:00              22815.384615
16  2000-01-01 12:16:00  2000-01-01 12:16:00              23323.076923
17  2000-01-01 12:17:00  2000-01-01 12:17:00              24292.307692
18  2000-01-01 12:18:00  2000-01-01 12:18:00              25046.875000
19  2000-01-01 12:19:00  2000-01-01 12:19:00              25939.393939
20  2000-01-01 12:20:00  2000-01-01 12:20:00              26815.384615
21  2000-01-01 12:21:00  2000-01-01 12:21:00              27800.000000
22  2000-01-01 12:22:00  2000-01-01 12:22:00              28430.769231
23  2000-01-01 12:23:00  2000-01-01 12:23:00              29292.307692
24  2000-01-01 12:24:00  2000-01-01 12:24:00              30060.606061
25  2000-01-01 12:25:00  2000-01-01 12:25:00              31000.000000
26  2000-01-01 12:26:00  2000-01-01 12:26:00              31939.393939
27  2000-01-01 12:27:00  2000-01-01 12:27:00              32707.692308
28  2000-01-01 12:28:00  2000-01-01 12:28:00              33569.230769
29  2000-01-01 12:29:00  2000-01-01 12:29:00              34200.000000
30  2000-01-01 12:30:00  2000-01-01 12:30:00              35184.615385
31  2000-01-01 12:31:00  2000-01-01 12:31:00              36060.606061
32  2000-01-01 12:32:00  2000-01-01 12:32:00              36953.125000
33  2000-01-01 12:33:00  2000-01-01 12:33:00              37707.692308
34  2000-01-01 12:34:00  2000-01-01 12:34:00              38676.923077
35  2000-01-01 12:35:00  2000-01-01 12:35:00              39184.615385
```

(continues on next page)

(continued from previous page)

```
36  2000-01-01 12:36:00  2000-01-01 12:36:00             40171.875000
37  2000-01-01 12:37:00  2000-01-01 12:37:00             41046.875000
38  2000-01-01 12:38:00  2000-01-01 12:38:00             41953.125000
39  2000-01-01 12:39:00  2000-01-01 12:39:00             42828.125000
40  2000-01-01 12:40:00  2000-01-01 12:40:00             43815.384615
41  2000-01-01 12:41:00  2000-01-01 12:41:00             44462.686567
42  2000-01-01 12:42:00  2000-01-01 12:42:00             45292.307692
43  2000-01-01 12:43:00  2000-01-01 12:43:00             45836.065574
44  2000-01-01 12:44:00  2000-01-01 12:44:00             46254.545455
45  2000-01-01 12:45:00  2000-01-01 12:45:00             46770.833333
46  2000-01-01 12:46:00  2000-01-01 12:46:00             47256.410256
47  2000-01-01 12:47:00  2000-01-01 12:47:00             47500.000000
48  2000-01-01 12:48:00  2000-01-01 12:48:00             47782.608696
49  2000-01-01 12:49:00  2000-01-01 12:49:00             48153.846154
50  2000-01-01 12:50:00  2000-01-01 12:50:00             48600.000000

    projection_x_coordinate   latitude   longitude  cell       time_cell
0              10000.000000  24.100000  150.100000     1 0 days 00:00:00
1              11848.484848  24.118485  150.109394     1 0 days 00:01:00
2              13661.538462  24.136615  150.117077     1 0 days 00:02:00
3              15353.846154  24.153538  150.125692     1 0 days 00:03:00
4              17107.692308  24.171077  150.132000     1 0 days 00:04:00
5              19000.000000  24.190000  150.141846     1 0 days 00:05:00
6              20848.484848  24.208485  150.150606     1 0 days 00:06:00
7              22765.625000  24.227656  150.159531     1 0 days 00:07:00
8              24338.461538  24.243385  150.167077     1 0 days 00:08:00
9              26169.230769  24.261692  150.176769     1 0 days 00:09:00
10             28000.000000  24.280000  150.181846     1 0 days 00:10:00
11             29828.125000  24.298281  150.191719     1 0 days 00:11:00
12             31765.625000  24.317656  150.200469     1 0 days 00:12:00
13             33234.375000  24.332344  150.209531     1 0 days 00:13:00
14             35171.875000  24.351719  150.218281     1 0 days 00:14:00
15             37000.000000  24.370000  150.228154     1 0 days 00:15:00
16             38830.769231  24.388308  150.233231     1 0 days 00:16:00
17             40661.538462  24.406615  150.242923     1 0 days 00:17:00
18             42234.375000  24.422344  150.250469     1 0 days 00:18:00
19             44151.515152  24.441515  150.259394     1 0 days 00:19:00
20             46000.000000  24.460000  150.268154     1 0 days 00:20:00
21             47892.307692  24.478923  150.278000     1 0 days 00:21:00
22             49646.153846  24.496462  150.284308     1 0 days 00:22:00
23             51338.461538  24.513385  150.292923     1 0 days 00:23:00
24             53151.515152  24.531515  150.300606     1 0 days 00:24:00
25             55000.000000  24.550000  150.310000     1 0 days 00:25:00
26             56848.484848  24.568485  150.319394     1 0 days 00:26:00
27             58661.538462  24.586615  150.327077     1 0 days 00:27:00
28             60353.846154  24.603538  150.335692     1 0 days 00:28:00
29             62107.692308  24.621077  150.342000     1 0 days 00:29:00
30             64000.000000  24.640000  150.351846     1 0 days 00:30:00
31             65848.484848  24.658485  150.360606     1 0 days 00:31:00
32             67765.625000  24.677656  150.369531     1 0 days 00:32:00
33             69338.461538  24.693385  150.377077     1 0 days 00:33:00
34             71169.230769  24.711692  150.386769     1 0 days 00:34:00
```

(continues on next page)

(continued from previous page)

```
35              73000.000000  24.730000  150.391846     1 0 days 00:35:00
36              74828.125000  24.748281  150.401719     1 0 days 00:36:00
37              76765.625000  24.767656  150.410469     1 0 days 00:37:00
38              78234.375000  24.782344  150.419531     1 0 days 00:38:00
39              80171.875000  24.801719  150.428281     1 0 days 00:39:00
40              82000.000000  24.820000  150.438154     1 0 days 00:40:00
41              83820.895522  24.838209  150.444627     1 0 days 00:41:00
42              85661.538462  24.856615  150.452923     1 0 days 00:42:00
43              87278.688525  24.872787  150.458361     1 0 days 00:43:00
44              89145.454545  24.891455  150.462545     1 0 days 00:44:00
45              91000.000000  24.910000  150.467708     1 0 days 00:45:00
46              93000.000000  24.930000  150.472564     1 0 days 00:46:00
47              94764.705882  24.947647  150.475000     1 0 days 00:47:00
48              96130.434783  24.961304  150.477826     1 0 days 00:48:00
49              97230.769231  24.972308  150.481538     1 0 days 00:49:00
50              98200.000000  24.982000  150.486000     1 0 days 00:50:00
```

The new variable cell now indexes the features in the different time steps. Therefore we can use it to create a mask for our moving feature:

[15]:
```
track_mask = trajectories["cell"] == 1.0
```

This mask can then be used - to select the track:

[16]:
```
track = trajectories.where(track_mask).dropna()
```

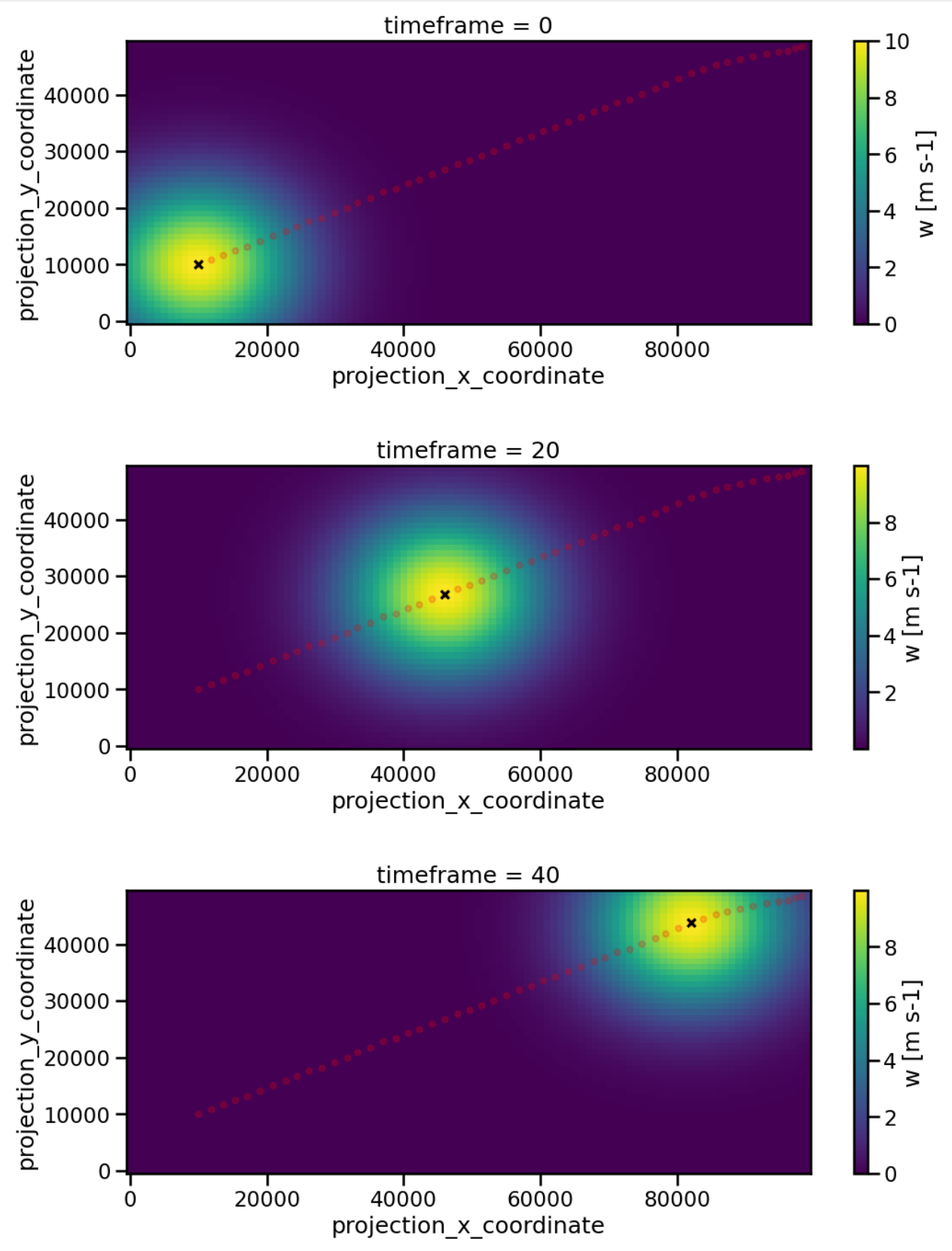
- and to show the track in our plots:

[17]:
```
fig, axs = plt.subplots(ncols=1, nrows=3, figsize=(12, 16), sharey=True)
plt.subplots_adjust(hspace=0.5)
# fig.tight_layout()


for i, itime in enumerate([0, 20, 40]):
    # plot the 2D blob field in colors
    test_data.isel(time=itime).plot(ax=axs[i])

    # plot the track
    track.plot.scatter(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=axs[i],
        color="red",
        marker="o",
        alpha=0.2,
    )

    # plot the detected feature as black cross
    f = features.loc[[itime]]
    f.plot.scatter(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        s=40,
```

(continues on next page)

(continued from previous page)

```
        ax=axs[i],
        color="black",
        marker="x",
    )

    axs[i].set_title(f"timeframe = {itime}")
```


timeframe = 0
projection_y_coordinate
projection_x_coordinate
w [m s-1]
0
10000
20000
30000
40000
0
20000
40000
60000
80000
0
2
4
6
8
10
timeframe = 20
projection_y_coordinate
projection_x_coordinate
w [m s-1]
timeframe = 40
projection_y_coordinate
projection_x_coordinate
w [m s-1]

6.1.5 4. Segmentation

Another step after the feature detection and tracking is the segmentation of the data. This means we find the area surrounding our features belonging to the same cluster. Logically, we now need the already detected features as an additional input.

Just a small thought here:

We often introduce the different steps as 1.feature detection, 2.segmentation and 3. tracking, so having the segmenation step actually before the trajectory linking. The order actually does not matter since the current segmentation approach is based on the feature detection output only and can be done before or after the linking.

On further extensions:

- Also, one could actually use the output of the segmentation (`segments` Dataset in the example below) as input for the tracking with the advantage that information on the area (ncells) is added in the dataframe.
- One could also use the output of the tracking in the segmentation (`trajectories` Dataset from above) with the advantage that mask will contain only the features that are also linked to trajectories.

These possibilities exist and could be utlized by you …

[18]:
```
%%capture
segment_labels, segments = tobac.segmentation_2D(features, test_data, dxy, threshold=9)
```

As the name implies, the first object returned is an array in which the segmented areas belonging to each feature have the same label value as that feature. The second output is a dataframe of the detected features updated with information about their segmented regions (currently only the number of pixels segmented)

[19]:
```
segment_labels
```

[19]:
```
<xarray.DataArray 'segmentation_mask' (time: 100, y: 50, x: 100)> Size: 2MB
[500000 values with dtype=int32]
Coordinates:
  * time       (time) datetime64[ns] 800B 2000-01-01T12:00:00 ... 2000-01-01T...
  * y          (y) float64 400B 0.0 1e+03 2e+03 ... 4.7e+04 4.8e+04 4.9e+04
  * x          (x) float64 800B 0.0 1e+03 2e+03 ... 9.7e+04 9.8e+04 9.9e+04
    latitude   (y, x) float64 40kB ...
    longitude  (y, x) float64 40kB ...
Attributes:
    long_name:  segmentation_mask
```

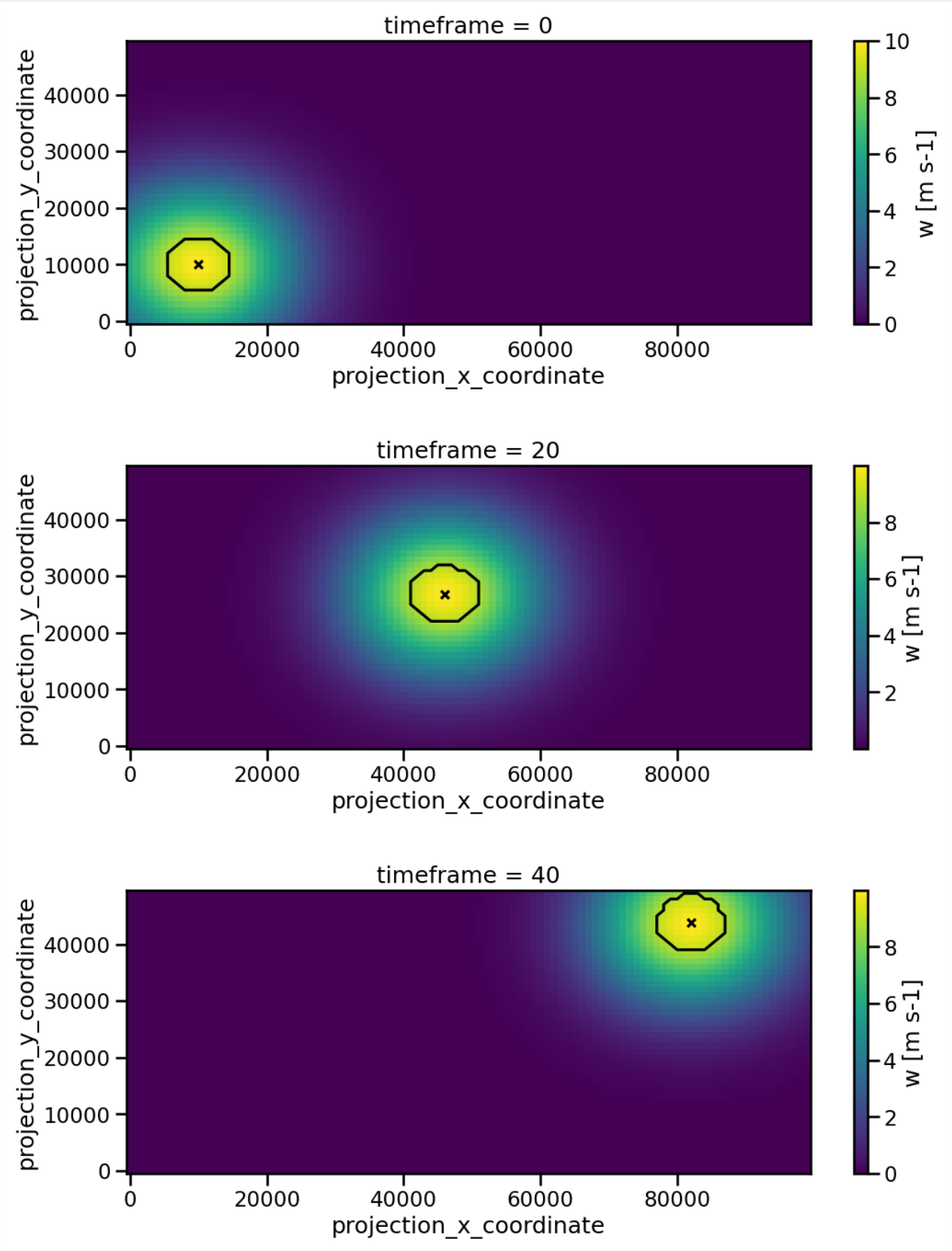
Since True/False corresponds to 1/0 in python, we can visualize the segments with a simple contour plot:

[20]:
```
fig, axs = plt.subplots(ncols=1, nrows=3, figsize=(12, 16), sharey=True)
plt.subplots_adjust(hspace=0.5)


for i, itime in enumerate([0, 20, 40]):
    # plot the 2D blob field in colors
    test_data.isel(time=itime).plot(ax=axs[i])

    # plot the mask outline
    segment_labels.isel(time=itime).plot.contour(levels=[0.5], ax=axs[i], colors="k")

    # plot the detected feature as black cross
    f = features.loc[[itime]]
```

(continues on next page)

(continued from previous page)

```
    f.plot.scatter(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        s=40,
        ax=axs[i],
        color="black",
        marker="x",
    )

    axs[i].set_title(f"timeframe = {itime}")
```



Keep in mind that the area of the resulting segments crucially depends on the defined thresholds.

6.1.6 5. Statistical Analysis

Blob Velocity / Motion Speed

Now different functions of tobacs analysis module can be used to calculate or plot properties of the tracks and the features. For example the velocity of the feature along the track can be calulated via:

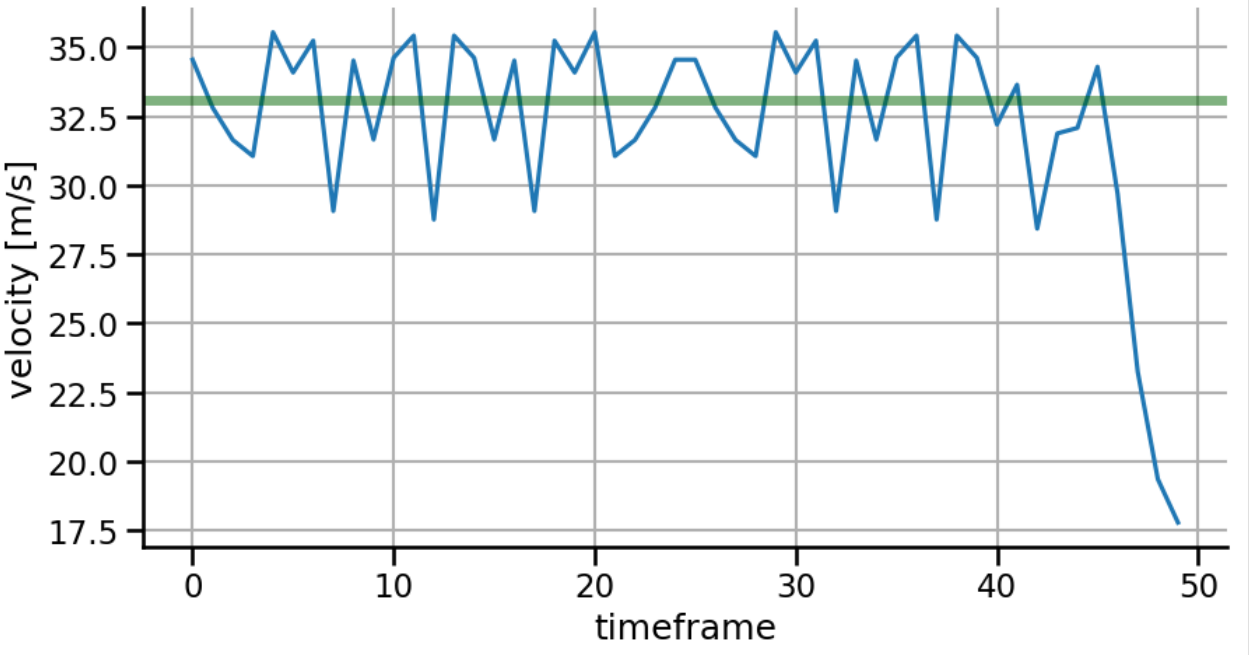
[21]:
```
vel = tobac.calculate_velocity(track)
```

The expected velocity (hard-coded in the test function) is:

[22]:
```
expected_velocity = np.sqrt(30**2 + 14**2)
```

Plotting the velocity vs the timeframe will give us

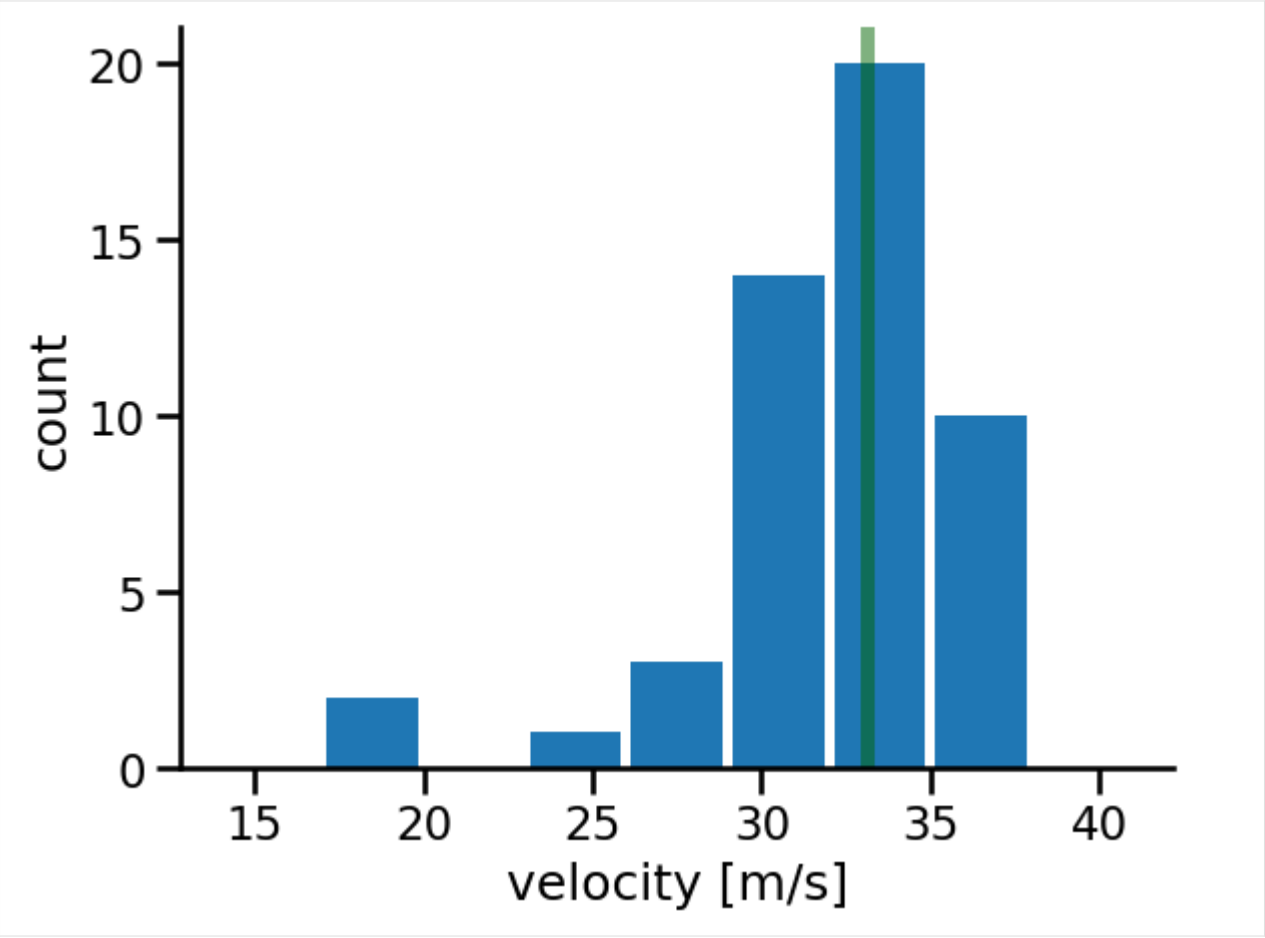
[23]:
```
plt.figure(figsize=(10, 5))
plt.tight_layout()
plt.plot(vel["frame"], vel["v"])

plt.xlabel("timeframe")
plt.ylabel("velocity [m/s]")
plt.grid()

plt.axhline(expected_velocity, color="darkgreen", lw=5, alpha=0.5)
sns.despine()
```



We can also create an histogram of the detected velocities:

[24]:
```
hist, edges = tobac.velocity_histogram(
    track,
    bin_edges=np.arange(14, 43, 3),
)
```

[25]:
```
width = 0.9 * (edges[1] - edges[0])
center = (edges[:-1] + edges[1:]) / 2

plt.tight_layout()
plt.bar(center, hist, width=width)
plt.ylabel("count")
plt.xlabel("velocity [m/s]")

plt.axvline(expected_velocity, color="darkgreen", lw=5, alpha=0.5)
sns.despine()
```



Area

The area of the features can also be calculated and plotted throughout time:

[26]:
```
area = tobac.calculate_area(features, segment_labels)
```

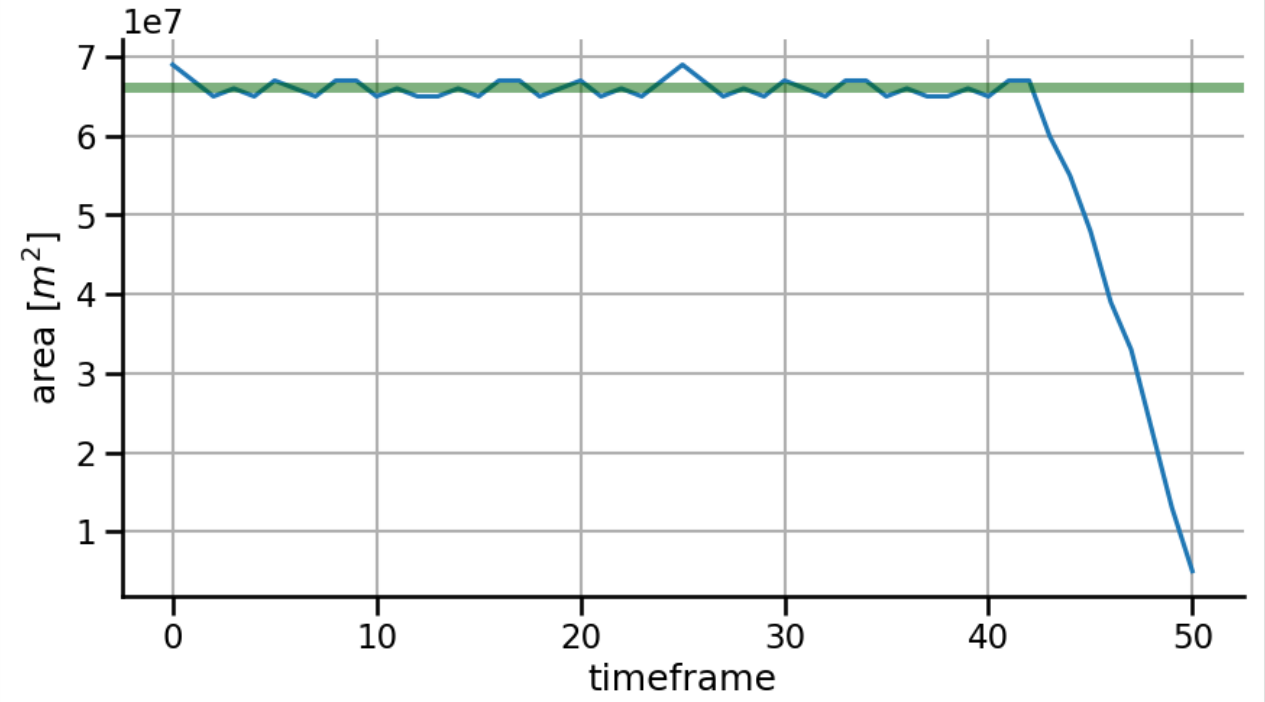
[27]:
```
blob_magitude = 10  # also hard-code in the test
blob_sigma = 10e3

normalized_circle_radius = np.sqrt(np.log(blob_magitude / threshold))
absolute_circle_radius = np.sqrt(2) * blob_sigma * normalized_circle_radius
```

(continues on next page)

(continued from previous page)

```
expected_area = np.pi * absolute_circle_radius**2
```

[28]:
```
plt.figure(figsize=(10, 5))
plt.tight_layout()
plt.plot(area["frame"], area["area"])
plt.xlabel("timeframe")
plt.ylabel(r"area [$m^2$]")
plt.grid()

plt.axhline(expected_area, color="darkgreen", lw=5, alpha=0.5)
sns.despine()
```

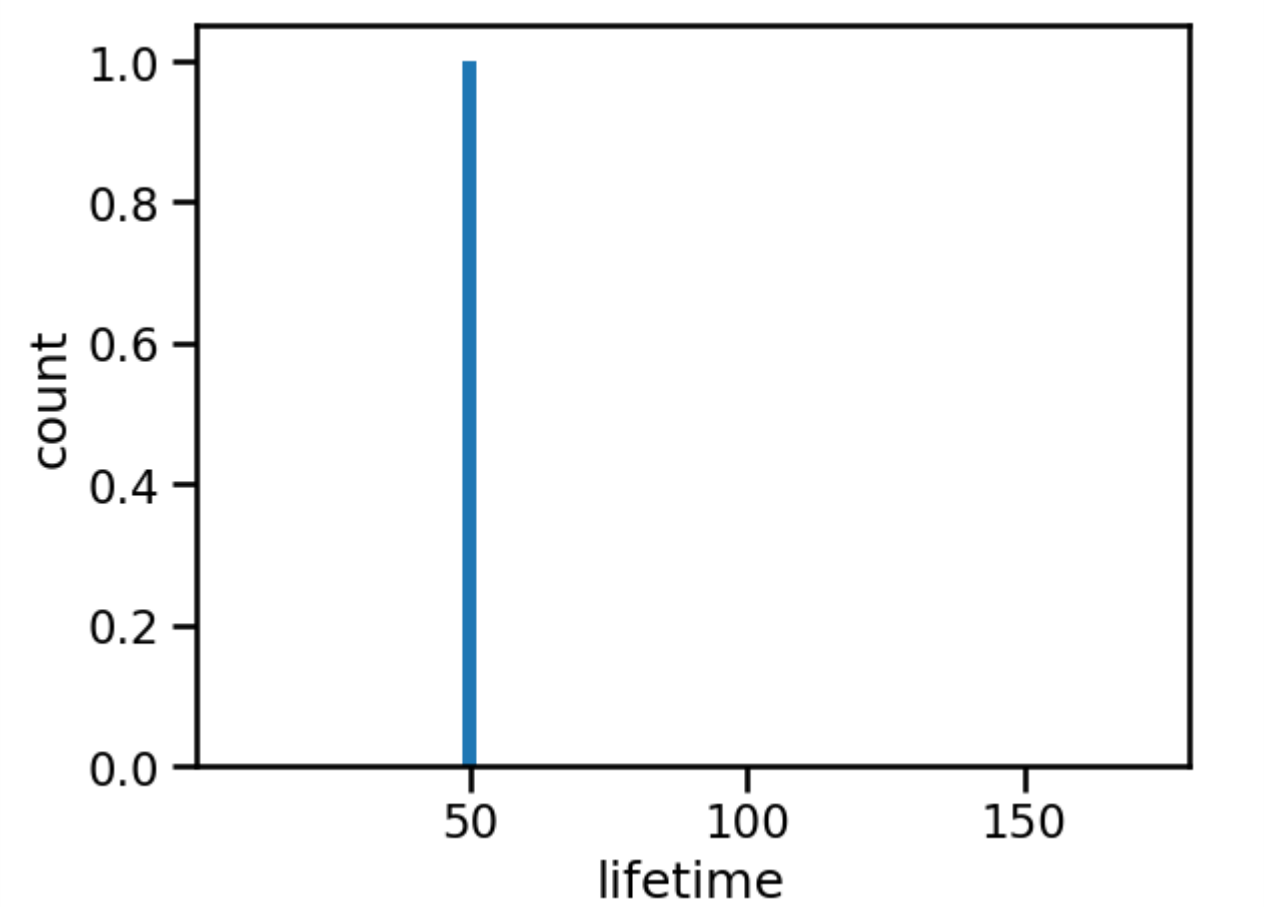


Lifetime

Another interesting property for real data are the lifetimes of our features. Tobac can also produce a histogram of this:

[29]:
```
hist, bins, centers = tobac.lifetime_histogram(
    track, bin_edges=np.arange(0, 200, 20)
)
```

[30]:
```
plt.tight_layout()
plt.bar(centers, hist, width=width)
plt.ylabel("count")
plt.xlabel("lifetime")
plt.show()
```



We can deduce, that our singular feature in the data has a lifetime of 50.

6.2 Idealized Case 2: Two crossing blobs

This tutorial explores the different methods of the linking process using an example of two crossing blobs. The following chapters will be covered:

1. *Data generation*
2. *Feature detection*
3. *Influence of tracking method*
4. *Analysis*

6.2.1 1. Data generation

We start by importing the usual libraries and adjusting some settings:

[1]:
```
import tobac
import numpy as np
import matplotlib.pyplot as plt
import datetime
import xarray as xr
import seaborn as sns

sns.set_context("talk")
%matplotlib inline
```

We will need to generate our own dataset for this tutorial. For this reason we define some bounds for our system:

[2]:
```
(
    x_min,
    y_min,
    x_max,
    y_max,
) = (
    0,
    0,
    1e5,
    1e5,
)
t_min, t_max = 0, 10000
```

We use these to create a mesh:

[3]:
```
def create_mesh(x_min, y_min, x_max, y_max, t_min, t_max, N_x=200, N_y=200, dt=520):
    x = np.linspace(x_min, x_max, N_x)
    y = np.linspace(y_min, y_max, N_y)
    t = np.arange(t_min, t_max, dt)
    mesh = np.meshgrid(t, y, x, indexing="ij")

    return mesh


mesh = create_mesh(x_min, y_min, x_max, y_max, t_min, t_max)
```

Additionally, we need to set velocities for our blobs:

[4]:
```
v_x = 10
v_y = 10
```

The dataset is created by using two functions. The first creates a wandering Gaussian blob as `numpy`-Array on our grid and the second transforms it into an `xarray`-DataArray with an arbitrary `datetime`.

[5]:
```
def create_wandering_blob(mesh, x_0, y_0, v_x, v_y, t_create, t_vanish, sigma=1e7):
    tt, yy, xx = mesh
    exponent = (xx - x_0 - v_x * (tt - t_create)) ** 2 + (
        yy - y_0 - v_y * (tt - t_create)
```

(continues on next page)

(continued from previous page)

```
    ) ** 2
    blob = np.exp(-exponent / sigma)
    blob = np.where(np.logical_and(tt >= t_create, tt <= t_vanish), blob, 0)

    return blob


def create_xarray(array, mesh, starting_time="2022-04-01T00:00"):
    tt, yy, xx = mesh
    t = np.unique(tt)
    y = np.unique(yy)
    x = np.unique(xx)

    N_t = len(t)
    dt = np.diff(t)[0]

    t_0 = np.datetime64(starting_time)
    t_delta = np.timedelta64(dt, "s")

    time = np.array([t_0 + i * t_delta for i in range(len(array))])

    dims = ("time", "projection_x_coordinate", "projection_y_coordinate")
    coords = {"time": time, "projection_x_coordinate": x, "projection_y_coordinate": y}
    attributes = {"units": ("m s-1")}

    data = xr.DataArray(data=array, dims=dims, coords=coords, attrs=attributes)
    # data = data.projection_x_coordinate.assign_attrs({"units": ("m")})
    # data = data.projection_y_coordinate.assign_attrs({"units": ("m")})
    data = data.assign_coords(latitude=("projection_x_coordinate", x / x.max() * 90))
    data = data.assign_coords(longitude=("projection_y_coordinate", y / y.max() * 90))

    return data
```

We use the first function to create two blobs whose paths will cross. To keep them detectable as seperate features in the dataset we don't want to just add them together. Instead we are going to use the highest value of each pixel by applying boolean masking and the resulting field is transformed into the `xarray` format.

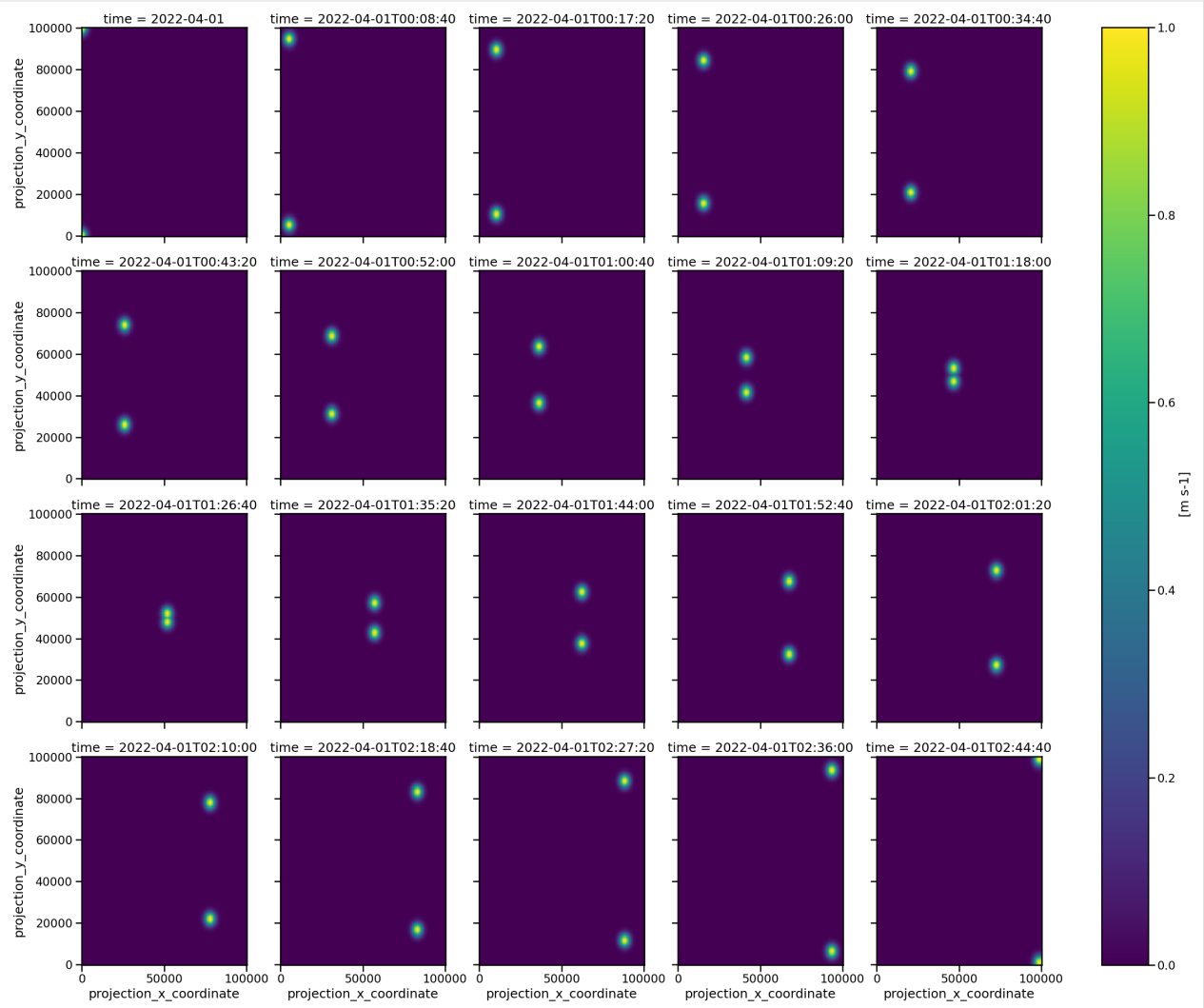
[6]:
```
blob_1 = create_wandering_blob(mesh, x_min, y_min, v_x, v_y, t_min, t_max)
blob_2 = create_wandering_blob(mesh, x_max, y_min, -v_x, v_y, t_min, t_max)
blob_mask = blob_1 > blob_2
blob = np.where(blob_mask, blob_1, blob_2)

data = create_xarray(blob, mesh)
```

```
/var/folders/40/kfr98p0j7n30fjp2n4ljjqbh0000gr/T/ipykernel_50349/1703872627.py:30:␣
↪UserWarning: Converting non-nanosecond precision datetime values to nanosecond␣
↪precision. This behavior can eventually be relaxed in xarray, as it is an artifact␣
↪from pandas which is now beginning to support non-nanosecond precision values. This␣
↪warning is caused by passing non-nanosecond np.datetime64 or np.timedelta64 values to␣
↪the DataArray or Variable constructor; it can be silenced by converting the values to␣
↪nanosecond precision ahead of time.
  data = xr.DataArray(data=array, dims=dims, coords=coords, attrs=attributes)
```

Let's check if we achived what we wanted by plotting the result:

[7]:
```
data.plot(
    cmap="viridis",
    col="time",
    col_wrap=5,
    x="projection_x_coordinate",
    y="projection_y_coordinate",
    size=5,
)
```

[7]:
```
<xarray.plot.facetgrid.FacetGrid at 0x13a7ad010>
```



Looks good! We see two features crossing each other, and they are clearly separable in every frame.

6.2.2 2. Feature detection

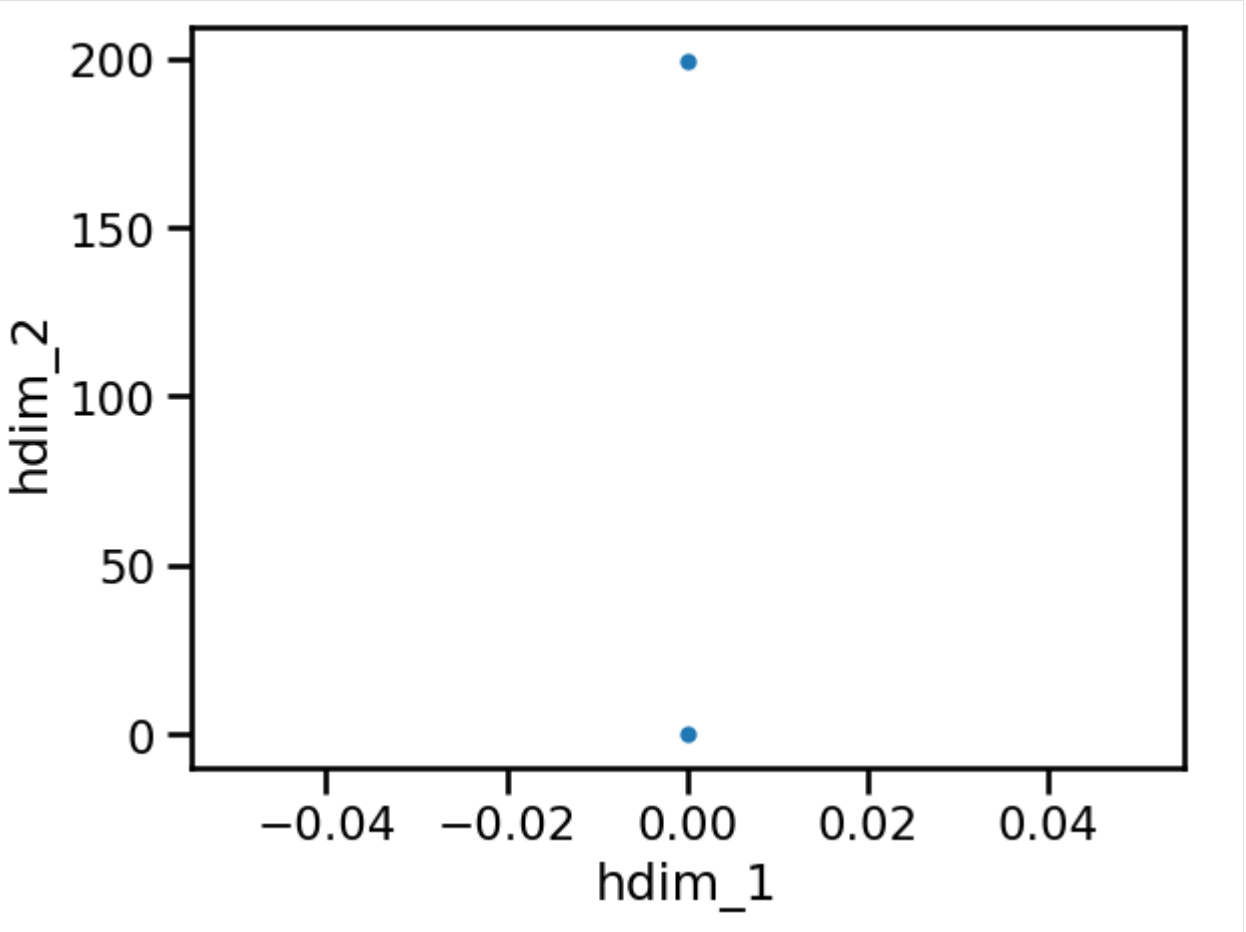
Before we can perform the tracking we need to detect the features with the usual function. The grid spacing is deduced from the generated field. We still need to find a reasonable threshold value. Let's try a really high one:

[8]:
```
%%capture

spacing = np.diff(np.unique(mesh[1]))[0]

dxy, dt = tobac.get_spacings(data, grid_spacing=spacing)
features = tobac.feature_detection_multithreshold(data, dxy, threshold=0.99)
```

[9]:
```
plt.figure(figsize=(10, 6))
features.plot.scatter(x="hdim_1", y="hdim_2")
```

[9]:
```
<Axes: xlabel='hdim_1', ylabel='hdim_2'>
```

```
<Figure size 1000x600 with 0 Axes>
```



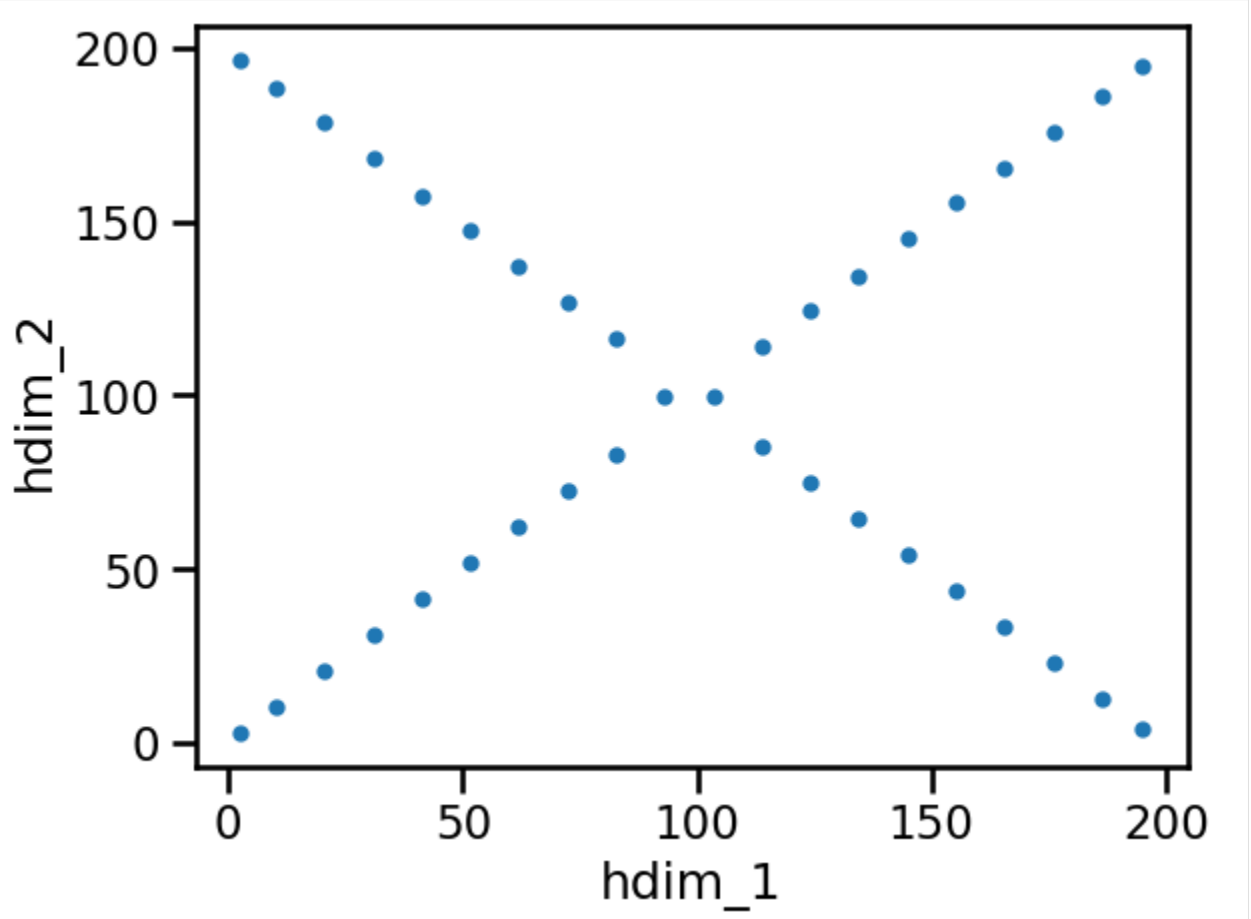
As you can see almost no features are detected. This means our threshold is too high and neglects many datapoints. Therefore it is a good idea to try a low threshold value:

[10]:
```
%%capture
features = tobac.feature_detection_multithreshold(data, dxy, threshold=0.3)
```

[11]:
```
plt.figure(figsize=(10, 6))
features.plot.scatter(x="hdim_1", y="hdim_2")
```

[11]:
```
<Axes: xlabel='hdim_1', ylabel='hdim_2'>
```

```
<Figure size 1000x600 with 0 Axes>
```



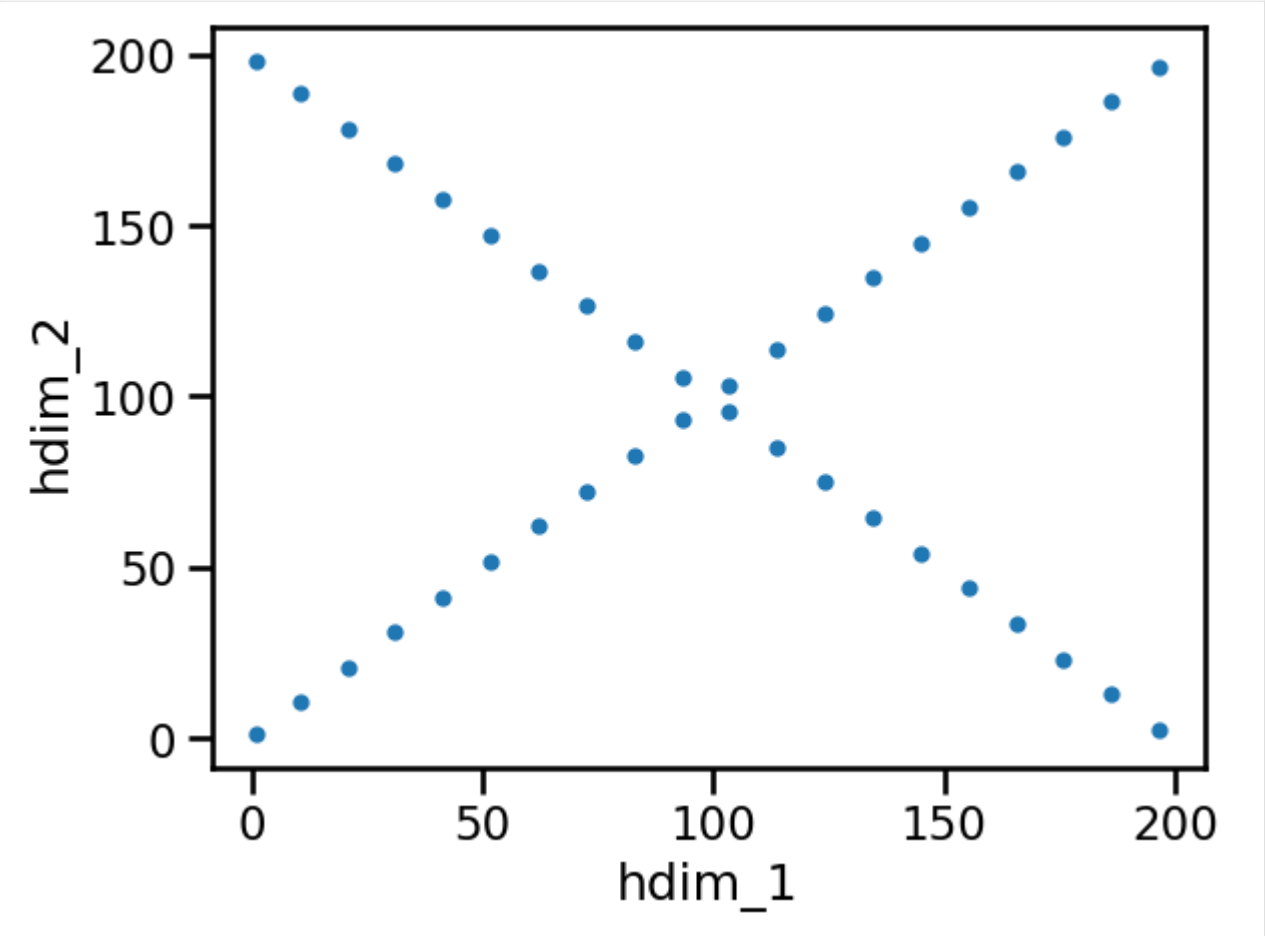
Here the paths of the blobs are clearly visible, but there is an area in the middle where both merge into one feature. This should be avoided. Therefore we try another value for the threshold somewhere in the middle of the available range:

[12]:
```
%%capture
features = tobac.feature_detection_multithreshold(data, dxy, threshold=0.8)
```

[13]:
```
plt.figure(figsize=(10, 6))
features.plot.scatter(x="hdim_1", y="hdim_2")
```

[13]:
```
<Axes: xlabel='hdim_1', ylabel='hdim_2'>
```

```
<Figure size 1000x600 with 0 Axes>
```



This is the picture we wanted to see. This means we can continue working with this set of features.

6.2.3 3. Influence of the tracking method

Now the tracking can be performed. We will create two outputs, one with `method = 'random'`, and the other one with `method = 'predict'`. Since we know what the velocities of our features are beforehand, we can select a reasonable value for `v_max`. Normally this would need to be finetuned.

[14]:
```
%matplotlib inline
v_max = 20

track_1 = tobac.linking_trackpy(
    features, data, dt, dxy, v_max=v_max, method_linking="random"
)

track_2 = tobac.linking_trackpy(
    features, data, dt, dxy, v_max=v_max, method_linking="predict"
)
```

```
Frame 19: 2 trajectories present.
```

[15]:
```
track_1
```

[15]:

	frame	idx	hdim_1	hdim_2	num	threshold_value	feature	time	timestr	projection_x_coordinate
0	0	1	1.000000	1.000000	9	0.8	1	2022-04-01 00:00:00	2022-04-01 00:00:00	502.512563
1	0	2	1.000000	198.000000	9	0.8	2	2022-04-01 00:00:00	2022-04-01 00:00:00	502.512563
2	1	1	10.321429	10.321429	28	0.8	3	2022-04-01 00:08:40	2022-04-01 00:08:40	5186.647523
3	1	2	10.321429	188.678571	28	0.8	4	2022-04-01 00:08:40	2022-04-01 00:08:40	5186.647523
4	2	1	20.678571	20.678571	28	0.8	5	2022-04-01 00:17:20	2022-04-01 00:17:20	10391.241924
5	2	2	20.678571	178.321429	28	0.8	6	2022-04-01 00:17:20	2022-04-01 00:17:20	10391.241924
6	3	1	31.000000	31.000000	25	0.8	7	2022-04-01 00:26:00	2022-04-01 00:26:00	15577.889447
7	3	2	31.000000	168.000000	25	0.8	8	2022-04-01 00:26:00	2022-04-01 00:26:00	15577.889447
9	4	2	41.321429	157.678571	28	0.8	10	2022-04-01 00:34:40	2022-04-01 00:34:40	20764.536971
8	4	1	41.321429	41.321429	28	0.8	9	2022-04-01 00:34:40	2022-04-01 00:34:40	20764.536971
10	5	1	51.678571	51.678571	28	0.8	11			
11	5	2	51.678571	147.321429	28	0.8	12			
12	6	1	62.192308	62.192308	26	0.8	13			
13	6	2	62.192308	136.807692	26	0.8	14			
14	7	1	72.321429	72.321429	28	0.8	15			
15	7	2	72.321429	126.678571	28	0.8	16			
16	8	1	82.678571	82.678571	28	0.8	17			
17	8	2	82.678571	116.321429	28	0.8	18			
19	9	2	93.192308	105.807692	26	0.8	20			
18	9	1	93.192308	93.192308	26	0.8	19			
20	10	1	103.321429	95.678571	28	0.8	21			
21	10	2	103.321429	103.321429	28	0.8	22			
22	11	1	113.807692	85.192308	26	0.8	23			
23	11	2	113.807692	113.807692	26	0.8	24			
24	12	1	124.192308	74.807692	26	0.8	25			
25	12	2	124.192308	124.192308	26	0.8	26			
26	13	1	134.678571	64.321429	28	0.8	27			
27	13	2	134.678571	134.678571	28	0.8	28			
29	14	2	144.807692	144.807692	26	0.8	30			
28	14	1	144.807692	54.192308	26	0.8	29			
30	15	1	155.321429	43.678571	28	0.8	31			
31	15	2	155.321429	155.321429	28	0.8	32			
32	16	1	165.678571	33.321429	28	0.8	33			
33	16	2	165.678571	165.678571	28	0.8	34			
34	17	1	175.916667	23.083333	24	0.8	35			
35	17	2	175.916667	175.916667	24	0.8	36			
36	18	1	186.321429	12.678571	28	0.8	37			
37	18	2	186.321429	186.321429	28	0.8	38			
38	19	1	196.678571	2.321429	28	0.8	39			
39	19	2	196.678571	196.678571	28	0.8	40			

(continues on next page)

(continued from previous page)

```
10  2022-04-01 00:43:20  2022-04-01 00:43:20             25969.131371
11  2022-04-01 00:43:20  2022-04-01 00:43:20             25969.131371
12  2022-04-01 00:52:00  2022-04-01 00:52:00             31252.415926
13  2022-04-01 00:52:00  2022-04-01 00:52:00             31252.415926
14  2022-04-01 01:00:40  2022-04-01 01:00:40             36342.426418
15  2022-04-01 01:00:40  2022-04-01 01:00:40             36342.426418
16  2022-04-01 01:09:20  2022-04-01 01:09:20             41547.020818
17  2022-04-01 01:09:20  2022-04-01 01:09:20             41547.020818
19  2022-04-01 01:18:00  2022-04-01 01:18:00             46830.305373
18  2022-04-01 01:18:00  2022-04-01 01:18:00             46830.305373
20  2022-04-01 01:26:40  2022-04-01 01:26:40             51920.315865
21  2022-04-01 01:26:40  2022-04-01 01:26:40             51920.315865
22  2022-04-01 01:35:20  2022-04-01 01:35:20             57189.795129
23  2022-04-01 01:35:20  2022-04-01 01:35:20             57189.795129
24  2022-04-01 01:44:00  2022-04-01 01:44:00             62408.194820
25  2022-04-01 01:44:00  2022-04-01 01:44:00             62408.194820
26  2022-04-01 01:52:40  2022-04-01 01:52:40             67677.674085
27  2022-04-01 01:52:40  2022-04-01 01:52:40             67677.674085
29  2022-04-01 02:01:20  2022-04-01 02:01:20             72767.684577
28  2022-04-01 02:01:20  2022-04-01 02:01:20             72767.684577
30  2022-04-01 02:10:00  2022-04-01 02:10:00             78050.969131
31  2022-04-01 02:10:00  2022-04-01 02:10:00             78050.969131
32  2022-04-01 02:18:40  2022-04-01 02:18:40             83255.563532
33  2022-04-01 02:18:40  2022-04-01 02:18:40             83255.563532
34  2022-04-01 02:27:20  2022-04-01 02:27:20             88400.335008
35  2022-04-01 02:27:20  2022-04-01 02:27:20             88400.335008
36  2022-04-01 02:36:00  2022-04-01 02:36:00             93628.858579
37  2022-04-01 02:36:00  2022-04-01 02:36:00             93628.858579
38  2022-04-01 02:44:40  2022-04-01 02:44:40             98833.452979
39  2022-04-01 02:44:40  2022-04-01 02:44:40             98833.452979

    projection_y_coordinate   latitude  longitude  cell       time_cell
0                502.512563   0.452261   0.452261     1 0 days 00:00:00
1              99497.487437   0.452261  89.547739     2 0 days 00:00:00
2               5186.647523   4.667983   4.667983     1 0 days 00:08:40
3              94813.352477   4.667983  85.332017     2 0 days 00:08:40
4              10391.241924   9.352118   9.352118     1 0 days 00:17:20
5              89608.758076   9.352118  80.647882     2 0 days 00:17:20
6              15577.889447  14.020101  14.020101     1 0 days 00:26:00
7              84422.110553  14.020101  75.979899     2 0 days 00:26:00
9              79235.463029  18.688083  71.311917     2 0 days 00:34:40
8              20764.536971  18.688083  18.688083     1 0 days 00:34:40
10             25969.131371  23.372218  23.372218     1 0 days 00:43:20
11             74030.868629  23.372218  66.627782     2 0 days 00:43:20
12             31252.415926  28.127174  28.127174     1 0 days 00:52:00
13             68747.584074  28.127174  61.872826     2 0 days 00:52:00
14             36342.426418  32.708184  32.708184     1 0 days 01:00:40
15             63657.573582  32.708184  57.291816     2 0 days 01:00:40
16             41547.020818  37.392319  37.392319     1 0 days 01:09:20
17             58452.979182  37.392319  52.607681     2 0 days 01:09:20
19             53169.694627  42.147275  47.852725     2 0 days 01:18:00
18             46830.305373  42.147275  42.147275     1 0 days 01:18:00
```

(continues on next page)

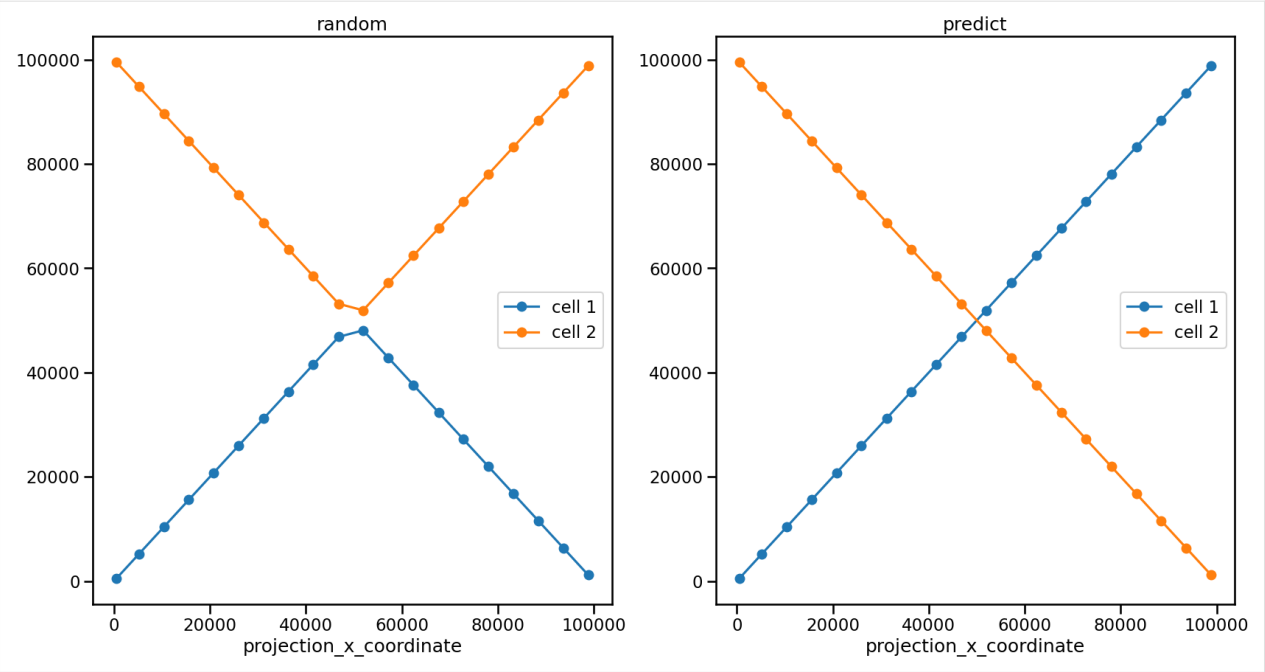
(continued from previous page)

```
20             48079.684135  46.728284  43.271716     1 0 days 01:26:40
21             51920.315865  46.728284  46.728284     2 0 days 01:26:40
22             42810.204871  51.470816  38.529184     1 0 days 01:35:20
23             57189.795129  51.470816  51.470816     2 0 days 01:35:20
24             37591.805180  56.167375  33.832625     1 0 days 01:44:00
25             62408.194820  56.167375  56.167375     2 0 days 01:44:00
26             32322.325915  60.909907  29.090093     1 0 days 01:52:40
27             67677.674085  60.909907  60.909907     2 0 days 01:52:40
29             72767.684577  65.490916  65.490916     2 0 days 02:01:20
28             27232.315423  65.490916  24.509084     1 0 days 02:01:20
30             21949.030869  70.245872  19.754128     1 0 days 02:10:00
31             78050.969131  70.245872  70.245872     2 0 days 02:10:00
32             16744.436468  74.930007  15.069993     1 0 days 02:18:40
33             83255.563532  74.930007  74.930007     2 0 days 02:18:40
34             11599.664992  79.560302  10.439698     1 0 days 02:27:20
35             88400.335008  79.560302  79.560302     2 0 days 02:27:20
36              6371.141421  84.265973   5.734027     1 0 days 02:36:00
37             93628.858579  84.265973  84.265973     2 0 days 02:36:00
38              1166.547021  88.950108   1.049892     1 0 days 02:44:40
39             98833.452979  88.950108  88.950108     2 0 days 02:44:40
```

Let's have a look at the resulting tracks:

[16]:
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(20, 10))

cell_color = [None, "C0", "C1", "C2", "C3"]
for i, cell_track in track_1.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax1,
        marker="o",
        color=cell_color[i],
        label="cell {0}".format(int(i)),
    )
ax1.legend()
ax1.set_title("random")

for i, cell_track in track_2.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax2,
        marker="o",
        color=cell_color[i],
        label="cell {0}".format(int(i)),
    )
ax2.legend()
ax2.set_title("predict")
```

[16]:
```
Text(0.5, 1.0, 'predict')
```



As you can see, there is a clear difference. While in the first link output the feature positions in the top half of the graph are linked into one cell, in the second output the path of the cell follows the actual way of the Gaussian blobs we created. This is possible because `method = "predict"` uses an extraploation to infer the next position from the previous timeframes.

6.2.4 4. Analysis

We know that the second option is “correct”, because we created the data. But can we also decude this by analyzing our tracks?

Let’s calculate the values for the velocities:

[17]:
```
track_1 = tobac.analysis.calculate_velocity(track_1)
track_2 = tobac.analysis.calculate_velocity(track_2)

v1 = track_1.where(track_1["cell"] == 1).dropna().v.values
v2 = track_2.where(track_1["cell"] == 1).dropna().v.values
```

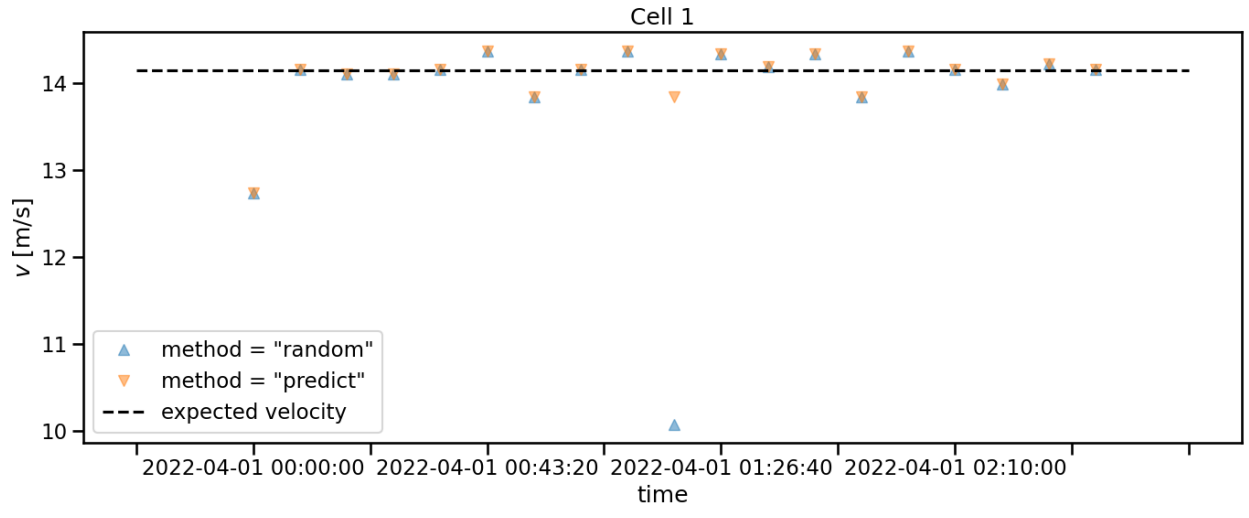
Visualizing these can help us with our investigation:

[18]:
```
fig, (ax) = plt.subplots(figsize=(14, 6))

ax.set_title("Cell 1")

mask_1 = track_1["cell"] == 1
mask_2 = track_2["cell"] == 1

track_1.where(mask_1).dropna().plot(
    x="time",
    y="v",
    ax=ax,
```

(continues on next page)

(continued from previous page)

```
    label='method = "random"',
    marker="^",
    linestyle="",
    alpha=0.5,
)
track_2.where(mask_2).dropna().plot(
    x="time",
    y="v",
    ax=ax,
    label='method = "predict"',
    marker="v",
    linestyle="",
    alpha=0.5,
)

ticks = ax.get_xticks()

plt.hlines(
    [np.sqrt(v_x**2 + v_y**2)],
    ticks.min(),
    ticks.max(),
    color="black",
    label="expected velocity",
    linestyle="--",
)


ax.set_ylabel("$v$ [m/s]")
plt.legend()
plt.tight_layout()
```



The expected velocity is just added for reference. But also without looking at this, we can see that the values for `method = "random"` have an outlier, that deviates far from the other values. This is a clear sign, that this method is not suited well for this case.

6.3 Methods and Parameters for Feature Detection: Part 1

In this notebook, we will take a detailed look at tobac's feature detection and examine some of its parameters. We concentrate on:

- Minima/Maxima and Multiple Thresholds for Feature Identification
- *Feature Position*
- *Sigma Parameter for Smoothing of Noisy Data*
- *Band Pass Filter for Input Fields*

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

%matplotlib inline

import seaborn as sns

sns.set_context("talk")

import warnings

warnings.filterwarnings("ignore")
```

[2]:
```
import tobac
import tobac.testing
```

6.3.1 Minima/Maxima and Multiple Thresholds for Feature Identification

Feature identification search for local maxima in the data.

When working different inputs it is sometimes necessary to switch the feature detection from finding maxima to minima. Furthermore, for more complex datasets containing multiple features differing in magnitude, a categorization according to this magnitude is desirable. Both will be demonstrated with the `make_sample_data_2D_3blobs()` function, which creates such a dataset. For the search for minima we will simply turn the dataset negative:

[3]:
```
data = tobac.testing.make_sample_data_2D_3blobs()
neg_data = -data
```

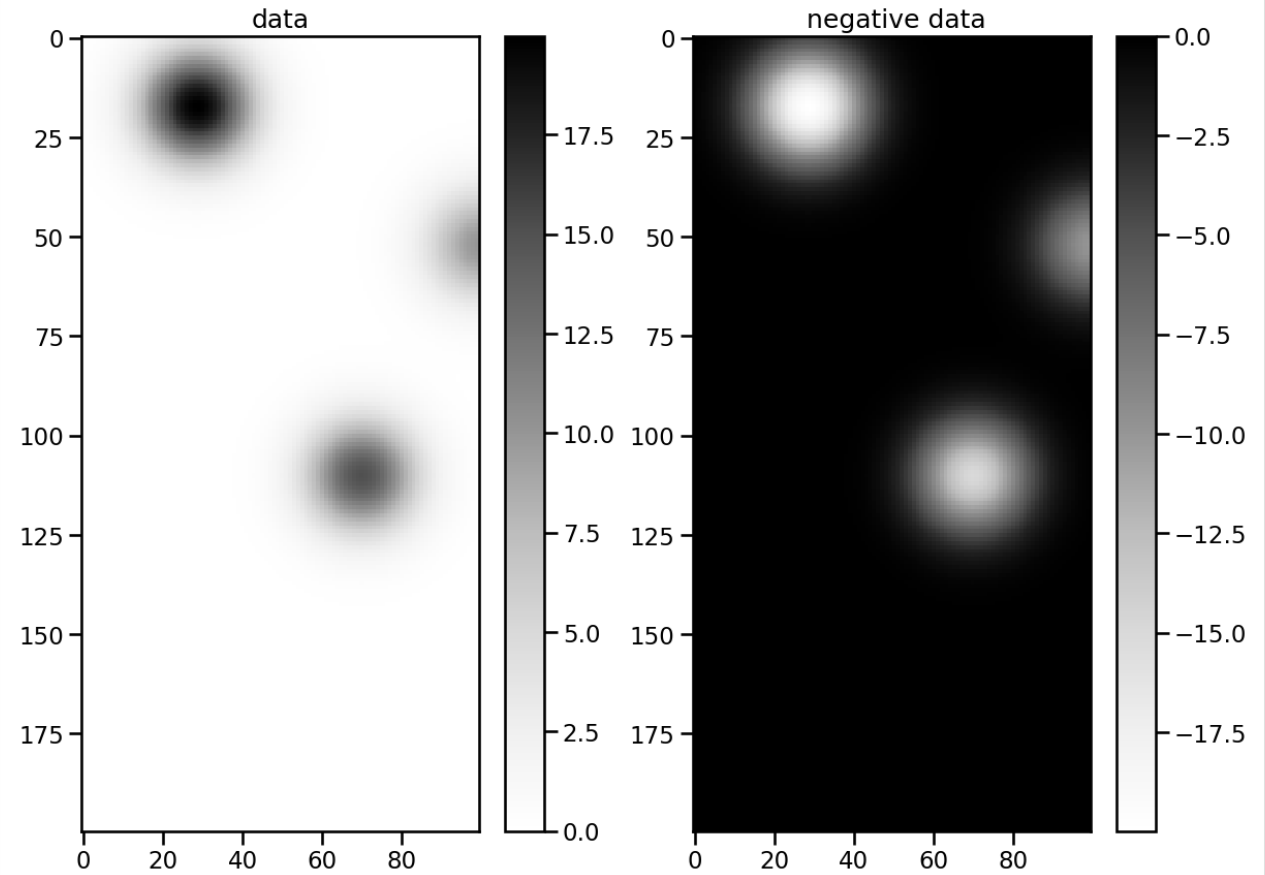
Let us have a look at frame number 50:

[4]:
```
n = 50

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(14, 10))

im1 = ax1.imshow(data.data[50], cmap="Greys")
ax1.set_title("data")
cbar = plt.colorbar(im1, ax=ax1)

im2 = ax2.imshow(neg_data.data[50], cmap="Greys")
```

(continues on next page)

(continued from previous page)

```
ax2.set_title(" negative data")
cbar = plt.colorbar(im2, ax=ax2)
```



As you can see the data has 3 maxima/minima with different extremal values. To capture these, we use list comprehensions to obtain multiple thresholds in the range of the data:

[5]:
```
thresholds = [i for i in range(9, 18)]
neg_thresholds = [-i for i in range(9, 18)]
```

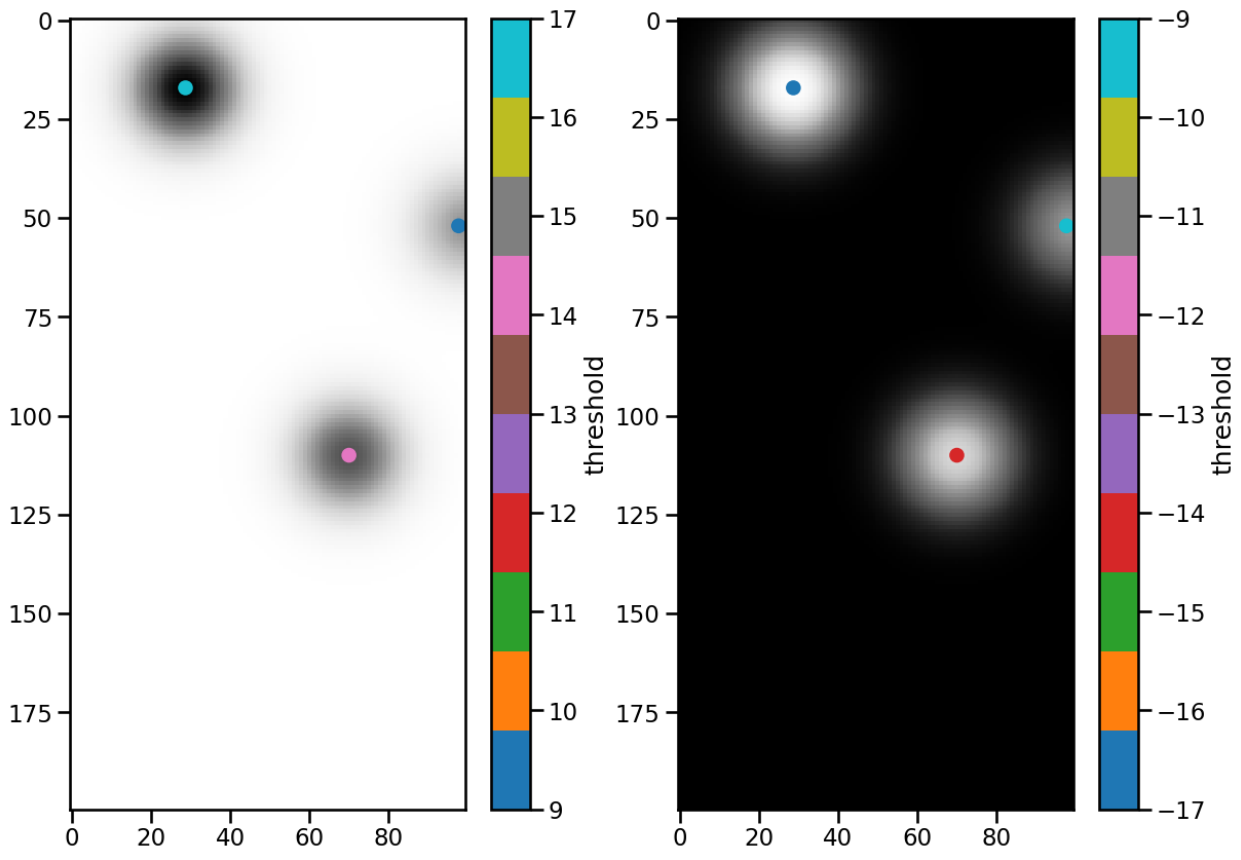
These can now be passed as arguments to `feature_detection_multithreshold()`. With the `target`-keyword we can set a flag whether to search for minima or maxima. The standard is `"maxima"`.

[6]:
```
%%capture

dxy, dt = tobac.utils.get_spacings(data)

features = tobac.feature_detection_multithreshold(
    data, dxy, thresholds, target="maximum"
)
features_inv = tobac.feature_detection_multithreshold(
    neg_data, dxy, neg_thresholds, target="minimum"
)
```

Let's scatter the detected features onto frame 50 of the dataset and create colorbars for the threshold values:

[7]:
```
mask_1 = features["frame"] == n
mask_2 = features_inv["frame"] == n

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(14, 10))

ax1.imshow(data.data[50], cmap="Greys")
im1 = ax1.scatter(
    features.where(mask_1)["hdim_2"],
    features.where(mask_1)["hdim_1"],
    c=features.where(mask_1)["threshold_value"],
    cmap="tab10",
)
cbar = plt.colorbar(im1, ax=ax1)
cbar.ax.set_ylabel("threshold")

ax2.imshow(neg_data.data[50], cmap="Greys")
im2 = ax2.scatter(
    features_inv.where(mask_2)["hdim_2"],
    features_inv.where(mask_2)["hdim_1"],
    c=features_inv.where(mask_2)["threshold_value"],
    cmap="tab10",
)
cbar = plt.colorbar(im2, ax=ax2)
cbar.ax.set_ylabel("threshold")
```

[7]:
```
Text(0, 0.5, 'threshold')
```



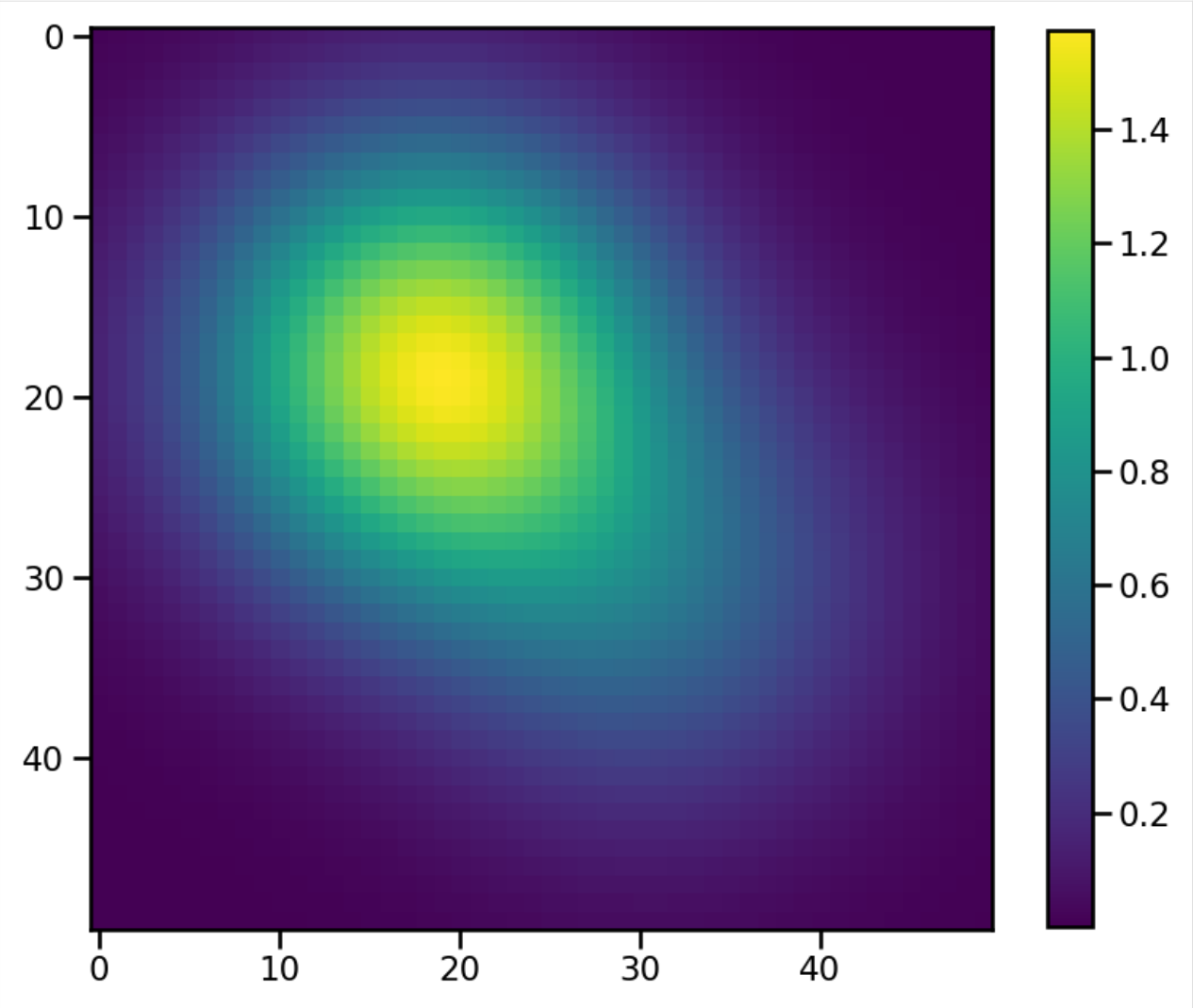
The three features were found in both data sets, and the color bars indicate which threshold they belong to. When using multiple thresholds, note that the order of the list is important. Each feature is assigned the threshold value that was reached last. Therefore, it makes sense to start with the lowest value in case of maxima.

6.3.2 Feature Position

To explore the influence of the `position_threshold` flag we need a radially asymmetric feature. Let's create a simple one by adding two 2d-gaussians and add an extra dimension for the time, which is required for working with tobac:

[8]:
```
x = np.linspace(-2, 2)
y = np.linspace(-2, 2)
xx, yy = np.meshgrid(x, y)

exp1 = 1.5 * np.exp(-((xx + 0.5) ** 2 + (yy + 0.5) ** 2))
exp2 = 0.5 * np.exp(-((0.5 - xx) ** 2 + (0.5 - yy) ** 2))

asymmetric_data = np.expand_dims(exp1 + exp2, axis=0)

plt.figure(figsize=(10, 10))
plt.imshow(asymmetric_data[0])
plt.colorbar(shrink=0.8)
```

[8]:
```
<matplotlib.colorbar.Colorbar at 0x1377ed290>
```



To feed this data into the feature detection we need to convert it into an `xarray.DataArray`. Before we do that we select an arbitrary time and date for the single frame of our synthetic field:

[9]:
```
date = np.datetime64(
    "2022-04-01T00:00",
)
assym = xr.DataArray(
    data=asymmetric_data, coords={"time": np.expand_dims(date, axis=0), "y": y, "x": x}
)
assym
```

[9]:
```
<xarray.DataArray (time: 1, y: 50, x: 50)> Size: 20kB
array([[[0.01666536, 0.02114886, 0.02648334, ..., 0.00083392,
         0.00058509, 0.00040694],
        [0.02114886, 0.02683866, 0.03360844, ..., 0.00109464,
         0.00077154, 0.0005393 ],
        [0.02648334, 0.03360844, 0.04208603, ..., 0.00142435,
         0.00100896, 0.00070907],
        ...,
```

(continues on next page)

(continued from previous page)

```
        [0.00083392, 0.00109464, 0.00142435, ..., 0.01405279,
         0.01121917, 0.00883872],
        [0.00058509, 0.00077154, 0.00100896, ..., 0.01121917,
         0.00895731, 0.00705704],
        [0.00040694, 0.0005393 , 0.00070907, ..., 0.00883872,
         0.00705704, 0.00556009]]])
Coordinates:
  * time     (time) datetime64[ns] 8B 2022-04-01
  * y        (y) float64 400B -2.0 -1.918 -1.837 -1.755 ... 1.837 1.918 2.0
  * x        (x) float64 400B -2.0 -1.918 -1.837 -1.755 ... 1.837 1.918 2.0
```

Since we do not have a dt in this dataset, we can not use the `get_spacings()`-utility this time and need to calculate the `dxy` spacing manually:

[10]:
```
dxy = assym.diff("x")
```

Finally, we choose a threshold in the datarange and apply the feature detection with the four `position_threshold` flags - 'center' - 'extreme' - 'weighted_diff' - 'weighted_abs'

[11]:
```
%%capture

threshold = 0.2
features_center = tobac.feature_detection_multithreshold(
    assym, dxy, threshold, position_threshold="center"
)
features_extreme = tobac.feature_detection_multithreshold(
    assym, dxy, threshold, position_threshold="extreme"
)
features_diff = tobac.feature_detection_multithreshold(
    assym, dxy, threshold, position_threshold="weighted_diff"
)
features_abs = tobac.feature_detection_multithreshold(
    assym, dxy, threshold, position_threshold="weighted_abs"
)
```

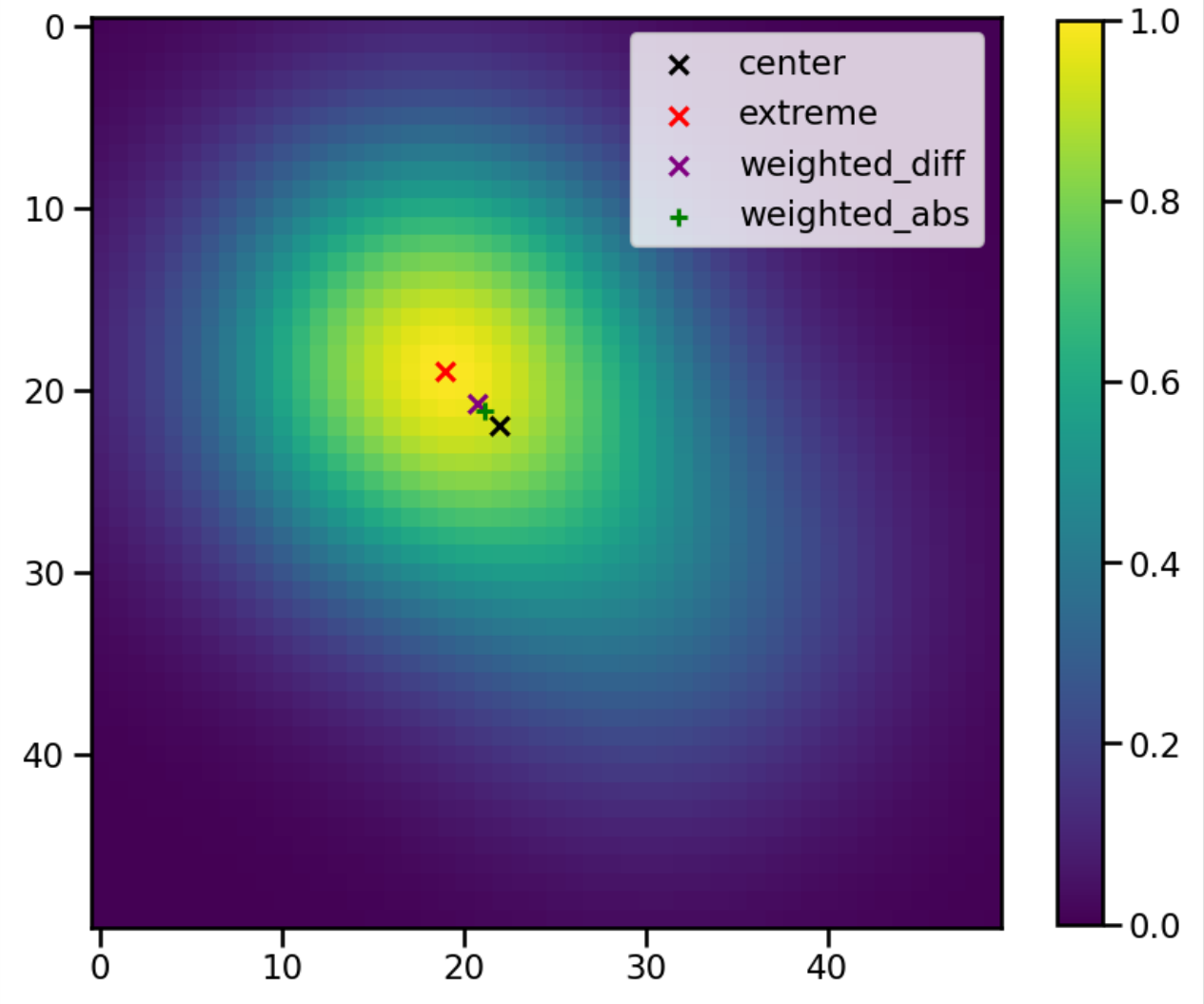
[12]:
```
plt.figure(figsize=(10, 10))
plt.imshow(assym[0])
plt.scatter(
    features_center["hdim_2"],
    features_center["hdim_1"],
    color="black",
    marker="x",
    label="center",
)
plt.scatter(
    features_extreme["hdim_2"],
    features_extreme["hdim_1"],
    color="red",
    marker="x",
    label="extreme",
)
plt.scatter(
```

(continues on next page)

(continued from previous page)

```
    features_diff["hdim_2"],
    features_diff["hdim_1"],
    color="purple",
    marker="x",
    label="weighted_diff",
)
plt.scatter(
    features_abs["hdim_2"],
    features_abs["hdim_1"],
    color="green",
    marker="+",
    label="weighted_abs",
)
plt.colorbar(shrink=0.8)
plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x1377f2b10>
```



As you can see this parameter specifies how the postion of the feature is defined. These are the descriptions given in

the code:

- extreme: get position as max/min position inside the identified region
- center : get position as geometrical centre of identified region
- weighted_diff: get position as centre of identified region, weighted by difference from the threshold
- weighted_abs: get position as centre of identified region, weighted by absolute values if the field

6.3.3 Sigma Parameter for Smoothing of Noisy Data

Before the features are searched a gausian filter is applied to the data in order to smooth it. So let's import the filter used by tobac for a demonstration:

[13]:
```
from scipy.ndimage import gaussian_filter
```

This filter works performing a convolution of a (in our case 2-dimensional) gaussian function

$$h(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

with our data and with this parameter we set the value of σ.

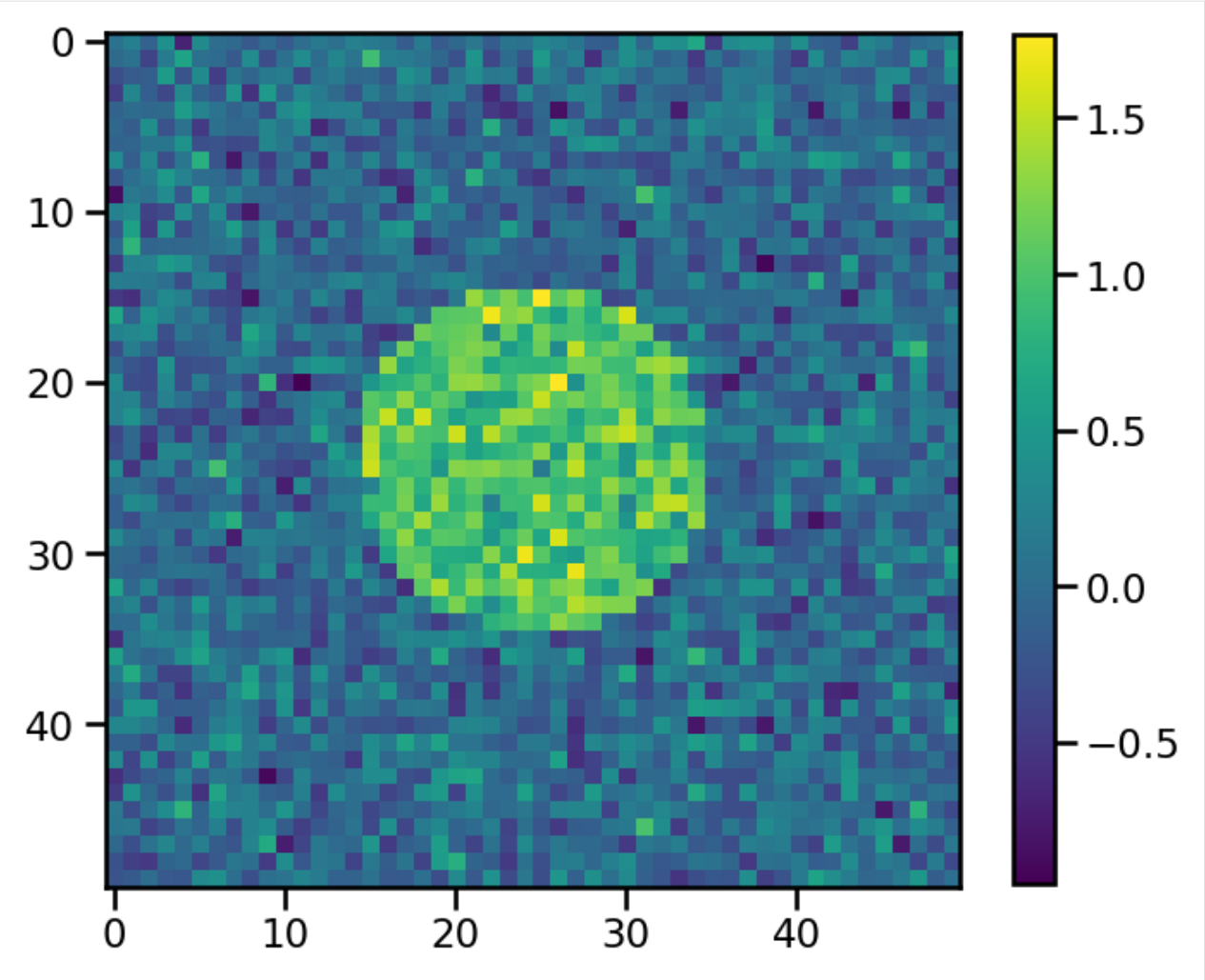
The effect of this filter can best be demonstrated on very sharp edges in the input. Therefore we create an array from a boolean mask of another 2d-Gaussian, which has only values of 0 or 1:

[14]:
```
x = np.linspace(-2, 2)
y = np.linspace(-2, 2)
xx, yy = np.meshgrid(x, y)

exp = np.exp(-(xx**2 + yy**2))

gaussian_data = np.expand_dims(exp, axis=0)
```

and we add some random noise to it:

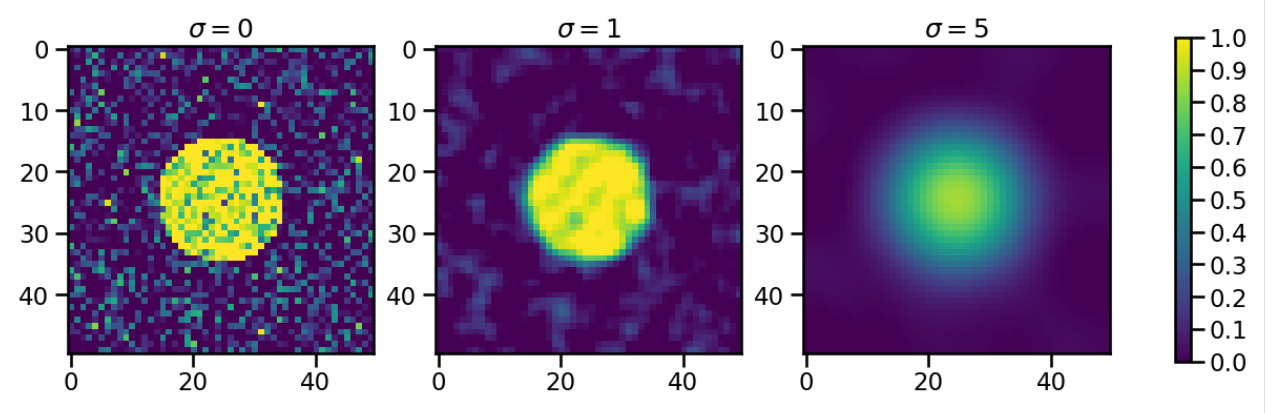
[15]:
```
np.random.seed(54321)

noise = 0.3 * np.random.randn(*gaussian_data.shape)
data_sharp = np.array(gaussian_data > 0.5, dtype="float32")

data_sharp += noise

plt.figure(figsize=(8, 8))
plt.imshow(data_sharp[0])
plt.colorbar(shrink=0.8)
```

[15]:
```
<matplotlib.colorbar.Colorbar at 0x137d40250>
```



If we apply this filter to the data with increasing sigmas, increasingly smoothed data will be the result:

[16]:
```
non_smooth_data = gaussian_filter(data_sharp, sigma=0)
smooth_data = gaussian_filter(data_sharp, sigma=1)
smoother_data = gaussian_filter(data_sharp, sigma=5)

fig, axes = plt.subplots(ncols=3, figsize=(16, 10))

im0 = axes[0].imshow(non_smooth_data[0], vmin=0, vmax=1)
axes[0].set_title(r"$\sigma = 0$")

im1 = axes[1].imshow(smooth_data[0], vmin=0, vmax=1)
axes[1].set_title(r"$\sigma = 1$")

im2 = axes[2].imshow(smoother_data[0], vmin=0, vmax=1)
axes[2].set_title(r"$\sigma = 5$")

cbar = fig.colorbar(im1, ax=axes.tolist(), shrink=0.4)
cbar.set_ticks(np.linspace(0, 1, 11))
```



This is what happens in the background, when the `feature_detection_multithreshold()` function is called. The default value of `sigma_threshold` is 0.5. The next step is trying to detect features of the dataset with these `sigma_threshold` values. We first need an xarray DataArray again:

[17]:
```
date = np.datetime64("2022-04-01T00:00")
input_data = xr.DataArray(
    data=data_sharp, coords={"time": np.expand_dims(date, axis=0), "y": y, "x": x}
)
```

Now we set a threshold and detect the features:

[18]:
```
%%capture

threshold = 0.9
features_sharp = tobac.feature_detection_multithreshold(
    input_data, dxy, threshold, sigma_threshold=0
)
features_smooth = tobac.feature_detection_multithreshold(
    input_data, dxy, threshold, sigma_threshold=1
)
features_smoother = tobac.feature_detection_multithreshold(
    input_data, dxy, threshold, sigma_threshold=5
)
```

Attempting to plot the results

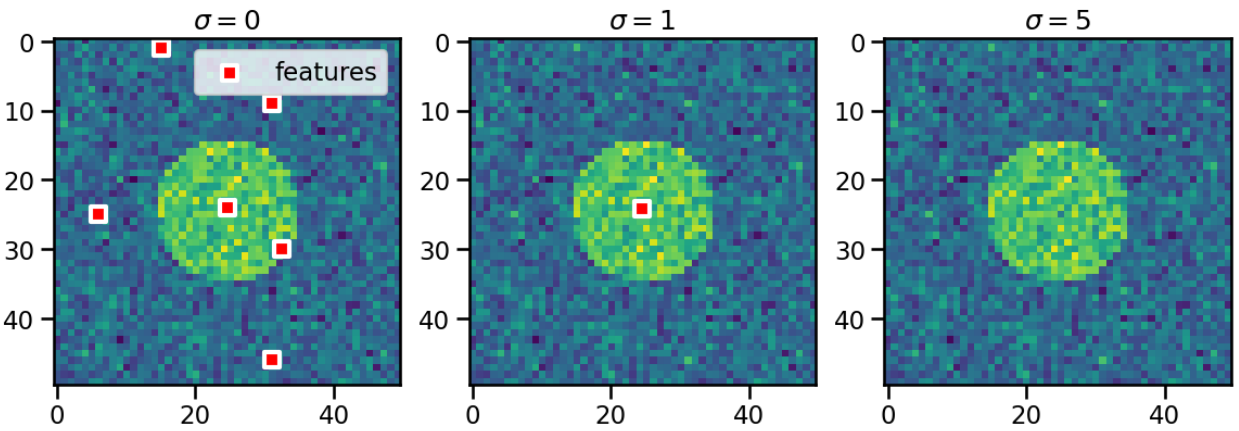
[19]:
```
fig, axes = plt.subplots(ncols=3, figsize=(14, 10))
plot_kws = dict(
    color="red",
    marker="s",
    s=100,
    edgecolors="w",
    linewidth=3,
)

im0 = axes[0].imshow(input_data[0])
axes[0].set_title(r"$\sigma = 0$")
axes[0].scatter(
    features_sharp["hdim_2"], features_sharp["hdim_1"], label="features", **plot_kws
)
```

(continues on next page)

(continued from previous page)

```
axes[0].legend()

im0 = axes[1].imshow(input_data[0])
axes[1].set_title(r"$\sigma = 1$")
axes[1].scatter(features_smooth["hdim_2"], features_smooth["hdim_1"], **plot_kws)

im0 = axes[2].imshow(input_data[0])
axes[2].set_title(r"$\sigma = 5$")
try:
    axes[2].scatter(
        features_smoother["hdim_2"], features_smoother["hdim_1"], **plot_kws
    )
except:
    print("WARNING: No Feature Detected!")
```

```
WARNING: No Feature Detected!
```



Noise may cause some false detections (left panel) that are significantly reduced when a suitable smoothing parameter is chosen (middle panel).

6.3.4 Band-Pass Filter for Input Fields via Parameter `wavelength_filtering`

This parameter can be understood best, when looking at real instead of snythethic data. An example of usage is given *here*

6.4 Methods and Parameters for Feature Detection: Part 2

In this notebook, we will contninue to look in detail at tobac's feature detection and examine the remaining parameters.

We will treat:

- *Object Erosion Parameter*
- *Minimum Object Pair Distance*

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import xarray as xr

%matplotlib inline

import seaborn as sns

sns.set_context("talk")


import warnings

warnings.filterwarnings("ignore")
```

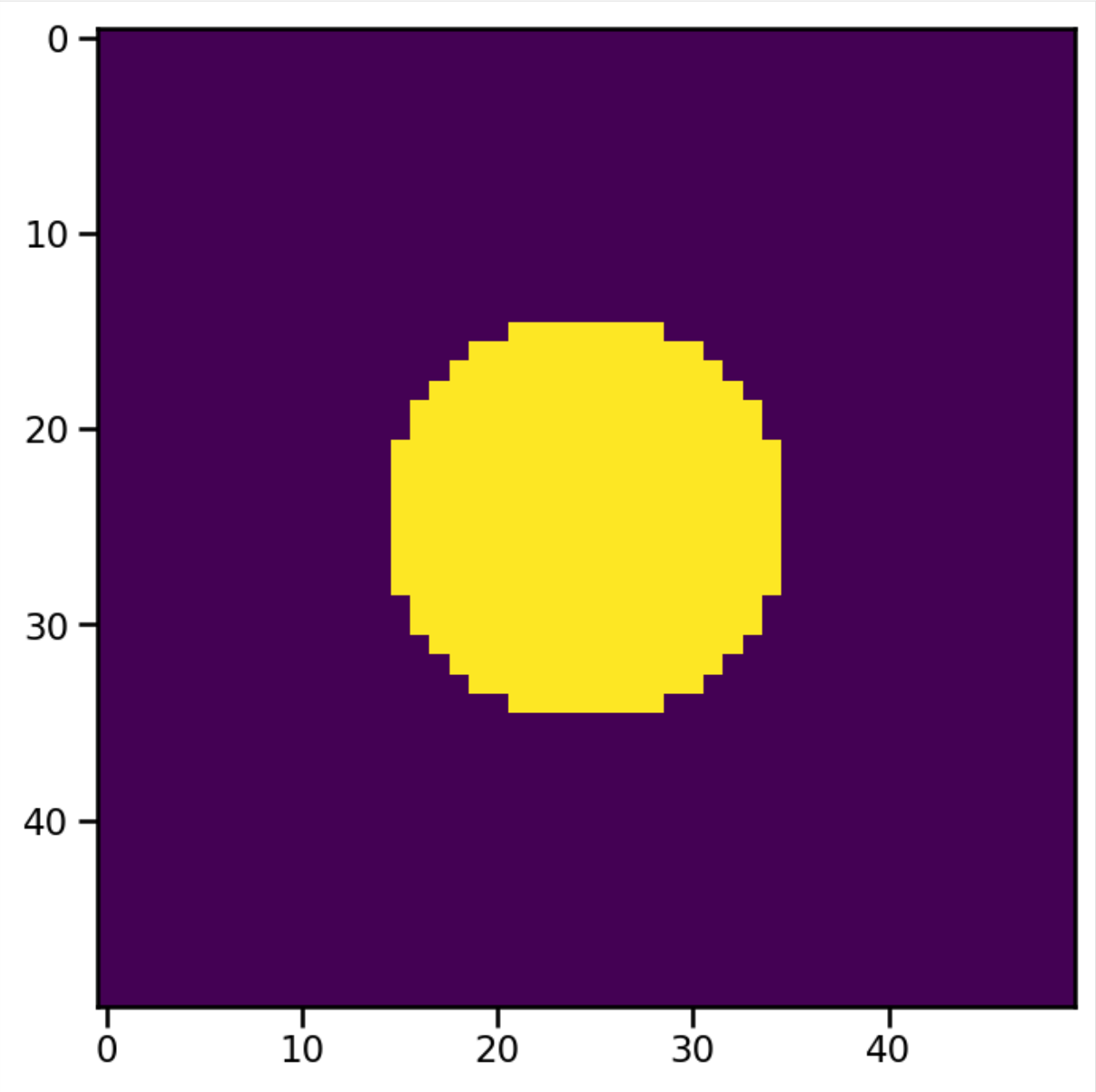
[2]:
```
import tobac
import tobac.testing
```

6.4.1 Object Erosion Parameter `n_erosion_threshold`

To understand this parameter we have to look at one varibale of the feature-Datasets we did not mention so far: *num*

The value of *num* for a specific feature tells us the number of datapoints exceeding the threshold. *n_erosion_threshold* reduces this number by eroding the mask of the feature on its boundary. Supose we are looking at the gaussian data again and we set a treshold of 0.5. The resulting mask of our feature will look like this:

[3]:
```
x = np.linspace(-2, 2)
y = np.linspace(-2, 2)
xx, yy = np.meshgrid(x, y)

exp = np.exp(-(xx**2 + yy**2))

gaussian_data = np.expand_dims(exp, axis=0)
threshold = 0.5

mask = gaussian_data > threshold
mask = mask[0]

plt.figure(figsize=(8, 8))
plt.imshow(mask)
plt.show()
```



The erosion algorithm used by tobac is imported from *skimage.morphology*:

[4]:
```
from skimage.morphology import binary_erosion
```

Applying this algorithm requires a quadratic matrix. The size of this matrix is provided by the *n_erosion_threshold* parameter. For a quick demonstration we can create the matrix by hand and apply the erosion for different values:

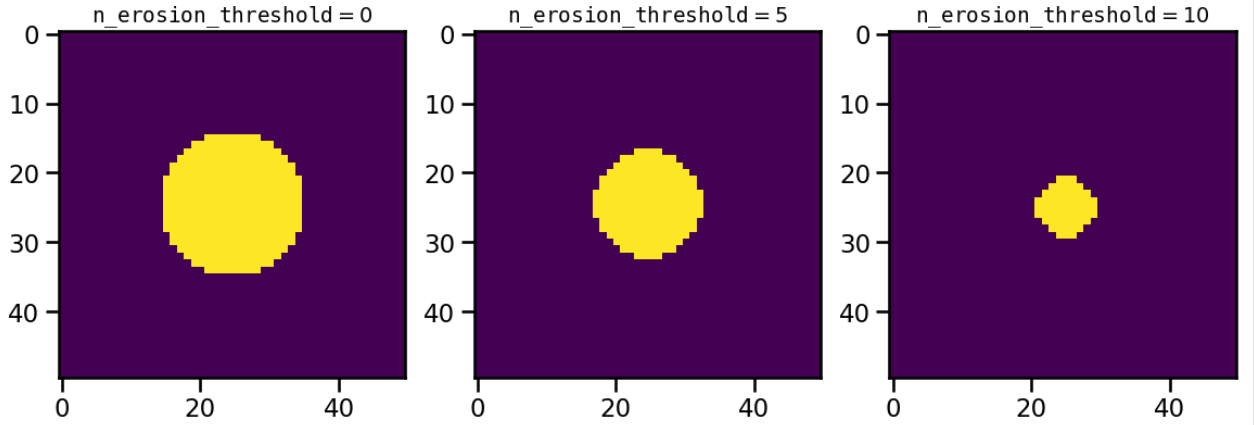
[5]:
```
fig, axes = plt.subplots(ncols=3, figsize=(14, 10))

im0 = axes[0].imshow(mask)
axes[0].set_title(r"$\mathtt{n\_erosion\_threshold} = 0$", fontsize=14)

n_erosion_threshold = 5
```

(continues on next page)

(continued from previous page)

```
selem = np.ones((n_erosion_threshold, n_erosion_threshold))
mask_er = binary_erosion(mask, selem).astype(np.int64)

im1 = axes[1].imshow(mask_er)
axes[1].set_title(r"$\mathtt{n\_erosion\_threshold} = 5$", fontsize=14)

n_erosion_threshold = 10
selem = np.ones((n_erosion_threshold, n_erosion_threshold))
mask_er_more = binary_erosion(mask, selem).astype(np.int64)

im2 = axes[2].imshow(mask_er_more)
axes[2].set_title(r"$\mathtt{n\_erosion\_threshold} = 10$", fontsize=14)
```

[5]:
```
Text(0.5, 1.0, '$\\mathtt{n\\_erosion\\_threshold} = 10$')
```



This means by using increasing values of *n_erosion_threshold* for a feature detection we will get lower values of *num*, which will match the number of **True**-values in the masks above:

[6]:
```
%%capture

date = np.datetime64("2022-04-01T00:00")
input_data = xr.DataArray(
    data=gaussian_data, coords={"time": np.expand_dims(date, axis=0), "y": y, "x": x}
)
dxy = input_data["x"][1] - input_data["x"][0]

features = tobac.feature_detection_multithreshold(
    input_data, dxy, threshold, n_erosion_threshold=0
)
features_eroded = tobac.feature_detection_multithreshold(
    input_data, dxy, threshold, n_erosion_threshold=5
)
features_eroded_more = tobac.feature_detection_multithreshold(
    input_data, dxy, threshold, n_erosion_threshold=10
)
```

[7]:
```
features["num"][0]
```

[7]:
```
332
```

[8]:
```
mask.sum()
```
[8]:
```
332
```

[9]:
```
features_eroded["num"][0]
```
[9]:
```
188
```

[10]:
```
mask_er.sum()
```
[10]:
```
188
```

[11]:
```
features_eroded_more["num"][0]
```
[11]:
```
57
```

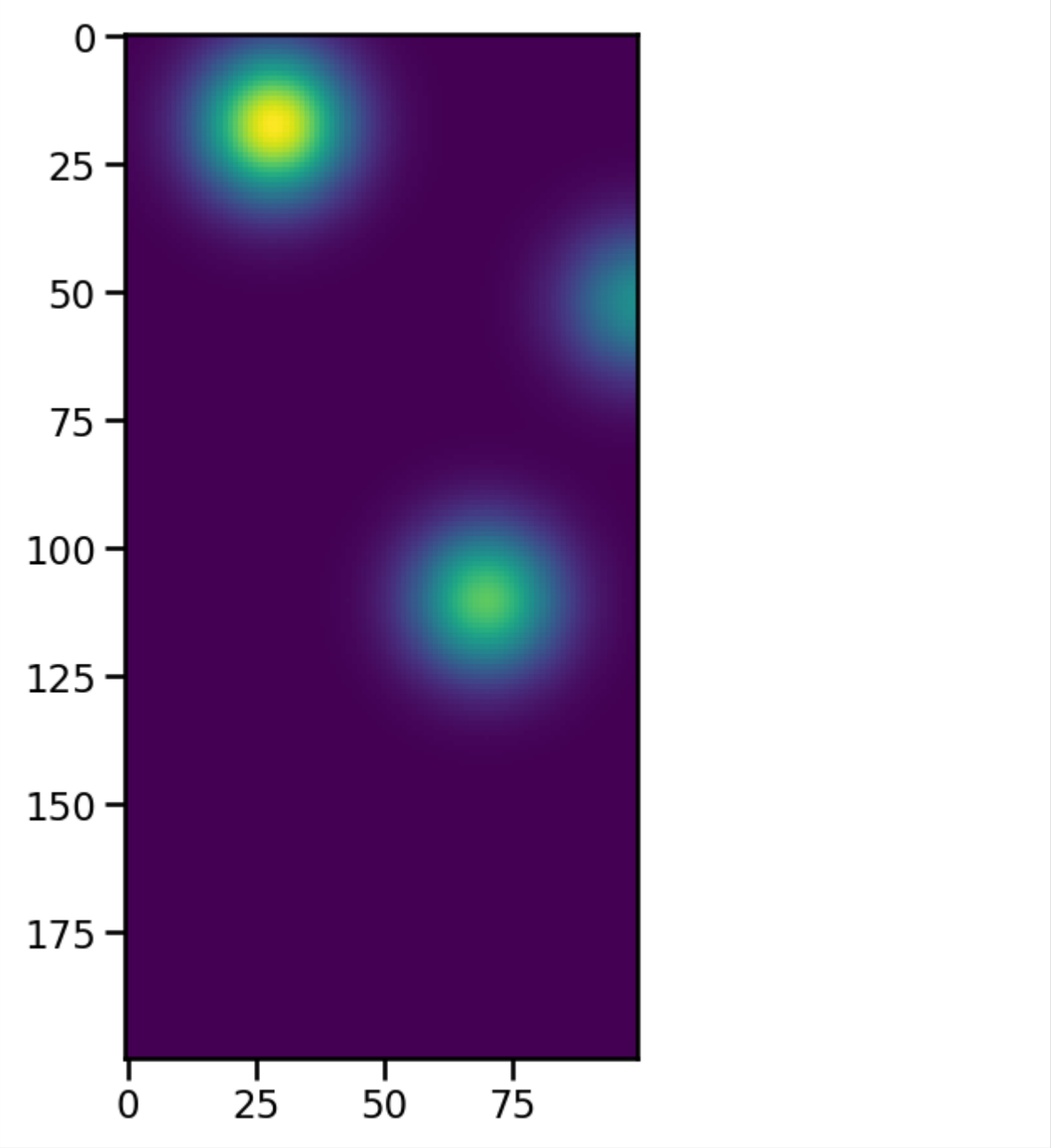
[12]:
```
mask_er_more.sum()
```
[12]:
```
57
```

This can be used to simplify the geometry of complex features.

6.4.2 Minimum Object Size Parameter `n_min_threshold`

With *n_min_threshold* parameter we can exclude smaller features by setting a minimum of datapoints that have to exceed the threshold for one feature. If we again detect the three blobs and check their *num* value at frame 50, we can see that one of them contains fewer pixels.

[13]:
```
data = tobac.testing.make_sample_data_2D_3blobs(data_type="xarray")

plt.figure(figsize=(8, 8))
plt.imshow(data[50])
plt.show()
```



[14]:
```
%%capture
threshold = 9
dxy, dt = tobac.utils.get_spacings(data)
features = tobac.feature_detection_multithreshold(data, dxy, threshold)
```

[15]:
```
features.where(features["frame"] == 50)["num"].dropna()
```

[15]:
```
60    501.0
61     30.0
```

(continues on next page)

(continued from previous page)

```
62    325.0
Name: num, dtype: float64
```

Obviously, the feature with only 30 datapoints is the rightmost feature that has almost left the imaging area. If we now use an *n_min-threshold* above or equal to 30, we will not detect this small feature:

[16]:
```
%%capture
features = tobac.feature_detection_multithreshold(
    data, dxy, threshold, n_min_threshold=30
)
```

[17]:
```
features.where(features["frame"] == 50)["num"].dropna()
```

[17]:
```
60    501.0
61    325.0
Name: num, dtype: float64
```

Noisy data can be cleaned using this method by ignoring smaller features of no significance or, as here, features that leave the detection range.

6.4.3 Minimum Object Pair Distance `min_distance`

Another way of getting rid of this specific feature is the *min_distance* parameter. It sets a minimal distance of our features. Lets detect the features as before:

[18]:
```
%%capture

data = tobac.testing.make_sample_data_2D_3blobs(data_type="xarray")

threshold = 9
dxy, dt = tobac.utils.get_spacings(data)
features = tobac.feature_detection_multithreshold(data, dxy, threshold)
```

A quick look at frame 50 tells us the found features and their indices:

[19]:
```
n = 50
mask = features["frame"] == n
features_frame = features.where(mask).dropna()
features_frame.index.values
```

[19]:
```
array([60, 61, 62])
```

Notice that *to_dataframe()* was used to convert the Dataset to a pandas dataframe, which is required to use the *calculate_distance* function of the analysis module. The distances bewteen our features are:

[20]:
```
tobac.analysis.calculate_distance(
    features_frame.loc[60], features_frame.loc[61], method_distance="xy"
)
```

[20]:
```
77307.67873610597
```

[21]:
```
tobac.analysis.calculate_distance(
    features_frame.loc[62], features_frame.loc[61], method_distance="xy"
)
```

[21]:
```
64289.48419281164
```

[22]:
```
tobac.analysis.calculate_distance(
    features_frame.loc[62], features_frame.loc[60], method_distance="xy"
)
```

[22]:
```
101607.57596570993
```

With this knowledge we can set reasonable values for *min_distance* to exclude the small feature:

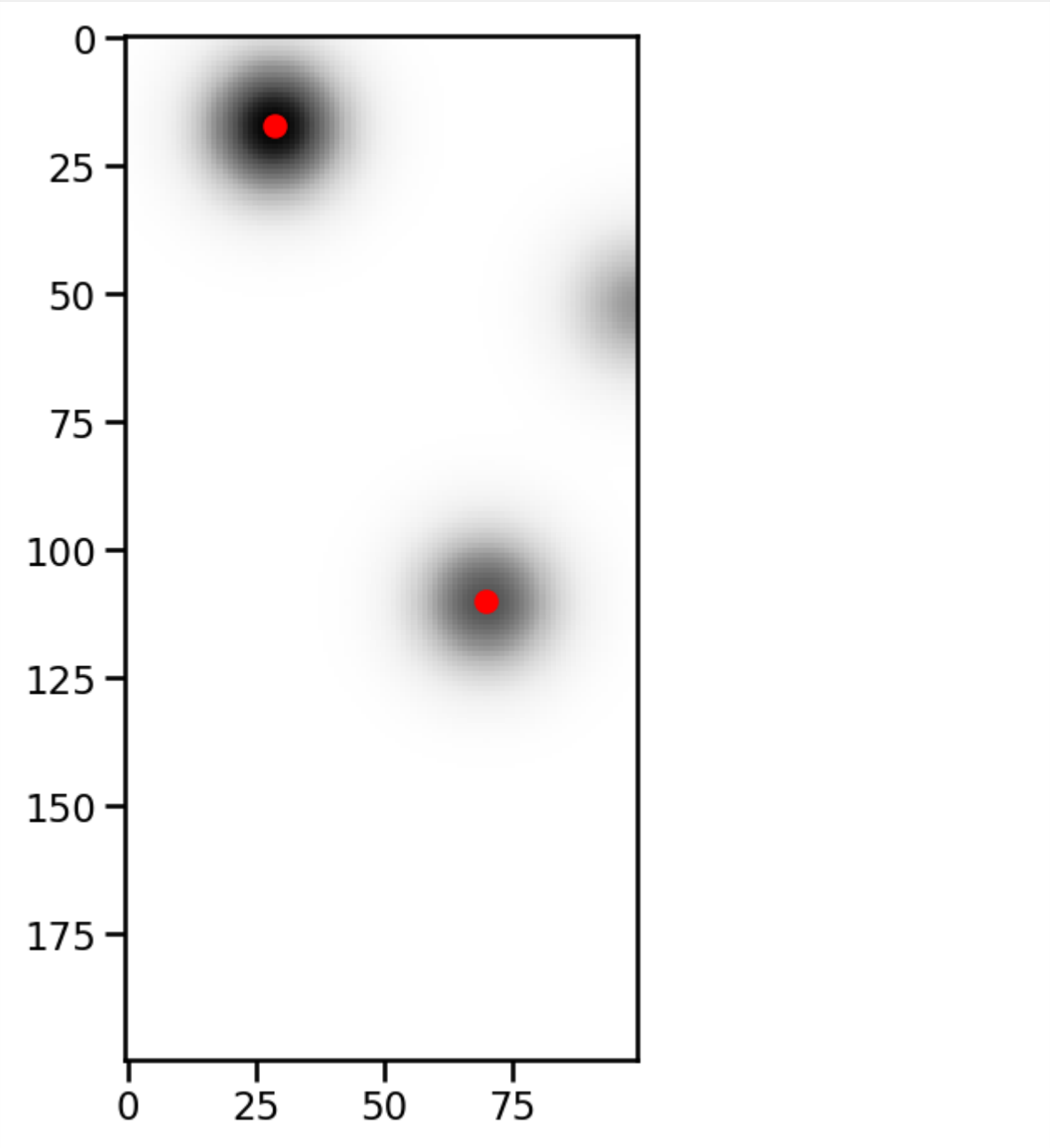
[23]:
```
min_distance = 70000
```

and perform the feature detection as usual:

[24]:
```
%%capture

data = tobac.testing.make_sample_data_2D_3blobs(data_type="xarray")

thresholds = [10]
dxy, dt = tobac.utils.get_spacings(data)
features = tobac.feature_detection_multithreshold(
    data, dxy, thresholds, min_distance=min_distance
)

n = 50
mask = features["frame"] == n
```

Plotting the result shows us that we now exclude the expected feature.

[25]:
```
fig, ax1 = plt.subplots(ncols=1, figsize=(8, 8))
ax1.imshow(data.data[50], cmap="Greys")

im1 = ax1.scatter(
    features.where(mask)["hdim_2"], features.where(mask)["hdim_1"], color="red"
)
```



If the features have the same threshold, tobac keeps the feature with the larger area. Otherwise the feature with the higher treshold is kept.

6.5 Methods and Parameters for Segmentation

This notebook explores the segmentation function of `tobac` and its parameters:

- *Required Inputs*
- *Different Thresholds*
- *Choosing Method and Target*
- *Setting a maximum Distance*
- *Handling 3d-Data*

We start with the usual imports:

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

%matplotlib inline

import seaborn as sns

sns.set_context("talk")

import warnings

warnings.filterwarnings("ignore")
```

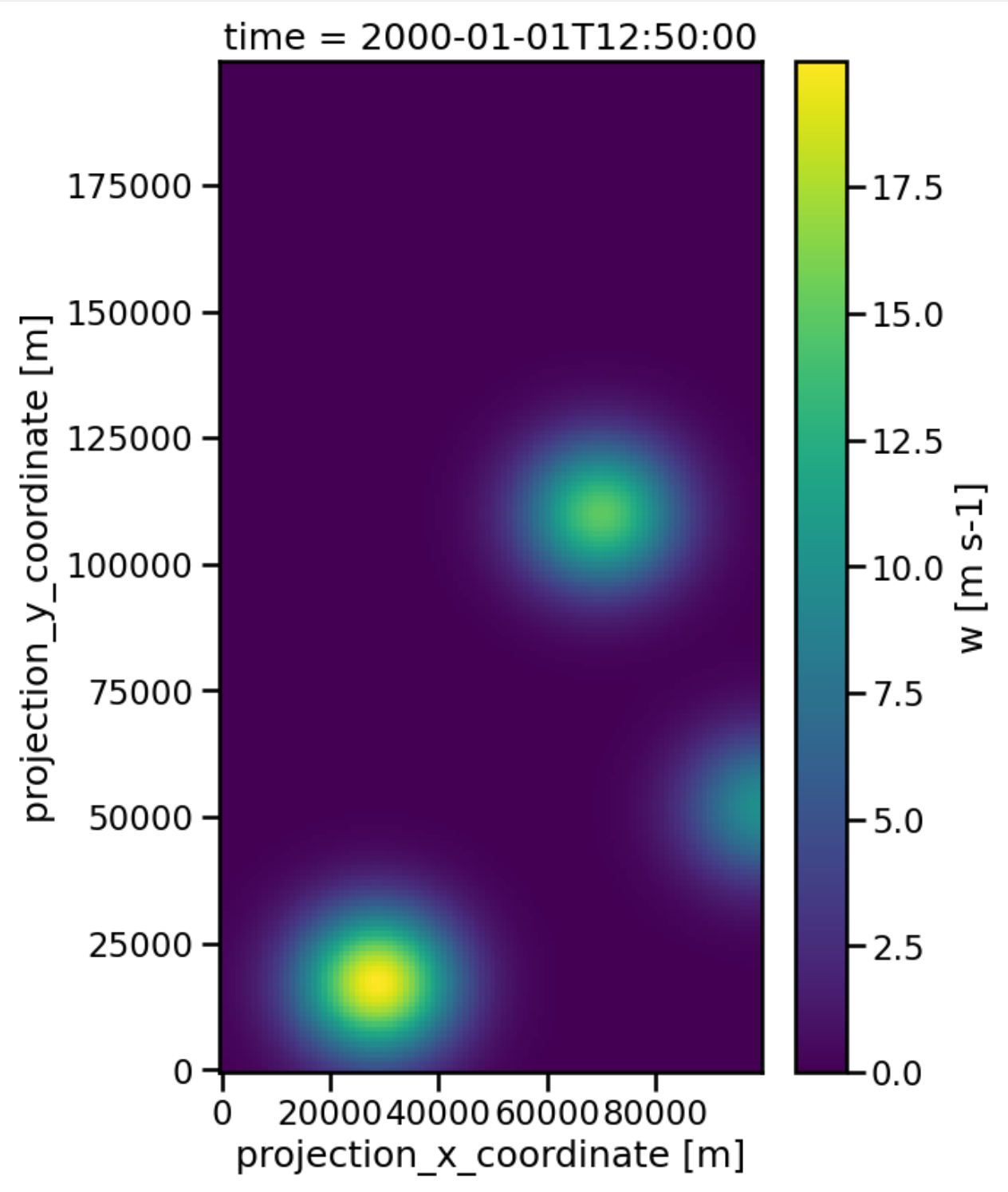
[2]:
```
import tobac
import tobac.testing
```

6.5.1 Required Inputs

To perform a segmentation we need a dataset with already detected features. Therefore, we take advantage of the `testing.make_sample_data_2D_3blobs_inv()`-utility and detect features with different thresholds:

[3]:
```
data = tobac.testing.make_sample_data_2D_3blobs_inv(data_type="xarray")
dxy, dt = tobac.utils.get_spacings(data)

plt.figure(figsize=(6, 9))
data.isel(time=50).plot(x="x", y="y")
```

[3]:
```
<matplotlib.collections.QuadMesh at 0x13d3275d0>
```



[4]:
```
%%capture
thresholds = [9, 14, 17]
features = tobac.feature_detection_multithreshold(
    data, dxy, thresholds, position_threshold="weighted_abs"
)
```

The resulting dataset can now be used as argument for the `segmentation()`-function. The other required inputs are the original dataset, the spacing and a threshold.

[5]:
```
mask, features_mask = tobac.segmentation_2D(features, data, dxy, threshold=9)
```

The created segments are provided as mask, which is the first returned object of the function. The second output is the features-dataset again, but with the additional *ncells*-variable, which gives us the number of datapoints belonging to the feature:

[6]:
```
features_mask["ncells"][1]
```

[6]:
```
67.0
```

Notice that this number can be deviate from the *num*-value, because watershedding works differently from just detecting the values exceeeding the threshold. For example, for the second feature *ncells* contains one additional datapoint compared to the original feature detection:

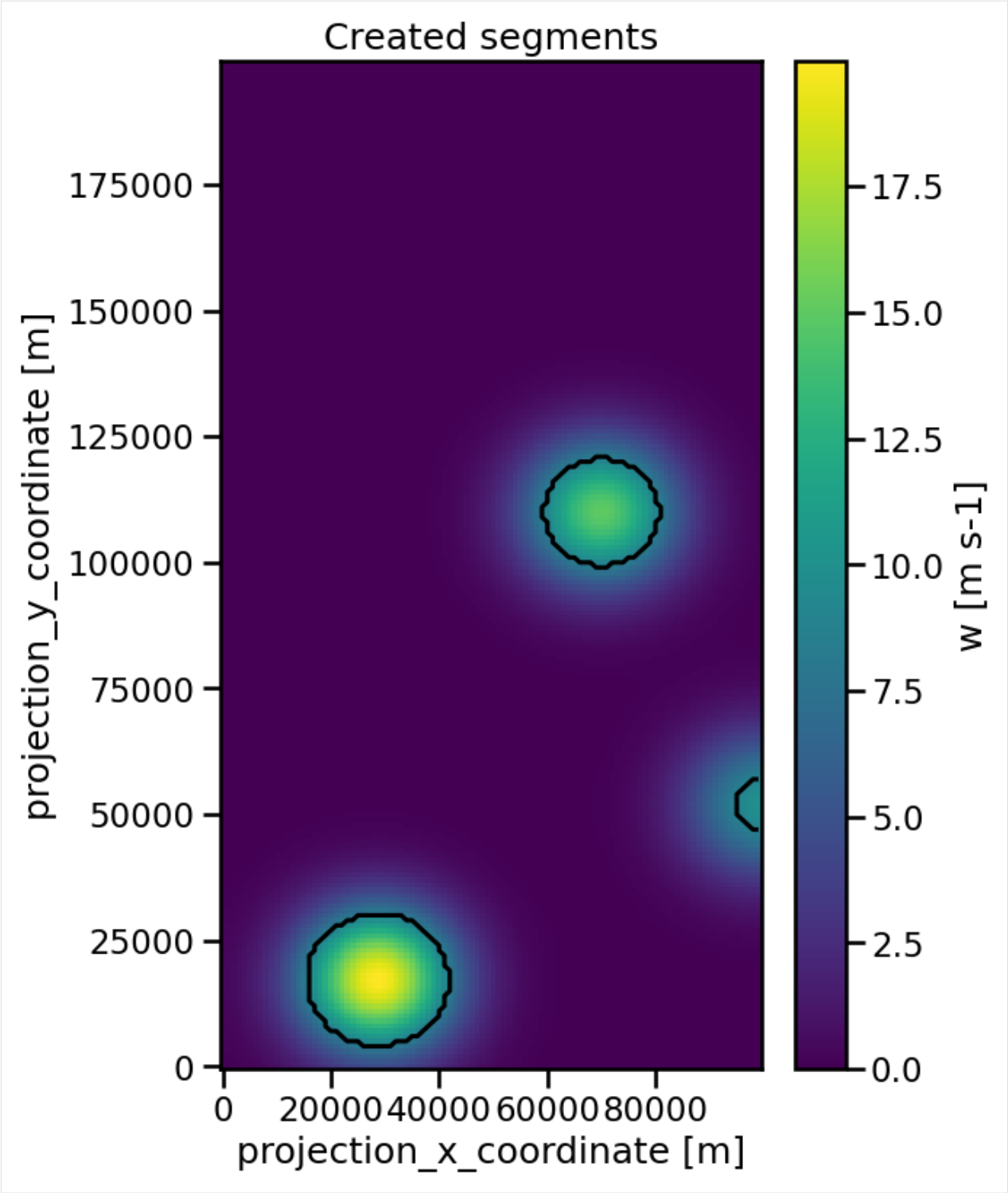
[7]:
```
features_mask["num"][1]
```

[7]:
```
66
```

The created segments can be visualized with a contour plot of tha mask:

[8]:
```
plt.figure(figsize=(6, 9))
data.isel(time=50).plot(x="x", y="y")
mask.isel(time=50).plot.contour(levels=[0.5], x="x", y="y", colors="k")
plt.title("Created segments")
```

[8]:
```
Text(0.5, 1.0, 'Created segments')
```


Created segments
175000
150000
125000
100000
75000
50000
25000
0
projection_y_coordinate [m]
0
20000
40000
60000
80000
projection_x_coordinate [m]
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
w [m s-1]

6.5.2 Different Thresholds

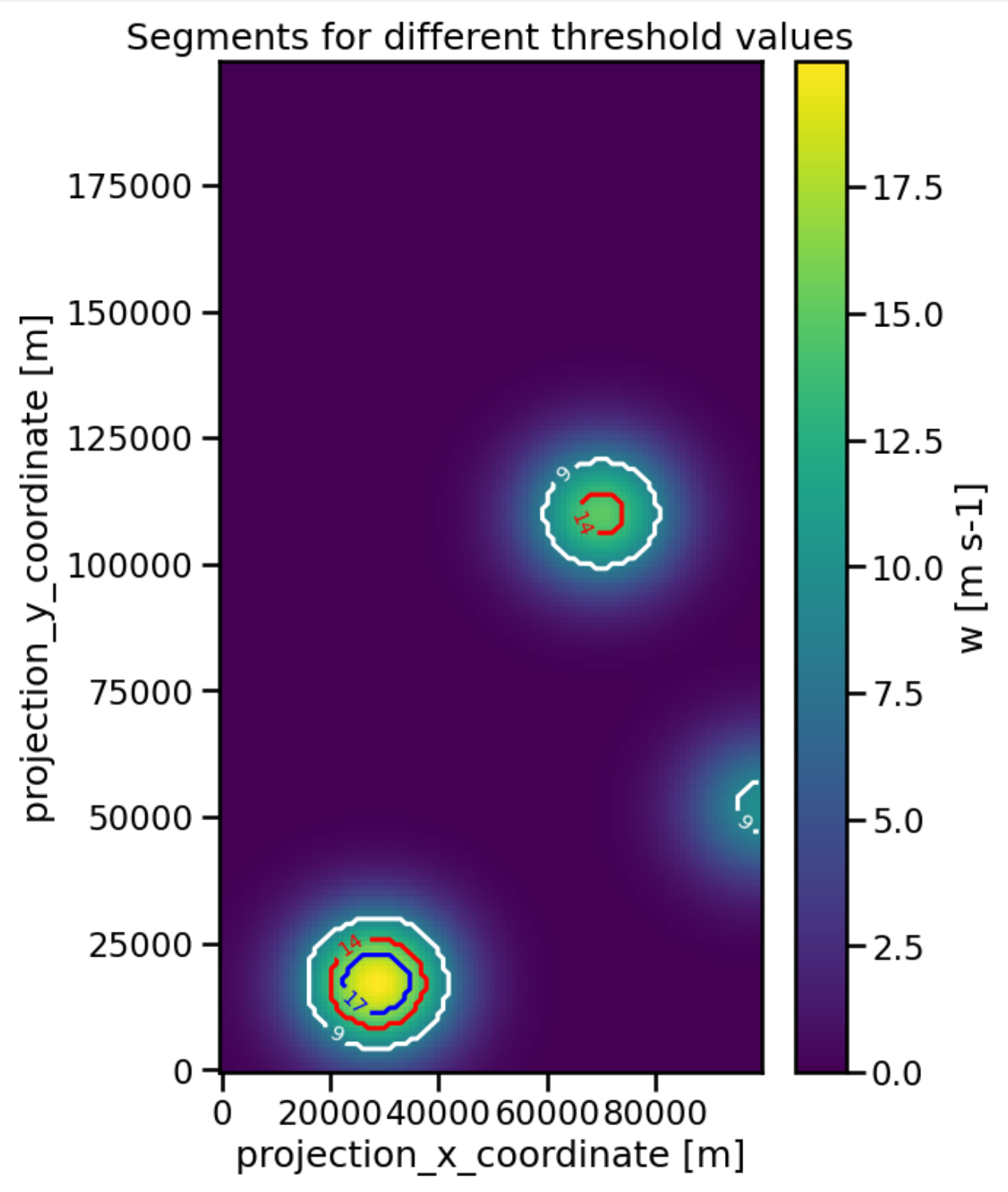
It is important to highlight that (in contrast to the feature detection), segmentation is only possible with **single threshold values**. Because of that, we have to call the function multiple times with different `threshold` values to explore the influence of this argument:

[9]:
```
%%capture
mask_1, features_mask_1 = tobac.segmentation_2D(features, data, dxy, threshold=9)
mask_2, features_mask_2 = tobac.segmentation_2D(features, data, dxy, threshold=14)
mask_3, features_mask_3 = tobac.segmentation_2D(features, data, dxy, threshold=17)
```

To visualize the segments we can use contour-plots of the masks:

[10]:
```
thresholds = [9, 14, 17]
masks = [mask_1, mask_2, mask_3]
colors = ["w", "r", "b"]

fig, ax = plt.subplots(ncols=1, figsize=(6, 9))
data.isel(time=50).plot(ax=ax, x="x", y="y")

for n, mask, color in zip(thresholds, masks, colors):
    contour = mask.isel(time=50).plot.contour(levels=[n], x="x", y="y", colors=color)
    ax.clabel(contour, inline=True, fontsize=10)

ax.set_title("Segments for different threshold values")
```

[10]:
```
Text(0.5, 1.0, 'Segments for different threshold values')
```



Obviously, a lower threshold value prodcuces a larger segment and if a feature does not exceed the value at all, no segment is associated.

6.5.3 Choosing Method and Target

The segmentation uses certain techniques to associate areas or volumes to each identified feature. Watershedding is the default and the only implemented option at the moment, but in future realeases the method will be selected by the `method`-keyword:

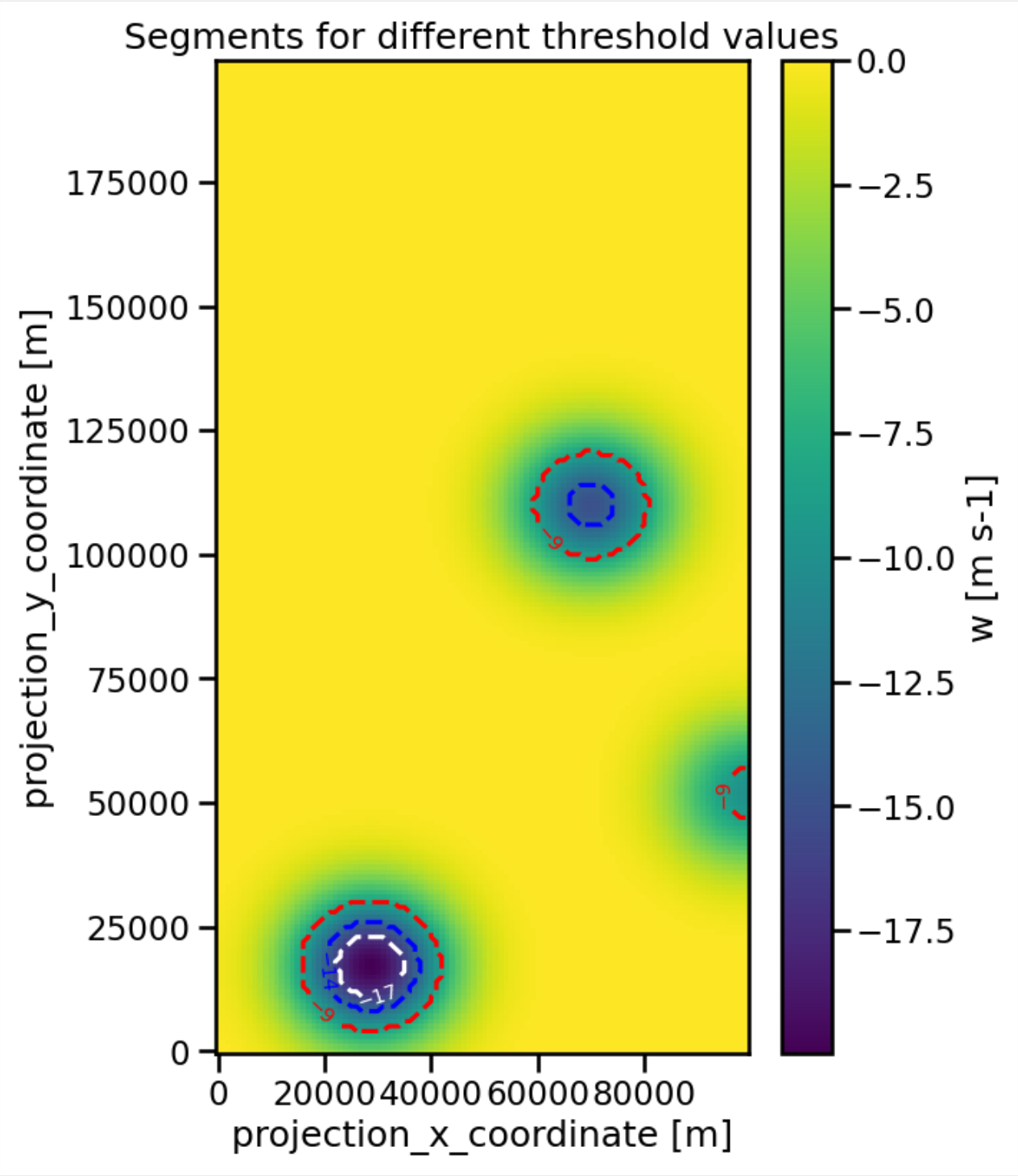
[11]:
```
%%capture
mask_1, features_mask_1 = tobac.segmentation_2D(
    features, data, dxy, threshold=9, method="watershed"
)
```

Analogous to the feature detection, it is also possible to apply the segmentation to minima by changing the *target* keyword:

[12]:
```
%%capture

data = -tobac.testing.make_sample_data_2D_3blobs_inv(data_type="xarray")
dxy, dt = tobac.utils.get_spacings(data)
thresholds = [-9, -14, -17]
features = tobac.feature_detection_multithreshold(
    data, dxy, thresholds, target="minimum"
)

mask_1, features_mask_1 = tobac.segmentation_2D(
    features, data, dxy, threshold=-9, target="minimum"
)
mask_2, features_mask_2 = tobac.segmentation_2D(
    features, data, dxy, threshold=-14, target="minimum"
)
mask_3, features_mask_3 = tobac.segmentation_2D(
    features, data, dxy, threshold=-17, target="minimum"
)
```

[13]:
```
masks = [mask_1, mask_2, mask_3]
colors = ["r", "b", "w"]
thresholds = [-9, -14, -17]

fig, ax = plt.subplots(ncols=1, figsize=(6, 9))
data.isel(time=50).plot(ax=ax, x="x", y="y")

for n, mask, color in zip(thresholds, masks, colors):
    contour = (
        (n * mask).isel(time=50).plot.contour(levels=[n], colors=color, x="x", y="y")
    )
    ax.clabel(contour, inline=True, fontsize=10)

ax.set_title("Segments for different threshold values")
```

[13]:
```
Text(0.5, 1.0, 'Segments for different threshold values')
```


Segments for different threshold values
projection_y_coordinate [m]
0
25000
50000
75000
100000
125000
150000
175000
projection_x_coordinate [m]
0
20000
40000
60000
80000
w [m s-1]
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−17.5
−9
−14
−17

6.5.4 Setting a maximum Distance

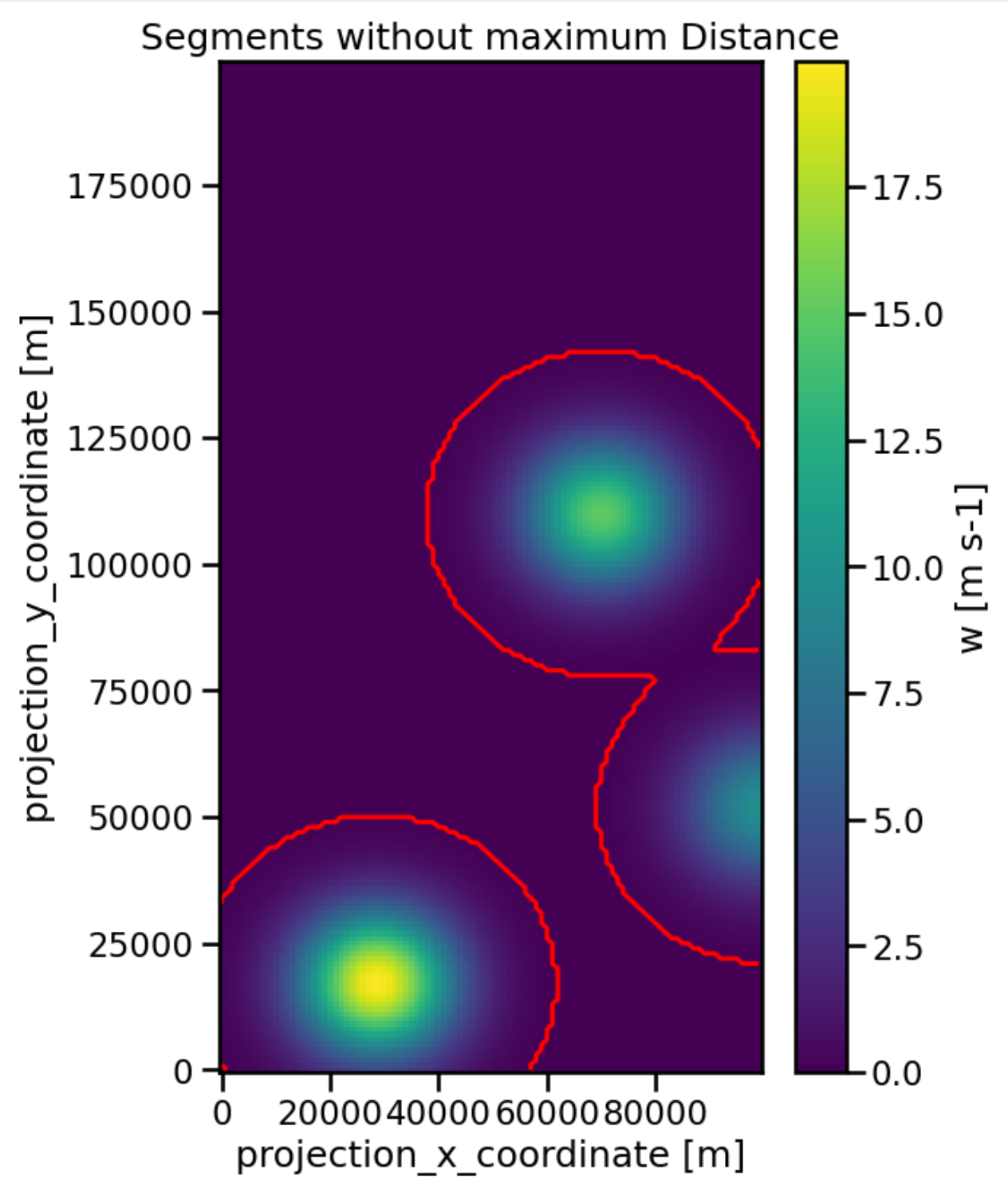
Another way of determining the size of our segments is the `max_distance`-parameter. It defines a maximum distance the segment can have from the coordinates of feature (in meters). This enables us, for example, to ensure that the segments of different features do not touch each other when we use a very low threshold value:

[14]:
```
%%capture

data = tobac.testing.make_sample_data_2D_3blobs_inv(data_type="xarray")
dxy, dt = tobac.utils.get_spacings(data)
thresh = 0.1

features = tobac.feature_detection_multithreshold(data, dxy, threshold=3)
mask_0, features_0 = tobac.segmentation_2D(features, data, dxy, threshold=thresh)
```

As you can see the threshold value was set to a value of 0.1. The result is that the segments of the two upper features will touch:

[15]:
```
fig, ax = plt.subplots(figsize=(6, 9))
data.isel(time=50).plot(ax=ax, x="x", y="y")
mask_0.isel(time=50).plot.contour(levels=[0.5], ax=ax, colors="r", x="x", y="y")
ax.set_title("Segments without maximum Distance")
```

[15]:
```
Text(0.5, 1.0, 'Segments without maximum Distance')
```



We can prevent this from happening by using the `max_distance` parameter to specify a maximum distance the border of the segment can have from the feature in meter:

[16]:
```
%%capture
```

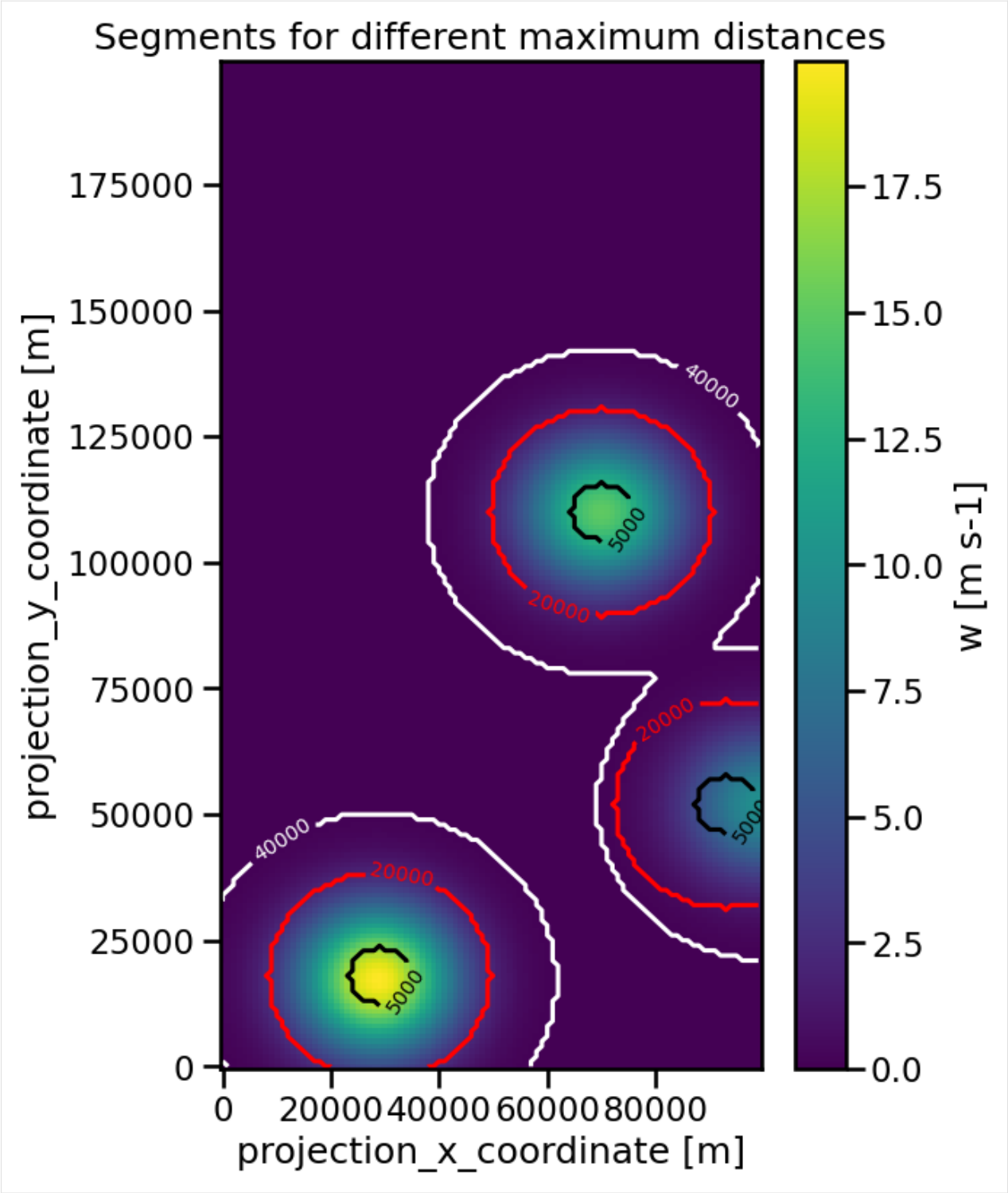
(continues on next page)

(continued from previous page)

```
mask_1, features_mask_1 = tobac.segmentation_2D(
    features, data, dxy, threshold=thresh, max_distance=40000
)
mask_2, features_mask_2 = tobac.segmentation_2D(
    features, data, dxy, threshold=thresh, max_distance=20000
)
mask_3, features_mask_3 = tobac.segmentation_2D(
    features, data, dxy, threshold=thresh, max_distance=5000
)
```

[17]:
```
masks = [mask_1, mask_2, mask_3]
colors = ["w", "r", "k"]
distances = [4e4, 2e4, 5e3]

fig, ax = plt.subplots(ncols=1, figsize=(6, 9))
data.isel(time=50).plot(ax=ax, x="x", y="y")

for n, mask, color in zip(distances, masks, colors):
    contour = (
        (n * mask).isel(time=50).plot.contour(levels=[n], colors=color, x="x", y="y")
    )
    ax.clabel(contour, inline=True, fontsize=10)

ax.set_title("Segments for different maximum distances")
```

[17]:
```
Text(0.5, 1.0, 'Segments for different maximum distances')
```


Segments for different maximum distances
projection_y_coordinate [m]
projection_x_coordinate [m]
w [m s-1]
0
25000
50000
75000
100000
125000
150000
175000
0
20000
40000
60000
80000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
40000
20000
5000
40000
20000
5000

6.5.5 Handling 3d-Data

The remaining parameters `level` and `vertical_coord` are useful only for the segemtation of 3-dimensional inputs and will be covered in the notebook for 3d-data (TBD).

6.6 Methods and Parameters for Linking

Linking assigns the detected features and segments in each timestep to trajectories, to enable an analysis of their time evolution. We will cover the topics:

- *Understanding the Linking Output*
- *Defining a Search Radius*
- *Working with the Subnetwork*
- *Timescale of the Tracks*
- *Gappy Tracks*
- *Other Parameters*

We start by loading the usual libraries:

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

%matplotlib inline

import seaborn as sns

sns.set_context("talk")


import warnings

warnings.filterwarnings("ignore")
```

[2]:
```
import tobac
import tobac.testing
```

6.6.1 Understanding the Linking Output

Since it has a time dimension, the sample data from the testing utilities is also suitable for a demonstration of this step. The chosen method creates 2-dimensional sample data with up to 3 moving blobs at the same. So, our expectation is to obtain up to 3 tracks.

At first, loading in the data and performing the usual feature detection is required:

[3]:
```
%%capture

data = tobac.testing.make_sample_data_2D_3blobs_inv(data_type="xarray")
dxy, dt = tobac.utils.get_spacings(data)
```

(continues on next page)

(continued from previous page)

```
features = tobac.feature_detection_multithreshold(
    data, dxy, threshold=1, position_threshold="weighted_abs"
)
```

Now the linking via trackpy can be performed. Notice that the temporal spacing is also a required input this time. Additionally, it is necessary to provide either a maximum speed *v_max* or a maximum search range *d_max*. Here we use a maximum speed of 100 m/s:

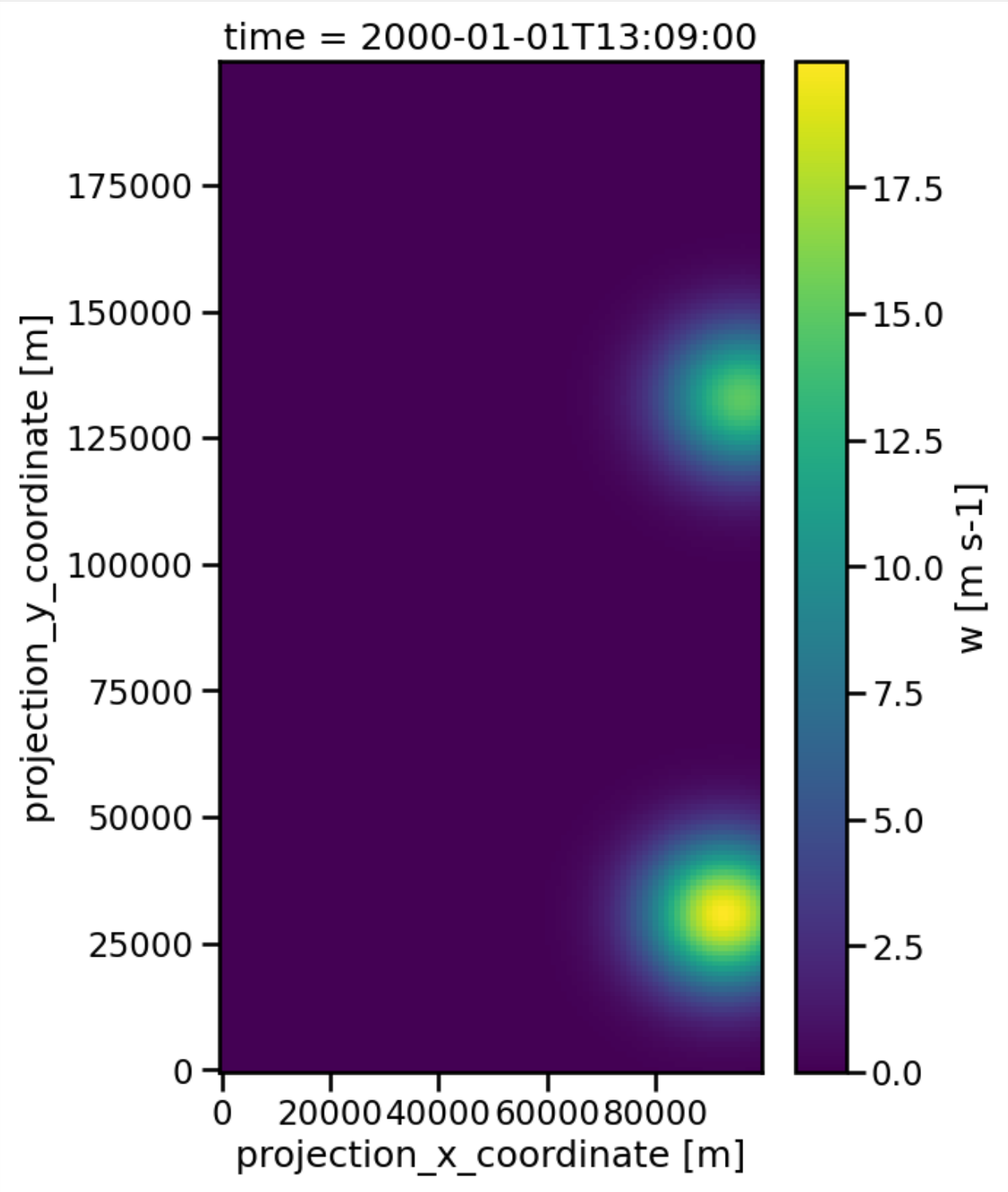
[4]:
```
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, v_max=100)
```

```
Frame 69: 2 trajectories present.
```

The output tells us, that in the final frame 69 two trajecteries where still present. If we checkout this frame via `xarray.plot` method, we can see that this corresponds to the two present features there, which are assigned to different trajectories.

[5]:
```
plt.figure(figsize=(6, 9))
data.isel(time=69).plot(x="x", y="y")
```

[5]:
```
<matplotlib.collections.QuadMesh at 0x1351a7050>
```



The track dataset contains two new variables, *cell* and *time_cell*. The first assigns the features to one of the found trajectories by an integer and the second specifies how long the feature has been present already in the data.

```
[6]: track[["cell", "time_cell"]][:20]
```

```
[6]:     cell       time_cell
```

(continues on next page)

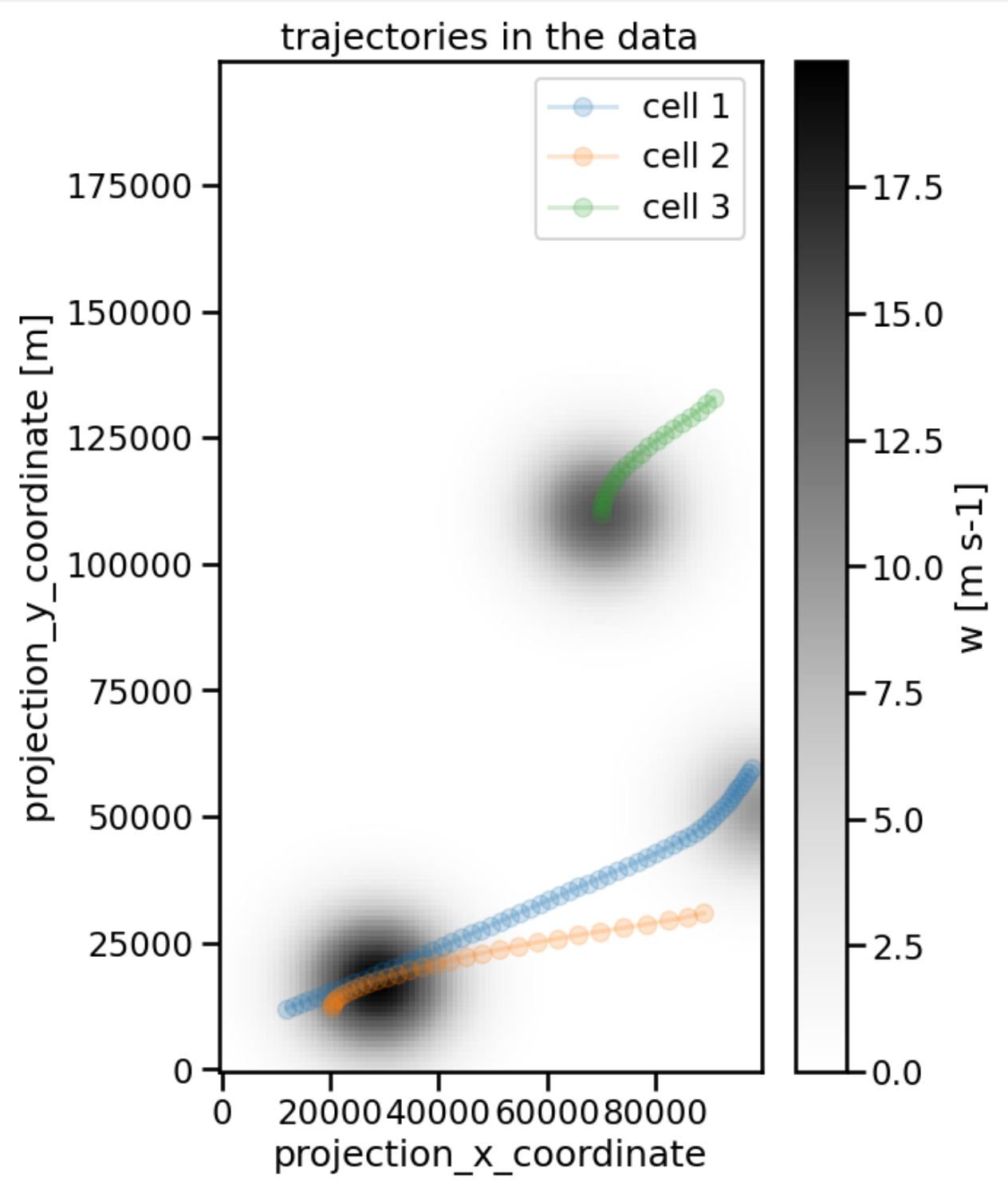
(continued from previous page)

```
0      1 0 days 00:00:00
1      1 0 days 00:01:00
2      1 0 days 00:02:00
3      1 0 days 00:03:00
4      1 0 days 00:04:00
5      1 0 days 00:05:00
6      1 0 days 00:06:00
7      1 0 days 00:07:00
8      1 0 days 00:08:00
9      1 0 days 00:09:00
10     1 0 days 00:10:00
11     1 0 days 00:11:00
12     1 0 days 00:12:00
13     1 0 days 00:13:00
14     1 0 days 00:14:00
15     1 0 days 00:15:00
16     1 0 days 00:16:00
17     1 0 days 00:17:00
18     1 0 days 00:18:00
19     1 0 days 00:19:00
```

Since we know that this dataset contains 3 features, we can visualize the found tracks with masks created from the *cell* variable:

[7]:
```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))

data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )


ax.set_title("trajectories in the data")
plt.legend()
```

[7]:
```
<matplotlib.legend.Legend at 0x1348a2350>
```

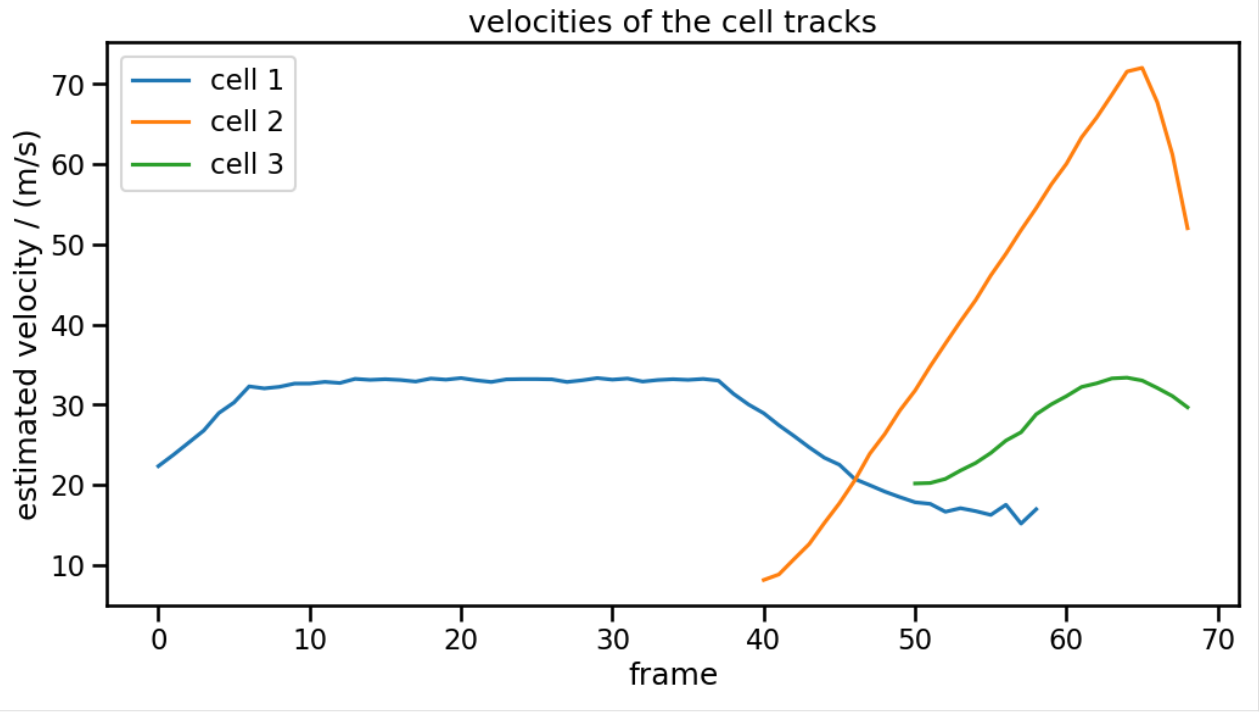


The cells have been specified with **different velocities** that also change in time. `tobac` retrieves the following values for our test data:

[8]:
```
vel = tobac.analysis.calculate_velocity(track)
```

[9]:
```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(12, 6))

for i, vc in vel.groupby("cell"):
    # get velocity as xarray Dataset
    v = vc.to_xarray()["v"]
    v = v.assign_coords({"frame": vc.to_xarray()["frame"]})

    v.plot(x="frame", ax=ax, label="cell {0}".format(int(i)))

    ax.set_ylabel("estimated velocity / (m/s)")

ax.set_title("velocities of the cell tracks")
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x1351bf250>
```



Interpretation:

- "cell 1" is actually specified with constant velocity. The initial increase and the final decrease come from edge effects, i.e. that the blob corresponding to "cell 1" has parts that that go beyond the border.
- "cell 2" and "cell 3" are specified with linearily increasing x-component of the velocity. The initial speed-up is due to the square-root behavior of the velocity magnitude. The final decrease however come again from edge effects.

The starting point of the cell index is of course arbitrary. If we want to change it from 1 to 0 we can do this with the `cell_number_start` parameter:

[10]:
```
track = tobac.linking_trackpy(
    features, data, dt=dt, dxy=dxy, v_max=100, cell_number_start=0
)
```

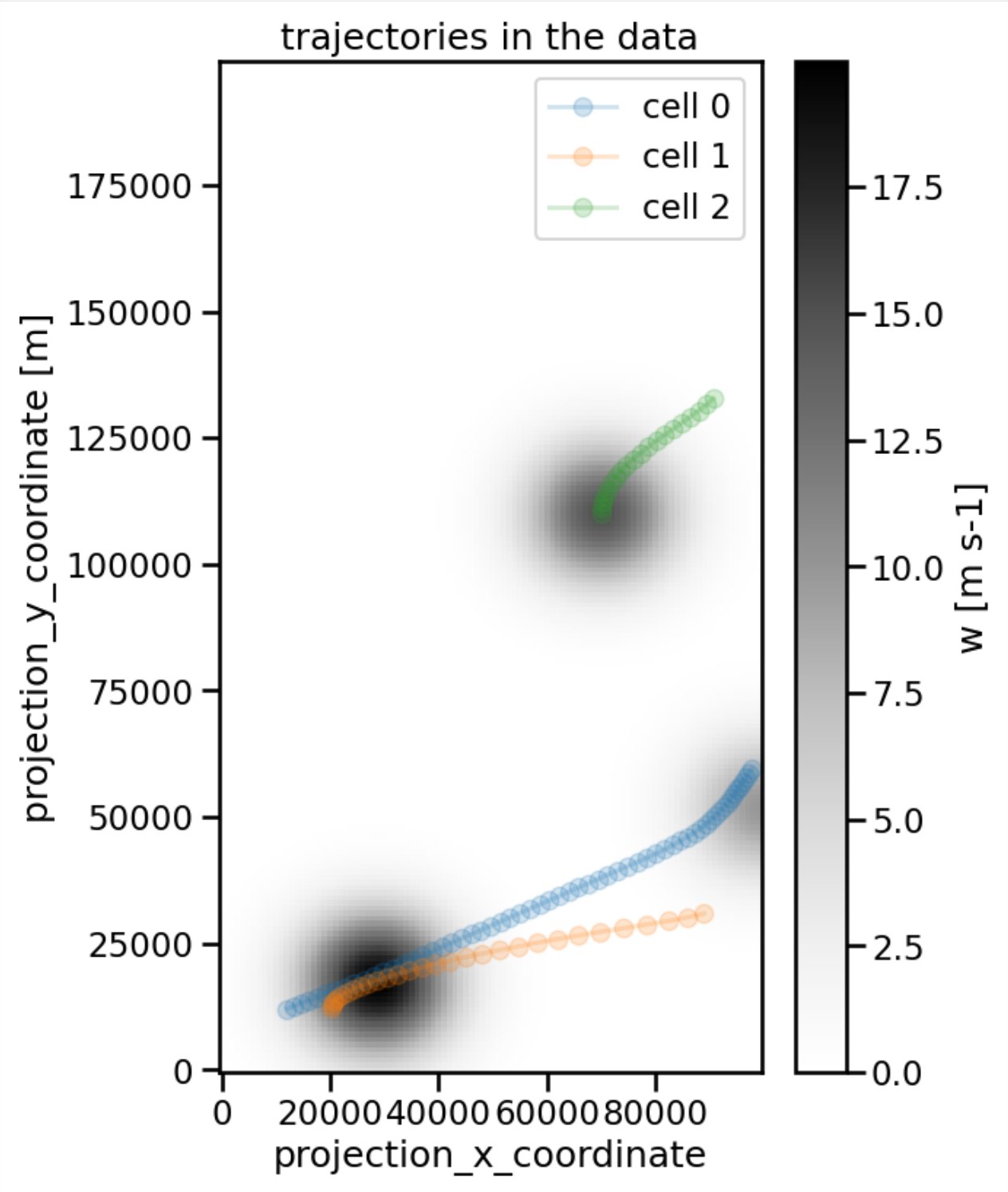
(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))

data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

ax.set_title("trajectories in the data")
plt.legend()
```

```
Frame 69: 2 trajectories present.
```

[10]:
```
<matplotlib.legend.Legend at 0x13598a350>
```



The linking in `tobac` is performed with methods of the trackpy-library. Currently, there are two methods implemented, 'random' and 'predict'. These can be selected with the `method` keyword. The default is 'random'.

[11]:
```
track_p = tobac.linking_trackpy(
    features, data, dt=dt, dxy=dxy, v_max=100, method_linking="predict"
)
```

```
Frame 69: 2 trajectories present.
```

6.6.2 Defining a Search Radius

If you paid attention to the input of the `linking_trackpy()` function you noticed that we used the parameter `v_max` there. The reason for this is that it would take a lot of computation time to check the whole data of a frame for the next position of a feature from the previous frame with the Crocker-Grier linking algorithm. Therefore the search is restricted to a circle around a position predicted by different methods implemented in *trackpy*.

The speed we defined before specifies such a radius timestep via

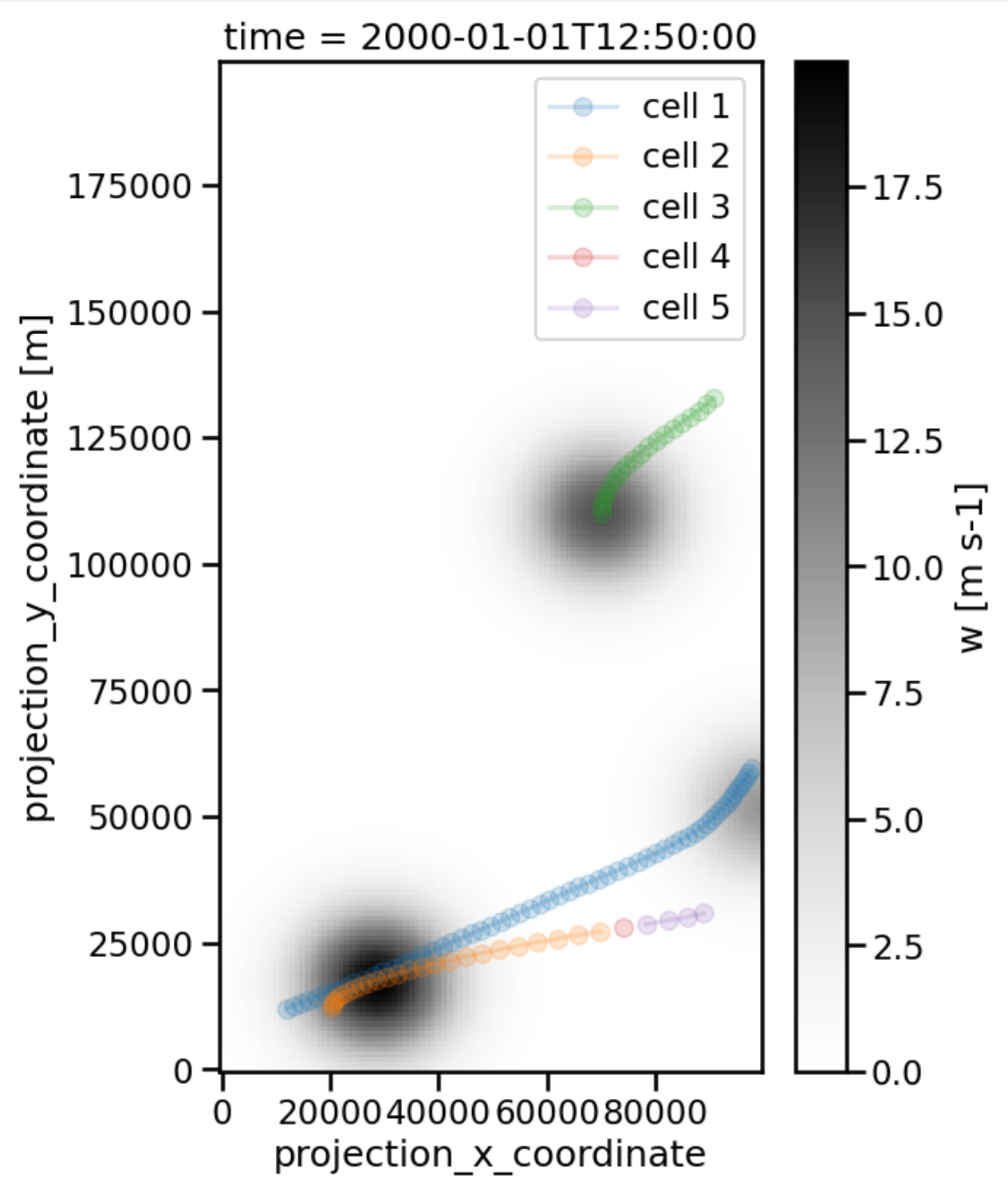
$$r_{max} = v_{max}\Delta t.$$

By reducing the **maximum speed** from 100 m/s to 70 m/s we will find more cell tracks, because a feature that is not assigned to an existing track will be used as a starting point for a new one:

[12]:
```
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, v_max=70)

fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))

data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

plt.legend()
```

```
Frame 69: 2 trajectories present.
```

[12]:
```
<matplotlib.legend.Legend at 0x13598a290>
```



Above you see that previously orange track is *split into three parts* because the considered cell moves so fast.

The same effect can be achieved by directly reducing the **maximum search range** with the `d_max` parameter to

$$d_{max} = 4200m$$

because

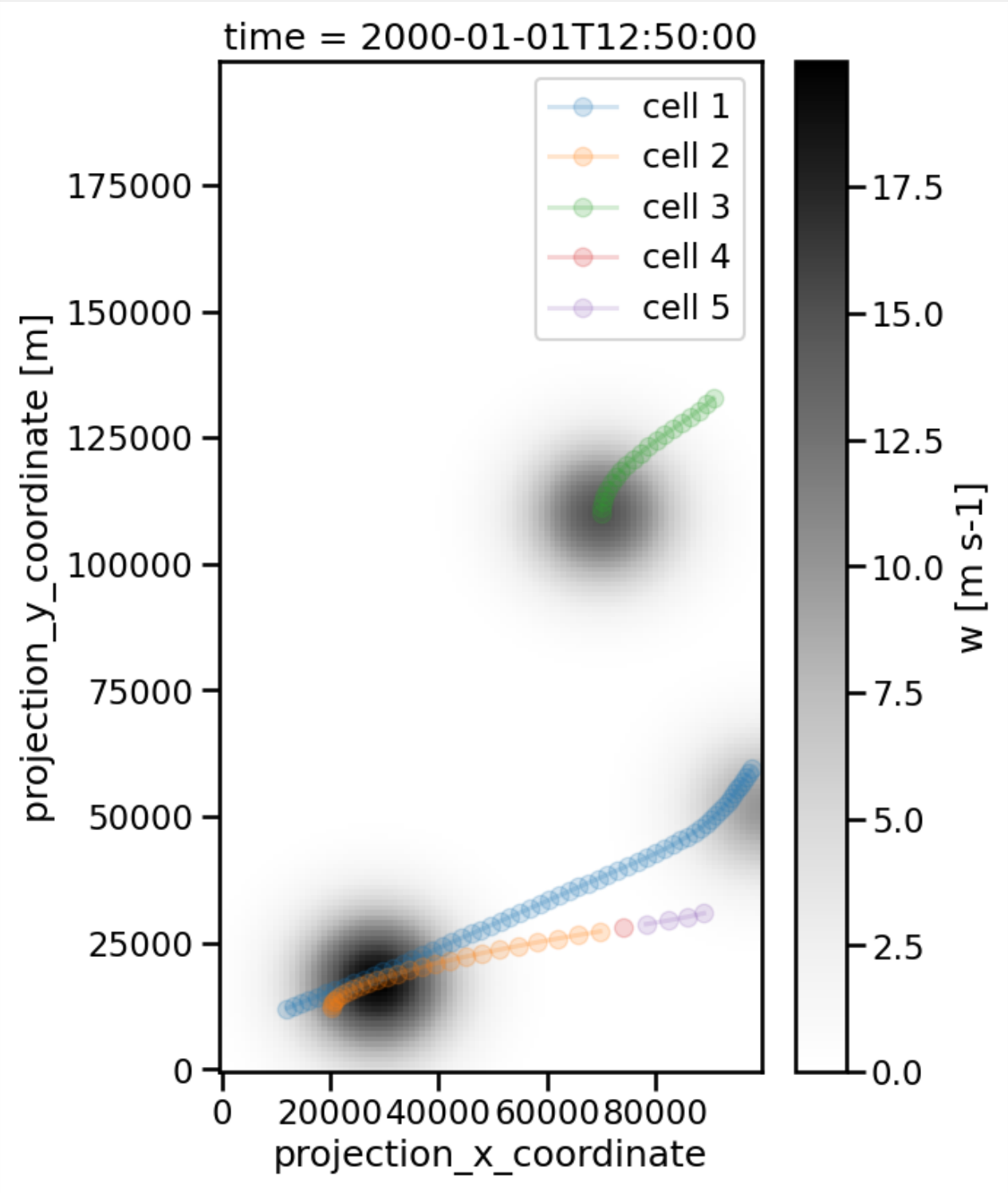
$$v_{max}\Delta t = 4200m$$

for $v_{max} = 70$ m/s in our case.

[13]:
```
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, d_max=4200)

fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))

data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

plt.legend()
```

```
Frame 69: 2 trajectories present.
```

[13]:
```
<matplotlib.legend.Legend at 0x1365638d0>
```



v_max and *d_max* should not be used together, because they both specify the same qunatity, namely the search range, but it is neccery to set at least one of these parameters.

6.6.3 Working with the Subnetwork

If we increase *v_max* or *d_max* it can happen that more than one feature of the previous timestep is in the search range. The selection of the best matching feature from the set of possible features (which is called the subnetwork, for a more in depth explanation have a look here) is the most time consuming part of the linking process. For this reason, it is possible to limit the number of features in the search range via the `subnetwork_size` parameter. The tracking will result in a **trackpy.SubnetOversizeException** if more features than specified there are in the search range. We can simulate this situation by setting a really high value for `v_max` and 1 for the `subnetwork_size`:

[14]:
```
from trackpy import SubnetOversizeException

try:
    track = tobac.linking_trackpy(
        features, data, dt=dt, dxy=dxy, v_max=1000, subnetwork_size=1
    )
    print("tracking successful")

except SubnetOversizeException as e:
    print("Tracking not possible because the {}".format(e))
```

```
Frame 57: 3 trajectories present.
Tracking not possible because the Subnetwork contains 2 points
```

The default value for this parameter is 30 and can be accessed via:

[15]:
```
import trackpy as tp

tp.linking.Linker.MAX_SUB_NET_SIZE
```

[15]:
```
1
```

It is important to highlight that if the `linking_trackpy()`-function is called with a `subnetwork_size` different from 30, this number will be the new default. This is the reason, why the value is 1 at this point.

Adaptive Search

A way of dealing with **SubnetOversizeException**s is adaptive search. An extensive description can be found here.

If the subnetwork is too large, the search range is reduced iteratively by multiplying it with the `adaptive_step`. The `adaptive_stop`-parameter is the lower limit of the search range. If it is reached and the subnetwork is still too large, a **SubnetOversizeException** is thrown. Notice that as soon as adaptive search is used, a different limit of the subnetwork size applies, which needs to be specified by hand at the moment:

[16]:
```
tp.linking.Linker.MAX_SUB_NET_SIZE_ADAPTIVE = 1
```

Now the tracking can be performed and will no longer result in an exception:

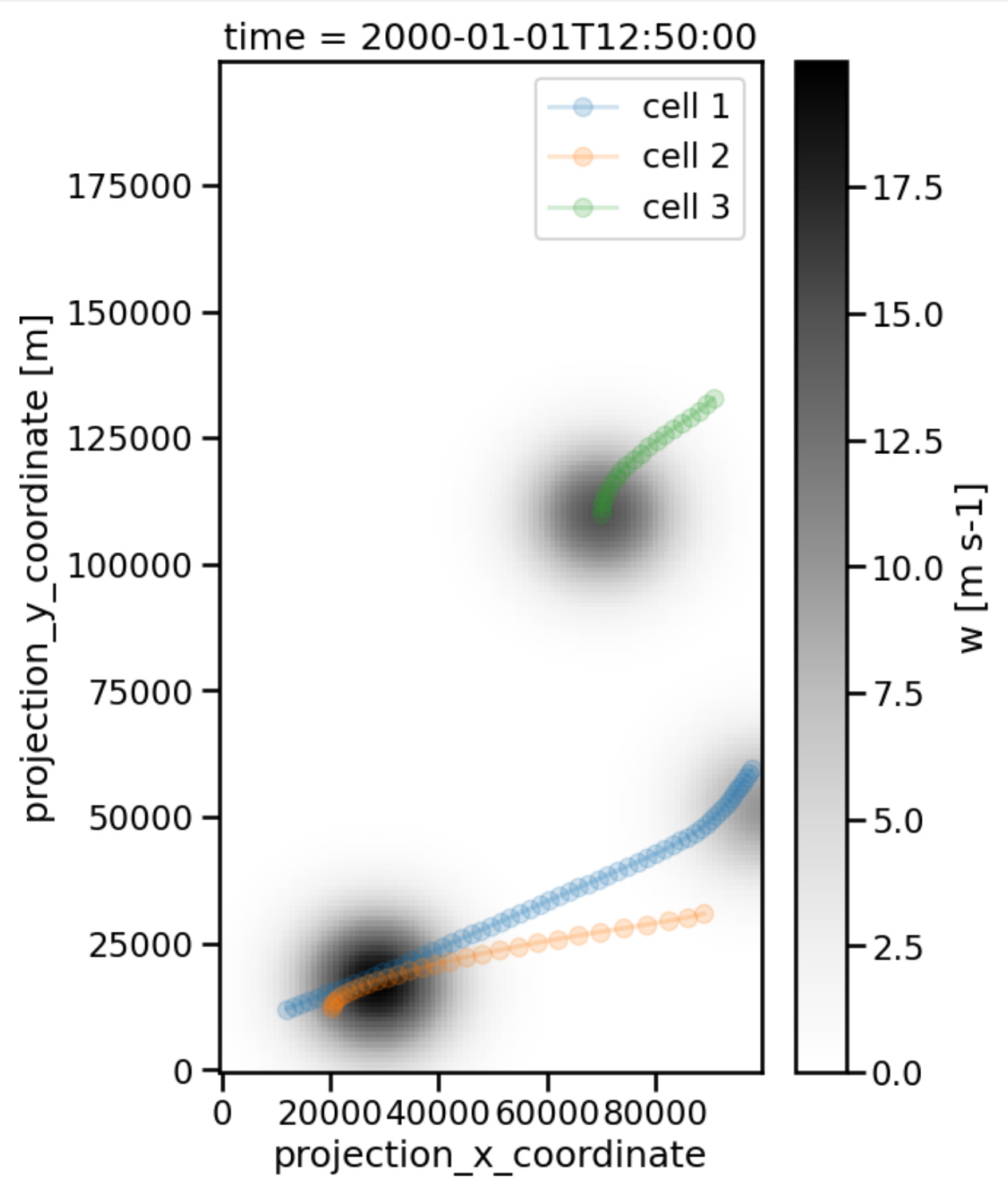
[17]:
```
try:
    track = tobac.linking_trackpy(
        features, data, dt=dt, dxy=dxy, v_max=1000, adaptive_stop=10, adaptive_step=0.9
    )
    print("tracking successful!")

    fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))
```

(continues on next page)

(continued from previous page)

```
    data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

    for i, cell_track in track.groupby("cell"):
        cell_track.plot(
            x="projection_x_coordinate",
            y="projection_y_coordinate",
            ax=ax,
            marker="o",
            alpha=0.2,
            label="cell {0}".format(int(i)),
        )

    plt.legend()

except SubnetOversizeException as e:
    print("Tracking not possible because the {}".format(e))
```

```
Frame 69: 2 trajectories present.
tracking successful!
```



If `adaptive_stop` is specified, `adaptive_step` needs to be specified as well.

6.6.4 Timescale of the Tracks

If we want to limit our analysis to long living features of the dataset it is possible to exclude tracks which only cover few timesteps. This is done by setting a lower bound for those via the `stups` parameter. In our test data, only the first cell exceeds 50 time steps:

[18]:
```
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, v_max=100, stubs=50)
```
```
Frame 69: 2 trajectories present.
```

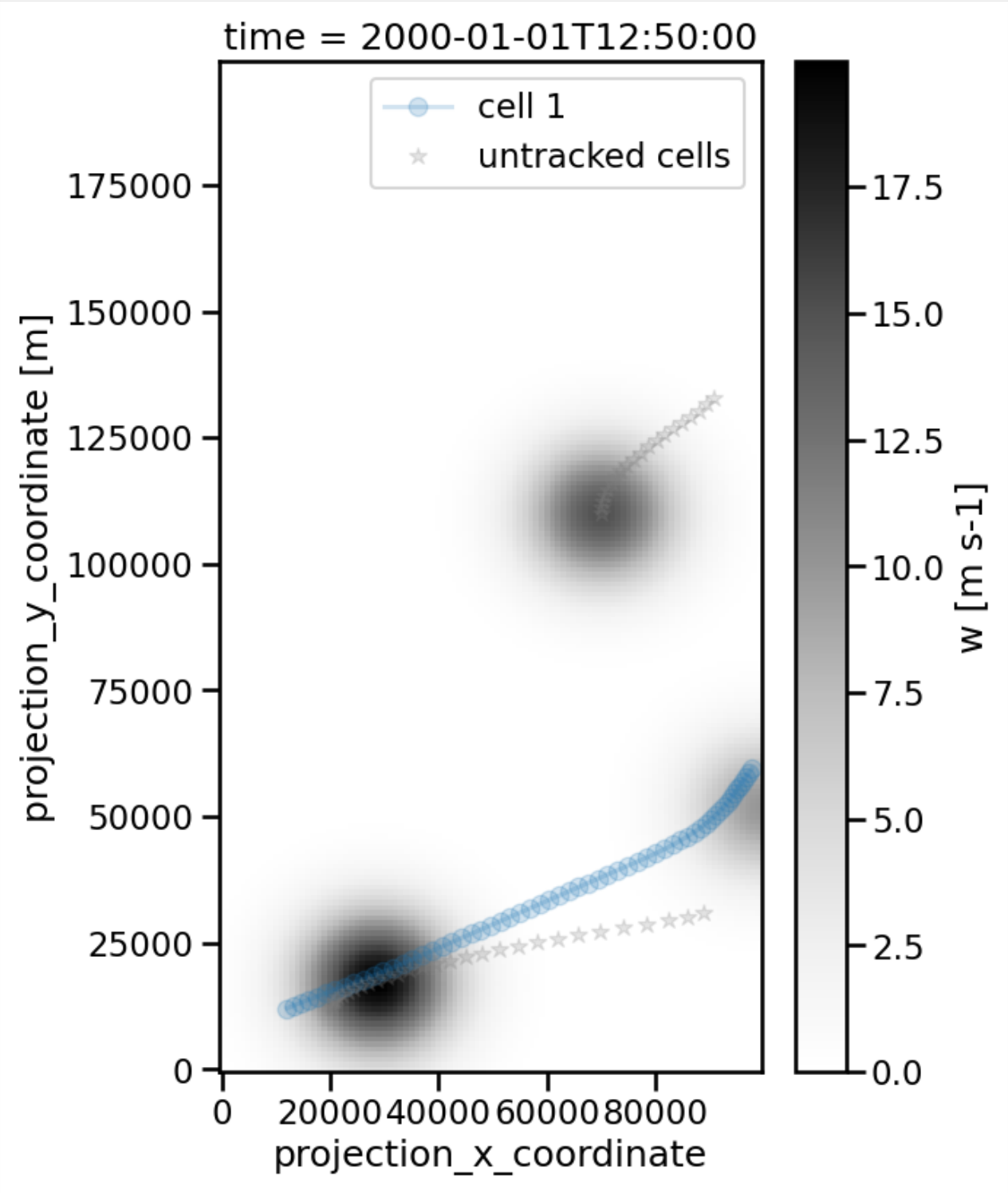
Now, the DataFrame `track` contains two types (two categories) of cells: 1. *tracked cells*: connected with cell tracks 2. *untracked cells*: not connected

The first type (*"tracked cells"*) is what you know and what you have already worked with. But the 2nd type (*"untracked cells"*) is new. All untracked cells are marked with the cell index -1 and thus collected in a buffer space.

[19]:
```
tracked_cells_only = track.where(track.cell > 0)
untracked_cells = track.where(track.cell == -1)
```

Now, the track dataset is split into two parts. Let's have a look:

[20]:
```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))

data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in tracked_cells_only.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

for i, cell_track in untracked_cells.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        color="gray",
        marker="*",
        label="untracked cells",
        lw=0,
        alpha=0.2,
    )

plt.legend()
```
[20]:
```
<matplotlib.legend.Legend at 0x137046190>
```



Analogius to the search range, there is a second option for that called *time_cell_min*. This defines a minimum of time in seconds for a cell to appear as tracked:

[21]:
```
track = tobac.linking_trackpy(
    features,
```

(continues on next page)

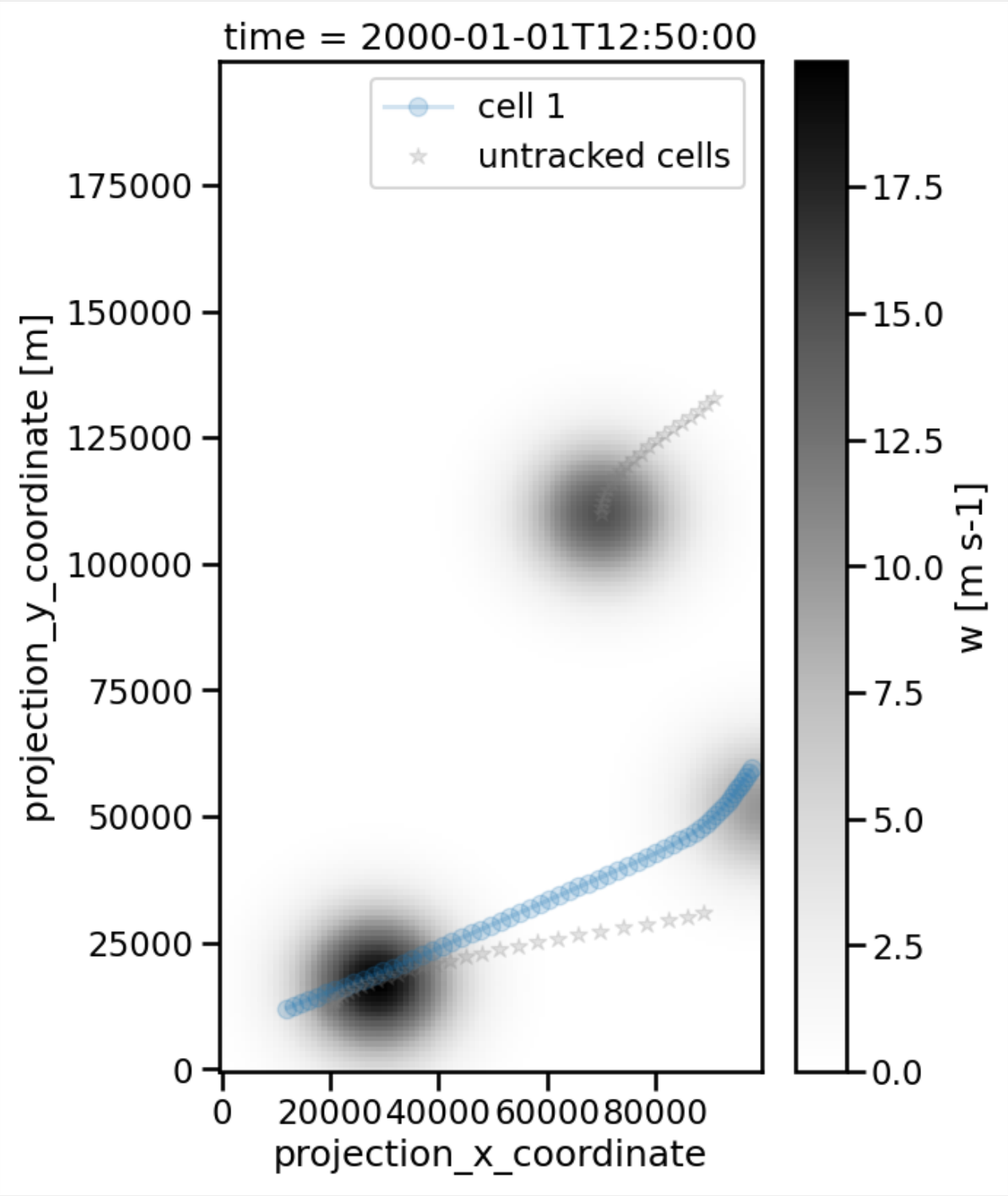
(continued from previous page)

```
    data,
    dt=dt,
    dxy=dxy,
    v_max=100,
    time_cell_min=60 * 50,  # means 50 steps of 60 seconds
)

tracked_cells_only = track.where(track.cell > 0)

untracked_cells = track.where(track.cell == -1)
```

```
Frame 69: 2 trajectories present.
```

[22]:
```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(6, 9))

data.isel(time=50).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in tracked_cells_only.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

for i, cell_track in untracked_cells.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        color="gray",
        marker="*",
        label="untracked cells",
        lw=0,
        alpha=0.2,
    )

plt.legend()
```

[22]:
```
<matplotlib.legend.Legend at 0x133e9f550>
```


time = 2000-01-01T12:50:00
cell 1
untracked cells
projection_y_coordinate [m]
0
25000
50000
75000
100000
125000
150000
175000
0
20000
40000
60000
80000
projection_x_coordinate
w [m s-1]
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

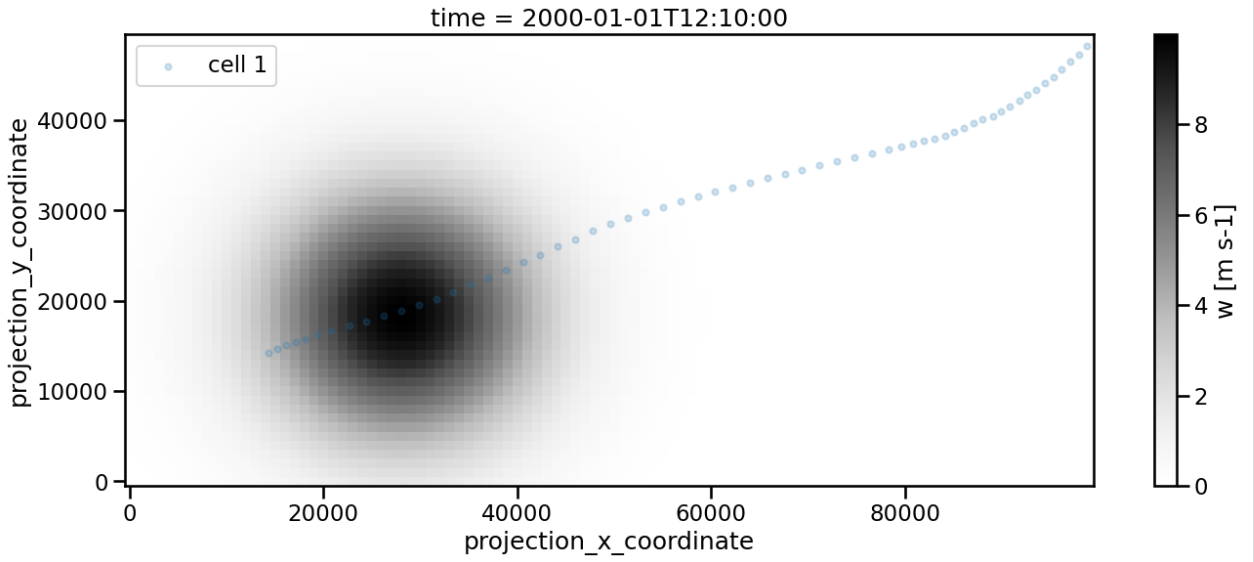
6.6.5 Gappy Tracks

If the data is noisy, an object may disappear for a few frames and then reappear. Such features can still be linked by setting the `memory` parameter with an integer. This defines the number of timeframes a feature can vanish and still get tracked as one cell. We demonstrate this on a simple dataset with only one feature:

[23]:
```
data = tobac.testing.make_simple_sample_data_2D(data_type="xarray")
dxy, dt = tobac.utils.get_spacings(data)
features = tobac.feature_detection_multithreshold(data, dxy, threshold=1)
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, v_max=1000)

fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(16, 6))

data.isel(time=10).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot.scatter(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

plt.legend()
```

```
Frame 59: 1 trajectories present.
```

[23]:
```
<matplotlib.legend.Legend at 0x136b09c10>
```



To simulate a gap in the data we set 5 timeframes to 0:

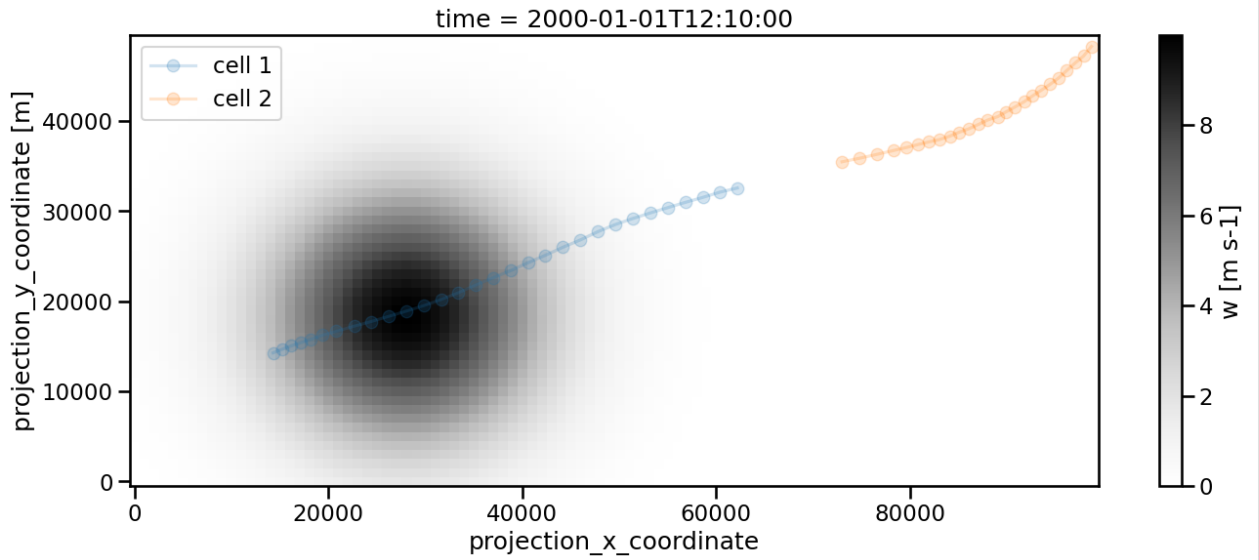
[24]:
```
data[30:35] = data[30] * 0
```

If we apply feature detection and linking to this modified dataset, we will now get two cells:

[25]:
```
features = tobac.feature_detection_multithreshold(data, dxy, threshold=1)
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, v_max=1000)

fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(16, 6))

data.isel(time=10).plot(ax=ax, x="x", y="y", cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot(
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

plt.legend()
```

```
Frame 59: 1 trajectories present.
```

[25]:
```
<matplotlib.legend.Legend at 0x136a5fe90>
```



We can avoid that by setting `memory` to a sufficiently high number. 5 is of course sufficient in our case as the situation is artificially created, but with real data this would require some fine tuning. Keep in mind that the search radius needs to be large enough to reach the next feature position. This is the reason for setting `v_max` to 1000 in the linking process.

[26]:
```
features = tobac.feature_detection_multithreshold(data, dxy, threshold=1)
track = tobac.linking_trackpy(features, data, dt=dt, dxy=dxy, v_max=1000, memory=5)

fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(16, 6))

data.isel(time=10).plot(ax=ax, cmap="Greys")

for i, cell_track in track.groupby("cell"):
    cell_track.plot(
```

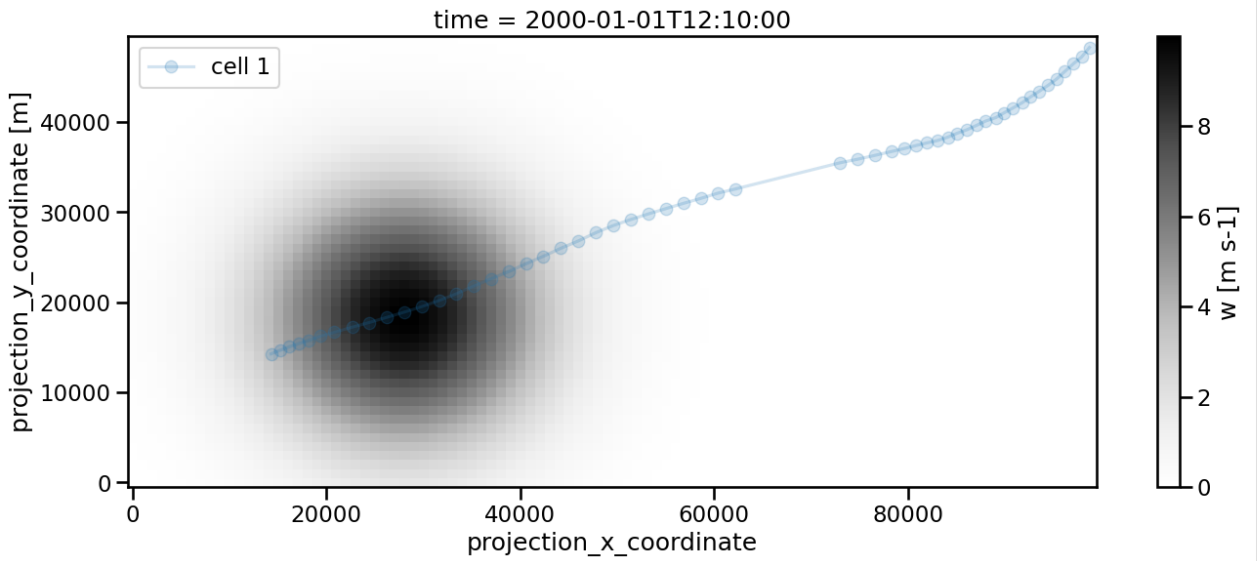
(continues on next page)

(continued from previous page)

```
        x="projection_x_coordinate",
        y="projection_y_coordinate",
        ax=ax,
        marker="o",
        alpha=0.2,
        label="cell {0}".format(int(i)),
    )

plt.legend()
```

```
Frame 59: 1 trajectories present.
```

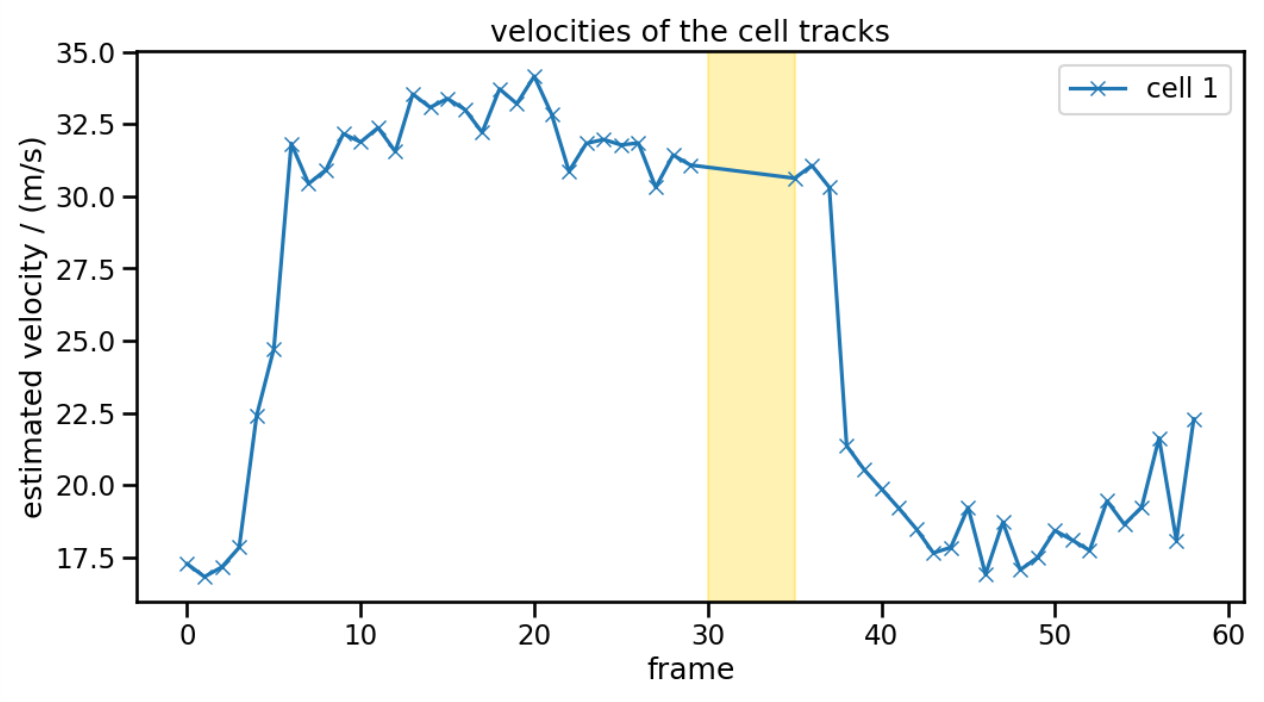
[26]:
```
<matplotlib.legend.Legend at 0x136bd2c10>
```



[27]:
```
vel = tobac.analysis.calculate_velocity(track)
```

[28]:
```
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(12, 6))

for i, vc in vel.groupby("cell"):
    # get velocity as xarray Dataset
    v = vc.to_xarray()["v"]
    v = v.assign_coords({"frame": vc.to_xarray()["frame"]})
    v.plot(x="frame", ax=ax, label="cell {0}".format(int(i)), marker="x")

    ax.set_ylabel("estimated velocity / (m/s)")

ax.axvspan(30, 35, color="gold", alpha=0.3)
ax.set_title("velocities of the cell tracks")
plt.legend()
```

[28]:
```
<matplotlib.legend.Legend at 0x136dce390>
```



Velocity calculations work well across the gap (marked with yellow above). The initial and final drop in speed comes again from edge effects.

6.6.6 Other Parameters

The parameters `d_min_`, `extrapolate` and `order` do not have a working implmentation in tobac V2 at the moment.

6.7 tobac example: Tracking deep convection based on OLR from geostationary satellite retrievals

This example notebook demonstrates the use of tobac to track isolated deep convective clouds based on outgoing longwave radiation (OLR) calculated based on a combination of two different channels of the GOES-13 imaging instrument.

The data used in this example is downloaded from "zenodo link" automatically as part of the notebooks (This only has to be done once for all the tobac example notebooks).

[1]:
```
# Import libraries:
import iris
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import iris.plot as iplt
import iris.quickplot as qplt
import shutil
from six.moves import urllib
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Import tobac itself:
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[3]:
```
# Disable a few warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

Download example data:
This has to be done only once for all tobac examples.

[4]:
```
data_out=Path('../')
```

[5]:
```
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_OLR_satellite.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_OLR_satellite.nc'))
```

Load data:

[6]:
```
# Load Data from downloaded file:
OLR=iris.load_cube(str(data_file[0]),'OLR')
```

[7]:
```
# Display information about the input data cube:
display(OLR)
```

```
<iris 'Cube' of OLR / (W m^-2) (time: 54; latitude: 131; longitude: 184)>
```

[8]:
```
#Set up directory to save output and plots:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
```

(continues on next page)

(continued from previous page)

```
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

Feature identification:
Identify features based on OLR field and a set of threshold values

[9]:
```
# Determine temporal and spatial sampling of the input data:
dxy,dt=tobac.get_spacings(OLR,grid_spacing=4000)
```

[10]:
```
# Keyword arguments for the feature detection step
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['n_min_threshold']=4
parameters_features['target']='minimum'
parameters_features['threshold']=[250,225,200,175,150]
```

[11]:
```
# Feature detection and save results to file:
print('starting feature detection')
Features=tobac.feature_detection_multithreshold(OLR,dxy,**parameters_features)
Features.to_hdf(savedir / 'Features.h5','table')
print('feature detection performed and saved')
```

```
starting feature detection
```

```
feature detection performed and saved
```

Segmentation:
Segmentation is performed based on the OLR field and a threshold value to determine the cloud areas.

[12]:
```
# Keyword arguments for the segmentation step:
parameters_segmentation={}
parameters_segmentation['target']='minimum'
parameters_segmentation['method']='watershed'
parameters_segmentation['threshold']=250
```

[13]:
```
# Perform segmentation and save results to files:
Mask_OLR,Features_OLR=tobac.segmentation_2D(Features,OLR,dxy,**parameters_segmentation)
print('segmentation OLR performed, start saving results to files')
iris.save([Mask_OLR], savedir / 'Mask_Segmentation_OLR.nc', zlib=True, complevel=4)
Features_OLR.to_hdf(savedir / 'Features_OLR.h5', 'table')
print('segmentation OLR performed and saved')
```

```
segmentation OLR performed, start saving results to files
segmentation OLR performed and saved
```

Trajectory linking:
The detected features are linked into cloud trajectories using the trackpy library (http://soft-matter.github.io/trackpy). This takes the feature positions determined in the feature detection step into account but does not include information on the shape of the identified objects.

[14]:
```
# keyword arguments for linking step
parameters_linking={}
parameters_linking['v_max']=20
parameters_linking['stubs']=2
parameters_linking['order']=1
parameters_linking['extrapolate']=0
parameters_linking['memory']=0
parameters_linking['adaptive_stop']=0.2
parameters_linking['adaptive_step']=0.95
parameters_linking['subnetwork_size']=100
parameters_linking['method_linking']= 'predict'
```

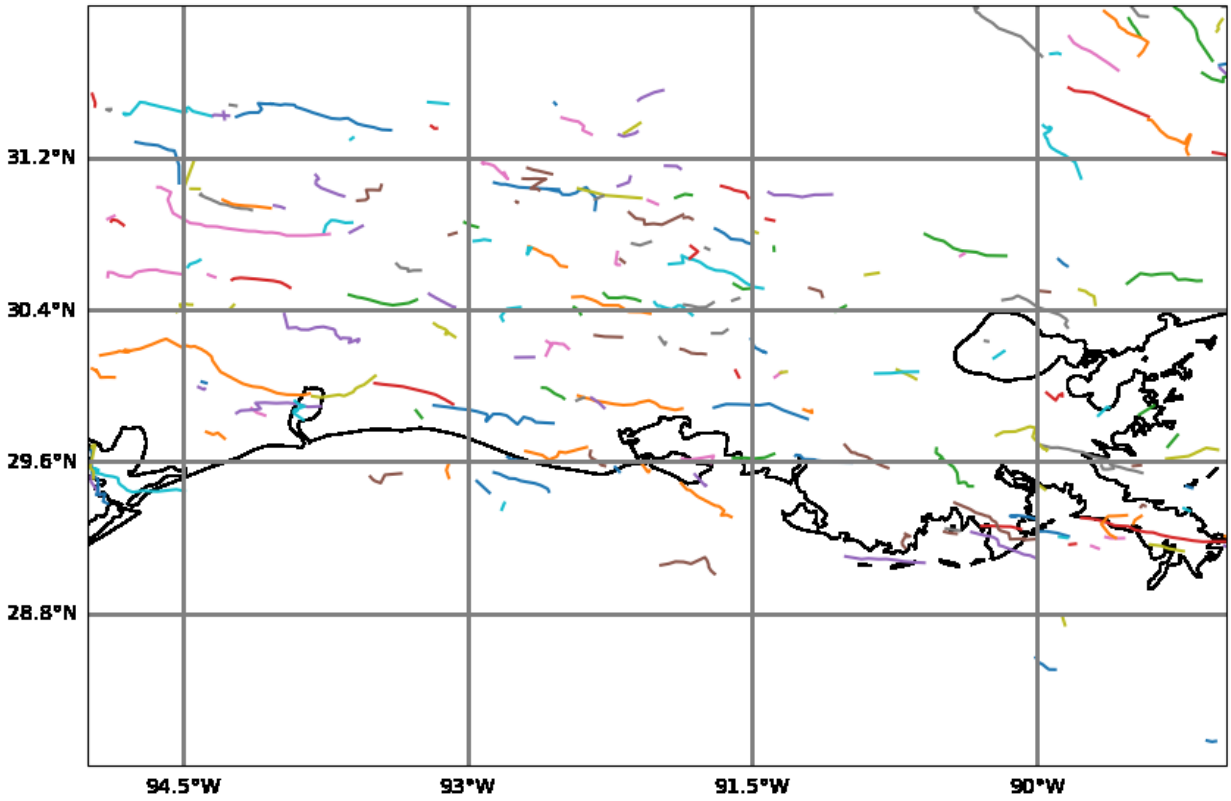
[15]:
```
# Perform linking and save results to file:
Track=tobac.linking_trackpy(Features,OLR,dt=dt,dxy=dxy,**parameters_linking)
Track.to_hdf(savedir / 'Track.h5','table')
```

```
Frame 53: 11 trajectories present.
```

Visualisation:

[16]:
```
# Set extent of maps created in the following cells:
axis_extent=[-95,-89,28,32]
```

[17]:
```
# Plot map with all individual tracks:
import cartopy.crs as ccrs
fig_map,ax_map=plt.subplots(figsize=(10,10),subplot_kw={'projection': ccrs.PlateCarree()}
↪)
ax_map=tobac.map_tracks(Track,axis_extent=axis_extent,axes=ax_map)
```



[18]:
```
# Create animation of tracked clouds and outlines with OLR as a background field
animation_test_tobac=tobac.animation_mask_field(Track,Features,OLR,Mask_OLR,
                                                axis_extent=axis_extent,#figsize=figsize,
↪orientation_colorbar='horizontal',pad_colorbar=0.2,
                                                vmin=80,vmax=330,cmap='Blues_r',
                                                plot_outline=True,plot_marker=True,marker_
↪track='x',plot_number=True,plot_features=True)
```

[19]:
```
# Display animation:
from IPython.display import HTML, Image, display
HTML(animation_test_tobac.to_html5_video())
```

[19]:
```
<IPython.core.display.HTML object>
```

```
<Figure size 640x480 with 0 Axes>
```

[20]:
```
# # Save animation to file:
# savefile_animation=plot_dir / 'Animation.mp4'
# animation_test_tobac.save(savefile_animation,dpi=200)
# print(f'animation saved to {savefile_animation}')
```

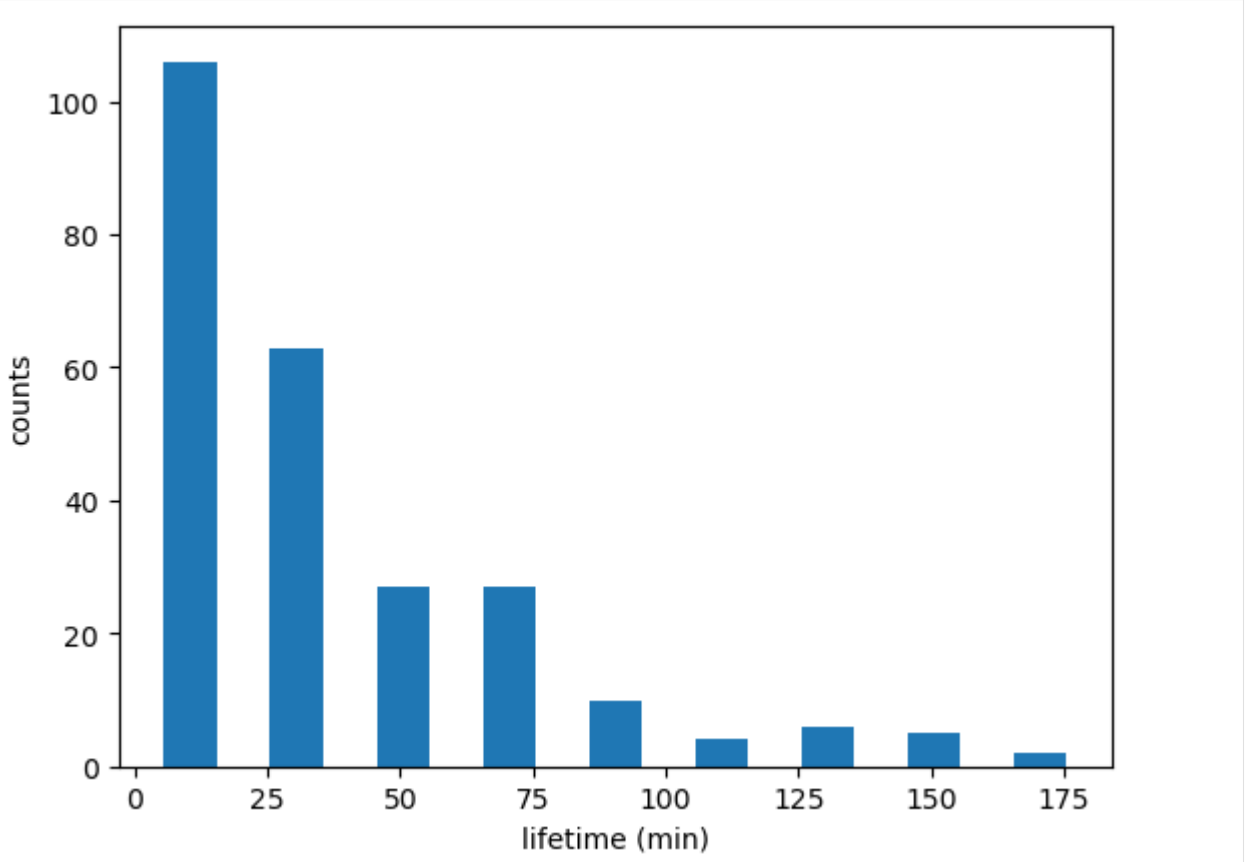
[21]:
```
# Lifetimes of tracked clouds:
fig_lifetime,ax_lifetime=plt.subplots()
tobac.plot_lifetime_histogram_bar(Track,axes=ax_lifetime,bin_edges=np.arange(0,200,20),
↪density=False,width_bar=10)
ax_lifetime.set_xlabel('lifetime (min)')
```

(continues on next page)

(continued from previous page)

```
ax_lifetime.set_ylabel('counts')
```

[21]:
```
Text(0, 0.5, 'counts')
```



[ ]:
```
```

6.8 Track on Radar data, Segment on satellite data

This notebooks shows: - how to input NEXRAD radar data and GOES satellite data from Amazon cloud service and prepare it for use in tobac - how to identify features on one data type (3D radar data) and to connect another data type (satellite data) to these features via segmentation

This is an idealized show case that ignores potential data mismatches, in our case the mismatch due to parallax effects.

6.8.1 Library Imports

*Note: In addition to the normal*tobac* requirements, this tutorial also requires that Py-ART and s3fs are installed.*

[1]:
```
# Import libraries:
import pyart
import xarray as xr
import s3fs
import numpy as np
import datetime
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
from matplotlib.pyplot import cm as cmaps
import pandas as pd
from pathlib import Path
import iris
from pyproj import Proj, Geod

%matplotlib inline
```

```
## You are using the Python ARM Radar Toolkit (Py-ART), an open source
## library for working with weather radar data. Py-ART is partly
## supported by the U.S. Department of Energy as part of the Atmospheric
## Radiation Measurement (ARM) Climate Research Facility, an Office of
## Science user facility.
##
## If you use this software to prepare a publication, please cite:
##
##     JJ Helmus and SM Collis, JORS 2016, doi: 10.5334/jors.119
```

[2]:
```
# Disable a few warnings:
import warnings

warnings.filterwarnings("ignore", category=UserWarning, append=True)
warnings.filterwarnings("ignore", category=RuntimeWarning, append=True)
warnings.filterwarnings("ignore", category=FutureWarning, append=True)
warnings.filterwarnings("ignore", category=pd.io.pytables.PerformanceWarning)
```

[3]:
```
# Import tobac itself:
import tobac
import tobac.utils

print("using tobac version", str(tobac.__version__))
```

```
using tobac version 1.5.3
```

6.8.2 Data Input and Preparations

Reading radar data and satellite data from Amazon S3

[4]:
```
# read in radar data
radar = pyart.io.read_nexrad_archive(
    "s3://noaa-nexrad-level2/2021/05/26/KGLD/KGLD20210526_155623_V06"
)
```

[5]:
```
# read in satellite data
fs = s3fs.S3FileSystem(anon=True)
aws_url = "s3://noaa-goes16/ABI-L2-MCMIPC/2021/146/15/OR_ABI-L2-MCMIPC-M6_G16_
↪s20211461556154_e20211461558539_c20211461559030.nc"

goes_data = xr.open_dataset(fs.open(aws_url), engine="h5netcdf")
```

[6]:
```
#Set up directory to save output and plots:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

Preparing GOES satellite data

In the next steps, we - calculate longitude and latitude values using geostationary satellite projection - construct a GOES dataset with lon/lat as coordinates - finally, convert the dataset into an Iris Cube (which is taken as tobac input)

[7]:
```
"""
Because the GOES data comes in without latitude/longitude values, we need to calculate␣
↪those.
"""


def lat_lon_reproj(g16nc):
    # GOES-R projection info and retrieving relevant constants
    proj_info = g16nc["goes_imager_projection"]
    lon_origin = proj_info.attrs["longitude_of_projection_origin"]
    H = proj_info.attrs["perspective_point_height"] + proj_info.attrs["semi_major_axis"]
    r_eq = proj_info.attrs["semi_major_axis"]
    r_pol = proj_info.attrs["semi_minor_axis"]

    # grid info
    lat_rad_1d = g16nc.variables["x"][:]
    lon_rad_1d = g16nc.variables["y"][:]

    # create meshgrid filled with radian angles
    lat_rad, lon_rad = np.meshgrid(lat_rad_1d, lon_rad_1d)

    # lat/lon calc routine from satellite radian angle vectors
```

(continues on next page)

(continued from previous page)

```
    lambda_0 = (lon_origin * np.pi) / 180.0

    a_var = np.power(np.sin(lat_rad), 2.0) + (
        np.power(np.cos(lat_rad), 2.0)
        * (
            np.power(np.cos(lon_rad), 2.0)
            + (((r_eq * r_eq) / (r_pol * r_pol)) * np.power(np.sin(lon_rad), 2.0))
        )
    )
    b_var = -2.0 * H * np.cos(lat_rad) * np.cos(lon_rad)
    c_var = (H**2.0) - (r_eq**2.0)

    r_s = (-1.0 * b_var - np.sqrt((b_var**2) - (4.0 * a_var * c_var))) / (2.0 * a_var)

    s_x = r_s * np.cos(lat_rad) * np.cos(lon_rad)
    s_y = -r_s * np.sin(lat_rad)
    s_z = r_s * np.cos(lat_rad) * np.sin(lon_rad)

    lat = (180.0 / np.pi) * (
        np.arctan(
            ((r_eq * r_eq) / (r_pol * r_pol))
            * ((s_z / np.sqrt(((H - s_x) * (H - s_x)) + (s_y * s_y))))
        )
    )
    lon = (lambda_0 - np.arctan(s_y / (H - s_x))) * (180.0 / np.pi)

    return lon, lat
```

[8]:
```
llons, llats = lat_lon_reproj(goes_data)

full_goes_data = goes_data["CMI_C14"].values
in_goes_for_tobac = xr.Dataset(
    data_vars={
        "C14_brightness_temperature": (
            ("time", "Y", "X"),
            [full_goes_data],
        )
    },
    coords={
        "time": [goes_data["time_bounds"][0].values],
        "longitude": (["Y", "X"], llons),
        "latitude": (["Y", "X"], llats),
    },
)
```

[9]:
```
def enhanced_colormap(vmin=200.0, vmed=240.0, vmax=300.0):
    '''
    Creates enhanced colormap typical of IR BTs.
    '''
    nfull = 256
```

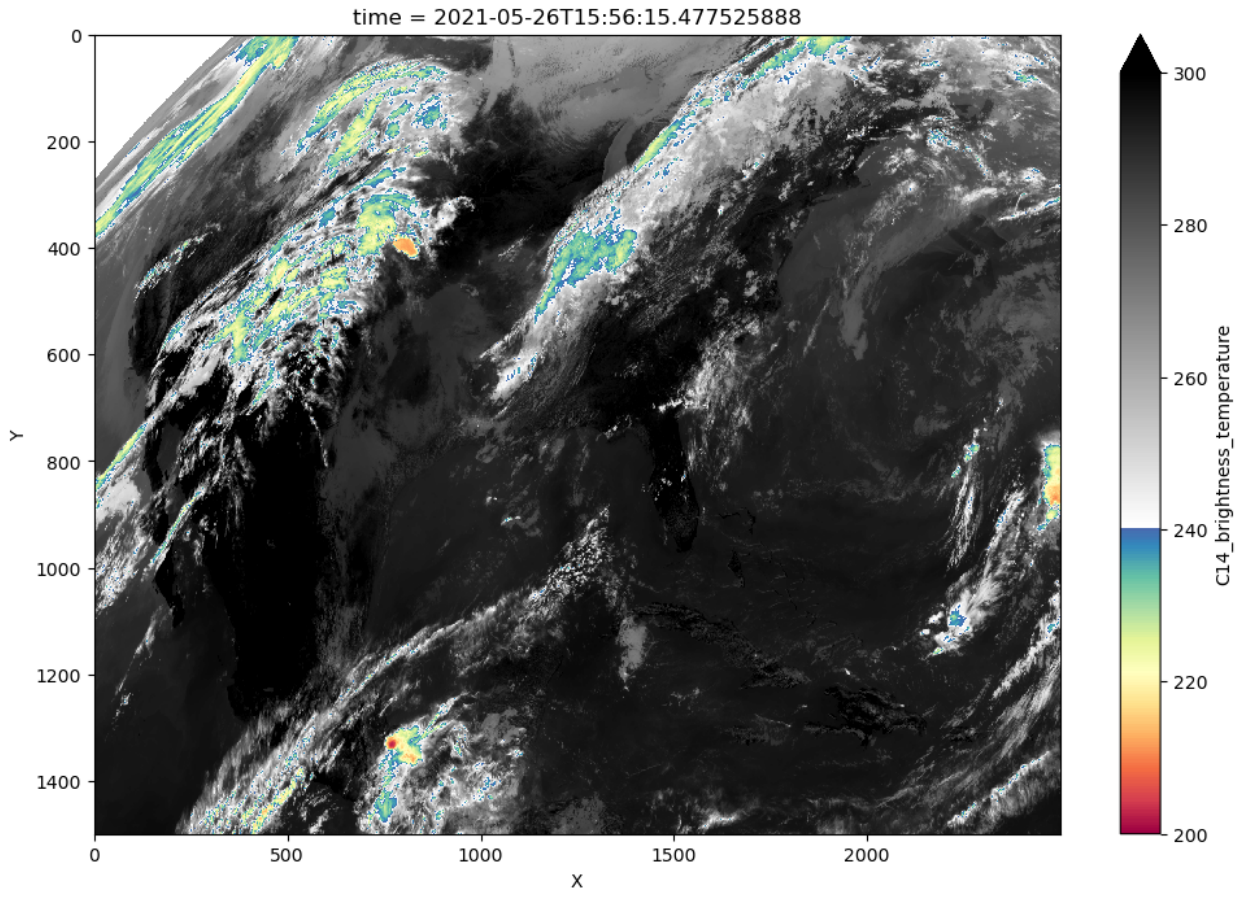
(continues on next page)

(continued from previous page)

```
    ngray = int(nfull * (vmax - vmed) / (vmax - vmin))
    ncol = nfull - ngray

    colors1 = plt.cm.gray_r(np.linspace(0.0, 1.0, ngray))
    colors2 = plt.cm.Spectral(np.linspace(0., 0.95, ncol))

    # combine them and build a new colormap
    colors = np.vstack((colors2, colors1))
    mymap = plt.matplotlib.colors.LinearSegmentedColormap.from_list(
        "enhanced_colormap", colors
    )

    return mymap
```

[10]:
```
in_goes_for_tobac["C14_brightness_temperature"].plot(
    yincrease=False, vmin=200, vmax=300.0, cmap=enhanced_colormap(), figsize=(12, 8)
)
```

[10]:
```
<matplotlib.collections.QuadMesh at 0x13c0c8bd0>
```



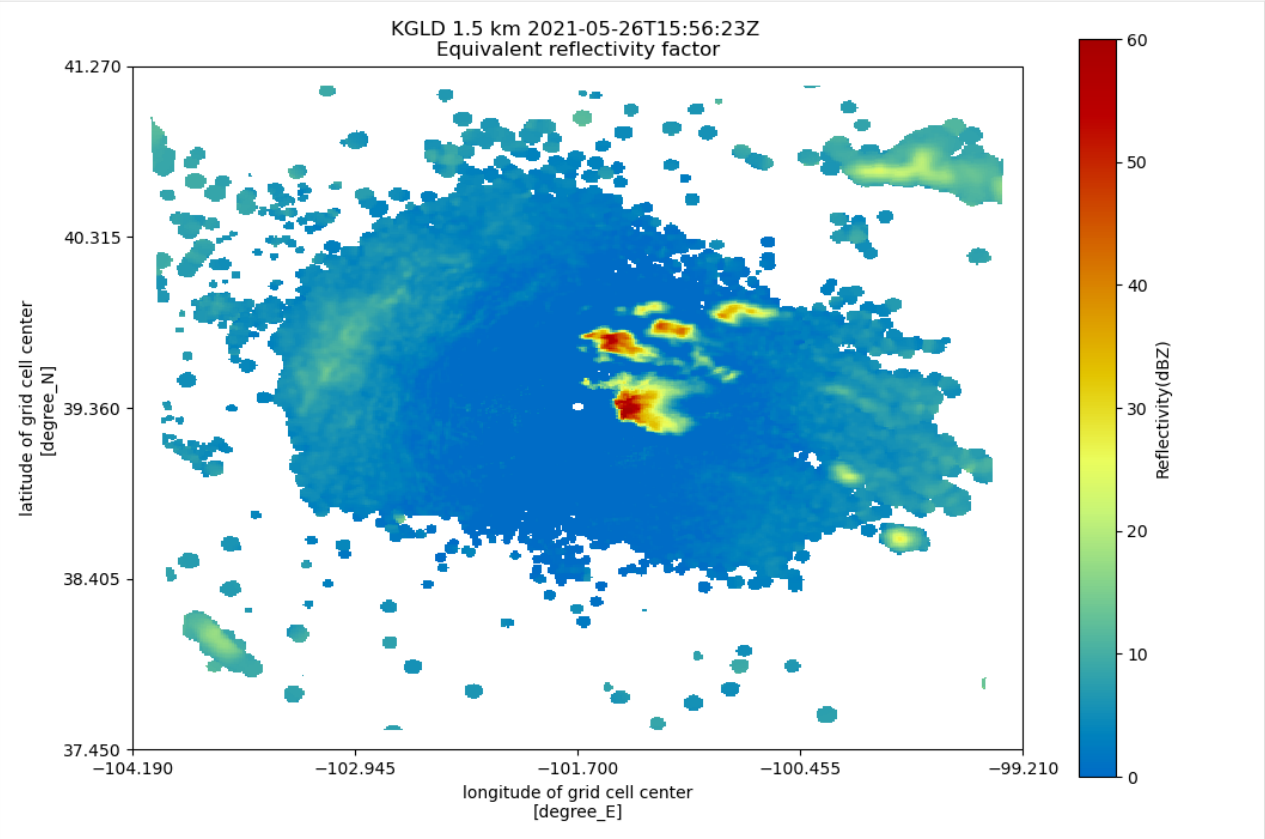
[11]:
```
goes_array_iris = in_goes_for_tobac["C14_brightness_temperature"].to_iris()
```

Preparing Radar Data

In the next steps, we - map radar data onto a regular grid - construct a 3D radar dataset and add longitude, latitude and altitude as coordinates - finally, convert the dataset into an Iris Cube (which is taken as tobac input)

[12]:
```
# First, we need to grid the input radar data.
radar_grid_pyart = pyart.map.grid_from_radars(
    radar,
    grid_shape=(41, 401, 401),
    grid_limits=(
        (
            0.0,
            20000,
        ),
        (-200000.0, 200000.0),
        (-200000, 200000.0),
    ),
)
```

[13]:
```
# In Py-ART's graphing suite, there is a display class similar to RadarMapDisplay,
# but for grids. To plot the grid:
fig = plt.figure(figsize=[12, 8])
plt.axis("off")

display = pyart.graph.GridMapDisplay(radar_grid_pyart)
display.plot_grid("reflectivity", level=3, vmin=0, vmax=60, fig=fig)
```



[14]:
```
radar_xarray_grid = radar_grid_pyart.to_xarray()
```

[15]:
```
radar_xr_grid_full = radar_xarray_grid["reflectivity"][:, 4]
radar_xr_grid_full["z"] = radar_xr_grid_full.z.assign_attrs(
    {"standard_name": "altitude"}
)
radar_xr_grid_full["lat"] = radar_xr_grid_full.lat.assign_attrs(
    {"standard_name": "latitude"}
)
radar_xr_grid_full["lon"] = radar_xr_grid_full.lon.assign_attrs(
    {"standard_name": "longitude"}
)
```

[16]:
```
radar_grid_iris = radar_xr_grid_full.to_iris()
```

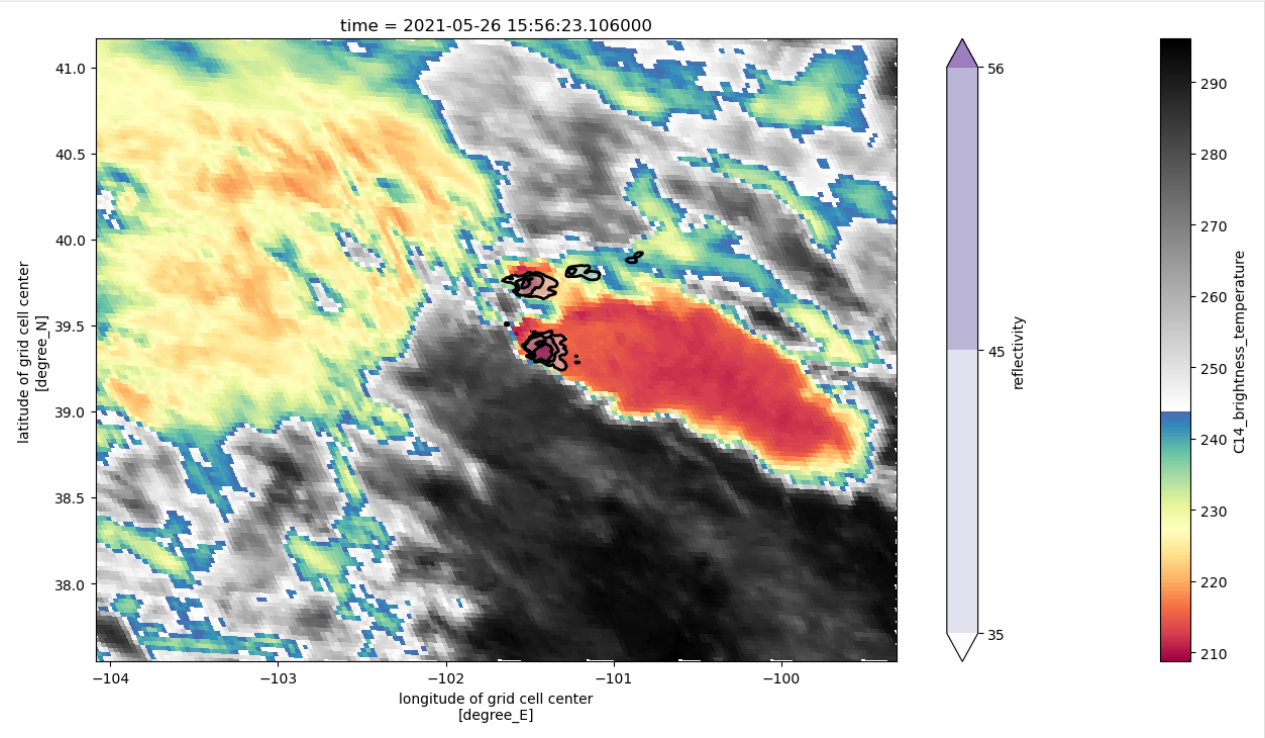
Combined Plot

[17]:
```
# first we cut satellite data a little bit
rlon_min, rlon_max = radar_xarray_grid.lon.min().data, radar_xarray_grid.lon.max().data
rlat_min, rlat_max = radar_xarray_grid.lat.min().data, radar_xarray_grid.lat.max().data

lon_mask = (llons >= rlon_min) & (llons <= rlon_max)
lat_mask = (llats >= rlat_min) & (llats <= rlat_max)
mask = lon_mask & lat_mask


goes_cutted = (
    in_goes_for_tobac.where(mask).dropna("X", how="all").dropna("Y", how="all")
)
```

[18]:
```
Zmax = radar_xarray_grid['reflectivity'].max('z').squeeze()
```

[19]:
```
goes_cutted["C14_brightness_temperature"].plot(
    x="longitude", y="latitude", cmap=enhanced_colormap(), figsize=(16, 8)
)

Zmax.where(Zmax > 34).plot.contourf(
    x="lon",
    y="lat",
    cmap=plt.cm.Purples,
    levels=[35, 45, 56],
    alpha=0.5,
)

Zmax.where(Zmax > 20).plot.contour(
    x="lon", y="lat", colors="k", levels=[35, 45, 56], linewidths=2
)

plt.xlim(rlon_min, rlon_max)
plt.ylim(rlat_min, rlat_max)
```

[19]:
```
(37.54600438517133, 41.16558741816462)
```



6.8.3 Tobac Cloud Tracking

Feature detection

We use multiple thresholds to detect features in the radar data.

[20]:
```
feature_detection_params = dict()
feature_detection_params["threshold"] = [30, 40, 50]
feature_detection_params["target"] = "maximum"
feature_detection_params["position_threshold"] = "weighted_diff"
feature_detection_params["n_erosion_threshold"] = 2
feature_detection_params["sigma_threshold"] = 1
feature_detection_params["n_min_threshold"] = 4
```

[21]:
```
# Perform feature detection:
print('starting feature detection')
radar_features = tobac.feature_detection.feature_detection_multithreshold(
    radar_grid_iris, 0, **feature_detection_params
)
radar_features.to_hdf(savedir / 'Features.h5', 'table')
print('feature detection performed and saved')
```

```
starting feature detection
```

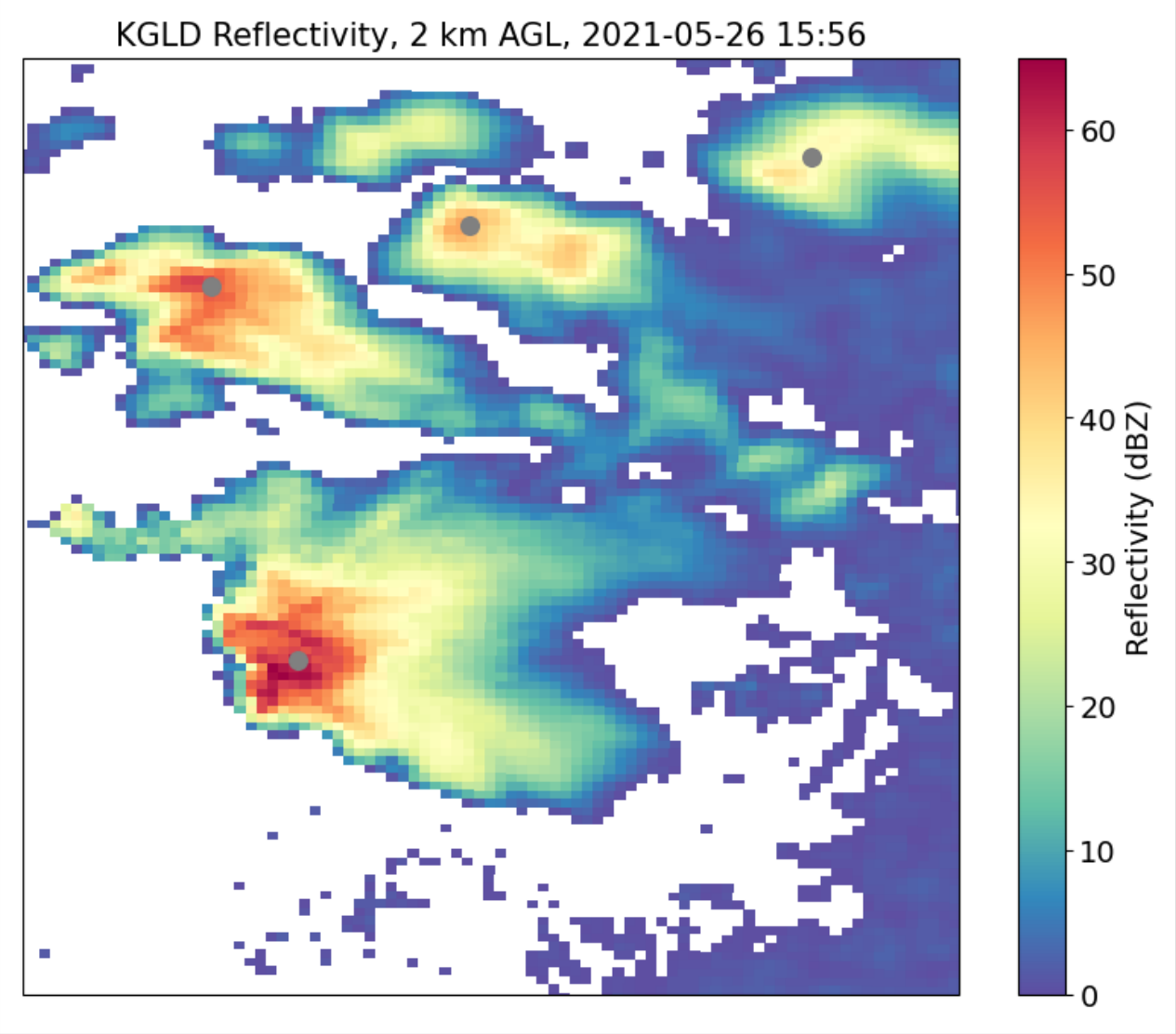
```
feature detection performed and saved
```

[22]:
```
radar_features
```

[22]:
```
   frame  idx      hdim_1      hdim_2  num  threshold_value  feature  \
0      0    5  258.862284  271.851509   19               30        1
1      0    9  250.469609  240.740283    6               40        2
2      0   11  198.778758  225.305252   54               50        3
3      0   12  243.316368  217.254572    7               50        4

                  time             timestr  projection_y_coordinate  \
0  2021-05-26 15:56:23  2021-05-26 15:56:23             58862.283732
1  2021-05-26 15:56:23  2021-05-26 15:56:23             50469.608638
2  2021-05-26 15:56:23  2021-05-26 15:56:23             -1221.242078
3  2021-05-26 15:56:23  2021-05-26 15:56:23             43316.368018

   projection_x_coordinate      altitude   latitude   longitude
0             71851.509127  71851.509127  39.893281 -100.858070
1             40740.283179  40740.283179  39.819857 -101.223255
2             25305.251764  25305.251764  39.355590 -101.405959
3             17254.571781  17254.571781  39.756323 -101.498434
```

[23]:
```
# we now have 4 detected features

fig = plt.figure(figsize=[10, 8])
ax = fig.add_subplot(1, 1, 1, projection=ccrs.PlateCarree())
ax.set_extent([-101.7, -100.7, 39, 40], crs=ccrs.PlateCarree())


Z = radar_xarray_grid["reflectivity"][0, 4]
cm = ax.pcolormesh(
    radar_xarray_grid["lon"],
    radar_xarray_grid["lat"],
    Z.where(Z> 0),
    vmin=0,
    vmax=65,
    transform=ccrs.PlateCarree(),
    cmap="Spectral_r",
)
plt.xlim(-101.7, -100.7)
plt.ylim(39.0, 40)
cb = plt.colorbar(cm)
cb.set_label("Reflectivity (dBZ)", size=14)
cb.ax.tick_params(labelsize=14)
plt.title("KGLD Reflectivity, 2 km AGL, 2021-05-26 15:56", size=15)
plt.scatter(
    radar_features["longitude"],
    radar_features["latitude"],
    70,
    transform=ccrs.PlateCarree(),
    color="grey",
)
```

[23]:
```
<matplotlib.collections.PathCollection at 0x13913f150>
```



[24]:
```
# maximum space away in m,  maximum time away as Python Datetime object
goes_adj_features = tobac.utils.transform_feature_points(
    radar_features,
    goes_array_iris,
    max_time_away=datetime.timedelta(minutes=1),
    max_space_away=20 * 1000,
)
```

[25]:
```
# the transformed dataframe needs to have identical time to the data to segment on.
# replacement_dt = np.datetime64(in_goes_for_tobac['time'][0].values, 's')
# however, iris cannot deal with times in ms, so we need to drop the ms values.

# goes_adj_features['time'] = replacement_dt
```

[26]:
```
goes_adj_features
```

[26]:
```
       frame   idx  hdim_1  hdim_2   num  threshold_value  feature  \
index
0        0.0   5.0   372.0   806.0  19.0             30.0        1
1        0.0   9.0   376.0   791.0   6.0             40.0        2
2        0.0  11.0   393.0   777.0  54.0             50.0        3
3        0.0  12.0   379.0   781.0   7.0             50.0        4

                     time              timestr  projection_y_coordinate  \
index
0     2021-05-26 15:56:15  2021-05-26 15:56:23              58862.283732
1     2021-05-26 15:56:15  2021-05-26 15:56:23              50469.608638
2     2021-05-26 15:56:15  2021-05-26 15:56:23              -1221.242078
3     2021-05-26 15:56:15  2021-05-26 15:56:23              43316.368018

       projection_x_coordinate      altitude   latitude   longitude
index
0                  71851.509127  71851.509127  39.893281 -100.858070
1                  40740.283179  40740.283179  39.819857 -101.223255
2                  25305.251764  25305.251764  39.355590 -101.405959
3                  17254.571781  17254.571781  39.756323 -101.498434
```

Segmentation:

[27]:
```
parameters_segmentation = dict()
parameters_segmentation["method"] = "watershed"
parameters_segmentation["threshold"] = 235
parameters_segmentation["target"] = "minimum"
```

[28]:
```
# Perform segmentation and save results:
print('Starting segmentation.')
seg_data, seg_feats = tobac.segmentation.segmentation(
    goes_adj_features, goes_array_iris, dxy=2000, **parameters_segmentation
)
print('segmentation performed, start saving results to files')
iris.save([seg_data], savedir / 'Mask_Segmentation_sat.nc', zlib=True, complevel=4)
seg_feats.to_hdf(savedir / 'Features_sat.h5', 'table')
print('segmentation performed and saved')
```

```
Starting segmentation.
```

```
segmentation performed, start saving results to files
```

```
segmentation performed and saved
```

[29]:
```
seg_data_xr = xr.DataArray.from_iris(seg_data)
```

[30]:
```
# In Py-ART's graphing suite, there is a display class similar to RadarMapDisplay,
# but for grids. To plot the grid:
fig = plt.figure(figsize=[10, 8])
ax = fig.add_subplot(1, 1, 1, projection=ccrs.PlateCarree())
ax.set_extent([-101.7, -100.7, 39, 40], crs=ccrs.PlateCarree())

contoured = ax.contourf(
    in_goes_for_tobac["longitude"],
```

(continues on next page)

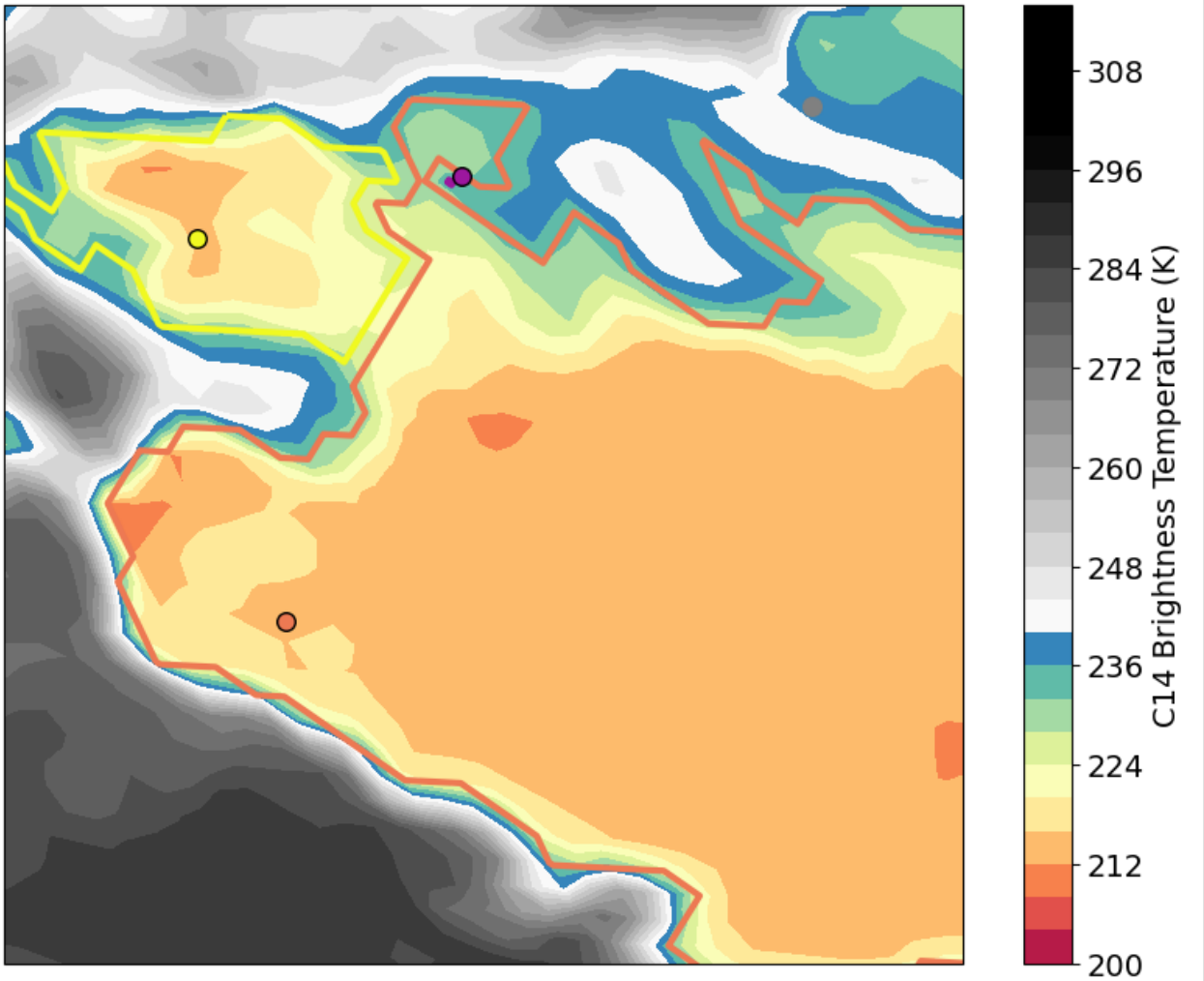
(continued from previous page)

```
    in_goes_for_tobac["latitude"],
    in_goes_for_tobac["C14_brightness_temperature"][0],
    transform=ccrs.PlateCarree(),
    cmap=enhanced_colormap(), vmin = 200, vmax = 300, levels = 31
)
plt.xlim(-101.7, -100.7)
plt.ylim(39.0, 40)
unique_seg = np.unique(seg_data_xr)
color_map = cmaps.plasma(np.linspace(0, 1, len(unique_seg)))

# we have one feature without a segmented area
curr_feat = goes_adj_features[goes_adj_features["feature"] == 1]
plt.scatter(
    curr_feat["longitude"],
    curr_feat["latitude"],
    70,
    transform=ccrs.PlateCarree(),
    color="grey",
)


for seg_num, color in zip(unique_seg, color_map):
    if seg_num == 0 or seg_num == -1:
        continue
    curr_seg = (seg_data_xr == seg_num).astype(int)
    ax.contour(
        seg_data_xr["longitude"],
        seg_data_xr["latitude"],
        curr_seg,
        colors=[
            color,
        ],
        levels=[
            0.9,1
        ],
        linewidths=3,
    )
    curr_feat = goes_adj_features[goes_adj_features["feature"] == seg_num]
    plt.scatter(
        curr_feat["longitude"],
        curr_feat["latitude"],
        70,
        transform=ccrs.PlateCarree(),
        color=color,
        edgecolors = 'black',
        zorder = 10
    )

cb = plt.colorbar(contoured)
cb.set_label("C14 Brightness Temperature (K)", size=14)
cb.ax.tick_params(labelsize=14)
# plt.title("KGLD Reflectivity, 2 km AGL, 2021-05-26 15:56", size=15)
```

(continues on next page)

(continued from previous page)

```
# plt.savefig("./radar_example_2/satellite_wseg.png", facecolor='w', bbox_inches='tight')
```



result: - convective anvils could be connected to two respective radar cells - ignoring parallax shifts can cause assignment problems for smaller anvils (see purple radar cell)

6.9 tobac example: Tracking of deep convection based on OLR from convection permitting model simulations

This example notebook demonstrates the use of tobac to track deep convection based on the outgoing longwave radiation (OLR) from convection permitting simulations.

The simulation results used in this example were performed as part of the ACPC deep convection intercomparison case study (http://acpcinitiative.org/Docs/ACPC_DCC_Roadmap_171019.pdf) with WRF using the Morrison microphysics scheme. Simulations were performed with a horizontal grid spacing of 4.5 km.

The data used in this example is downloaded from "zenodo link" automatically as part of the notebooks (This only has to be done once for all the tobac example notebooks).

Import libraries:

[1]:
```
# Import a range of python libraries used in this notebook:
import iris
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import iris.plot as iplt
import iris.quickplot as qplt
import shutil
import datetime
from six.moves import urllib
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Import tobac itself:
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[3]:
```
# Disable a few warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

Download example data:
The actual download is only necessary once for all example notebooks.

[4]:
```
data_out=Path('../')
```

[5]:
```
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_OLR_model.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_OLR_model.nc'))
```

[6]:
```
#Load Data from downloaded file:
OLR=iris.load_cube(str(data_file[0]),'OLR')
```

[7]:
```
#Set up directory to save output and plots:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

Feature detection:
Feature detection is performed based on OLR field and a set of thresholds.

[8]:
```
# Determine temporal and spatial sampling:
dxy,dt=tobac.get_spacings(OLR)
```

[9]:
```
# Dictionary containing keyword arguments for feature detection step (Keywords could␣
↪also be given directly in the function call).
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['n_min_threshold']=4
parameters_features['target']='minimum'
parameters_features['threshold']=[250,225,200,175,150]
```

[10]:
```
# Perform feature detection:
print('starting feature detection')
Features=tobac.feature_detection_multithreshold(OLR,dxy, **parameters_features)
Features.to_hdf(savedir / 'Features.h5', 'table')
print('feature detection performed and saved')
```

```
starting feature detection
```

```
feature detection performed and saved
```

Segmentation:
Segmentation is performed with watershedding based on the detected features and a single threshold value.

[11]:
```
# Dictionary containing keyword options for the segmentation step:
parameters_segmentation={}
parameters_segmentation['target']='minimum'
parameters_segmentation['method']='watershed'
parameters_segmentation['threshold']=250
```

[12]:
```
# Perform segmentation and save results:
print('Starting segmentation based on OLR.')
Mask_OLR,Features_OLR=tobac.segmentation_2D(Features,OLR,dxy,**parameters_segmentation)
print('segmentation OLR performed, start saving results to files')
iris.save([Mask_OLR], savedir / 'Mask_Segmentation_OLR.nc', zlib=True, complevel=4)
```

(continues on next page)

(continued from previous page)

```
Features_OLR.to_hdf(savedir / 'Features_OLR.h5', 'table')
print('segmentation OLR performed and saved')
```

```
Starting segmentation based on OLR.
```

```
segmentation OLR performed, start saving results to files
segmentation OLR performed and saved
```

Trajectory linking:

Features are linked into cloud trajectories using the trackpy library (http://soft-matter.github.io/trackpy). This takes the feature positions determined in the feature detection step into account but does not include information on the shape of the identified objects.**

[13]:
```
# Arguments for trajectory linking:
parameters_linking={}
parameters_linking['v_max']=20
parameters_linking['stubs']=2
parameters_linking['order']=1
parameters_linking['extrapolate']=0
parameters_linking['memory']=0
parameters_linking['adaptive_stop']=0.2
parameters_linking['adaptive_step']=0.95
parameters_linking['subnetwork_size']=100
parameters_linking['method_linking']= 'predict'
```

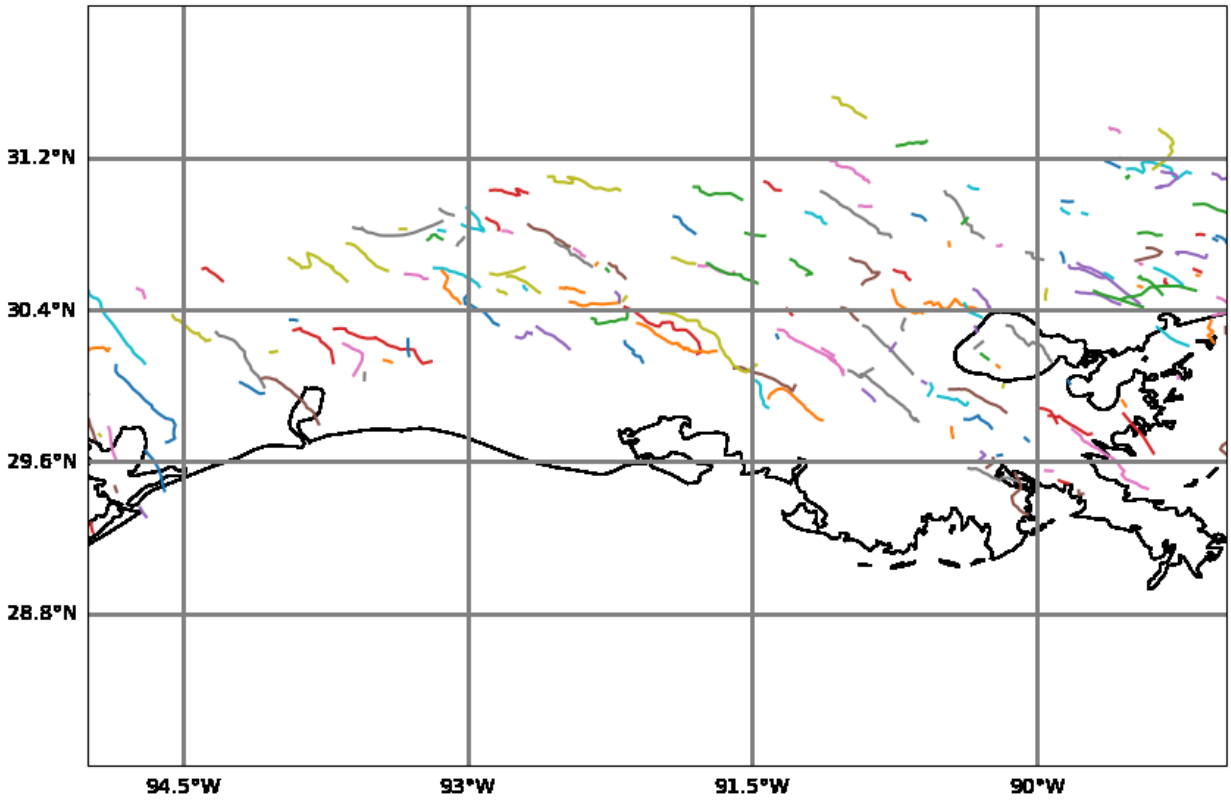
[14]:
```
# Perform linking and save results to file:
Track=tobac.linking_trackpy(Features, OLR, dt=dt, dxy=dxy, **parameters_linking)
Track.to_hdf(savedir / 'Track.h5', 'table')
```

```
Frame 95: 12 trajectories present.
```

Visualisation:

[15]:
```
# Set extent of maps created in the following cells:
axis_extent = [-95, -89, 28, 32]
```

[16]:
```
# Plot map with all individual tracks:
import cartopy.crs as ccrs
fig_map, ax_map = plt.subplots(figsize=(10,10), subplot_kw={'projection': ccrs.
↪PlateCarree()})
ax_map = tobac.map_tracks(Track, axis_extent=axis_extent,  axes=ax_map)
```



[17]:
```
# Create animation of tracked clouds and outlines with OLR as a background field
animation_test_tobac=tobac.animation_mask_field(Track,Features,OLR,Mask_OLR,
                                                axis_extent=axis_extent,#figsize=figsize,
↪orientation_colorbar='horizontal',pad_colorbar=0.2,
                                                vmin=80,vmax=330,
                                                plot_outline=True,plot_marker=True,marker_
↪track='x',plot_number=True,plot_features=True)
```

[18]:
```
# Display animation:
from IPython.display import HTML, Image, display
HTML(animation_test_tobac.to_html5_video())
```

[18]:
```
<IPython.core.display.HTML object>
```

```
<Figure size 640x480 with 0 Axes>
```

[19]:
```
# # Save animation to file:
# savefile_animation = plot_dir /'Animation.mp4'
# animation_test_tobac.save(savefile_animation,dpi=200)
# print(f'animation saved to {savefile_animation}')
```

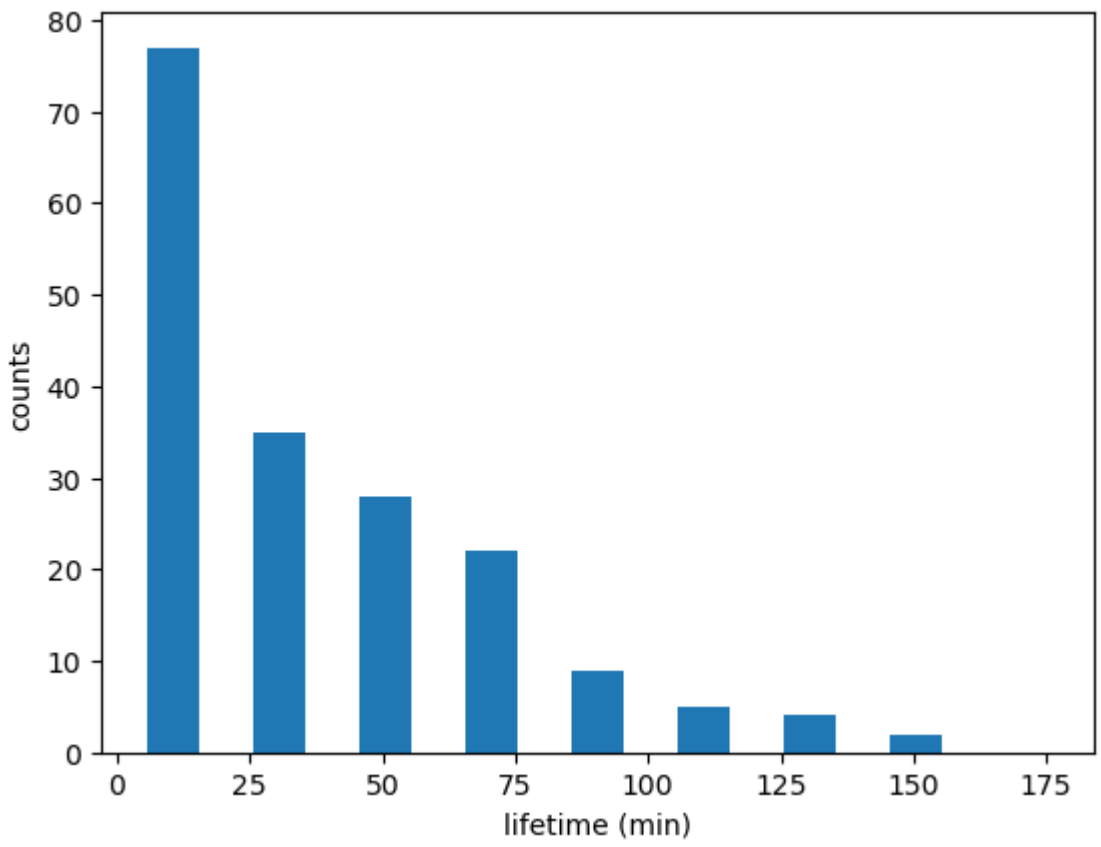
[20]:
```
# Lifetimes of tracked clouds:
fig_lifetime,ax_lifetime=plt.subplots()
tobac.plot_lifetime_histogram_bar(Track,axes=ax_lifetime,bin_edges=np.arange(0,200,20),
↪density=False,width_bar=10)
ax_lifetime.set_xlabel('lifetime (min)')
```

(continues on next page)

(continued from previous page)

```
ax_lifetime.set_ylabel('counts')
```

[20]:
```
Text(0, 0.5, 'counts')
```



[ ]:
```
```

6.10 tobac example: Tracking of precipitation features

This example notebook demonstrates the use of tobac to track precipitation features from isolated deep convective clouds.

The simulation results used in this example were performed as part of the ACPC deep convection intercomparison case study (http://acpcinitiative.org/Docs/ACPC_DCC_Roadmap_171019.pdf) with WRF using the Morrison microphysics scheme.

The data used in this example is downloaded from "zenodo link" automatically as part of the notebooks (This only has to be done once for all the tobac example notebooks).

Import necessary python libraries:

[1]:
```
# Import libraries
import iris
import numpy as np
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import iris.plot as iplt
import iris.quickplot as qplt
import datetime
import shutil
from six.moves import urllib
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Import tobac itself
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[3]:
```
# Disable a few warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

Download example data:

Actual download has to be performed only once for all example notebooks!

[4]:
```
data_out=Path('../')
```

[5]:
```
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
```

Load Data from downloaded file:

[6]:
```
Precip=iris.load_cube(str(data_file[0]),'surface_precipitation_average')
```

[7]:
```
#display information about the iris cube containing the surface precipitation data:
display(Precip)
```

```
<iris 'Cube' of surface_precipitation_average / (mm h-1) (time: 47; south_north: 198;␣
↪west_east: 198)>
```

[8]:
```
#Set up directory to save output and plots:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

Feature detection:

Feature detection is perfomed based on surface precipitation field and a range of thresholds

[9]:
```
# Dictionary containing keyword options (could also be directly given to the function)
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['min_distance']=0
parameters_features['sigma_threshold']=1
parameters_features['threshold']=[1,2,3,4,5,10,15] #mm/h
parameters_features['n_erosion_threshold']=0
parameters_features['n_min_threshold']=3
```

[10]:
```
# get temporal and spation resolution of the data
dxy,dt=tobac.get_spacings(Precip)
```

[11]:
```
# Feature detection based on based on surface precipitation field and a range of␣
↪thresholds
print('starting feature detection based on multiple thresholds')
Features=tobac.feature_detection_multithreshold(Precip,dxy,**parameters_features)
print('feature detection done')
Features.to_hdf(savedir / 'Features.h5','table')
print('features saved')
```

```
starting feature detection based on multiple thresholds
```

```
feature detection done
```

```
features saved
```

Segmentation:

Segmentation is performed based on a watershedding and a threshold value:

[12]:
```
# Dictionary containing keyword arguments for segmentation step:
parameters_segmentation={}
parameters_segmentation['method']='watershed'
parameters_segmentation['threshold']=1  # mm/h mixing ratio
```

[13]:
```
# Perform Segmentation and save resulting mask to NetCDF file:
print('Starting segmentation based on surface precipitation')
Mask,Features_Precip=tobac.segmentation_2D(Features,Precip,dxy,**parameters_segmentation)
print('segmentation based on surface precipitation performed, start saving results to␣
↪files')
iris.save([Mask], savedir / 'Mask_Segmentation_precip.nc', zlib=True, complevel=4)
Features_Precip.to_hdf(savedir / 'Features_Precip.h5', 'table')
```

(continues on next page)

(continued from previous page)

```
print('segmentation surface precipitation performed and saved')
```

```
Starting segmentation based on surface precipitation
```

```
segmentation based on surface precipitation performed, start saving results to files
segmentation surface precipitation performed and saved
```

Trajectory linking:

Trajectory linking is performed using the trackpy library (http://soft-matter.github.io/trackpy). This takes the feature positions determined in the feature detection step into account but does not include information on the shape of the identified objects.

[14]:
```
# Dictionary containing keyword arguments for the linking step:
parameters_linking={}
parameters_linking['method_linking']='predict'
parameters_linking['adaptive_stop']=0.2
parameters_linking['adaptive_step']=0.95
parameters_linking['extrapolate']=0
parameters_linking['order']=1
parameters_linking['subnetwork_size']=100
parameters_linking['memory']=0
parameters_linking['time_cell_min']=5*60
parameters_linking['method_linking']='predict'
parameters_linking['v_max']=10
```

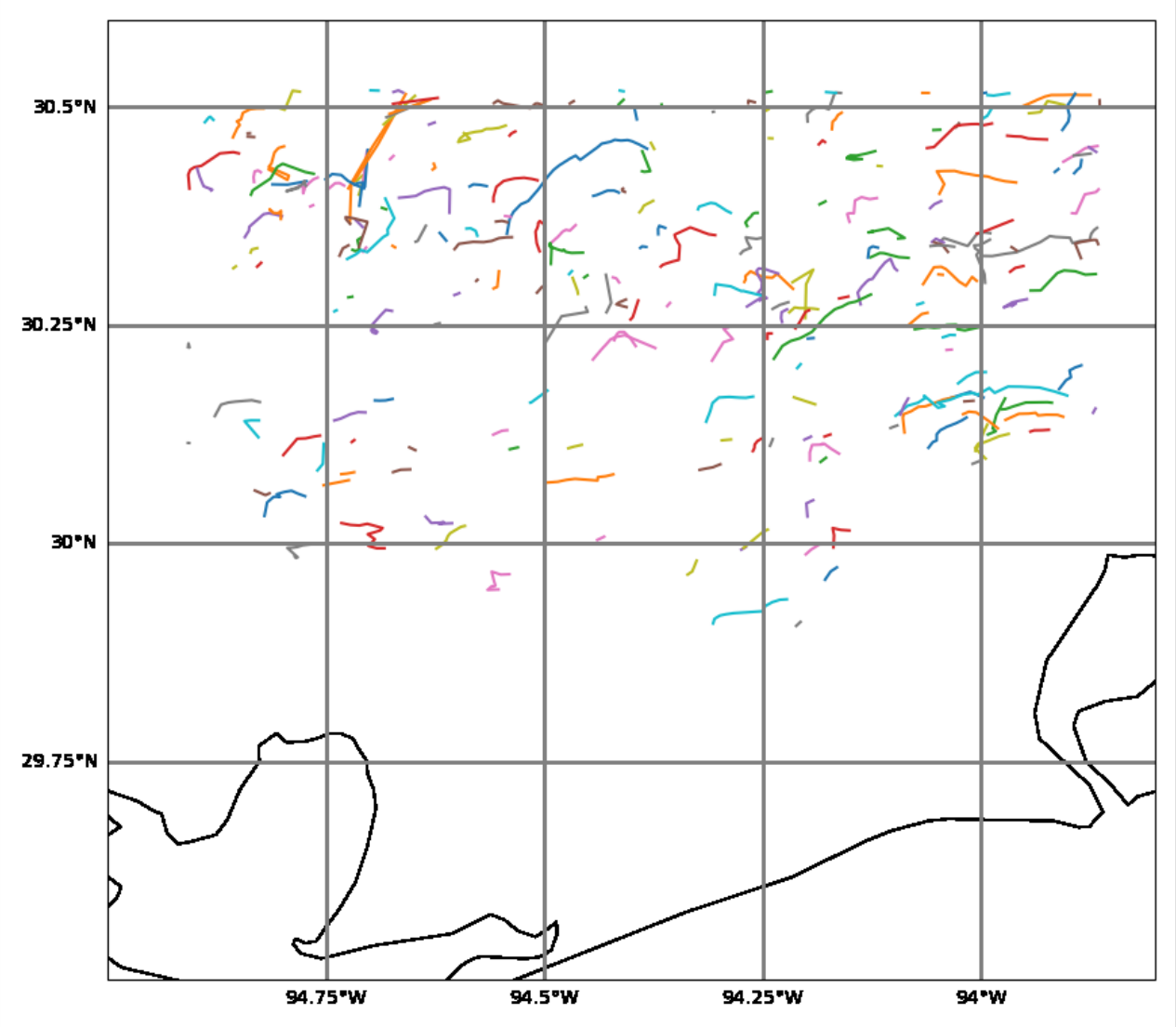
[15]:
```
# Perform trajectory linking using trackpy and save the resulting DataFrame:
Track=tobac.linking_trackpy(Features,Precip,dt=dt,dxy=dxy,**parameters_linking)
Track.to_hdf(savedir / 'Track.h5', 'table')
```

```
Frame 46: 18 trajectories present.
```

Visualistation:

[16]:
```
# Set extent for maps plotted in the following cells ( in the form [lon_min,lon_max,lat_
↪min,lat_max])
axis_extent=[-95,-93.8,29.5,30.6]
```

[17]:
```
# Plot map with all individual tracks:
import cartopy.crs as ccrs
fig_map,ax_map=plt.subplots(figsize=(10,10),subplot_kw={'projection': ccrs.PlateCarree()}
↪)
ax_map=tobac.map_tracks(Track,axis_extent=axis_extent,axes=ax_map)
```



[18]:
```
# Create animation showing tracked cells with outline of precipitation features and the␣
↪and surface precipitation as a background field:
animation_tobac=tobac.animation_mask_field(track=Track,features=Features,field=Precip,
↪mask=Mask,
                                           axis_extent=axis_extent,#figsize=figsize,
↪orientation_colorbar='horizontal',pad_colorbar=0.2,
                                           vmin=0,vmax=60,extend='both',cmap='Blues',
                                           interval=500,figsize=(10,10),
                                           plot_outline=True,plot_marker=True,marker_
↪track='x',plot_number=True,plot_features=True)
```

[19]:
```
# Display animation:
from IPython.display import HTML, Image, display
HTML(animation_tobac.to_html5_video())
```

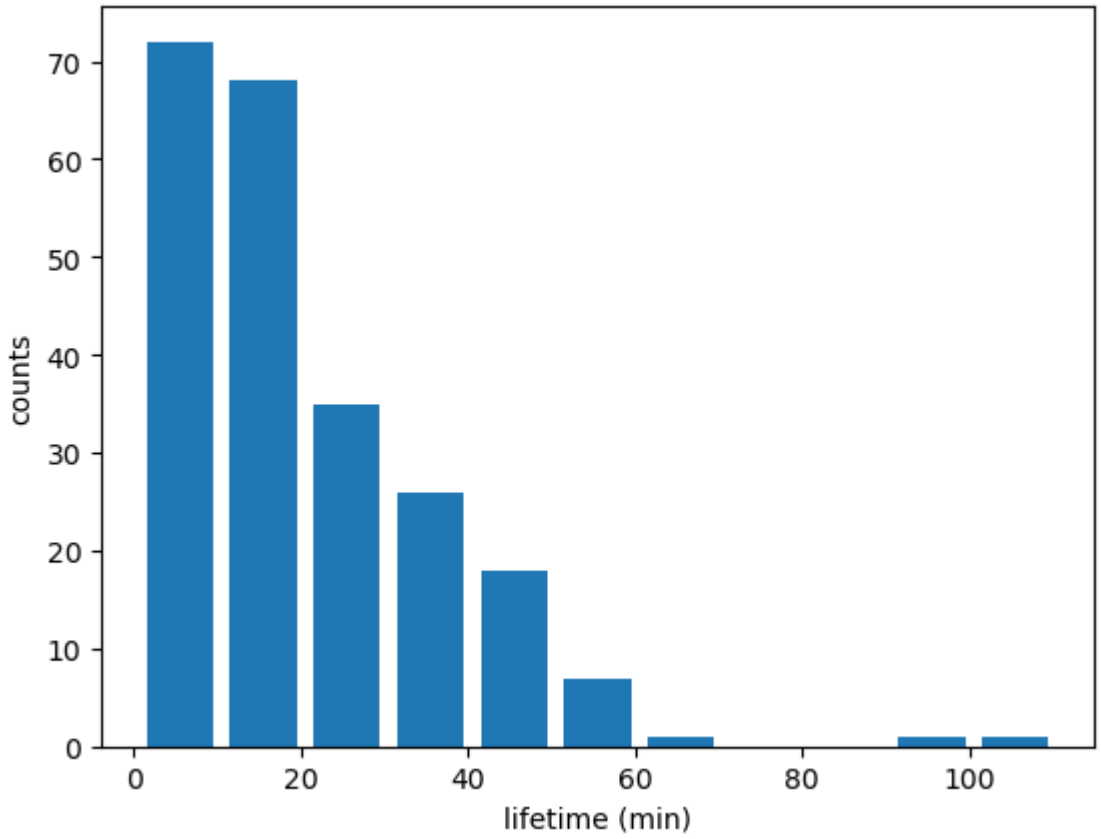
[19]:
```
<IPython.core.display.HTML object>
```

```
<Figure size 640x480 with 0 Axes>
```

[20]:
```
# # Save animation to file
# savefile_animation=plot_dir / 'Animation.mp4'
# animation_tobac.save(savefile_animation,dpi=200)
# print(f'animation saved to {savefile_animation}')
```

[21]:
```
# Lifetimes of tracked features:
fig_lifetime,ax_lifetime=plt.subplots()
tobac.plot_lifetime_histogram_bar(Track,axes=ax_lifetime,bin_edges=np.arange(0,120,10),
↪density=False,width_bar=8)
ax_lifetime.set_xlabel('lifetime (min)')
ax_lifetime.set_ylabel('counts')
```

[21]:
```
Text(0, 0.5, 'counts')
```



[ ]:

6.11 tobac example: Tracking isolated convection based on updraft velocity and total condensate

This example notebook demonstrates the use of tobac to track isolated deep convective clouds in cloud-resolving model simulation output based on vertical velocity and total condensate mixing ratio.

The simulation results used in this example were performed as part of the ACPC deep convection intercomparison case study (http://acpcinitiative.org/Docs/ACPC_DCC_Roadmap_171019.pdf) with WRF using the Morrison microphysics scheme.

The data used in this example is downloaded from "zenodo link" automatically as part of the notebooks (This only has to be done once for all the tobac example notebooks).

Import libraries:

[1]:
```
import iris
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import iris.plot as iplt
import iris.quickplot as qplt
import shutil
from six.moves import urllib
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Import tobac itself:
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[3]:
```
#Disable a couple of warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

Download and load example data:

The actual dowloading is only necessary once for all example notebooks.

[4]:
```
data_out=Path('../')
```

[5]:
```
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_midlevelUpdraft.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
```

(continues on next page)

(continued from previous page)

```
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_midlevelUpdraft.nc'))
```

Load Data from downloaded file:

[6]:
```
data_file_W_mid_max = list(data_out.rglob('data/Example_input_midlevelUpdraft.nc'))[0]
data_file_TWC = list(data_out.rglob('data/Example_input_Condensate.nc'))[0]
```

[7]:
```
W_mid_max=iris.load_cube(str(data_file_W_mid_max),'w')
TWC=iris.load_cube(str(data_file_TWC),'TWC')
```

[8]:
```
# Display information about the two cubes for vertical velocity and total condensate␣
↪mixing ratio:
display(W_mid_max)
display(TWC)
```

```
<iris 'Cube' of w / (m s-1) (time: 47; south_north: 198; west_east: 198)>
```

```
<iris 'Cube' of TWC / (kg kg-1) (time: 47; bottom_top: 94; south_north: 198; west_east:␣
↪198)>
```

[9]:
```
#Set up directory to save output and plots:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

Feature detection:

Perform feature detection based on midlevel maximum vertical velocity and a range of threshold values.

[10]:
```
# Determine temporal and spatial sampling of the input data:
dxy,dt=tobac.get_spacings(W_mid_max)
```

[11]:
```
# Keyword arguments for feature detection step:
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['min_distance']=0
parameters_features['sigma_threshold']=1
parameters_features['threshold']=[3,5,10] #m/s
parameters_features['n_erosion_threshold']=0
parameters_features['n_min_threshold']=3
```

[12]:
```
# Perform feature detection and save results:
print('start feature detection based on midlevel column maximum vertical velocity')
dxy,dt=tobac.get_spacings(W_mid_max)
Features=tobac.feature_detection_multithreshold(W_mid_max,dxy,**parameters_features)
print('feature detection performed start saving features')
Features.to_hdf(savedir / 'Features.h5', 'table')
print('features saved')
```

```
start feature detection based on midlevel column maximum vertical velocity
```

```
feature detection performed start saving features
```

```
features saved
```

Segmentation:
Perform segmentation based on 3D total condensate field to determine cloud volumes associated to identified features:

[13]:
```
parameters_segmentation_TWC={}
parameters_segmentation_TWC['method']='watershed'
parameters_segmentation_TWC['threshold']=0.1e-3  # kg/kg mixing ratio
```

[14]:
```
print('Start segmentation based on total water content')
Mask_TWC,Features_TWC=tobac.segmentation_3D(Features,TWC,dxy,**parameters_segmentation_
↪TWC)
print('segmentation TWC performed, start saving results to files')
iris.save([Mask_TWC], savedir / 'Mask_Segmentation_TWC.nc',zlib=True,complevel=4)
Features_TWC.to_hdf(savedir / 'Features_TWC.h5','table')
print('segmentation TWC performed and saved')
```

```
Start segmentation based on total water content
```

```
segmentation TWC performed, start saving results to files
```

```
segmentation TWC performed and saved
```

Trajectory linking:
Detected features are linked into cloud trajectories using the trackpy library (http://soft-matter.github.io/trackpy). This takes the feature positions determined in the feature detection step into account but does not include information on the shape of the identified objects.

[15]:
```
# Keyword arguments for linking step:
parameters_linking={}
parameters_linking['method_linking']='predict'
parameters_linking['adaptive_stop']=0.2
parameters_linking['adaptive_step']=0.95
parameters_linking['extrapolate']=0
parameters_linking['order']=1
parameters_linking['subnetwork_size']=100
```

(continues on next page)

(continued from previous page)

```
parameters_linking['memory']=0
parameters_linking['time_cell_min']=5*60
parameters_linking['method_linking']='predict'
parameters_linking['v_max']=10
```

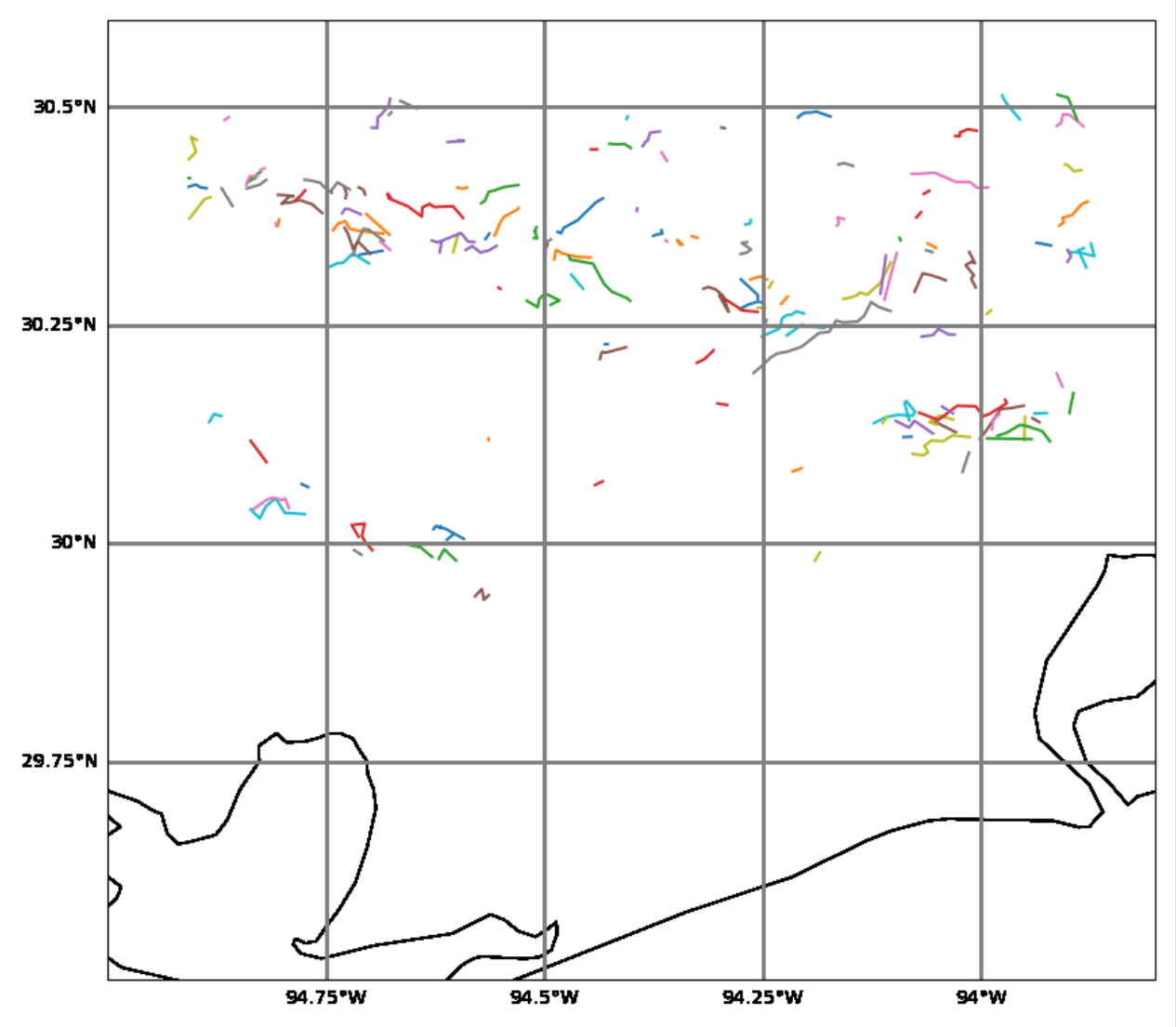
[16]:
```
# Perform linking and save results:
Track=tobac.linking_trackpy(Features,W_mid_max,dt=dt,dxy=dxy,**parameters_linking)
Track.to_hdf(savedir / 'Track.h5', 'table')
```

```
Frame 46: 18 trajectories present.
```

Visualisation:

[17]:
```
# Set extent for maps plotted in the following cells ( in the form [lon_min,lon_max,lat_
↪min,lat_max])
axis_extent=[-95,-93.8,29.5,30.6]
```

[18]:
```
# Plot map with all individual tracks:
import cartopy.crs as ccrs
fig_map,ax_map=plt.subplots(figsize=(10,10),subplot_kw={'projection': ccrs.PlateCarree()}
↪)
ax_map=tobac.map_tracks(Track,axis_extent=axis_extent,axes=ax_map)
```



[19]:
```
# Create animation showing tracked cells with outline of cloud volumes and the midlevel␣
↪vertical velocity as a background field:
animation_tobac=tobac.animation_mask_field(track=Track,features=Features,field=W_mid_max,
↪mask=Mask_TWC,
                                           axis_extent=axis_extent,#figsize=figsize,
↪orientation_colorbar='horizontal',pad_colorbar=0.2,
                                           vmin=0,vmax=20,extend='both',cmap='Blues',
                                           interval=500,figsize=(10,7),
                                           plot_outline=True,plot_marker=True,marker_
↪track='x',plot_number=True,plot_features=True)
```

[20]:
```
# Display animation:
from IPython.display import HTML, Image, display
HTML(animation_tobac.to_html5_video())
```

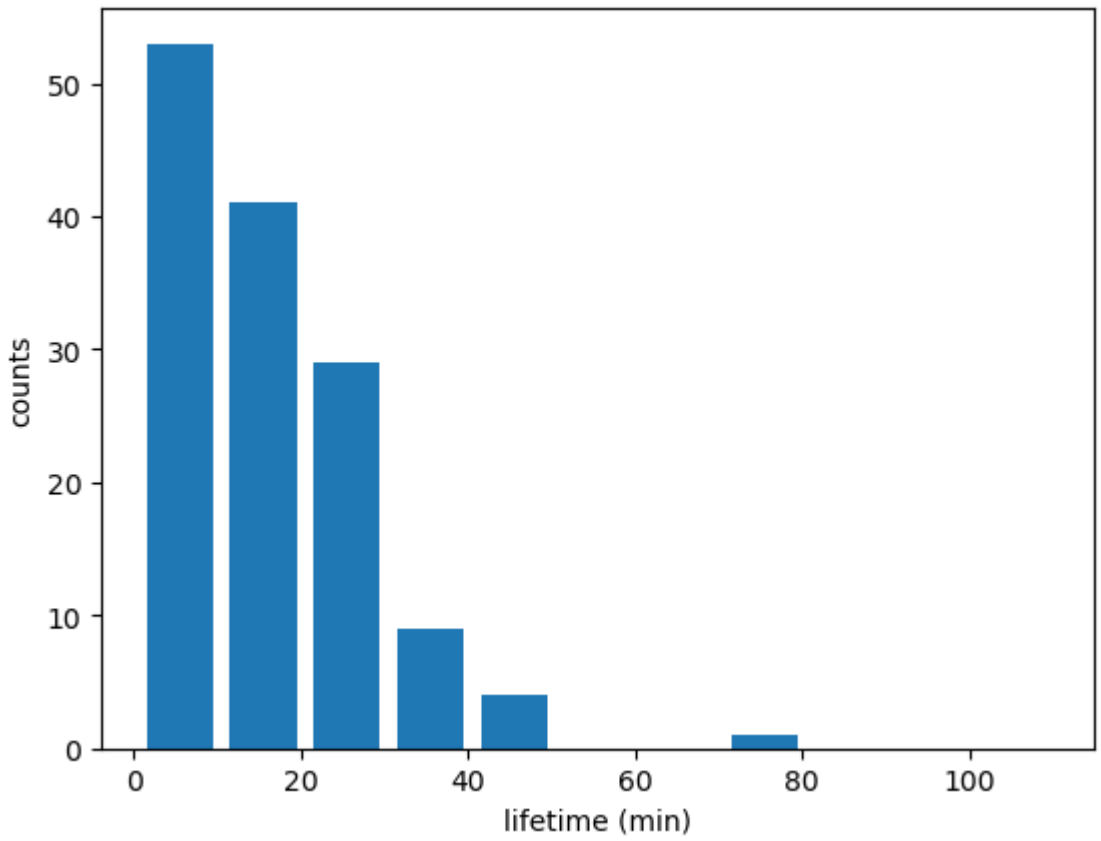
[20]:
```
<IPython.core.display.HTML object>
```

```
<Figure size 640x480 with 0 Axes>
```

[21]:
```
# # Save animation to file
# savefile_animation=plot_dir / 'Animation.mp4')
# animation_tobac.save(savefile_animation,dpi=200)
# print(f'animation saved to {savefile_animation}')
```

[22]:
```
# Updraft lifetimes of tracked cells:
fig_lifetime,ax_lifetime=plt.subplots()
tobac.plot_lifetime_histogram_bar(Track,axes=ax_lifetime,bin_edges=np.arange(0,120,10),
↪density=False,width_bar=8)
ax_lifetime.set_xlabel('lifetime (min)')
ax_lifetime.set_ylabel('counts')
```

[22]:
```
Text(0, 0.5, 'counts')
```



[ ]:

6.12 Cyclone tracking based on relative vorticity in kilometer-scale simulations

This example notebook demonstrates the use of tobac to track meso-scale vortices, based on the relative vorticity field in kilometer-scale simulations. Since such simulations are characterized by high frequencies in the vorticity field (especially in regions with complex terrain), *tobac* allows you to spectrally filter the input data by applying a bandpass filter based on user-specified wavelengths. This results in the removal of submeso-scale noise. For more details about the used filter method and the **discrete cosine transformation** that is used to transfer input data to the spectral space is given in Denis et al. 2002.

Data description

The data used in this example is relative vorticity from a 4km WRF simulation in the Tibetan Plateau-Himalaya region. This data is part of the CORDEX Flagship Pilot Study CPTP ("Convection-Permitting Third Pole"). The target weather system, which we want to track here are shallow meso-scale vortices at 500 hPa. The WRF simulation that produced the input data used Thompson microphysics, the Yonsei University (YSU) planetary boundary layer scheme, the RRTMG longwave and shortwave radiation scheme, and the Noah-MP land surface scheme.

Other applications for the spectral filtering tool in *tobac* could be to detect: - MJO - equatorial waves - atmospheric rivers - ...

You can access the data from the zenodo link, which is provided in the notebook.

[1]:
```
# Import a range of python libraries used in this notebook:
import xarray as xr
import numpy as np
import pandas as pd
from pathlib import Path
import urllib
```

[2]:
```
%matplotlib inline

import matplotlib.pyplot as plt
```

[3]:
```
# Disable a few warnings:
import warnings

warnings.filterwarnings("ignore", category=UserWarning, append=True)
warnings.filterwarnings("ignore", category=RuntimeWarning, append=True)
warnings.filterwarnings("ignore", category=FutureWarning, append=True)
warnings.filterwarnings("ignore", category=pd.io.pytables.PerformanceWarning)
```

[4]:
```
# Import tobac itself:
import tobac

print("using tobac version", str(tobac.__version__))
```

```
using tobac version 1.5.3
```

Download example data:

[5]:
```
# change this location if you want to save the downloaded data elsewhere
data_out = Path("../data")
if data_out.exists() is False:
    data_out.mkdir()
```

[6]:
```
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob("Example_input_vorticity_model.nc"))
if len(data_file) == 0:
    file_path = (
        "https://zenodo.org/record/6459542/files/Example_input_vorticity_model.nc"
    )
    print("start downloading data")
    request = urllib.request.urlretrieve(
        file_path, data_out / ("Example_input_vorticity_model.nc")
    )
    data_file = list(data_out.rglob("Example_input_vorticity_model.nc"))
```

```
start downloading data
```

[7]:
```
# Load Data from downloaded file:
data_file = Path(data_out / "Example_input_vorticity_model.nc")
ds = xr.open_dataset(data_file)
# get variables
relative_vorticity = ds.rv500
lats = ds.latitude
lons = ds.longitude

# check out what the data looks like
ds
```

[7]:
```
<xarray.Dataset> Size: 549MB
Dimensions:      (time: 168, south_north: 469, west_east: 866)
Coordinates:
  * time         (time) datetime64[ns] 1kB 2008-07-14 ... 2008-07-20T23:00:00
  * south_north  (south_north) int64 4kB 33 34 35 36 37 ... 497 498 499 500 501
  * west_east    (west_east) int64 7kB 741 742 743 744 ... 1603 1604 1605 1606
    latitude     (south_north, west_east) float32 2MB ...
    longitude    (south_north, west_east) float32 2MB ...
Data variables:
    rv500        (time, south_north, west_east) float64 546MB ...
Attributes:
    simulation:  WRF 4km
    forcing:     ECMWF-ERA5
    institute:   National Center for Atmospheric Research, Boulder
    contact:     julia.kukulies@gu.se
```

[8]:
```
# Set up directory to save output and plots:
savedir = Path("Save")
plotdir = Path("Plots")

savedir.mkdir(parents=True, exist_ok=True)
plotdir.mkdir(parents=True, exist_ok=True)
```

6.12.1 Check how the spectrally input field would look like

If you want to check how the filter and your filtered data looked like, you can do that by using the method *tobac.utils.general.spectral_filtering()* directly on your input data. This can be helpful in the development process, if you want to try out different ranges of wavelengths and see how this changes your data. In the example, we use use the spectral filtering method to remove wavelengths < 400 km and > 1000 km, because the focus is on meso-scale vortices.

Create your filter

[9]:
```
help(tobac.utils.general.spectral_filtering)
```

```
Help on function spectral_filtering in module 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).

    Parameters:
    -----------
    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.

    Returns:
    --------
    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.
```

[10]:
```
# define minimum and maximum wavelength
lambda_min, lambda_max = 400e3, 1000e3
dxy = 4e3
```

(continues on next page)

(continued from previous page)

```
# use spectral filtering method on input data to check how the applied filter changes␣
↪the data
transfer_function, relative_vorticity_meso = tobac.utils.general.spectral_filtering(
    dxy, relative_vorticity, lambda_min, lambda_max, return_transfer_function=True
)
```

Example for unfiltered vs. filtered input data

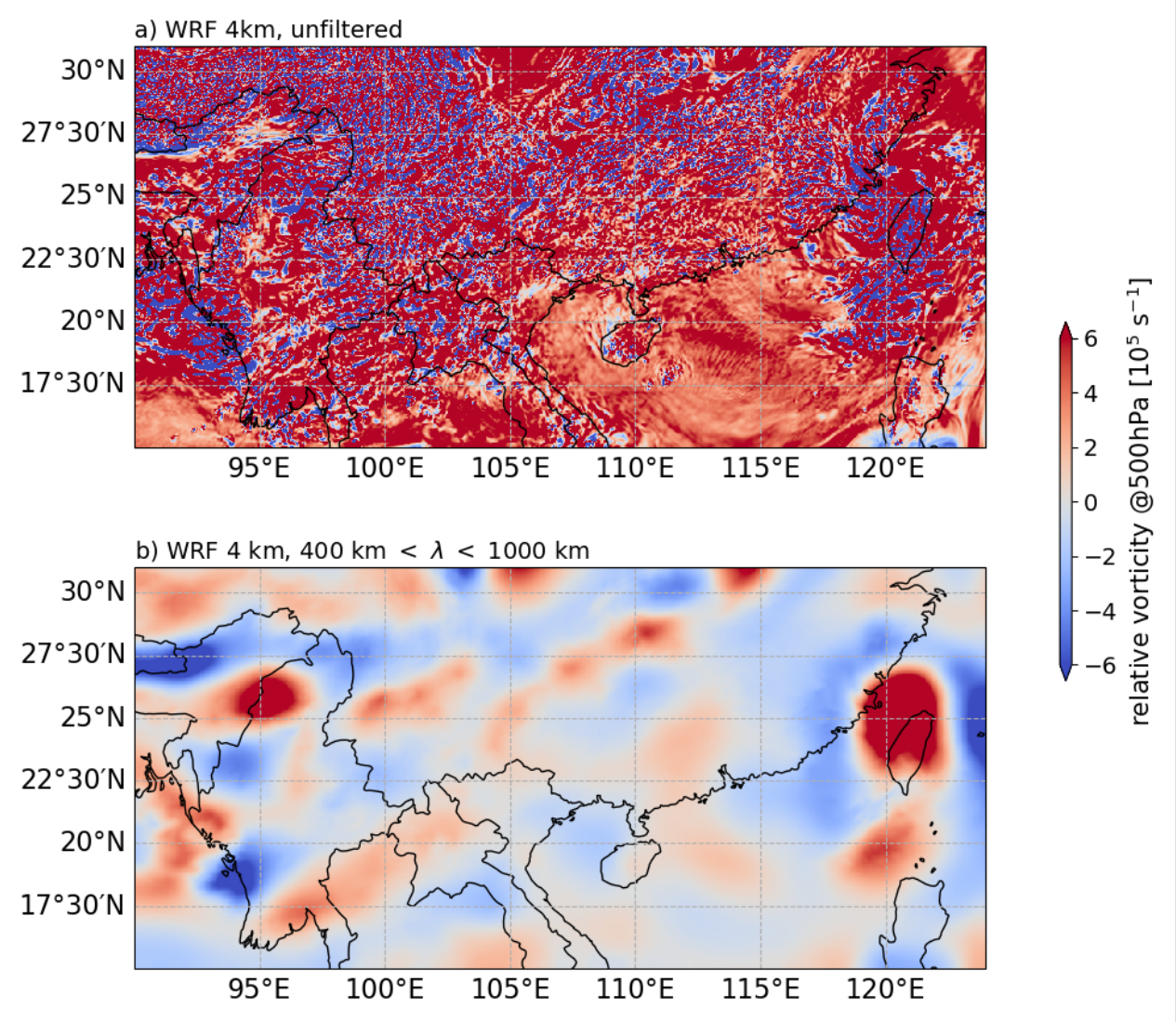
The example shows typhoon *Kalmaegi* over Taiwan on July 18^{th}, 2008 ; as can be seen the corresponding meso-scale vortex becomes only visible in the spectrally filtered field.

[11]:
```
import cartopy.crs as ccrs
import cartopy.feature as cfeature

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

fs = 14
axes = dict()
subplots = 2
ax1 = plt.subplot2grid(
    shape=(subplots, 1), loc=(0, 0), rowspan=1, projection=ccrs.PlateCarree()
)
ax2 = plt.subplot2grid(
    shape=(subplots, 1), loc=(1, 0), rowspan=1, projection=ccrs.PlateCarree()
)

plt.subplots_adjust( hspace = 0.3, right = 0.9)
########################################## vortex ##############################

cmap = "coolwarm"
# time step to plot
tt = 101
col = "black"
extent = [90, 124, 15, 30]

ax1.set_extent(extent)
ax1.set_title("a) WRF 4km, unfiltered", fontsize=fs, loc="left")
ax1.pcolormesh(lons, lats, relative_vorticity[tt], cmap=cmap, vmin=-6, vmax=6)
ax1.add_feature(cfeature.BORDERS, color="black")
ax1.add_feature(cfeature.COASTLINE, color=col)
gl = ax1.gridlines(
    draw_labels=True, dms=True, x_inline=False, y_inline=False, linestyle="--"
)
gl.top_labels = False
gl.right_labels = False
gl.xlabel_style = {"size": 16, "color": "black"}
gl.ylabel_style = {"size": 16, "color": "black"}

ax2.set_extent(extent)
ax2.set_title("b) WRF 4 km, 400 km $<$ $\lambda$ $<$ 1000 km", fontsize=fs, loc="left")
vort = ax2.pcolormesh(
```

(continues on next page)

(continued from previous page)

```
    lons, lats, relative_vorticity_meso[tt], cmap=cmap, vmin=-6, vmax=6
)
ax2.add_feature(cfeature.COASTLINE, color=col)
ax2.add_feature(cfeature.BORDERS, color="black")
gl = ax2.gridlines(
    draw_labels=True, dms=True, x_inline=False, y_inline=False, linestyle="--"
)
gl.top_labels = False
gl.right_labels = False
gl.xlabel_style = {"size": 16, "color": "black"}
gl.ylabel_style = {"size": 16, "color": "black"}

### colorbar ###
cb_ax2 = fig.add_axes([0.93, 0.35, 0.01, 0.3])
cbar = fig.colorbar(vort, cax=cb_ax2, extend="both")
cbar.ax.tick_params(labelsize=fs)
cbar.set_label(r"relative vorticity @500hPa [10$^5$ s$^{-1}$]", size=15)
plt.rcParams.update({"font.size": fs})
plt.show()
```

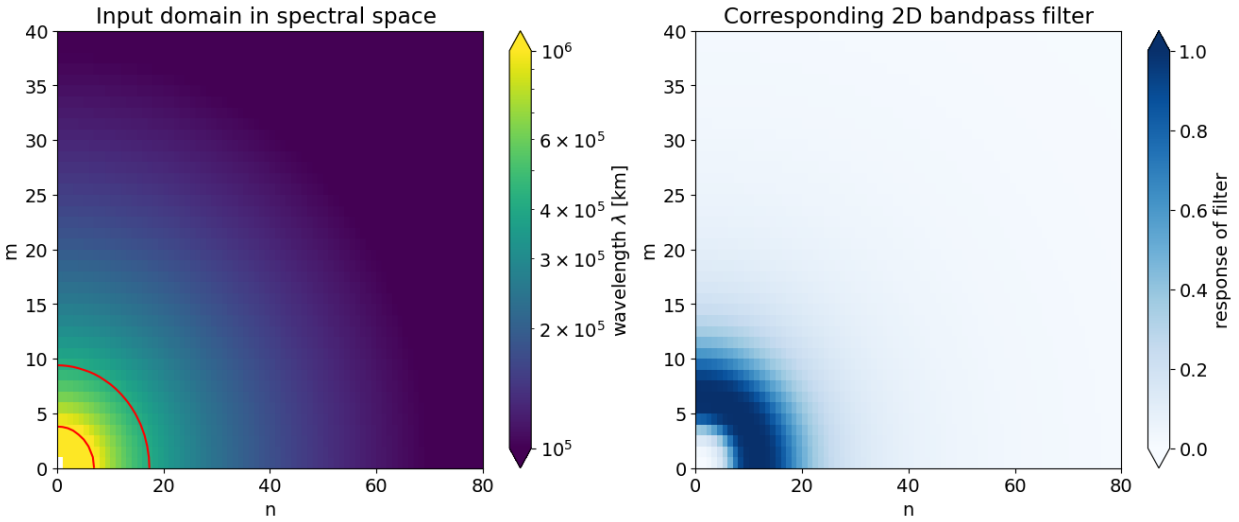


[12]:
```
# checkout how the 2D filter looks like
import matplotlib.colors as colors

plt.figure(figsize=(16, 6))

ax = plt.subplot(1, 2, 1)
# 2D field in spectral space
k = ax.pcolormesh(transfer_function[0], norm=colors.LogNorm(1e5, 1e6), shading="auto")
ax.contour(transfer_function[0], lt="-", levels=[lambda_min, lambda_max], colors="r")
plt.colorbar(k, label="wavelength $\lambda$ [km]", extend="both")
ax.set_ylabel("m")
ax.set_xlabel("n")
# zoom in to see relevant wavelengths
ax.set_xlim([0, 80])
ax.set_ylim([0, 40])
ax.set_title("Input domain in spectral space ")
```

(continues on next page)

(continued from previous page)

```
ax = plt.subplot(1, 2, 2)
# 2D filter
tf = ax.pcolormesh(
    transfer_function[1] / np.nanmax(transfer_function[1]), vmin=0, vmax=1, cmap="Blues"
)
plt.colorbar(tf, label="response of filter", extend="both")
ax.set_title("Corresponding 2D bandpass filter")
ax.set_ylabel("m")
ax.set_xlabel("n")
ax.set_xlim([0, 80])
ax.set_ylim([0, 40])

plt.show()
```



The response of the filter is 1 at locations, where wavelengths are within acceptable range and 0, when the wavelengths are outside of this range (here for: 400 km < λ < 1000 km). The transition is smoothed. To better understand this transition, one could also look at the same filter in one dimension (with the red lines indicating the wavelength truncations):

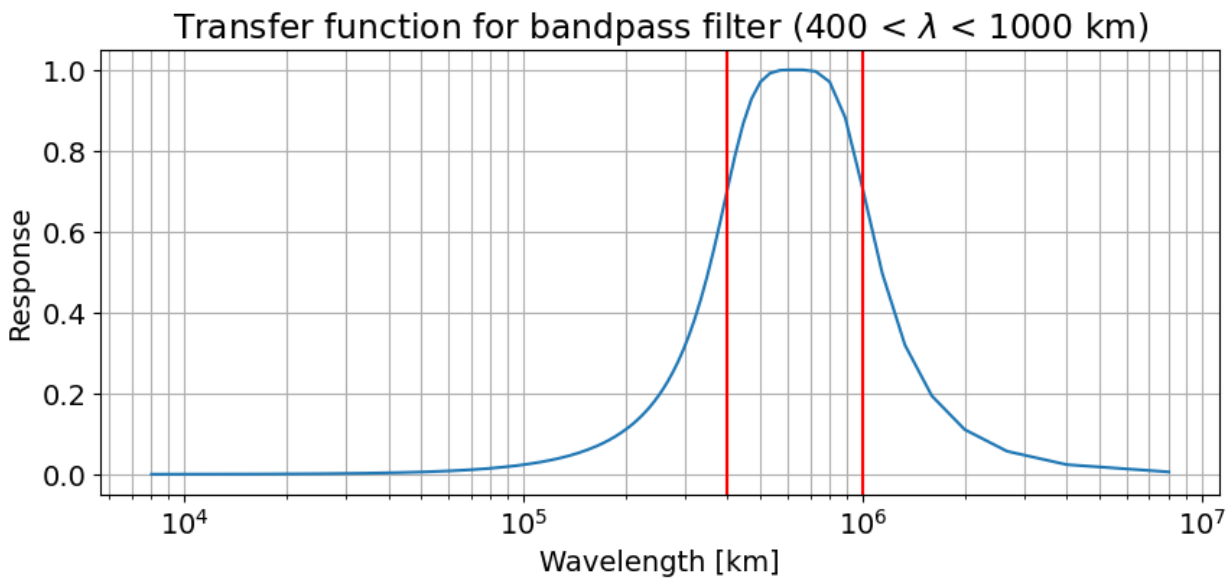
The same filter in 1D:

[13]:
```
from scipy import signal

# calculate filter according to lambda_min and lambda_max
b, a = signal.iirfilter(
    2,
    [1 / lambda_max, 1 / lambda_min],
    btype="band",
    ftype="butter",
    fs=1 / dxy,
    output="ba",
)
w, h = signal.freqz(b, a, 1000, fs=1 / dxy)
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(10, 4))
ax = fig.add_subplot(1, 1, 1)
plt.semilogx(1 / w, abs(h))
# plt.plot(w, h)
ax.set_title("Transfer function for bandpass filter (400 < $\lambda$ < 1000 km)")
ax.set_xlabel("Wavelength [km]")
ax.set_ylabel("Response")
ax.axvline(lambda_min, c="r")
ax.axvline(lambda_max, c="r")
ax.grid(which="both", axis="both")
plt.show()
```



If you are happy with how the filter changes your input field, continue as usual with the feature detection and segmentation:

6.12.2 Feature detection:

Feature detection is then performed based on the filtered relative vorticity field and your chosen set of thresholds.

[14]:
```
# you can use this function to get the grid spacings from lats and lons
dxy, dt = tobac.utils.general.get_spacings(relative_vorticity, grid_spacing=4e3)
# but better to define exact grid spacing if known, since get_spacings() uses haversine␣
↪approximation
```

[15]:
```
# if you want your WRF data to indcate the track locations in lons and lats:
relative_vorticity
```

[15]:
```
<xarray.DataArray 'rv500' (time: 168, south_north: 469, west_east: 866)> Size: 546MB
array([[[  3.138932,   3.150467, ...,   5.20407 ,   5.191312],
        [  3.202842,   3.218159, ...,   5.133   ,   5.111225],
        ...,
        [ 11.466752,  11.446242, ...,   2.244073,   2.288456],
```

(continues on next page)

(continued from previous page)

```
        [  6.06062 ,   6.087327, ...,   2.238939,   2.280738]],

       [[  3.063716,   3.038443, ...,   4.815409,   5.123763],
        [  3.141597,   3.124234, ...,   4.726799,   5.030745],
        ...,
        [ 12.680849,  12.979313, ...,   2.141634,   2.294254],
        [ -1.421874,  -1.235242, ...,   2.125223,   2.277909]],

       ...,

       [[ 10.169939,   9.744318, ...,   3.209985,   3.176951],
        [ 10.194508,   9.936515, ...,   3.136149,   3.103187],
        ...,
        [  3.718061,  -0.572581, ..., -28.510893,  -8.78719 ],
        [ 14.069323,  15.725659, ..., -28.109968,  -8.83858 ]],

       [[  9.703144,   8.762362, ...,   2.785694,   2.797884],
        [  9.489581,   8.667569, ...,   2.672183,   2.686641],
        ...,
        [  9.156374,   7.913566, ..., -32.878235, -10.757242],
        [ 20.767054,  15.039678, ..., -33.59285 , -11.135064]]])
Coordinates:
  * time         (time) datetime64[ns] 1kB 2008-07-14 ... 2008-07-20T23:00:00
  * south_north  (south_north) int64 4kB 33 34 35 36 37 ... 497 498 499 500 501
  * west_east    (west_east) int64 7kB 741 742 743 744 ... 1603 1604 1605 1606
    latitude     (south_north, west_east) float32 2MB 15.03 15.03 ... 31.98
    longitude    (south_north, west_east) float32 2MB 90.04 90.08 ... 124.4
Attributes:
    long name:      Relative vorticity 500 hPA
    standard name:  relative_vorticity
    units:          10^-5 s-1
```

[16]:
```
# Dictionary containing keyword arguments for feature detection step (Keywords could␣
↪also be given directly in the function call).
parameters_features = {}
parameters_features["position_threshold"] = "weighted_diff"
parameters_features["sigma_threshold"] = 0.5
parameters_features["min_num"] = 5
parameters_features["n_min_threshold"] = 20
parameters_features["target"] = "maximum"
parameters_features["threshold"] = [3, 5, 8, 10]
```

[17]:
```
# Perform feature detection:
print("starting feature detection")
Features = tobac.feature_detection_multithreshold(
    relative_vorticity,
    dxy=4000,
    **parameters_features,
    wavelength_filtering=(400e3, 1000e3)
)
Features.to_xarray().to_netcdf(savedir / "Features.nc")
print("feature detection performed and saved")
```

```
starting feature detection
```

```
feature detection performed and saved
```

6.12.3 Segmentation

Segmentation is performed with watershedding based on the detected features and a single threshold value.

[18]:
```
# Dictionary containing keyword options for the segmentation step:
parameters_segmentation = {}
parameters_segmentation["target"] = "maximum"
parameters_segmentation["method"] = "watershed"
parameters_segmentation["threshold"] = 1.5
```

[19]:
```
# Perform segmentation and save results:
print("Starting segmentation based on relative vorticity.")
Mask_rv, Features_rv = tobac.segmentation_2D(
    Features, relative_vorticity, dxy, **parameters_segmentation
)
print("segmentation performed, start saving results to files")
Mask_rv.to_netcdf(savedir / "Mask_Segmentation_rv.nc")
Features_rv.to_xarray().to_netcdf(savedir / "Features_rv.nc")
print("segmentation performed and saved")
```

```
Starting segmentation based on relative vorticity.
```

```
segmentation performed, start saving results to files
```

```
segmentation performed and saved
```

6.12.4 Trajectory linking

Features are linked into trajectories using the trackpy library (http://soft-matter.github.io/trackpy). This takes the feature positions determined in the feature detection step into account but does not include information on the shape of the identified objects.**

[20]:
```
# Arguments for trajectory linking:
parameters_linking = {}
parameters_linking["v_max"] = 80
parameters_linking["stubs"] = 2
parameters_linking["order"] = 1
parameters_linking["memory"] = 0
parameters_linking["adaptive_stop"] = 0.2
parameters_linking["adaptive_step"] = 0.95
parameters_linking["subnetwork_size"] = 1000
# require that the vortex has to persist during at least 12 hours
parameters_linking["time_cell_min"] = 12 * dt
parameters_linking["method_linking"] = "predict"
```

[21]:
```
# Perform linking and save results to file:
Track = tobac.linking_trackpy(
    Features, relative_vorticity, dt=dt, dxy=dxy, **parameters_linking
```

(continues on next page)

(continued from previous page)

```
)
Track.to_xarray().to_netcdf(savedir / "Track.nc")
```

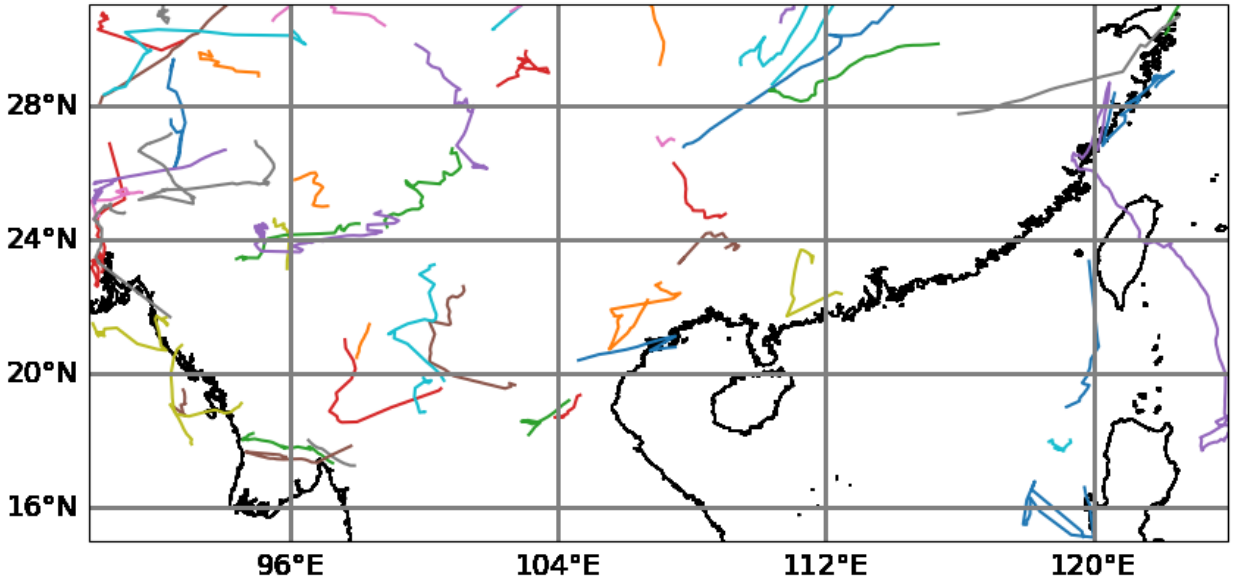
```
Frame 167: 16 trajectories present.
```

6.12.5 Visualisation of tracks

The long track in the east is the mesoscale vortex associated with **typhoon Kalmeagi** that passed Taiwan in July 2008.

The tracks in the west correspond to vortices that frequently form in the higher mountains over the Tibetan Plateau.

[22]:
```
# Plot map with all individual tracks:
import cartopy.crs as ccrs

fig_map, ax_map = plt.subplots(
    figsize=(10, 10), subplot_kw={"projection": ccrs.PlateCarree()}
)
ax_map = tobac.map_tracks(Track, axis_extent=extent, axes=ax_map)
```



The notebooks can be found in the **examples** folder as part of the python package. The necessary input data for these examples is avaliable on zenodo and can be downloaded automatically by the Jupyter notebooks.

CHAPTER

SEVEN

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. *Journal of Geophysical Research: Atmospheres*, 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. *Journal of Geophysical Research: Atmospheres*, 126(23), e2021JD035279. doi:10.1029/2021JD035279.

Kukulies, J., Lai, H. W., Curio, J., Feng, Z., Lin, C., Li, P., Ou, T., Sugimoto, S. & Chen, D. (2023). Mesoscale convective systems in the Third pole region: Characteristics, mechanisms and impact on precipitation. *Frontiers in Earth Science*, 11, 1143380.

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. *Remote Sensing*, 13(21), 4289. doi:10.3390/rs13214289.

Leung, G. R., Saleeby, S. M., Sokolowsky, G. A., Freeman, S. W., & van den Heever, S. C. (2023). Aerosol–cloud impacts on aerosol detrainment and rainout in shallow maritime tropical clouds. *Atmospheric Chemistry and Physics*, 23(9), 5263-5278.

Marinescu, P. J., Van Den Heever, S. C., Heikenfeld, M., Barrett, A. I., Barthlott, C., Hoose, C., Fan, J., Fridlind, A. M., Matsui, T., Miltenberger, A. K., Stier, P., Vie, B., White, B. A., & Zhang, Y. (2021). Impacts of varying concentrations of cloud condensation nuclei on deep convective cloud updrafts—a multimodel assessment. *Journal of the Atmospheric Sciences*, 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).

Oue, M., Saleeby, S. M., Marinescu, P. J., Kollias, P., & van den Heever, S. C. (2022). Optimizing radar scan strategies for tracking isolated deep convection using observing system simulation experiments. *Atmospheric Measurement Techniques*, 15(16), 4931-4950.

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. *Journal of Applied Meteorology and Climatology*, 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. *Remote Sensing*, 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!

CHAPTER

EIGHT

FEATURE DETECTION BASICS

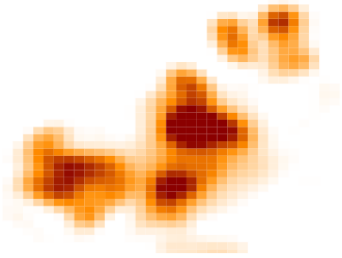
The feature detection is the first step in using **tobac**.

Currently implemented methods:

Multiple thresholds:

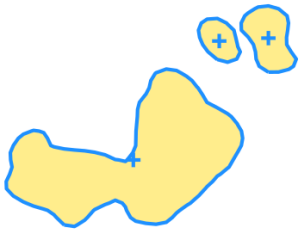
Features are identified as regions above or below a sequence of subsequent thresholds (if searching for eather maxima or minima in the data). Subsequently more restrictive threshold values are used to further refine the resulting features and allow for separation of features that are connected through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

a) Input data

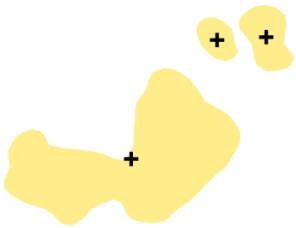


new features
old features
deleted features
final features

b) Threshold 1

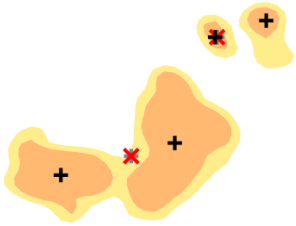


c) Intermediate features 1

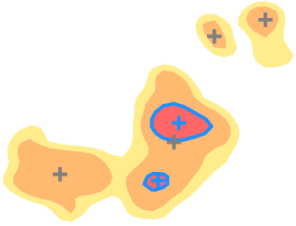


d) Threshold 2

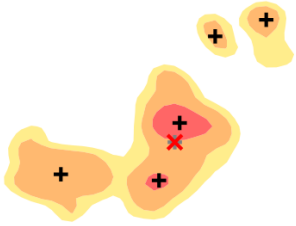
e) Intermediate features 2



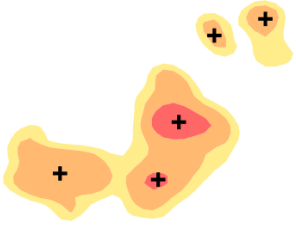
f) Threshold 3



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.

CHAPTER

NINE

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. The feature detection uses the threshold value in an inclusive way, which means that if you set `target='maximum', threshold=[10,]` and your field does not only contain values lower than `10`, it will include all values **greater than and equal to** `10`.

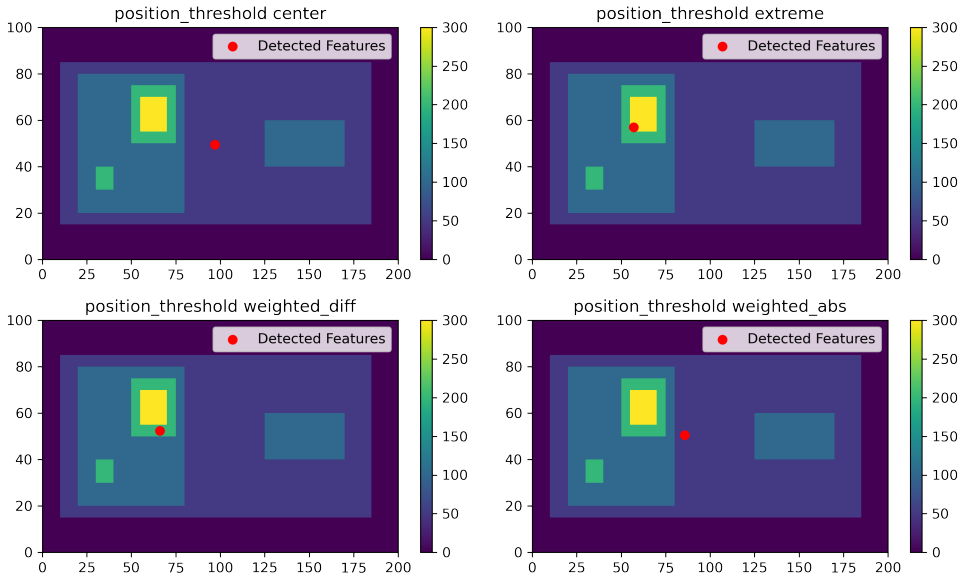
Including *multiple* thresholds will allow *tobac* to refine the detection of features and detect multiple features that are connected through a contiguous region of less restrictive threshold values. You can see a conceptual diagram of that here: *Feature Detection Basics*. To examine how setting different 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*.

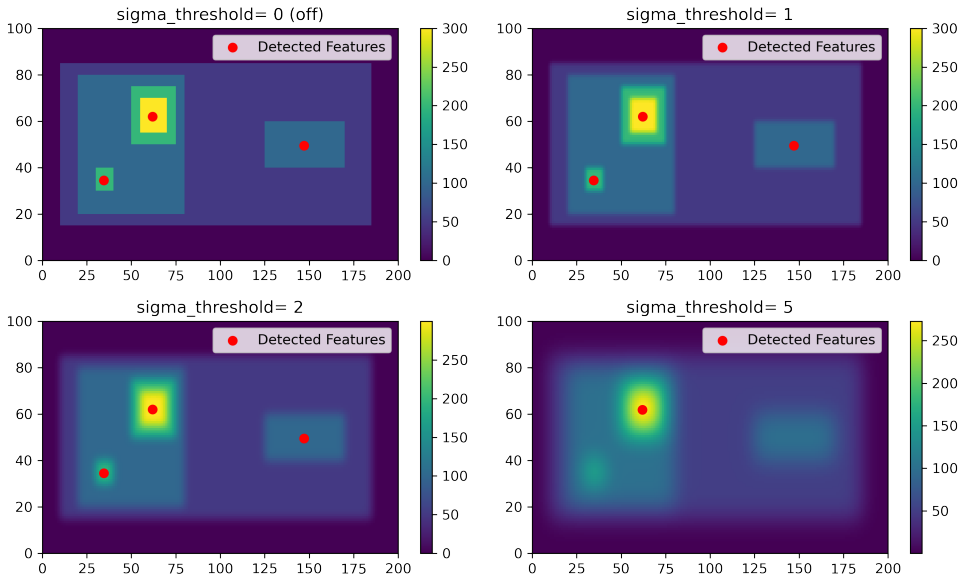
9.5 Feature Position

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` 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 more extreme value is kept, and the feature with the less extreme value is discarded.

CHAPTER

TEN

FEATURE DETECTION PARAMETER EXAMPLES

10.1 How multiple thresholds changes the features detected

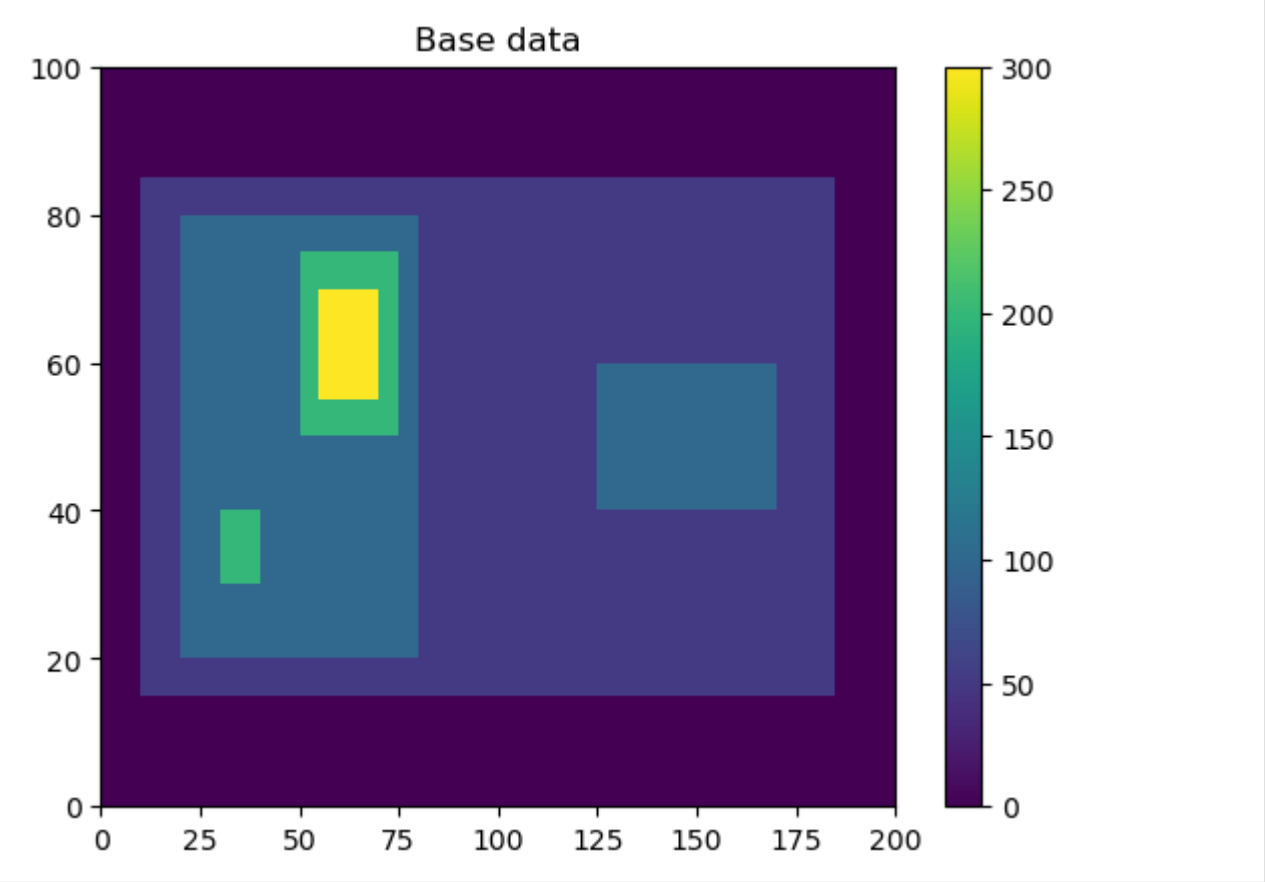
10.1.1 Imports

[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.

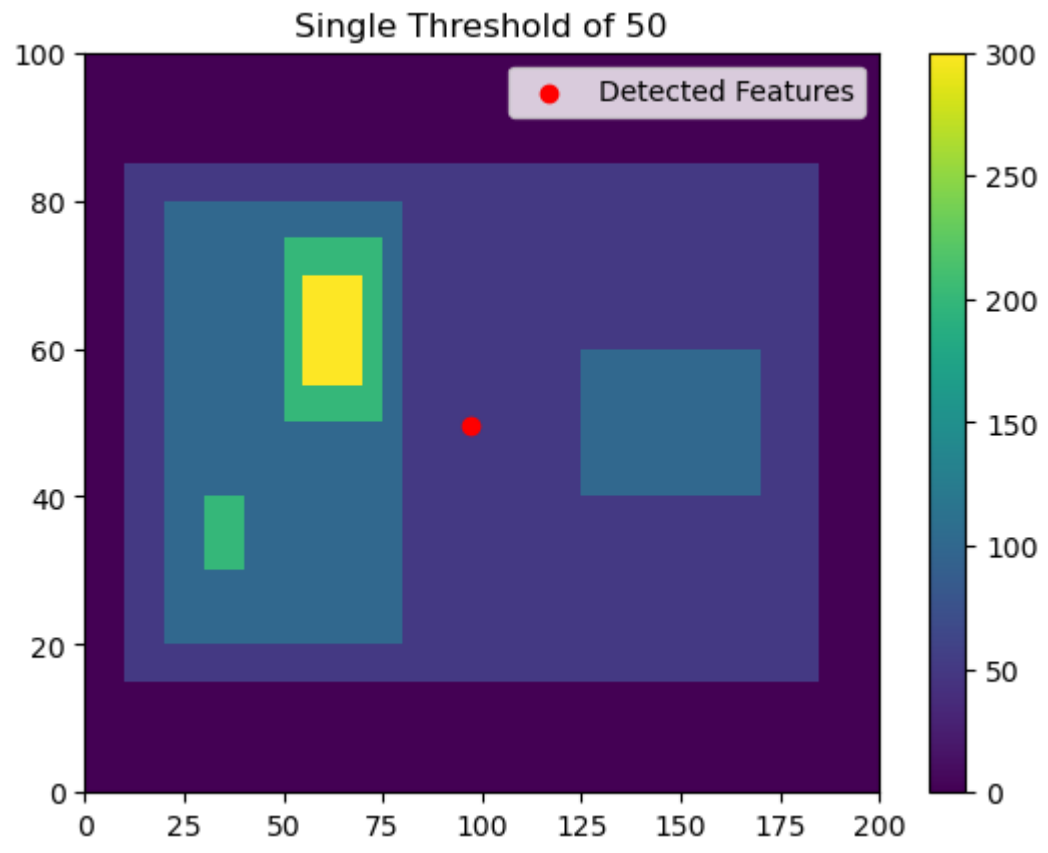
[3]:
```
input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={'time':
↪ [np.datetime64('2019-01-01T00:00:00')]}).to_iris()
```

```
/var/folders/40/kfr98p0j7n30fjp2n4ljjqbh0000gr/T/ipykernel_51508/3426653690.py:1:␣
↪UserWarning: Converting non-nanosecond precision datetime values to nanosecond␣
↪precision. This behavior can eventually be relaxed in xarray, as it is an artifact␣
↪from pandas which is now beginning to support non-nanosecond precision values. This␣
↪warning is caused by passing non-nanosecond np.datetime64 or np.timedelta64 values to␣
↪the DataArray or Variable constructor; it can be silenced by converting the values to␣
↪nanosecond precision ahead of time.
  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")
plt.legend()
plt.title("Single Threshold of 50")
plt.show()
```



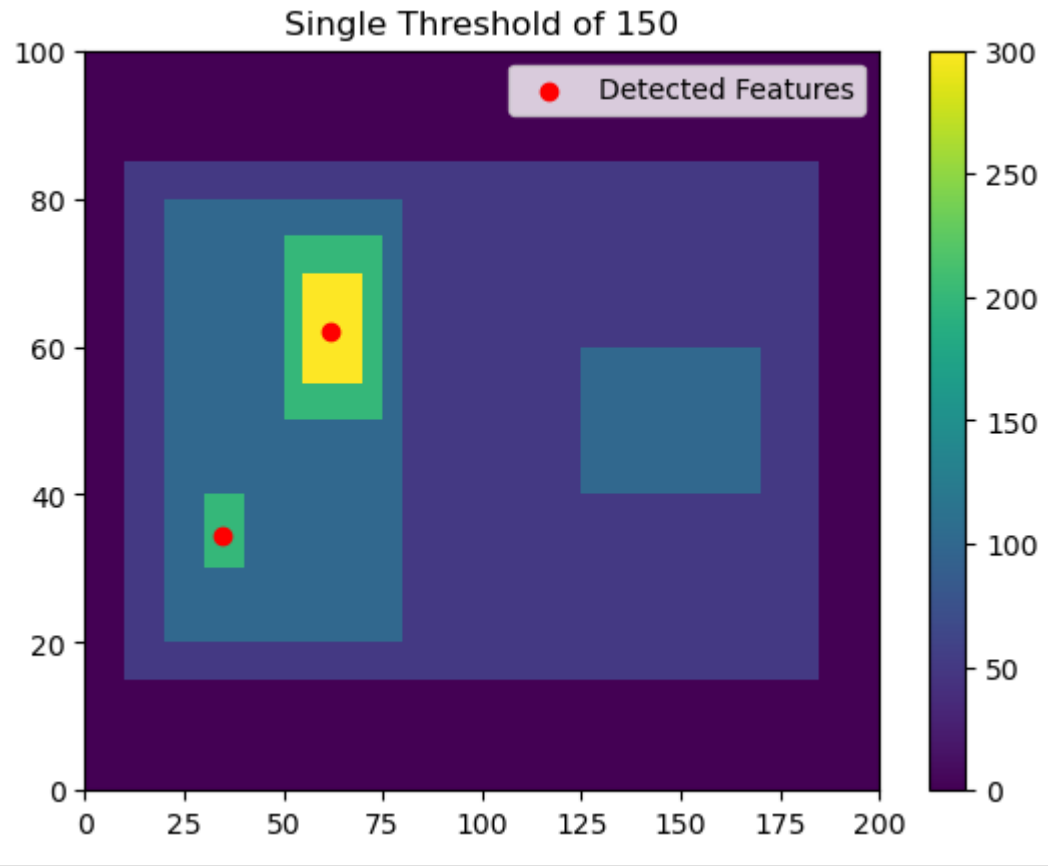
Now, let's try a single threshold of 150, which will give us two features on the left side of the image.

[5]:
```
thresholds = [150,]
# Using 'center' here outputs the feature location as the arithmetic center of the␣
↪detected feature
```

(continues on next page)

(continued from previous page)

```
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()
```

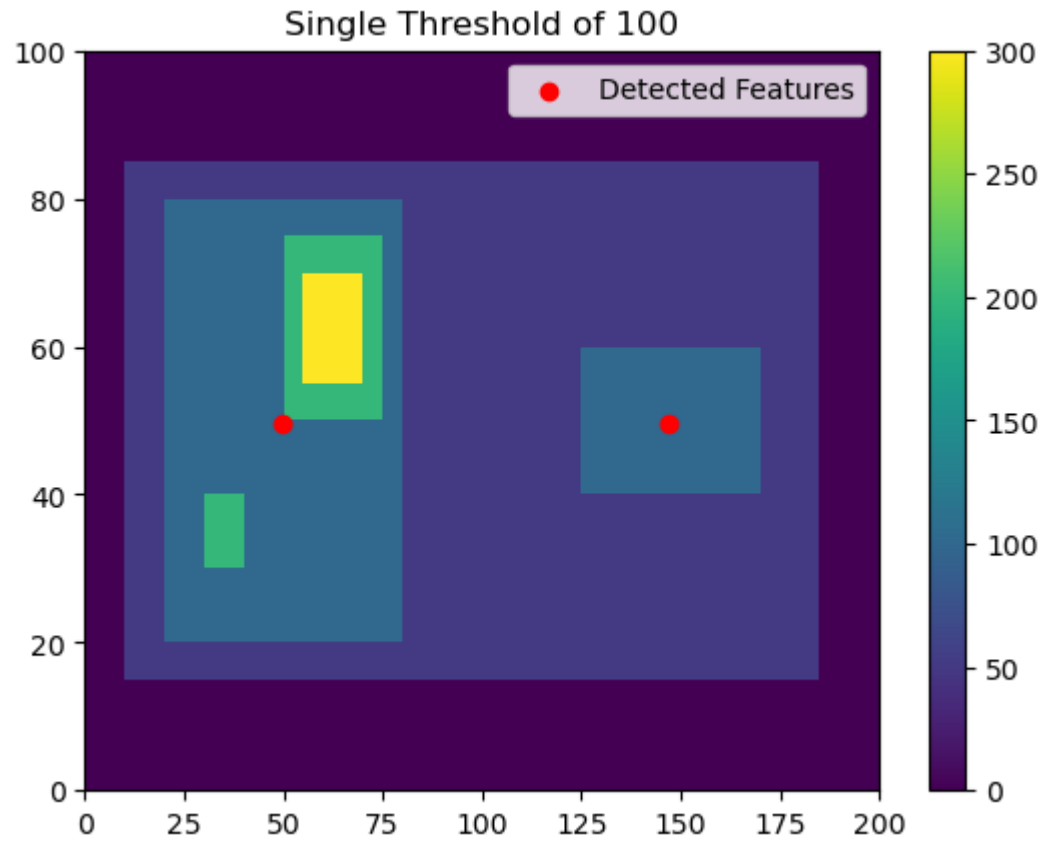


This gives us two detected features with minimum values >150.

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 overhshooting 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()
```



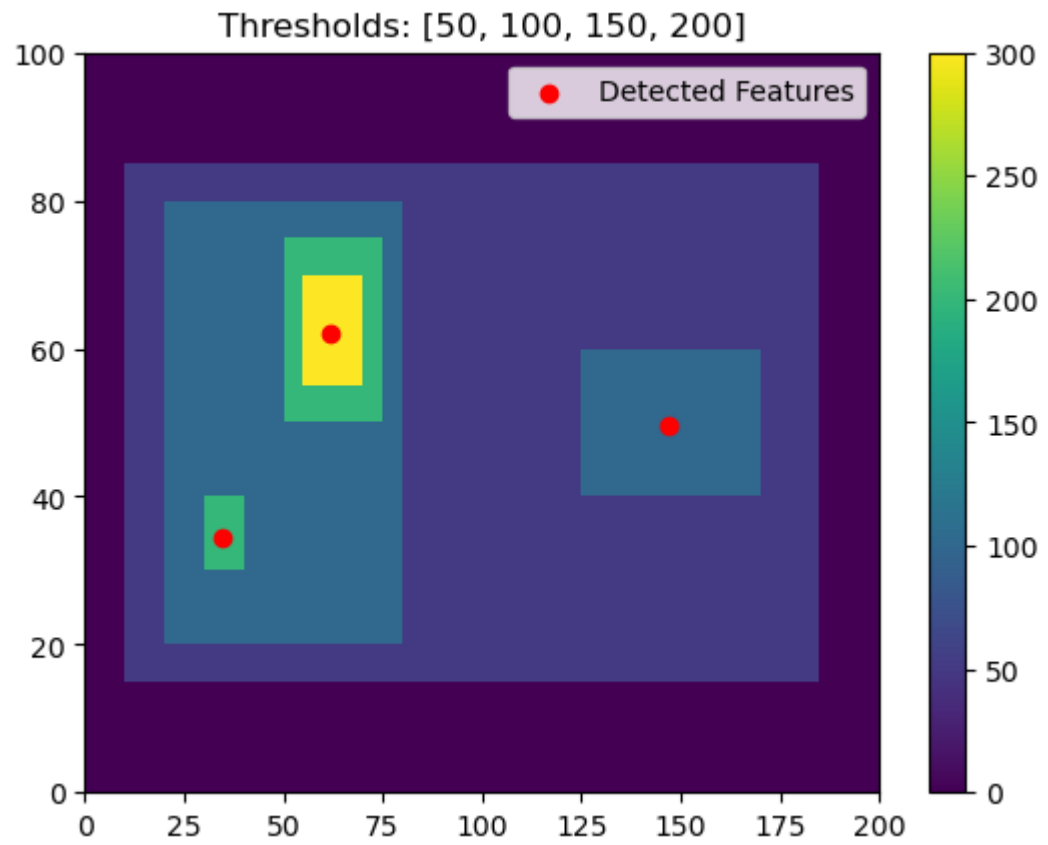
This is the power of having multiple thresholds. We can set thresholds of 50, 100, 150, 200 and capture both:

[7]:
```
thresholds = [50, 100, 150, 200]
# Using 'center' here outputs the feature location as the arithmetic center of the␣
```

(continues on next page)

(continued from previous page)

```
↪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()
```

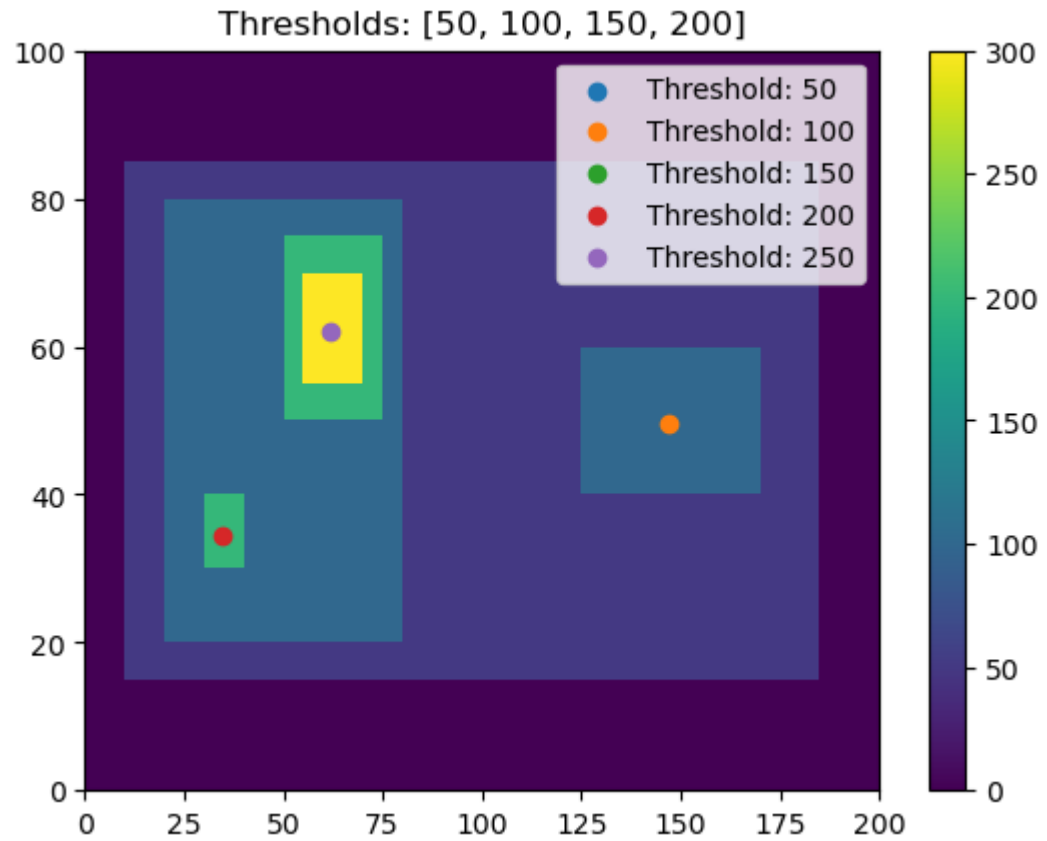


[8]:
```
thresholds = [50, 100, 150, 200, 250]
# Using 'center' here outputs the feature location as the arithmetic center of the␣
↪detected feature
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↪field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_threshold=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

# Plot all features detected
```

(continues on next page)

(continued from previous page)

```
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

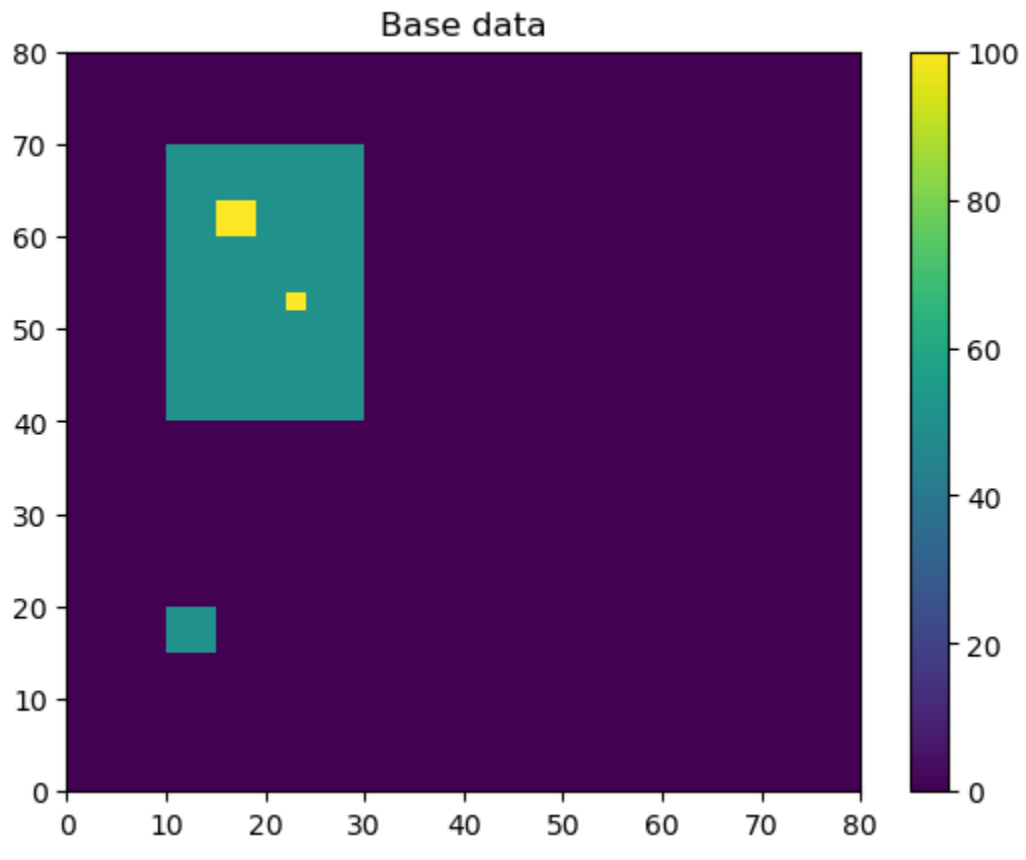
10.2.1 Imports

[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
# 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,
```

(continues on next page)

(continued from previous page)

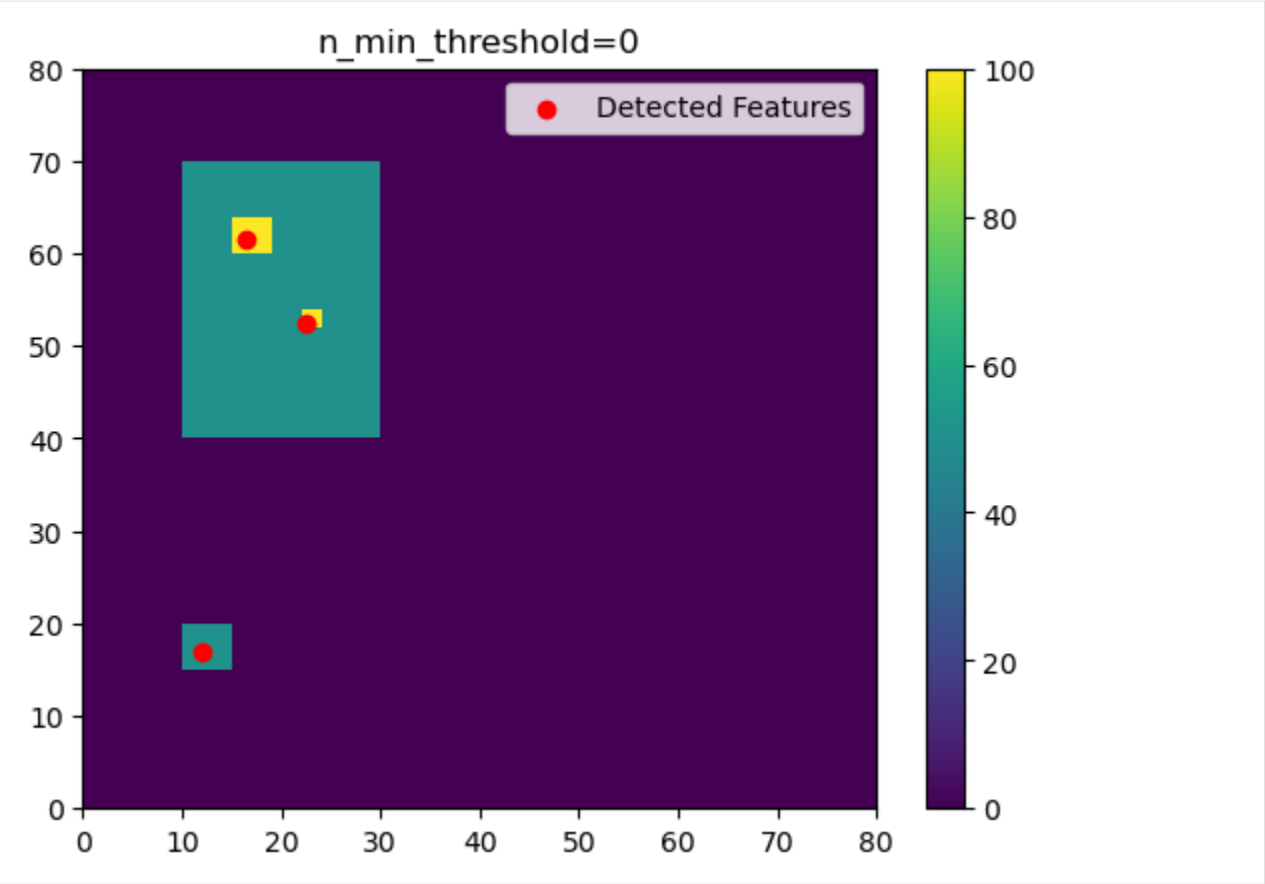
```
    dims=["time", "Y", "X"],
    coords={"time": [np.datetime64("2019-01-01T00:00:00")]},
).to_iris()
# Version 2.0 of tobac (currently in development) will allow the use of xarray directly␣
↪with tobac.
```

```
/var/folders/40/kfr98p0j7n30fjp2n4ljjqbh0000gr/T/ipykernel_51434/1759418141.py:3:␣
↪UserWarning: Converting non-nanosecond precision datetime values to nanosecond␣
↪precision. This behavior can eventually be relaxed in xarray, as it is an artifact␣
↪from pandas which is now beginning to support non-nanosecond precision values. This␣
↪warning is caused by passing non-nanosecond np.datetime64 or np.timedelta64 values to␣
↪the DataArray or Variable constructor; it can be silenced by converting the values to␣
↪nanosecond precision ahead of time.
  input_field_iris = xr.DataArray(
```

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`

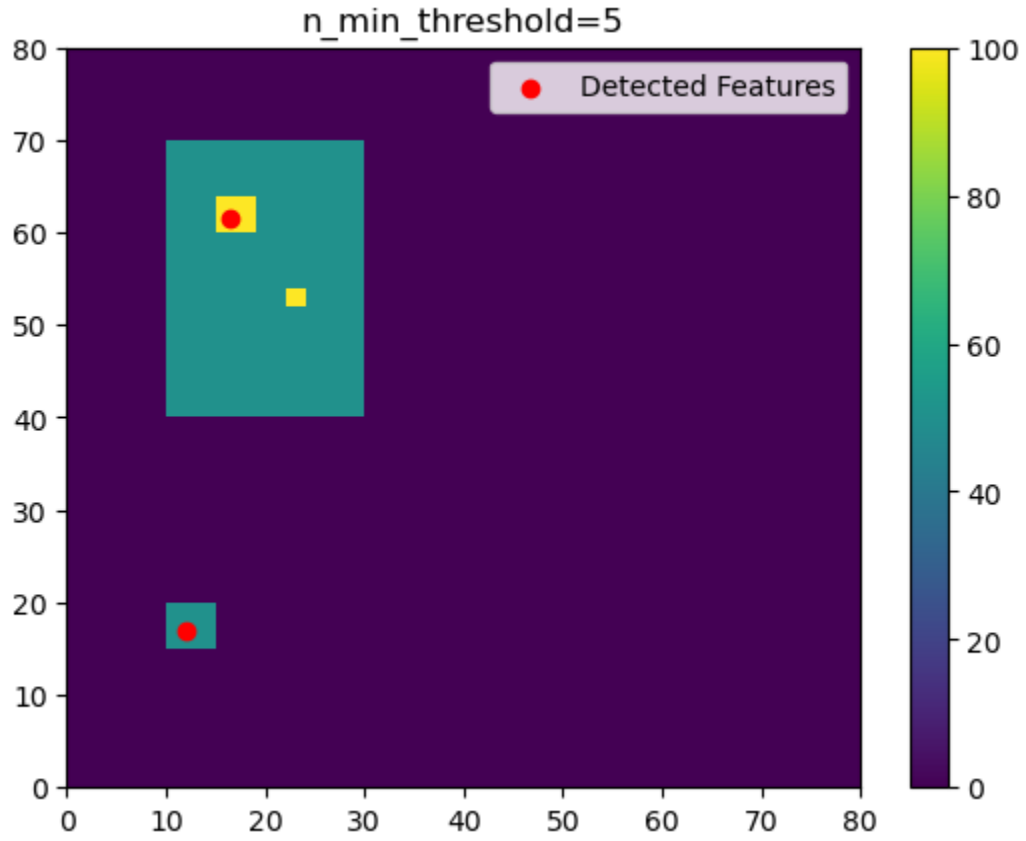
As we increase `n_min_threshold`, fewer of these separate features are detected. In this example, if we set `n_min_threshold` to 5, the smallest detected feature goes away.

[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
```

(continues on next page)

(continued from previous page)

```
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()
```



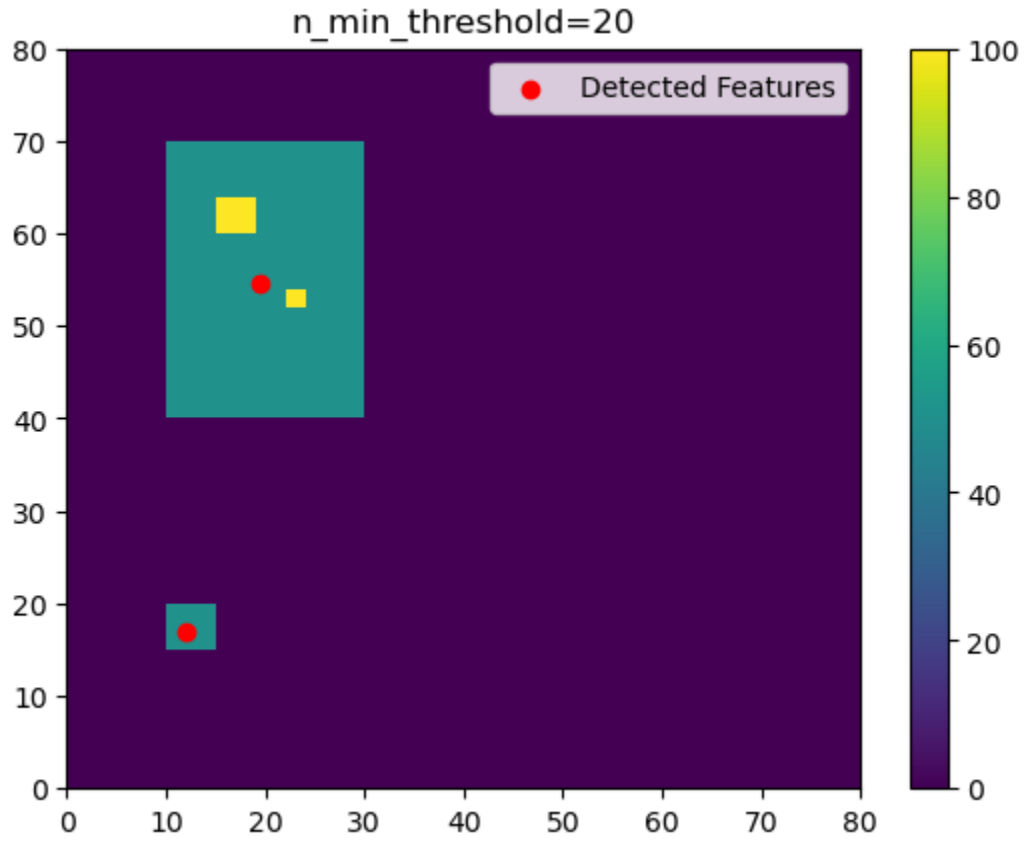
If we increase `n_min_threshold` to 20, only the large 50-valued feature is detected, rather than the two higher-valued squares.

[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",
```

(continues on next page)

(continued from previous page)

```
    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()
```



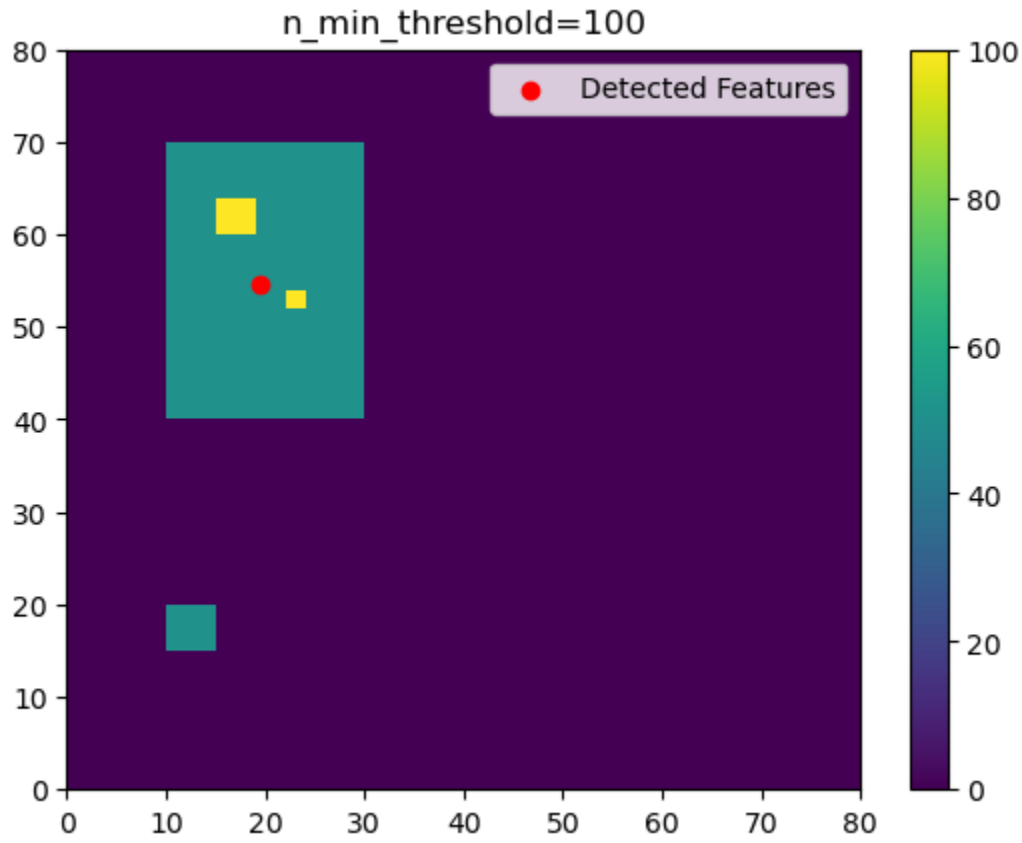
If we set `n_min_threshold` to 100, only the largest feature is detected.

[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(
```

(continues on next page)

(continued from previous page)

```
    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()
```

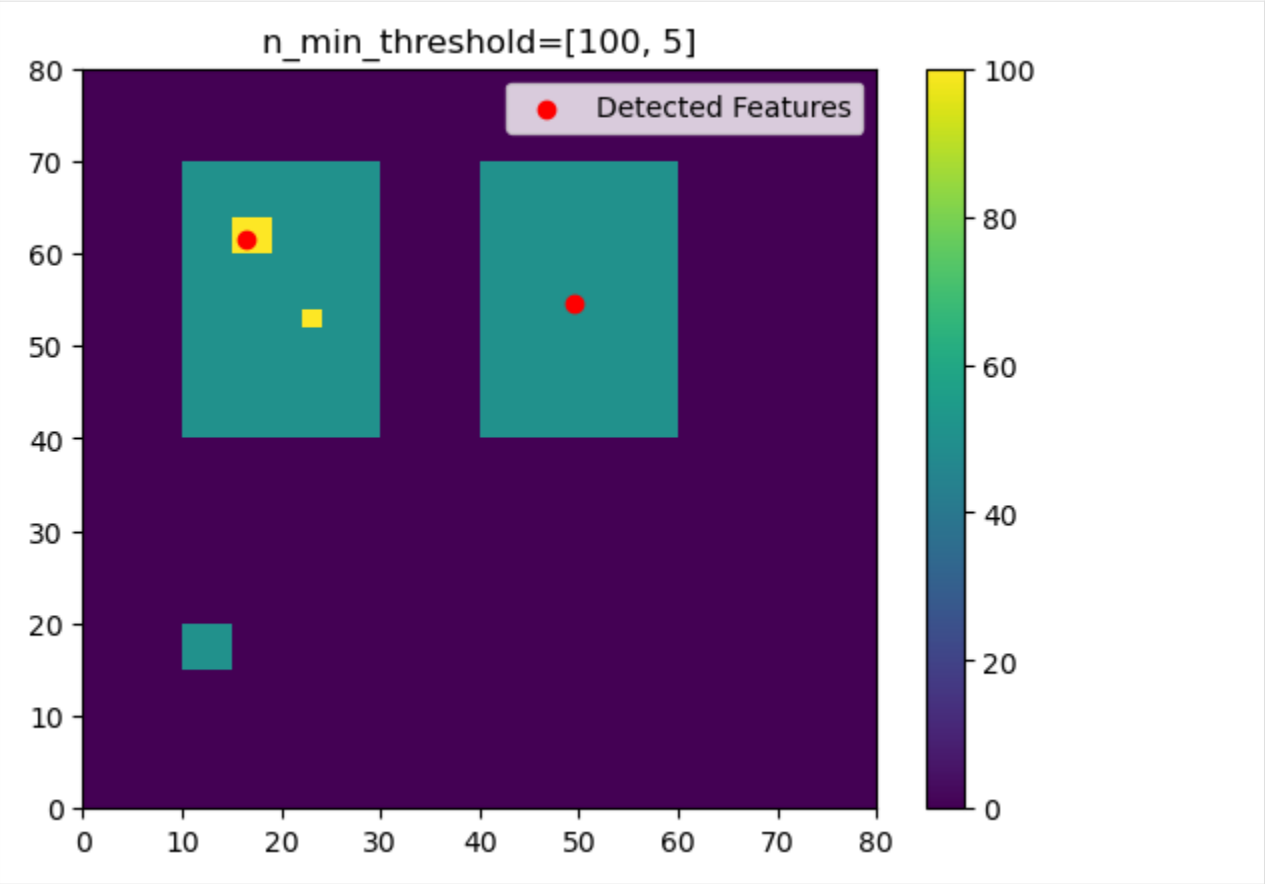


10.2.5 Different n_min_threshold for different threshold values

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

# 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`.

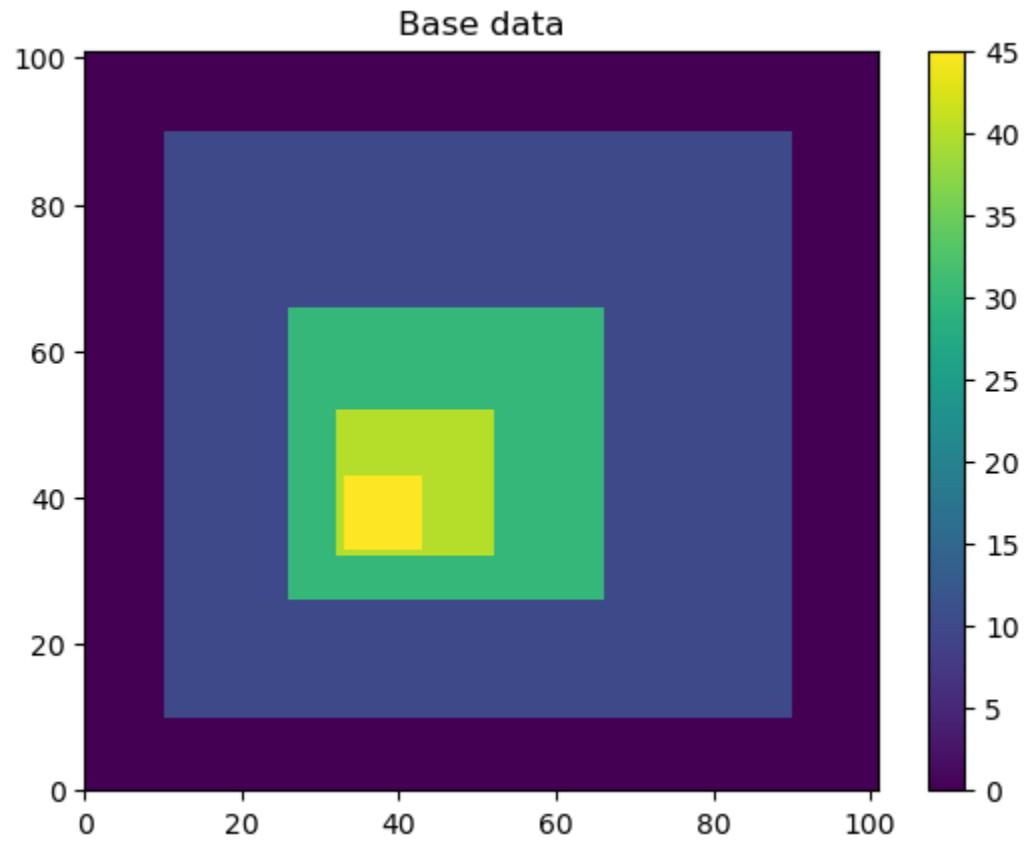
[9]:
```
# As above, we create test data to demonstrate the effect of strict_thresholding
input_field_arr = np.zeros((1, 101, 101))

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()
```

(continues on next page)

(continued from previous page)

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

```
/var/folders/40/kfr98p0j7n30fjp2n4ljjqbh0000gr/T/ipykernel_51434/418987827.py:13:␣
↪UserWarning: Converting non-nanosecond precision datetime values to nanosecond␣
↪precision. This behavior can eventually be relaxed in xarray, as it is an artifact␣
↪from pandas which is now beginning to support non-nanosecond precision values. This␣
↪warning is caused by passing non-nanosecond np.datetime64 or np.timedelta64 values to␣
↪the DataArray or Variable constructor; it can be silenced by converting the values to␣
↪nanosecond precision ahead of time.
  input_field_iris = xr.DataArray(
```

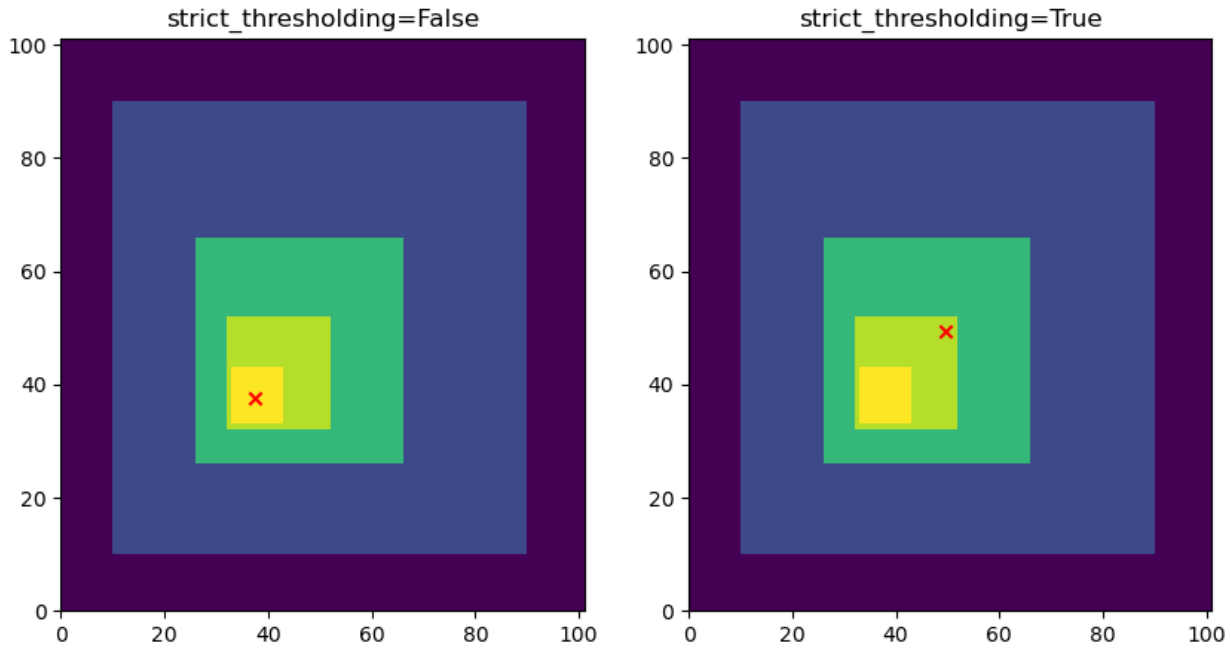


[10]:
```
thresholds = [8, 29, 39, 44]

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)):
```

(continues on next page)

(continued from previous page)

```
    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

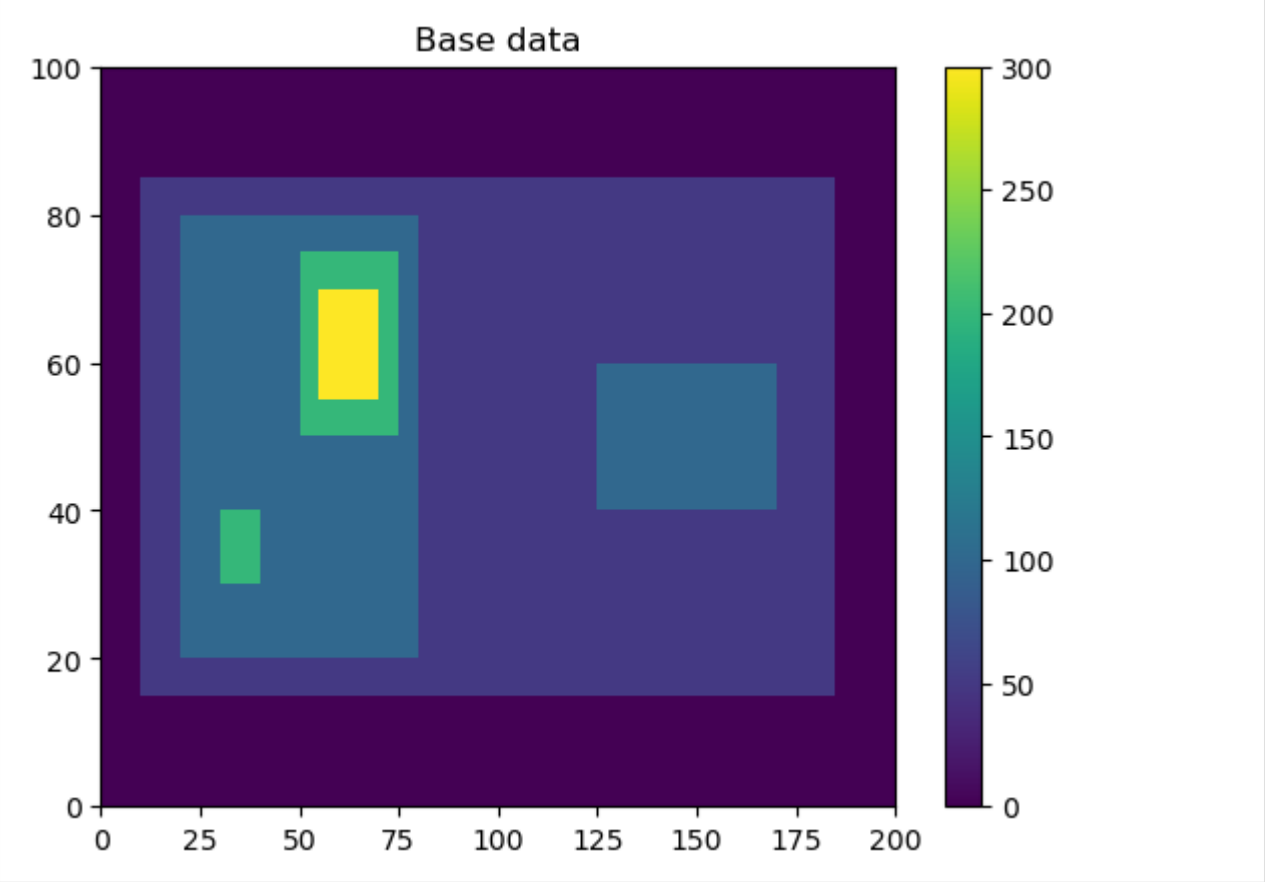
10.3.1 Imports

[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()
```



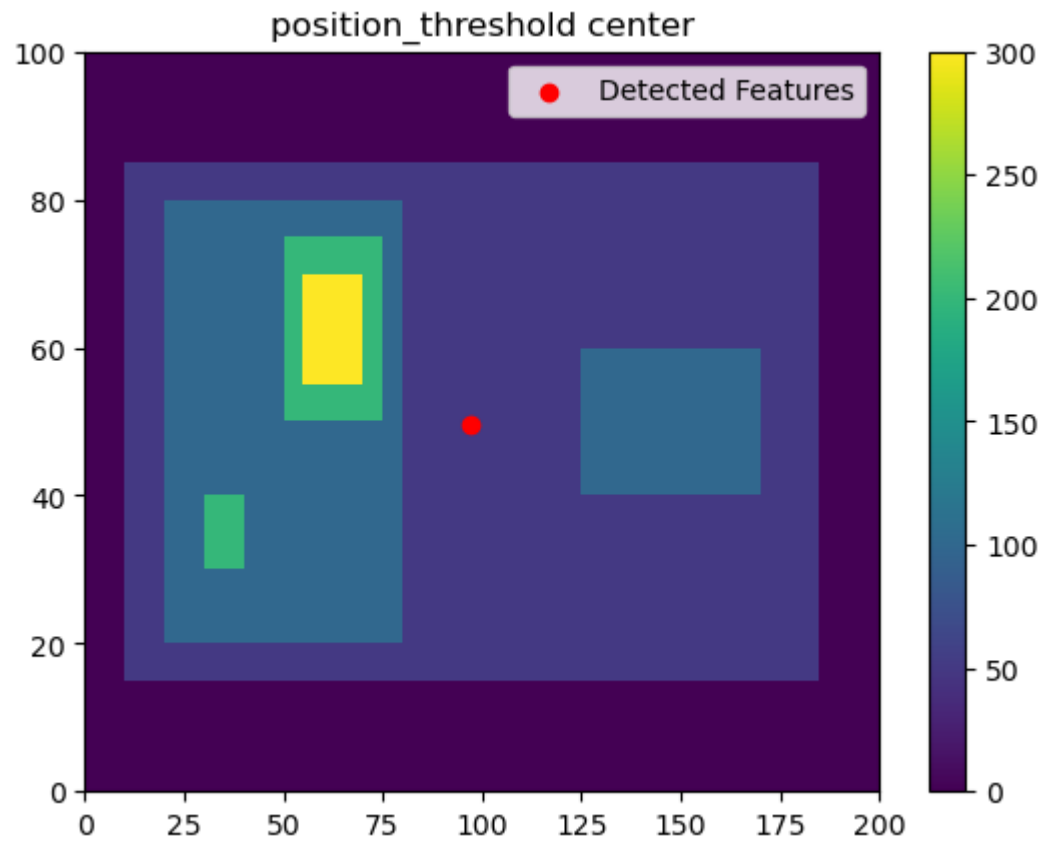
[3]:
```
# We now need to generate an Iris DataCube out of this dataset to run tobac feature detection.
# One can use xarray to generate a DataArray and then convert it to Iris, as done here.
input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()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

```
/var/folders/40/kfr98p0j7n30fjp2n4ljjqbh0000gr/T/ipykernel_51458/1365866487.py:3: UserWarning: Converting non-nanosecond precision datetime values to nanosecond precision. This behavior can eventually be relaxed in xarray, as it is an artifact from pandas which is now beginning to support non-nanosecond precision values. This warning is caused by passing non-nanosecond np.datetime64 or np.timedelta64 values to the DataArray or Variable constructor; it can be silenced by converting the values to nanosecond precision ahead of time.
  input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()
```

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.

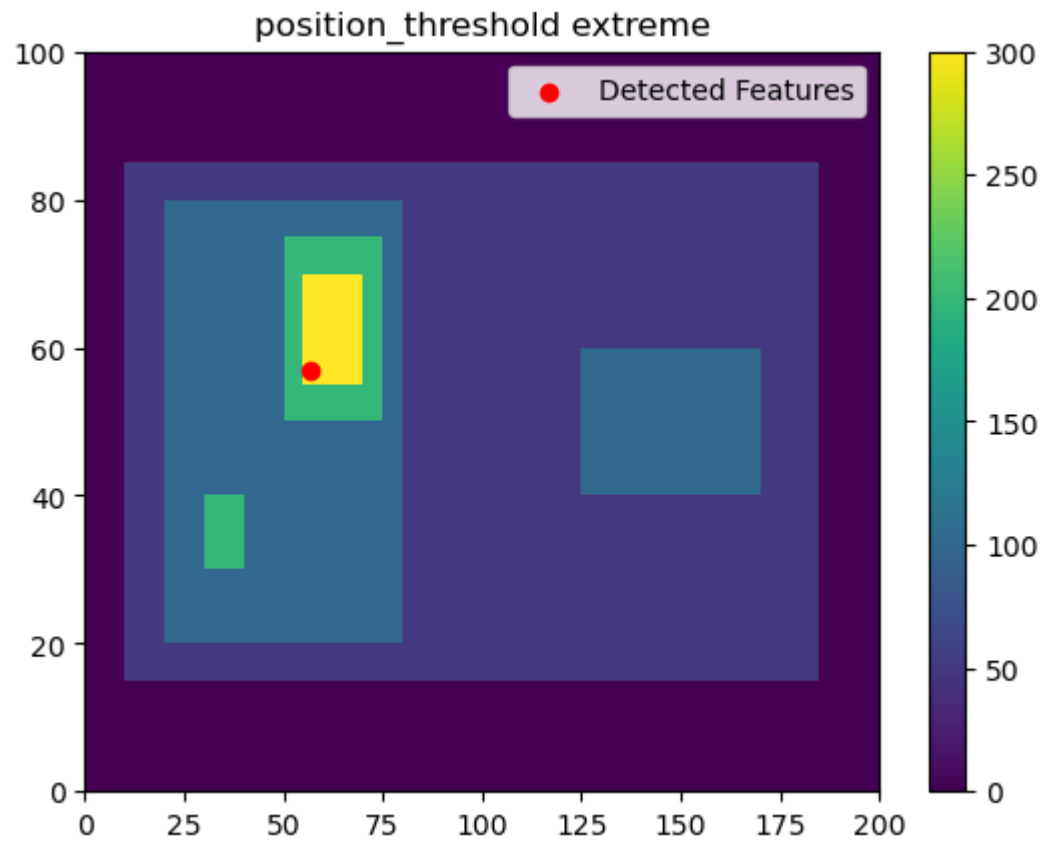
[4]:
```
thresholds = [50,]
position_threshold = 'center'
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↪field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
↪threshold=position_threshold)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# 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")
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.

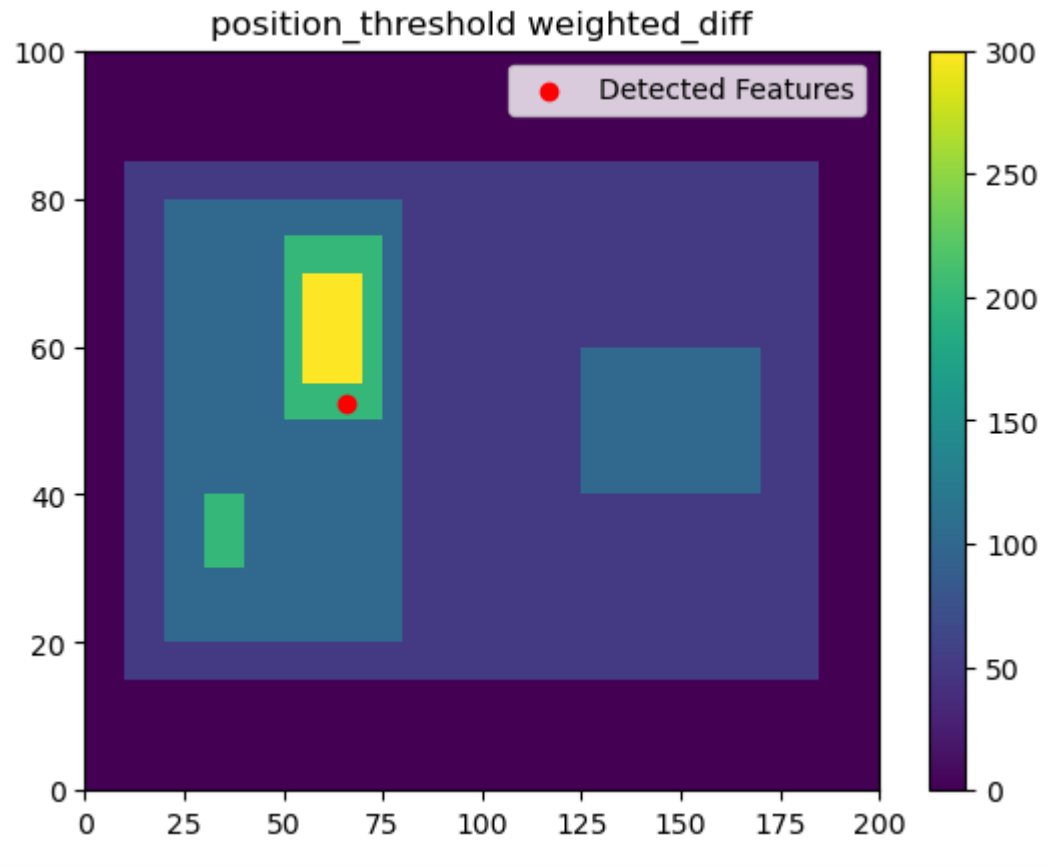
[5]:
```
thresholds = [50,]
position_threshold = 'extreme'
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↪field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
↪threshold=position_threshold)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# 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.

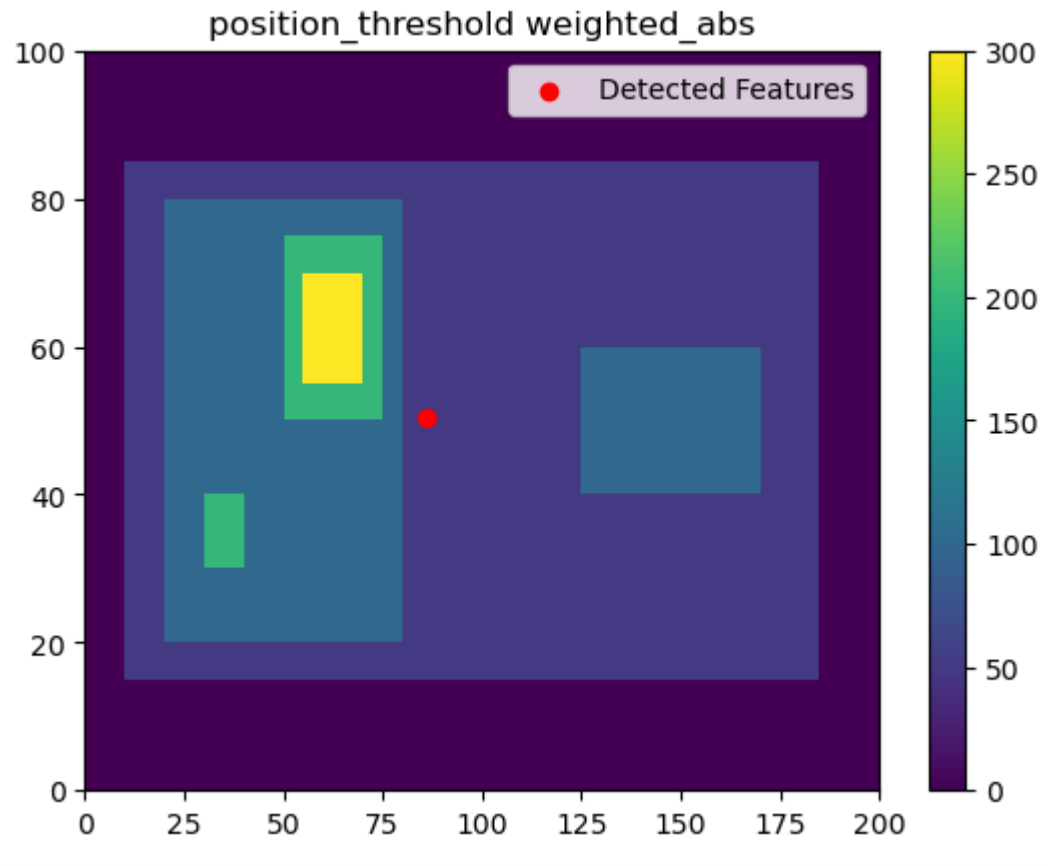
[6]:
```
thresholds = [50,]
position_threshold = 'weighted_diff'
single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↪field_iris, dxy = 1000, threshold=thresholds, target='maximum', position_
↪threshold=position_threshold)
plt.pcolormesh(input_field_arr[0])
plt.colorbar()
# 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'

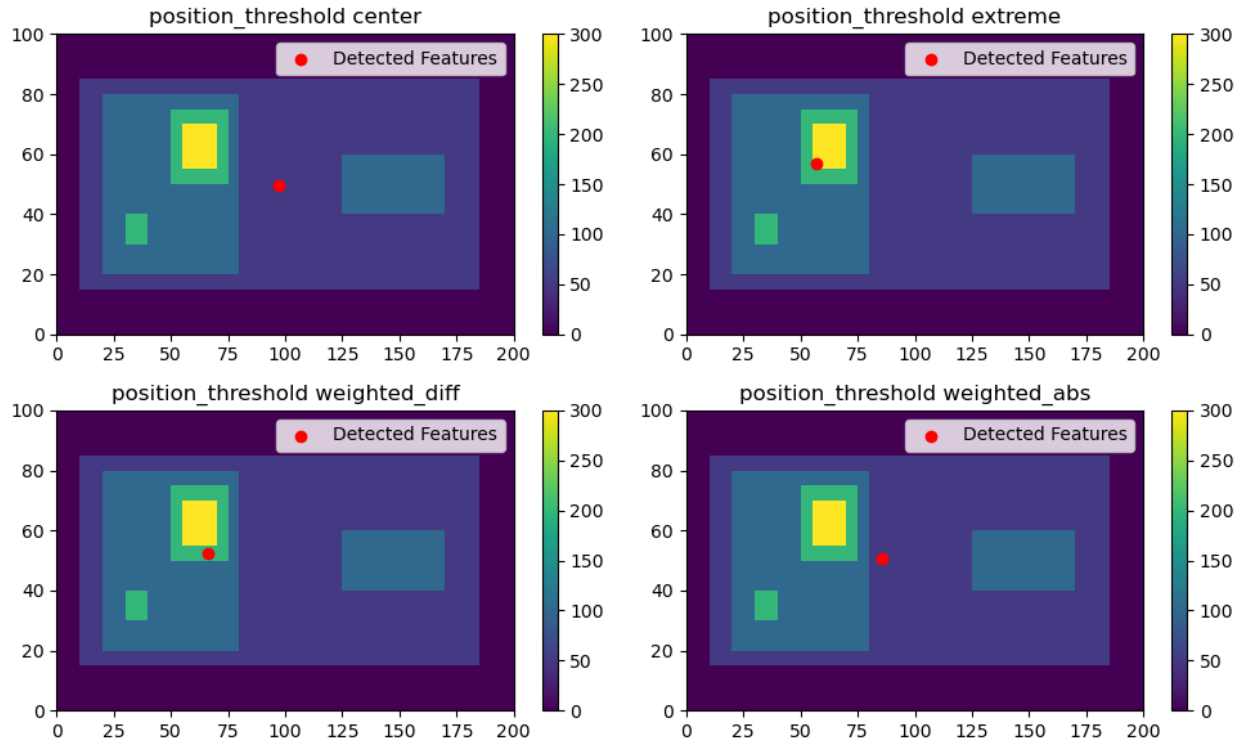
This option will choose the center of the region weighted by the absolute values of the field.

[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.

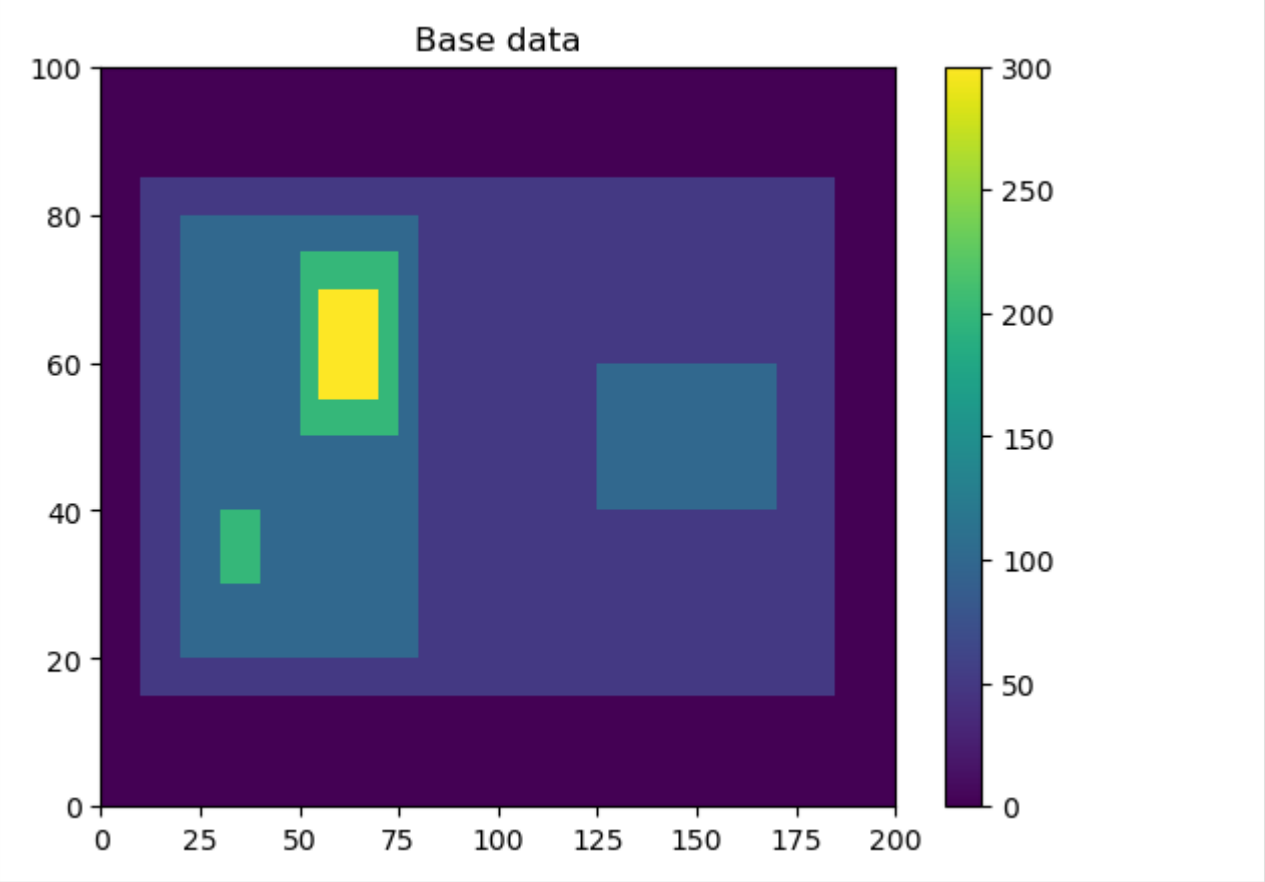
10.4.1 Imports

[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
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()
```



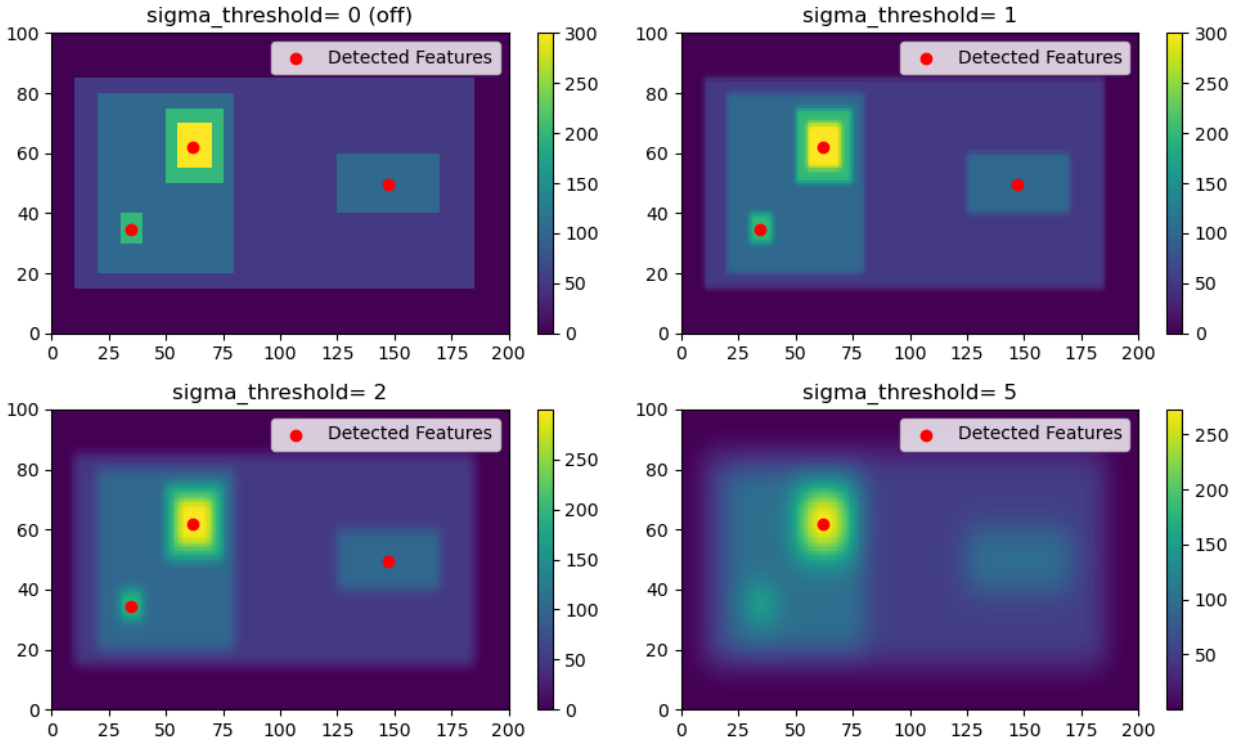
[3]:
```
# We now need to generate an Iris DataCube out of this dataset to run tobac feature detection.
# One can use xarray to generate a DataArray and then convert it to Iris, as done here.
input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()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

```
/var/folders/40/kfr98p0j7n30fjp2n4ljjqbh0000gr/T/ipykernel_51487/1365866487.py:3: UserWarning: Converting non-nanosecond precision datetime values to nanosecond precision. This behavior can eventually be relaxed in xarray, as it is an artifact from pandas which is now beginning to support non-nanosecond precision values. This warning is caused by passing non-nanosecond np.datetime64 or np.timedelta64 values to the DataArray or Variable constructor; it can be silenced by converting the values to nanosecond precision ahead of time.
  input_field_iris = xr.DataArray(input_field_arr, dims=['time', 'Y', 'X'], coords={'time': [np.datetime64('2019-01-01T00:00:00')]}).to_iris()
```

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.

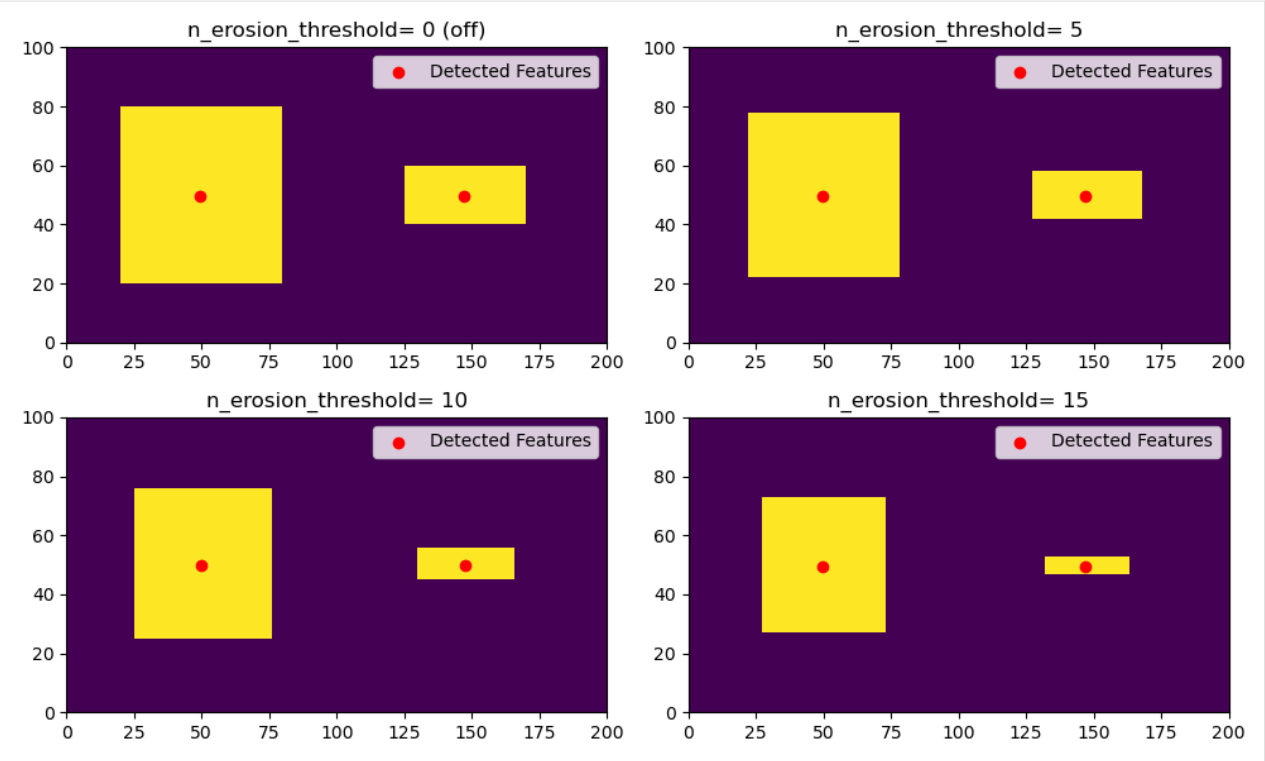
[4]:
```
thresholds = [50, 100, 150, 200]
fig, axarr = plt.subplots(2,2, figsize=(10,6))
sigma_values = [0, 1, 2, 5]
for sigma_value, ax in zip(sigma_values, axarr.flatten()):
    single_threshold_features = tobac.feature_detection_multithreshold(field_in = input_
↪field_iris, dxy = 1000, threshold=thresholds,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


n_erosion_threshold= 0 (off)
n_erosion_threshold= 5
n_erosion_threshold= 10
n_erosion_threshold= 15
Detected Features
0
20
40
60
80
100
0
25
50
75
100
125
150
175
200

CHAPTER

ELEVEN

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_	First horizontal dimension in grid point space (typically, although not always, N/S or y space)	Number of grid points	float
hdim_	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_va	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 datetime
timesti	String representation of the feature time	YYYY MM-DD HH:M	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
pro	X location of the feature in projection coodinates	Pro	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

One can optionally get the bulk statistics of the data points belonging to each feature regiion or volume. This is done using the *statistics* parameter when calling **:ufunc:`tobac.feature_detection_multithreshold`** . The user-defined metrics are then added as columns to the output dataframe.

CHAPTER

TWELVE

SEGMENTATION

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.

CHAPTER

THIRTEEN

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 `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.

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 and with 2D or 3D feature detection output, 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. Note that the segmentation is not inclusive of the threshold value, meaning that only values greater than (for `target='maximum'`) or smaller than (for `target='minimum'`) the threshold are included in the segmented region.

13.4 Projecting 2D Spatial Features into 3D Segmentation

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, 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 Projecting 3D Spatial Features into 2D Segmentation

When running feature detection on a 3D dataset and then using these detected features to segment data in 2D, the vertical coordinate is ignored. In case of vertically overlapping features, the larger `Feature` value is currently seeded.

13.6 Projecting 3D Spatial Features into 3D Segmentation

When running feature detection on a 3D dataset and then using these detected features to segment data in 3D, there are currently two options for determining how to seed the watershedding algorithm: *column* seeding (set by `seed_3D_flag='column'`) and *box* seeding (set by `seed_3D_flag='box'`). We generally recommend *box* seeding when running feature detection and segmentation in 3D.

Column seeding (`seed_3D_flag='column'`) works by setting seed markers throughout some or all of the vertical column at all detected feature centroids (i.e., one column per feature detected). While the default behavior is to seed throughout the full vertical column, the vertical extent of the seeds can be set by passing a slice into the `level` parameter. Note that `level` operates in *array* coordinates rather than physical coordinates.

Box seeding (`seed_3D_flag='box'`) sets a cube or rectangular seed markers around the detected feature in 3D space. The marker size is user defined (in array coordinates) by `seed_3D_size` as either an integer (for a cube) or a tuple of `(int, int, int)`, ordered `(vertical, hdim_1, hdim_2)`. Note that `seed_3D_size` must be an odd number to avoid the box becoming biased to one side. If two seed boxes overlap, the seeded area is marked with the closest feature centroid.

13.7 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`.

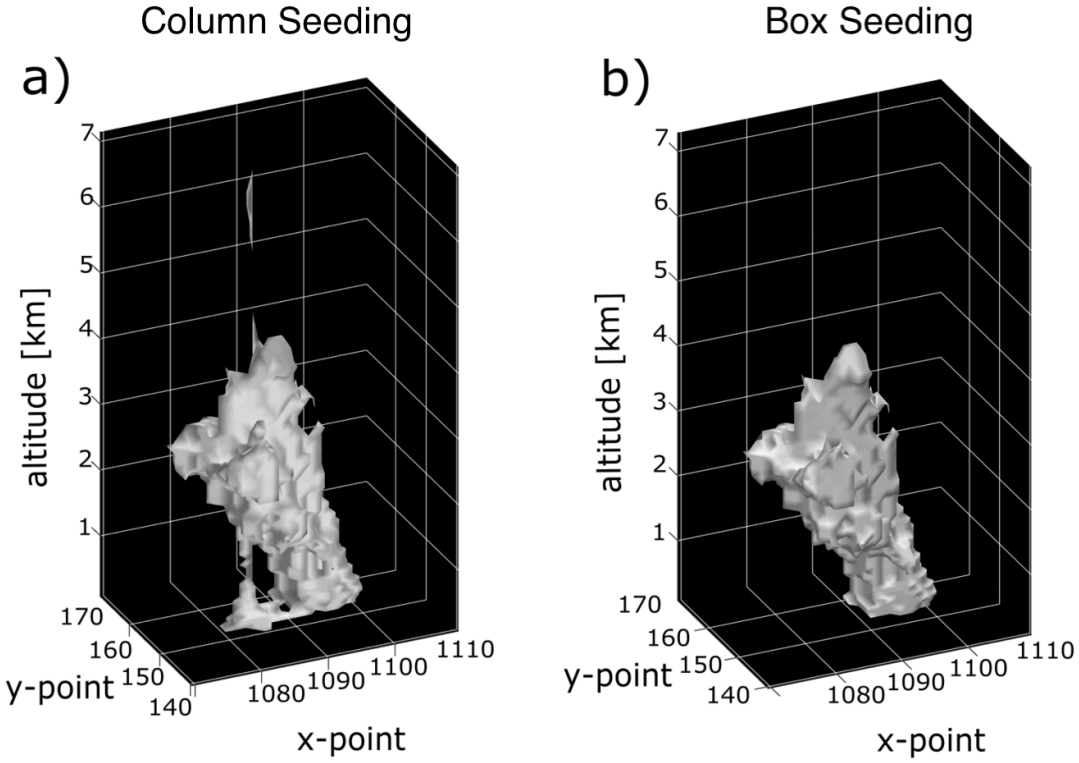


Fig. 1: An example plot from a numerical model simulation showing the real-world difference between column and box seeding with a tilted feature. As this image demonstrates, box seeding is typically recommended for 3D data.

CHAPTER

FOURTEEN

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

One can optionally get the bulk statistics of the data points belonging to each segmented feature (i.e. either the 2D area or the 3D volume assigned to the feature). This is done using the *statistics* parameter when calling **:ufunc:`tobac.segmentation.segmentation`** . The user-defined metrics are then added as columns to the output dataframe, for example:

Table 2: tobac Segmentation Output Variables

Variable Name	Description	Units	Type
feature_m	Mean of feature data points	same as input field	float
feature_m	Maximum value of feature data points	same as input field	float
feature_m	Minimum value of feature data points	same as input field	float
feature_su	Sum of feature data points	same as input field	float
major_ax	The length of the major axis of the ellipse that has the same normalized second central moments as the feature area	number of grid cells, multiply by dx to get distance unit	float
feature_p	Percentiles from 0 to 100 (with increment 1) of feature data distribution	same as input field	ndarray

Note that these statistics refer to the data fields that are used as input for the segmentation. It is possible to run the segmentation with different input (see transform segmentation) data to get statistics of a feature based on different variables (e.g. get statistics of cloud top temperatures as well as rain rates for a certain storm object).

CHAPTER

FIFTEEN

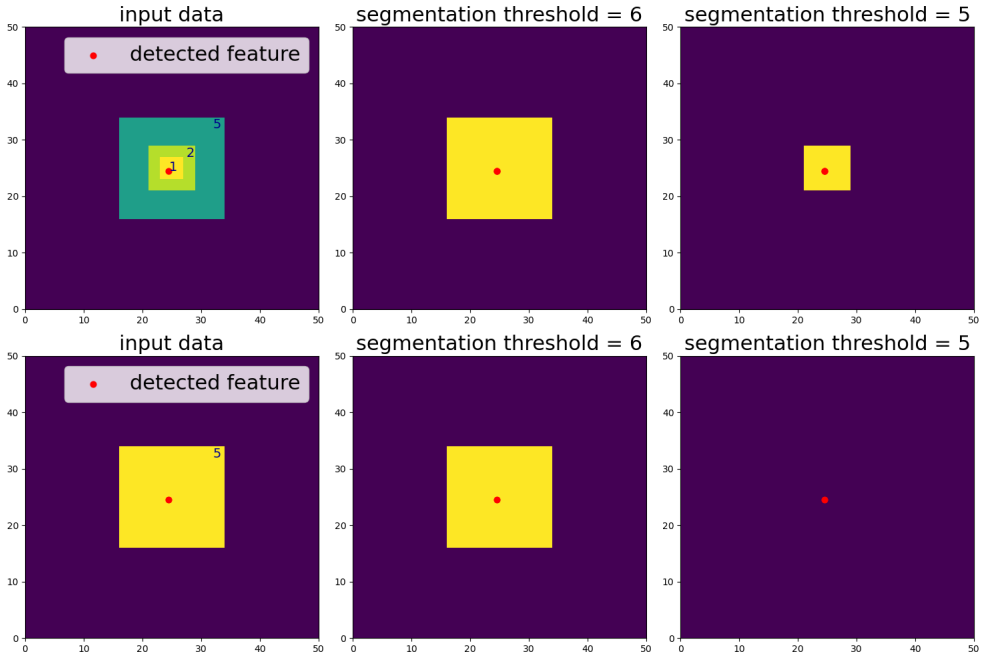
FEATURES WITHOUT SEGMENTED AREAS

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).

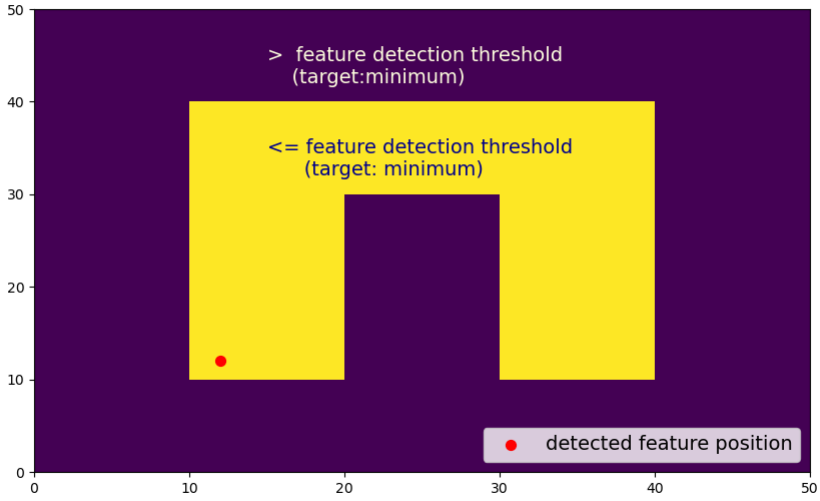
If the segmentation threshold is lower than or equal to the highest feature detection threshold (e.g. *threshold = 5*), features with threshold values lower than 5 still get a segmented area associated with them (upper panels). However,

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*:

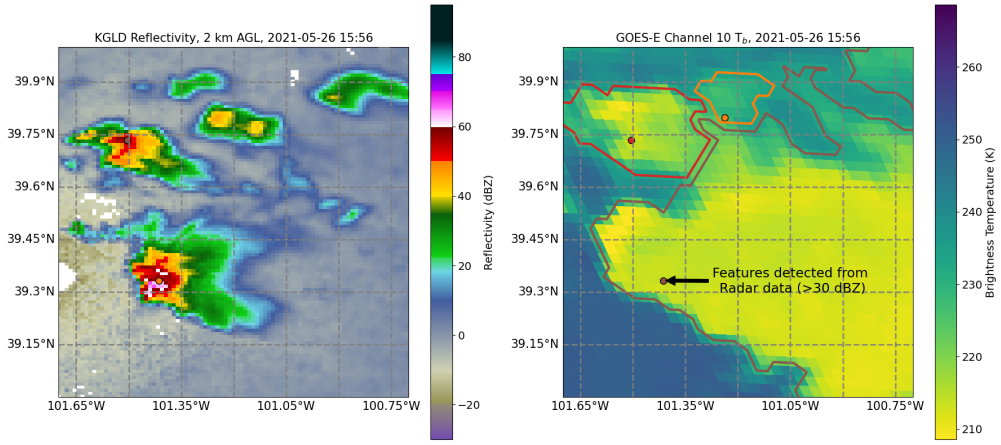


CHAPTER

SIXTEEN

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.

CHAPTER

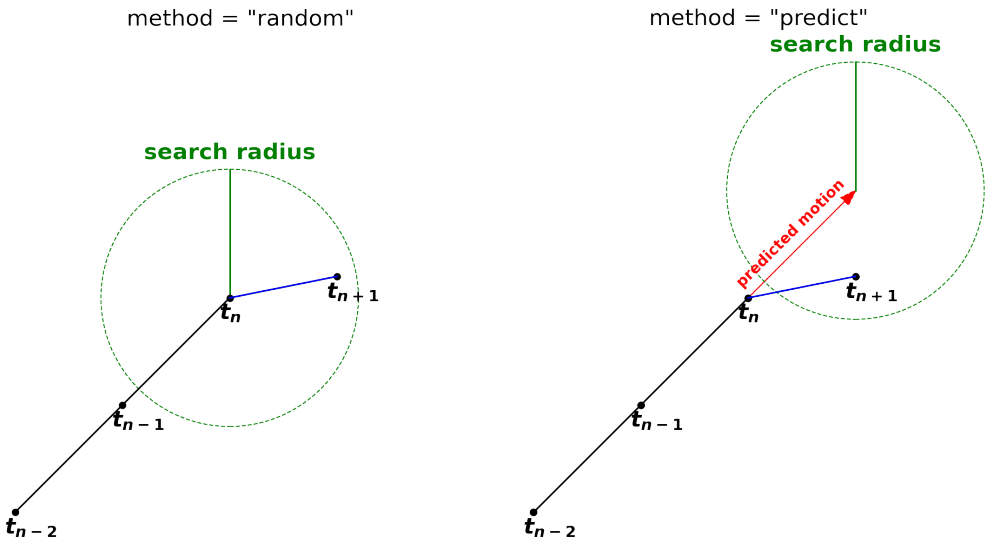
SEVENTEEN

LINKING

Currently implemented options for linking detected features into tracks:

Trackpy:

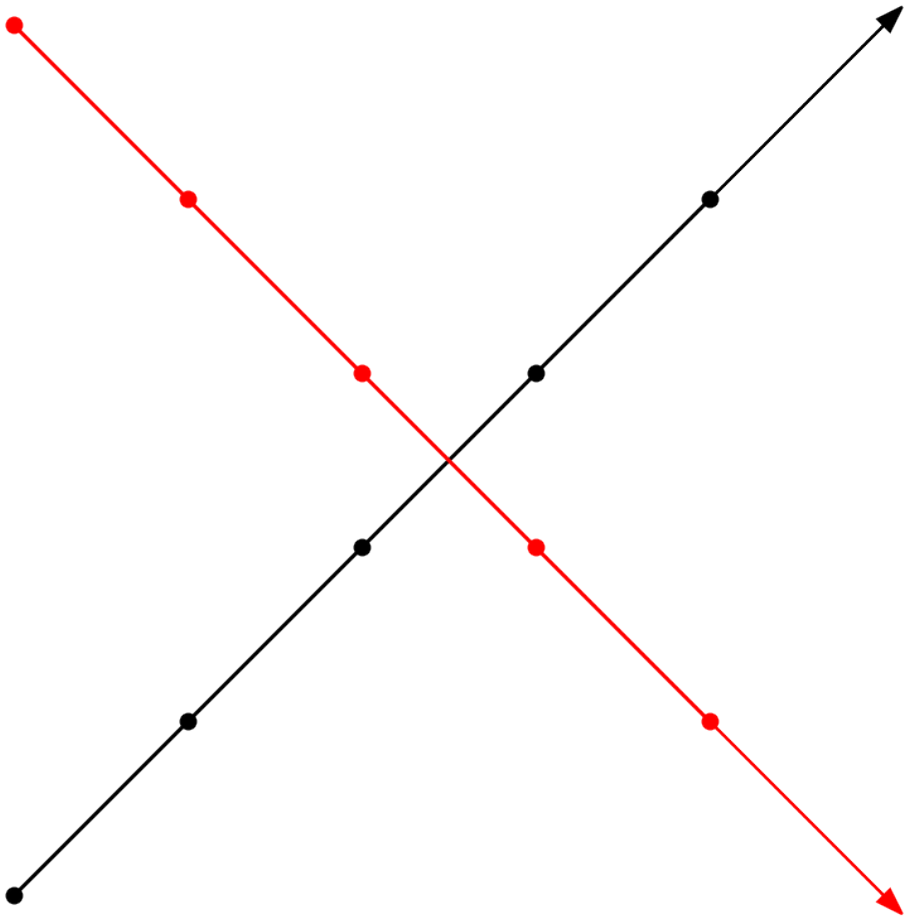
This method uses the trackpy library (http://soft-matter.github.io/trackpy). This approach only takes the point-like position of the feature, e.g. determined as the weighted mean, into account. Features to link with are looked for in a search radius defined by the parameters v_max or d_max. The position 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:



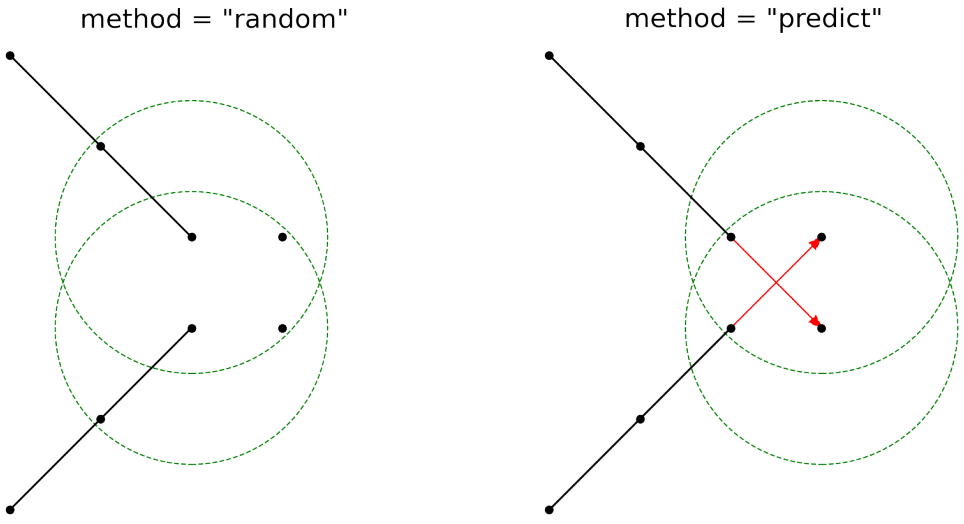
If there is only one feature in the search radius, the linking can happen immediately. If there are none, the track ends at this timestep. If there are more options, trackpy performs a decision process. Assume there are N features in the current and also N in the next timeframe and they are all within 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 $\delta_n, n = 1, 2, ..., N$ 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:

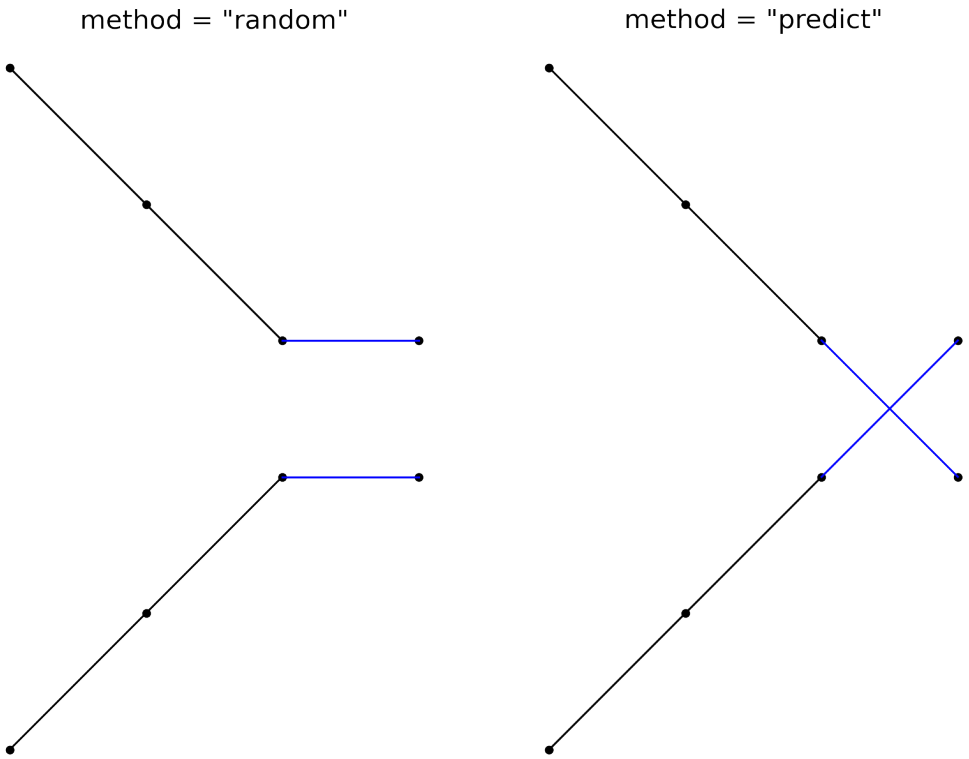


If the search radius is chosen large enough, each will contain two options, a horizontal and a diagonal feature:



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 excatly 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

EIGHTEEN

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_c	Time since cell was first detected.	minutes	object/python timedelta

CHAPTER

NINETEEN

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_p	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.	n/a	int64
feature_p	The associated parent cell id for each feature. All feature in a given cell will have the same cell id.	n/a	int64
feature_p	The associated parent track id for each feature. This is not the same as the cell id number.	n/a	int64
track_	The number of features belonging to all child cells of a given track id.	n/a	int64
cell_cl	The number of features for each cell.	n/a	int64

CHAPTER

TWENTY

COMPUTE BULK STATISTICS

Bulk statistics allow for a wide range of properties of detected objects to be calculated during feature detection and segmentation or as a postprocessing step. The *tobac.utils.bulk_statistics.get_statistics_from_mask()* function applies one or more functions over one or more data fields for each detected object. For example, one could calculate the convective mass flux for each detected feature by providing fields of vertical velocity, cloud water content and area. Numpy-like broadcasting is supported, allowing 2D and 3D data to be combined.

20.1 tobac example: Compute bulk statistics during feature detection

You can compute bulk statistics for features from the feature detection or the masked features from the segmentation mask.

This example shows how to derive some basic statistics for precipitation features associated with isolated deep convective clouds using the same data as in our example for precipitation tracking. The data used in this example is downloaded from automatically as part of the notebook.

[1]:
```
# Import libraries
import iris
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import shutil
from six.moves import urllib
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Import tobac itself
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[3]:
```
# Disable a few warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

Download example data:

Actual download has to be performed only once for all example notebooks!

[4]:
```
data_out=Path('../')
```

[5]:
```
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
```

Load Data from downloaded file:

[6]:
```
Precip=iris.load_cube(str(data_file[0]),'surface_precipitation_average')
```

[7]:
```
#display information about the iris cube containing the surface precipitation data:
display(Precip)
```

```
<iris 'Cube' of surface_precipitation_average / (mm h-1) (time: 47; south_north: 198;␣
↪west_east: 198)>
```

[8]:
```
#Set up directory to save output:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

Feature detection with bulk statistics

Feature detection is perfomed based on surface precipitation field and a range of thresholds. The statistics are calculated for each individual feature region and added to the feature output dataframe. You can decide which statistics to calculate by providing a dictionary with the name of the metric as keys (this will be the name of the column added to the dataframe) and functions as values. Note that it is also possible to provide input parameter to these functions.

20.1.1 Setting parameters for feature detection:

[9]:
```
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['min_distance']=0
parameters_features['sigma_threshold']=1
parameters_features['threshold']=[1,2,3,4,5,10,15] #mm/h
parameters_features['n_erosion_threshold']=0
parameters_features['n_min_threshold']=3

# get temporal and spation resolution of the data
dxy,dt=tobac.get_spacings(Precip)
```

20.1.2 Defining the dictionary for the statistics to be calculated

[10]:
```
statistics = {}
statistics['mean_precip'] = np.mean
statistics['total_precip'] = np.sum
statistics['max_precip'] = np.max
```

for some functions, we need to provide additional input parameters, e.g. np.percentile(). These can be provided as key word arguments in form of a dictionary. So instead of the function, you can provide a tuple with both the function and its respective input parameters:

[11]:
```
statistics['percentiles'] = (np.percentile, {'q': [95,99]})
```

[12]:
```
# Feature detection based on based on surface precipitation field and a range of␣
↪thresholds
print('starting feature detection based on multiple thresholds')
Features= tobac.feature_detection_multithreshold(Precip,dxy,**parameters_features,␣
↪statistic=statistics)
print('feature detection done')
Features.to_hdf(savedir / 'Features.h5','table')
print('features saved')
```

```
starting feature detection based on multiple thresholds
```

```
feature detection done
```

```
features saved
```

Look at the output:

[13]:
```
Features.head()
```

[13]:

	frame	idx	hdim_1	hdim_2	num	threshold_value	mean_precip	\
0	0	1	50.065727	139.857477	9	1	1.241012	
1	0	15	120.527119	172.500325	4	1	1.25016	
2	0	18	126.779273	145.368401	15	1	1.564113	
3	0	34	111.611369	155.452030	4	2	2.313658	

(continues on next page)

(continued from previous page)

```
4      0   35  111.765231  164.938866    8               2    2.610886

  total_precip max_precip                                    percentiles  ...  \
0    11.169106   1.528488  ([1.4821563005447387, 1.5192213106155394],)  ...
1     5.000638   1.267255    ([1.266197031736374, 1.267043651342392],)  ...
2    23.461691   2.321664    ([2.268769121170044, 2.311084909439087],)  ...
3      9.25463   2.409467  ([2.4016830801963804, 2.4079100108146667],)  ...
4    20.887089   3.081343    ([2.995926022529602, 3.064259362220764],)  ...

                  time              timestr south_north   west_east  \
0  2013-06-19 20:05:00  2013-06-19 20:05:00  331.065727  420.857477
1  2013-06-19 20:05:00  2013-06-19 20:05:00  401.527119  453.500325
2  2013-06-19 20:05:00  2013-06-19 20:05:00  407.779273  426.368401
3  2013-06-19 20:05:00  2013-06-19 20:05:00  392.611369  436.452030
4  2013-06-19 20:05:00  2013-06-19 20:05:00  392.765231  445.938866

   projection_y_coordinate           y   latitude  longitude  \
0            165782.863285  331.065727  29.846362 -94.172015
1            201013.559414  401.527119  30.166929 -93.996892
2            204139.636582  407.779273  30.196499 -94.139960
3            196555.684682  392.611369  30.126871 -94.087317
4            196632.615461  392.765231  30.127221 -94.037226

   projection_x_coordinate           x
0            210678.738492  420.857477
1            227000.162468  453.500325
2            213434.200454  426.368401
3            218476.015240  436.452030
4            223219.433218  445.938866

[5 rows x 21 columns]
```

[ ]:

20.2 tobac example: Compute bulk statistics during segmentation

This example shows how to derive some basic statistics for the segmented precipitation features associated with isolated deep convective clouds using the same data as in our example for precipitation tracking. As usual, we perform the segmentation step using the output from the feature detection, but we require some statistics to be calculated for the segmented features.

[1]:
```
# Import libraries
import iris
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import shutil
from six.moves import urllib
```

(continues on next page)

(continued from previous page)

```
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Disable a few warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

[3]:
```
# Import tobac itself
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[4]:
```
data_out=Path('../')
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
```

[5]:
```
#Set up directory to save output:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

[6]:
```
Precip=iris.load_cube(str(data_file[0]),'surface_precipitation_average')
```

Feature detection

[7]:
```
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['min_distance']=0
parameters_features['sigma_threshold']=1
parameters_features['threshold']=[1,2,3,4,5,10,15] #mm/h
```

(continues on next page)

(continued from previous page)

```
parameters_features['n_erosion_threshold']=0
parameters_features['n_min_threshold']=3

# get temporal and spation resolution of the data
dxy,dt=tobac.get_spacings(Precip)
```

[8]:
```
# Feature detection based on based on surface precipitation field and a range of␣
↪thresholds
print('starting feature detection based on multiple thresholds')
Features= tobac.feature_detection_multithreshold(Precip,dxy,**parameters_features)
print('feature detection done')
Features.to_hdf(savedir / 'Features.h5','table')
print('features saved')
```

```
starting feature detection based on multiple thresholds
```

```
feature detection done
```

```
features saved
```

Segmentation with bulk statistics

Segmentation is performed based on a watershedding and a threshold value. The statistics are calculated for each individual feature region and added to the feature output dataframe from the segmentation process. You can decide which statistics to calculate by providing a dictionary with the name of the metric as keys (this will be the name of the column added to the dataframe) and functions as values. Note that it is also possible to provide input parameter to these functions.

20.2.1 Setting parameters for segmentation:

[9]:
```
# Dictionary containing keyword arguments for segmentation step:
parameters_segmentation={}
parameters_segmentation['method']='watershed'
parameters_segmentation['threshold']=1  # mm/h mixing ratio

# get temporal and spation resolution of the data
dxy,dt=tobac.get_spacings(Precip)
```

20.2.2 Defining the dictionary for the statistics to be calculated

[10]:
```
statistics = {}
statistics['mean_precip'] = np.mean
statistics['total_precip'] = np.sum
statistics['max_precip'] = np.max
```

For some functions, we need to provide additional input parameters, e.g. np.percentile(). These can be provided as key word arguments in form of a dictionary. So instead of the function, you can provide a tuple with both the function and its respective input parameters:

[11]:
```
statistics['percentiles'] = (np.percentile, {'q': [95,99]})
```

[12]:
```
# Perform Segmentation and save resulting mask to NetCDF file:
print('Starting segmentation based on surface precipitation')
Mask,Features_Precip=tobac.segmentation_2D(Features,Precip,dxy,**parameters_segmentation,
↪ statistic=statistics)
print('segmentation based on surface precipitation performed, start saving results to␣
↪files')
iris.save([Mask], savedir / 'Mask_Segmentation_precip.nc', zlib=True, complevel=4)
Features_Precip.to_hdf(savedir / 'Features_Precip.h5', 'table')
print('segmentation surface precipitation performed and saved')
```

```
Starting segmentation based on surface precipitation
```

```
segmentation based on surface precipitation performed, start saving results to files
```

```
segmentation surface precipitation performed and saved
```

Look at the output:

[13]:
```
Features_Precip.head()
```

[13]:
```
   frame  idx       hdim_1      hdim_2  num  threshold_value  feature  \
0      0    1   50.065727  139.857477    9                1        1
1      0   15  120.527119  172.500325    4                1        2
2      0   18  126.779273  145.368401   15                1        3
3      0   34  111.611369  155.452030    4                2        4
4      0   35  111.765231  164.938866    8                2        5

                 time              timestr  south_north  ...           y  \
0  2013-06-19 20:05:00  2013-06-19 20:05:00   331.065727  ...  331.065727
1  2013-06-19 20:05:00  2013-06-19 20:05:00   401.527119  ...  401.527119
2  2013-06-19 20:05:00  2013-06-19 20:05:00   407.779273  ...  407.779273
3  2013-06-19 20:05:00  2013-06-19 20:05:00   392.611369  ...  392.611369
4  2013-06-19 20:05:00  2013-06-19 20:05:00   392.765231  ...  392.765231

    latitude  longitude  projection_x_coordinate           x  ncells  \
0  29.846362 -94.172015            210678.738492  420.857477      10
1  30.166929 -93.996892            227000.162468  453.500325      10
2  30.196499 -94.139960            213434.200454  426.368401      11
3  30.126871 -94.087317            218476.015240  436.452030      19
4  30.127221 -94.037226            223219.433218  445.938866      20

   mean_precip  total_precip max_precip  \
0     1.629695     16.296951   2.289786
1     1.409547     14.095468   1.819811
2     2.441526     26.856783   3.771701
3     1.938501     36.831512   4.067666
4     2.486886     49.737709   4.380943

                                 percentiles
0    ([2.221776068210602, 2.276183712482452],)
1  ([1.8030404090881347, 1.8164567756652832],)
2    ([3.710712432861328, 3.759503173828125],)
3    ([3.940941762924194, 4.042321195602417],)
```

(continues on next page)

(continued from previous page)

```
4   ([4.087516045570374, 4.3222578477859495],)

[5 rows x 22 columns]
```

[ ]:

20.3 tobac example: Compute bulk statistics as a postprocessing step

Instead of during the feature detection or segmentation process, you can also calculate bulk statistics of your detected/tracked objects as a postprocessing step, i.e. based on a segmentation mask. This makes it possible to combine different datasets and derive statistics for your detected features based on other input fields (e.g., get precipitation statistics under cloud features that were segmented based on brightness temperatures or outgoing longwave radiation).

This notebook shows an example for how to compute bulk statistics for detected features as a postprocessing step, that is based on the segmentation mask that we have already created. We perform the feature detection and segmentation with data from our example for precipitation tracking.

[1]:
```
# Import libraries
import iris
import numpy as np
import pandas as pd
import xarray as xr
import matplotlib.pyplot as plt
import datetime
import shutil
from six.moves import urllib
from pathlib import Path
%matplotlib inline
```

[2]:
```
# Import tobac itself
import tobac
print('using tobac version', str(tobac.__version__))
```

```
using tobac version 1.5.3
```

[3]:
```
# Disable a few warnings:
import warnings
warnings.filterwarnings('ignore', category=UserWarning, append=True)
warnings.filterwarnings('ignore', category=RuntimeWarning, append=True)
warnings.filterwarnings('ignore', category=FutureWarning, append=True)
warnings.filterwarnings('ignore',category=pd.io.pytables.PerformanceWarning)
```

Feature detection

[4]:
```
#Set up directory to save output:
savedir=Path("Save")
if not savedir.is_dir():
    savedir.mkdir()
plot_dir=Path("Plot")
```

(continues on next page)

(continued from previous page)

```
if not plot_dir.is_dir():
    plot_dir.mkdir()
```

[5]:
```
data_out=Path('../')
# Download the data: This only has to be done once for all tobac examples and can take a␣
↪while
data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
if len(data_file) == 0:
    file_path='https://zenodo.org/records/3195910/files/climate-processes/tobac_example_
↪data-v1.0.1.zip'
    #file_path='http://zenodo..'
    tempfile=Path('temp.zip')
    print('start downloading data')
    request=urllib.request.urlretrieve(file_path, tempfile)
    print('start extracting data')
    shutil.unpack_archive(tempfile, data_out)
    tempfile.unlink()
    print('data extracted')
    data_file = list(data_out.rglob('data/Example_input_Precip.nc'))
```

[6]:
```
Precip=iris.load_cube(str(data_file[0]),'surface_precipitation_average')
```

[7]:
```
parameters_features={}
parameters_features['position_threshold']='weighted_diff'
parameters_features['sigma_threshold']=0.5
parameters_features['min_distance']=0
parameters_features['sigma_threshold']=1
parameters_features['threshold']=[1,2,3,4,5,10,15] #mm/h
parameters_features['n_erosion_threshold']=0
parameters_features['n_min_threshold']=3

# get temporal and spation resolution of the data
dxy,dt=tobac.get_spacings(Precip)
```

[8]:
```
# Feature detection based on based on surface precipitation field and a range of␣
↪thresholds
print('starting feature detection based on multiple thresholds')
Features= tobac.feature_detection_multithreshold(Precip,dxy,**parameters_features)
print('feature detection done')
Features.to_hdf(savedir / 'Features.h5','table')
print('features saved')
```

```
starting feature detection based on multiple thresholds
```

```
feature detection done
```

```
features saved
```

Segmentation

[9]:
```
# Dictionary containing keyword arguments for segmentation step:
parameters_segmentation={}
parameters_segmentation['method']='watershed'
```

(continues on next page)

(continued from previous page)

```
parameters_segmentation['threshold']=1  # mm/h mixing ratio

# get temporal and spation resolution of the data
dxy,dt=tobac.get_spacings(Precip)
```

[10]:
```
# Perform Segmentation and save resulting mask to NetCDF file:
print('Starting segmentation based on surface precipitation')
Mask_Precip,Features_Precip=tobac.segmentation_2D(Features,Precip,dxy,**parameters_
↪segmentation)
print('segmentation based on surface precipitation performed, start saving results to␣
↪files')
iris.save([Mask_Precip], savedir / 'Mask_segmentation_precip.nc', zlib=True, complevel=4)
Features_Precip.to_hdf(savedir / 'Features_Precip.h5', 'table')
print('segmentation surface precipitation performed and saved')
```

```
Starting segmentation based on surface precipitation
```

```
segmentation based on surface precipitation performed, start saving results to files
```

```
segmentation surface precipitation performed and saved
```

Get bulk statistics from segmentation mask file

You can decide which statistics to calculate by providing a dictionary with the name of the metric as keys (this will be the name of the column added to the dataframe) and functions as values. Note that it is also possible to provide input parameter to these functions.

[11]:
```
from tobac.utils import get_statistics_from_mask
```

20.3.1 Defining the dictionary for the statistics to be calculated

[12]:
```
statistics = {}
statistics['mean_precip'] = np.mean
statistics['total_precip'] = np.sum
statistics['max_precip'] = np.max
```

For some functions, we need to provide additional input parameters, e.g. np.percentile(). These can be provided as key word arguments in form of a dictionary. So instead of the function, you can provide a tuple with both the function and its respective input parameters:

[13]:
```
statistics['percentiles'] = (np.percentile, {'q': [95,99]})
```

[14]:
```
features_with_stats = get_statistics_from_mask(Features_Precip, Mask_Precip, Precip,␣
↪statistic=statistics)
```

Look at the output:

[15]:
```
features_with_stats.mean_precip.head()
```

[15]:
```
0    1.629695
1    1.409547
2    2.441526
3    1.938501
4    2.486886
Name: mean_precip, dtype: object
```

[16]:
```
features_with_stats.total_precip.head()
```

[16]:
```
0    16.296951
1    14.095468
2    26.856783
3    36.831512
4    49.737709
Name: total_precip, dtype: object
```

[17]:
```
features_with_stats.percentiles.head()
```

[17]:
```
0      ([2.221776068210602, 2.276183712482452],)
1    ([1.8030404090881347, 1.8164567756652832],)
2      ([3.710712432861328, 3.759503173828125],)
3      ([3.940941762924194, 4.042321195602417],)
4     ([4.087516045570374, 4.3222578477859495],)
Name: percentiles, dtype: object
```

[ ]:

CHAPTER

TWENTYONE

TOBAC PACKAGE

21.1 Submodules

21.2 tobac.analysis module

21.3 tobac.analysis.cell_analysis module

Perform analysis on the properties of tracked cells

tobac.analysis.cell_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_1** (*pandas.DataFrame*) – The tracks conaining the cells to analyze.
- **track_2** (*pandas.DataFrame*) – The tracks conaining 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.cell_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** –

Return type
None

tobac.analysis.cell_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** –

Return type
None

tobac.analysis.cell_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.

Return type
None

tobac.analysis.cell_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 wether 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 intervalls

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

tobac.analysis.cell_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 tracked cells.

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 wether 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 intervalls.
- **minutes, optional** (*ndarray*) – Numpy.array of the lifetime of each feature in minutes. Returned if return_values is True.

tobac.analysis.cell_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*, *return_values=False*)

Create an velocity histogram of the tracked cells. 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 wether the required coordinates are present and starts with 'xy'. Default is None.
- **return_values** (*bool, optional*) – Bool determining wether 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.

21.4 tobac.analysis.feature_analysis module

Perform analysis on the properties of detected features

tobac.analysis.feature_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 wether the areas of the features are returned from this function. Default is False.
- **representive_area** (*bool, optional*) – If False, no weights will 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 intervalls.
- **areas** (*ndarray, optional*) – A numpy array approximating the area of each feature.

tobac.analysis.feature_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 intervalls

tobac.analysis.feature_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])*, *density=False*, *method_distance=None*, *return_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.

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 wether the required coordinates are present and starts with 'xy'. Default is None.

return_values
[bool, optional] Bool determining wether 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.

21.5 tobac.analysis.spatial module

Calculate spatial properties (distances, velocities, areas, volumes) of tracked objects

`tobac.analysis.spatial.`**`calculate_area`**(*features*, *mask*, *method_area=None*, *vertical_coord=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 wether the required coordinates are present and starts with 'xy'. Default is None.
- **`vertical_coord`** (*`None | str, optional (default: None)`*) – Name of the vertical coordinate. If None, tries to auto-detect. It looks for the coordinate or the dimension name corresponding to the string.

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_coords.

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.spatial.`**`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 assuems 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 auxilliary coordinate containing a 2d grid of latitudes for each point.

- **_2Dlon_coord** (*AuxCoord*) – Iris auxilliary 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.spatial.**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_1** (*pandas.DataFrame or pandas.Series*) – 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.
- **feature_2** (*pandas.DataFrame or pandas.Series*) – 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 wether 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.spatial.**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 wether 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.spatial.**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 timeframe. 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 timeframe. 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 wether the required coordinates are present and starts with 'xy'. Default is None.

Returns

velocity – Value of the approximate velocity.

Return type

float

tobac.analysis.spatial.**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

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

21.6 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.

21.7 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: iris.cube.Cube*, *dxy: float = None*, *threshold: list[float] = None*, *min_num: int = 0*, *target: typing_extensions.Literal[maximum, minimum] = 'maximum'*, *position_threshold: typing_extensions.Literal[center, extreme, weighted_diff, weighted abs] = 'center'*, *sigma_threshold: float = 0.5*, *n_erosion_threshold: int = 0*, *n_min_threshold: int = 0*, *min_distance: float = 0*, *feature_number_start: int = 1*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*, *vertical_coord: str = None*, *vertical_axis: int = None*, *detect_subset: dict = None*, *wavelength_filtering: tuple = None*, *dz: float | None = None*, *strict_thresholding: bool = False*, *statistic: dict[str, ~typing.Callable | tuple[~typing.Callable, dict]] | None = None*) → pandas.DataFrame

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),
- **dxy** (*float*) – Grid spacing of the input data (in meter).
- **thresholds** (*list of floats, optional*) – Threshold values used to select target regions to track. The feature detection is inclusive of the threshold value(s), i.e. values greater/less than or equal are included in the target region. 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 pixels by which to erode the identified features. Default is 0.
- **n_min_threshold** (*int, optional*) – Minimum number of identified contiguous pixels for a feature to be detected. Default is 0.
- **min_distance** (*float, optional*) – Minimum distance between detected features (in meters). 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: ~numpy.array*, *i_time: int*, *threshold: list[float] | None = None*, *min_num: int = 0*, *target: typing_extensions.Literal[maximum, minimum] = 'maximum'*, *position_threshold: typing_extensions.Literal[center, extreme, weighted_diff, weighted abs] = 'center'*, *sigma_threshold: float = 0.5*, *n_erosion_threshold: int = 0*, *n_min_threshold: int = 0*, *min_distance: float = 0*, *feature_number_start: int = 1*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*, *vertical_axis: int | None = None*, *dxy: float = -1*, *wavelength_filtering: tuple[float] | None = None*, *strict_thresholding: bool = False*, *statistic: dict[str, ~typing.Callable | tuple[~typing.Callable, dict]] | None = None*) → pandas.DataFrame

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*) – 3D field to perform the feature detection (single timestep) on.
- **i_time** (*int*) – Number of the current timestep.
- **threshold** (*list of floats, optional*) – Threshold value used to select target regions to track. The feature detection is inclusive of the threshold value(s), i.e. values greater/less than or equal are included in the target region. Default is None.
- **min_num** (*int, optional*) – This parameter is not used in the function. Default is 0.
- **target** (*{'maximum', 'minimum'}, optinal*) – 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 pixels by which to erode the identified features. Default is 0.
- **n_min_threshold** (*int, optional*) – Minimum number of identified contiguous pixels for a feature to be detected. Default is 0.

- **min_distance** (*float, optional*) – Minimum distance between detected features (in meters). 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 meters.
- **wavelength_filtering** (*tuple, optional*) – Minimum and maximum wavelength for spectral filtering in meters. 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.
- **statistic** (*dict, optional*) – Default is None. Optional parameter to calculate bulk statistics within feature detection. Dictionary with callable function(s) to apply over the region of each detected feature and the name of the statistics to appear in the feature ou tput dataframe. The functions should be the values and the names of the metric the keys (e.g. {'mean': np.mean})

Returns

features_threshold – Detected features for individual timestep.

Return type

pandas DataFrame

tobac.feature_detection.**feature_detection_threshold**(*data_i: array*, *i_time: int*, *threshold: float | None = None*, *min_num: int = 0*, *target: typing_extensions.Literal[maximum, minimum] = 'maximum'*, *position_threshold: typing_extensions.Literal[center, extreme, weighted_diff, weighted_abs] = 'center'*, *sigma_threshold: float = 0.5*, *n_erosion_threshold: int = 0*, *n_min_threshold: int = 0*, *min_distance: float = 0*, *idx_start: int = 0*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*, *vertical_axis: int = 0*) → tuple[pandas.DataFrame, dict]

Find features based on individual threshold value.

Parameters

- **data_i** (*np.array*) – 2D or 3D 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. The feature detection is inclusive of the threshold value(s), i.e. values greater/less than or equal are included in the target region. The feature detection is inclusive of the threshold value(s), i.e. values greater/less than or equal are included in the target region. 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 pixels by which to erode the identified features. Default is 0.
- **n_min_threshold** (*int, optional*) – Minimum number of identified contiguous pixels for a feature to be detected. Default is 0.
- **min_distance** (*float, optional*) – Minimum distance between detected features (in meters). 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: list[int], hdim2_indices: list[int], vdim_indices: list[int] | None = None, region_small: ~numpy.ndarray | None = None, region_bbox: list[int] | tuple[int] | None = None, track_data: ~numpy.ndarray | None = None, threshold_i: float | None = None, position_threshold: typing_extensions.Literal[center, extreme, weighted_diff, weighted abs] = 'center', target: typing_extensions.Literal[maximum, minimum] | None = None, PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none', hdim1_min: int = 0, hdim1_max: int = 0, hdim2_min: int = 0, hdim2_max: int = 0*) → tuple[float]

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 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: pandas.DataFrame*, *dxy: float | None = None*, *dz: float | None = None*, *min_distance: float | None = None*, *x_coordinate_name: str | None = None*, *y_coordinate_name: str | None = None*, *z_coordinate_name: str | None = None*, *target: typing_extensions.Literal[maximum, minimum] = 'maximum'*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*, *min_h1: int = 0*, *max_h1: int = 0*, *min_h2: int = 0*, *max_h2: int = 0*) → pandas.DataFrame

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 horzontal grid spacing (meters).
- **dz** (*float*) – Constant vertical grid spacing (meters), 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 (meters)
- **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 targeting 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: pandas.DataFrame*, *regions_i: dict*, *regions_old: dict*, *strict_thresholding: bool = False*) → pandas.DataFrame

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 greater/lower than and equal to threshold for the newly detected feature (feature ids as keys).
- **regions_old** (*dict*) – Dictionary containing the regions greater/lower than and equal to threshold from previous threshold (feature ids as keys).
- **strict_thresholding** (*Bool, optional*) – If True, a feature can only be detected if all previous thresholds have been met. Default is False.

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: list[tuple[int]]*, *region_outer: list[tuple[int]]*) → bool

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

21.8 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'))

21.9 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** (*tupel 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 wether 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 liftetime 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 liftetime 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 contrours
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 contrours 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*, *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*)

Make plots for cell in fixed frame and with one background field as filling and one background field as contrours 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'*, *plotdir='./'*, *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 contrours 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'*, *plotdir='./'*, *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 contrours 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 contrours 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'*, *plotdir='./'*, *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 contrours 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 contrours 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 wether the index of the cells is plotted next to the individual feature position. Default is True.
- **plot_features** (*bool, optional*) – Boolean defining wether 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 not 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 countour 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*, *margin_left=0.05*, *margin_right=0.05*, *margin_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** –

Return type

None

21.10 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-dimentional 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 visialization.

References

tobac.segmentation.**add_markers**(*features: pandas.DataFrame*, *marker_arr: array*, *seed_3D_flag: typing_extensions.Literal[column, box]*, *seed_3D_size: int | tuple[int] = 5*, *level: None | slice = None*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*) → array

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 (units of number of pixels), the seed box is identical in all dimensions. If it's a tuple, it specifies the seed area for each dimension separately, in units of pixels. Note: we strongly 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: array*, *unseeded_labels: array*, *border_min: int*, *border_max: int*, *markers_arr: array*, *inplace: bool = True*) → array

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 incomming 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

Return type
markers_arr with new markers added

tobac.segmentation.**segmentation**(*features: pandas.DataFrame*, *field: iris.cube.Cube*, *dxy: float*, *threshold: float = 0.003*, *target: typing_extensions.Literal[maximum, minimum] = 'maximum'*, *level: None | slice = None*, *method: typing_extensions.Literal[watershed] = 'watershed'*, *max_distance: None | float = None*, *vertical_coord: str | None = None*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*, *seed_3D_flag: typing_extensions.Literal[column, box] = 'column'*, *seed_3D_size: int | tuple[int] = 5*, *segment_number_below_threshold: int = 0*, *segment_number_unassigned: int = 0*, *statistic: dict[str, Callable | tuple[Callable, dict]] | None = None*) → tuple[iris.cube.Cube, pandas.DataFrame]

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 individal timestep of the input data.

Parameters

- **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 in meters.
- **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 in meters. 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 (units of number of pixels), the seed box is identical in all dimensions. If it's a tuple, it specifies the seed area for each dimension separately, in units of pixels. Note: we strongly 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.
- **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.
- **statistic** (*dict, optional*) – Default is None. Optional parameter to calculate bulk statistics within feature detection. Dictionary with callable function(s) to apply over the region of each detected feature and the name of the statistics to appear in the feature output dataframe. The functions should be the values and the names of the metric the keys (e.g. {'mean': np.mean})

Returns

- **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.

Raises

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'*, *statistic=None*)

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'*, *statistic=None*)

Wrapper for the segmentation()-function.

tobac.segmentation.**segmentation_timestep**(*field_in: iris.cube.Cube*, *features_in: pandas.DataFrame*, *dxy: float*, *threshold: float = 0.003*, *target: typing_extensions.Literal[maximum, minimum] = 'maximum'*, *level: None | slice = None*, *method: typing_extensions.Literal[watershed] = 'watershed'*, *max_distance: None | float = None*, *vertical_coord: str | None = None*, *PBC_flag: typing_extensions.Literal[none, hdim_1, hdim_2, both] = 'none'*, *seed_3D_flag: typing_extensions.Literal[column, box] = 'column'*, *seed_3D_size: int | tuple[int] = 5*, *segment_number_below_threshold: int = 0*, *segment_number_unassigned: int = 0*, *statistic: dict[str, Callable | tuple[Callable, dict]] | None = None*) → tuple[iris.cube.Cube, pandas.DataFrame]

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. The watershedding is exclusive of the threshold value, i.e. values greater (less) than the threshold are included in the target region, while values equal to the threshold value are excluded. Default is 3e-3.
- **target** (*{'maximum', 'minimum'}, optional*) – Flag to determine if tracking is targeting 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).
- **max_distance** (*float, optional*) – Maximum distance from a marker allowed to be classified as belonging to that cell in meters. 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 (units of number of pixels), the seed box is identical in all dimensions. If it's a tuple, it specifies the seed area for each dimension separately, in units of pixels. Note: we strongly 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.

- **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. This can be the same as *segment_number_below_threshold*, but can also be set separately.
- **statistics** (*boolean, optional*) – Default is None. If True, bulk statistics for the data points assigned to each feature are saved in output.

Returns

- **segmentation_out** (*iris.cube.Cube*) – Mask, 0 outside and integer numbers according to track inside the ojects.
- **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.

21.11 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

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 realisitic 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 abritraty date (2000-01-01 12:00). The longitude and latitude coordinates are added as 2D aux coordinates and arbitrary, but in realisitic 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 wether 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 abritraty date (2000-01-01 12:00). The longitude and latitude coordinates are added as 2D aux coordinates and arbitrary, but

in realisitic 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

21.12 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: pandas.DataFrame*, *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 or timesteps to extrapolate trajectories. Default is 0.
- **frame_max** (*int, optional*) – Size of input data along time axis. Default is None.
- **hdim_1_max** (*int, optional*) – Size of input data along first and second horizontal axis. Default is None.
- **hdim2_max** (*int, optional*) – 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** (*None*) – Input field. Not currently used; can be set to *None*.
- **dt** (*float*) – Time resolution of tracked features in seconds.
- **dxy** (*float*) – Horizontal grid spacing of the input data in meters.
- **dz** (*float*) – Constant vertical grid spacing (meters), 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 in meters. Only one of *d_max*, *d_min*, or *v_max* can be set. Default is None.
- **d_min** (*float, optional*) – Deprecated. Only one of *d_max*, *d_min*, or *v_max* can be set. 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 in meters per second. Only one of *d_max*, *d_min*, or *v_max* can be set. 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 that a cell must be tracked for to be considered a valid cell in seconds. Default is None.

- **order** (*int, optional*) – Order of polynomial used to extrapolate trajectory into gaps and ond start and end point. Default is 1.
- **extrapolate** (*int, optional*) – Number or timesteps to extrapolate trajectories. Currently unused. Default is 0.
- **method_linking** (*{'random', 'predict'}, optional*) – Flag choosing method used for trajectory linking. Default is 'random', although we typically encourage users to use 'predict'.
- **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 <= 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

21.13 tobac.utils modules

21.14 tobac.utils.bulk_statistics module

Support functions to compute bulk statistics of features, either as a postprocessing step or within feature detection or segmentation.

tobac.utils.bulk_statistics.**get_statistics**(*features: pandas.DataFrame, labels: ~numpy.ndarray[int], *fields: tuple[~numpy.ndarray], statistic: dict[str, ~typing.Callable | tuple[~typing.Callable, dict]] = {'ncells': <function count_nonzero>}, index: list[int] | None = None, default: None | float = None, id_column: str = 'feature', collapse_axis: None | int | list[int] = None*) → pandas.DataFrame

Get bulk statistics for objects (e.g. features or segmented features) given a labelled mask of the objects and any input field with the same dimensions or that can be broadcast with labels according to numpy-like broadcasting rules.

The statistics are added as a new column to the existing feature dataframe. Users can specify which statistics are computed by providing a dictionary with the column name of the metric and the respective function.

Parameters

- **features** (*pd.DataFrame*) – Dataframe with features or segmented features (output from feature detection or segmentation), which can be for the specific timestep or for the whole dataset
- **labels** (*np.ndarray[int]*) – Mask with labels of each regions to apply function to (e.g. output of segmentation for a specific timestep)
- ***fields** (*tuple[np.ndarray]*) – Fields to give as arguments to each function call. If the shape does not match that of labels, numpy-style broadcasting will be applied.
- **statistic** (*dict[str, Callable], optional (default: {'ncells':np.count_nonzero})*) – Dictionary with function(s) to apply over each region as values and the name of the respective statistics as keys. Default is to just count the number of cells associated with each feature and write it to the feature dataframe.
- **index** (*None | list[int], optional (default: None)*) – list of indices of regions in labels to apply function to. If None, will default to all integer feature labels in labels.
- **default** (*None | float, optional (default: None)*) – default value to return in a region that has no values.
- **id_column** (*str, optional (default: "feature")*) – Name of the column in feature dataframe that contains IDs that match with the labels in mask. The default is the column "feature".
- **collapse_axis** (*None | int | list[int], optional (default: None):*) – Index or indices of axes of labels to collapse. This will reduce the dimensionality of labels while allowing labelled features to overlap. This can be used, for example, to calculate the footprint area (2D) of 3D labels

Returns

features – Updated feature dataframe with bulk statistics for each feature saved in a new column.

Return type

pd.DataFrame

tobac.utils.bulk_statistics.**get_statistics_from_mask**(*features: pandas.DataFrame*, *segmentation_mask: xarray.DataArray*, **fields: xarray.DataArray*, *statistic: dict[str, tuple[~typing.Callable]] = {'Mean': <function mean>}*, *index: None | list[int] = None*, *default: None | float = None*, *id_column: str = 'feature'*, *collapse_dim: None | str | list[str] = None*) → pandas.DataFrame

Derives bulk statistics for each object in the segmentation mask, and returns a features Dataframe with these properties for each feature.

Parameters

- **features** (*pd.DataFrame*) – Dataframe with segmented features (output from feature detection or segmentation). Timesteps must not be exactly the same as in segmentation mask but all labels in the mask need to be present in the feature dataframe.
- **segmentation_mask** (*xr.DataArray*) – Segmentation mask output
- ***fields** (*xr.DataArray[np.ndarray]*) – Field(s) with input data. If field does not have a time dimension it will be considered time invariant, and the entire field will be passed for each time step in segmentation_mask. If the shape does not match that of labels, numpy-style broadcasting will be applied.
- **statistic** (*dict[str, Callable], optional (default: {'ncells':np.count_nonzero})*) – Dictionary with function(s) to apply over each region as values and the name of the respective statistics as keys. Default is to calculate the mean value of the field over each feature.
- **index** (*None | list[int], optional (default: None)*) – list of indexes of regions in labels to apply function to. If None, will default to all integers between 1 and the maximum value in labels
- **default** (*None | float, optional (default: None)*) – default value to return in a region that has no values
- **id_column** (*str, optional (default: "feature")*) – Name of the column in feature dataframe that contains IDs that match with the labels in mask. The default is the column "feature".
- **collapse_dim** (*None | str | list[str], optional (defailt: None)*) –

  Dimension names of labels to collapse, allowing, e.g. calulcation of statistics on 2D
  fields for the footprint of 3D objects

  features: pd.DataFrame
  Updated feature dataframe with bulk statistics for each feature saved in a new column

21.15 tobac.utils.decorators module

Decorators for use with other tobac functions

tobac.utils.decorators.**convert_cube_to_dataarray**(*cube*)

Convert an iris cube to an xarray dataarray, averting error for integer dtype cubes in xarray<v2023.06

Parameters
cube (*iris.cube.Cube*) – Iris data cube

Returns
dataarray – dataarray converted from cube. If the cube's core data is a masked array and has integer dtype, the returned datarray will have a numpy array with masked values filled with the minimum value for that integer dtype. Otherwise the data will be identical to that produced using xr.DataArray.from_iris

Return type
xr.DataArray

tobac.utils.decorators.**iris_to_xarray**(*save_iris_info: bool = False*)

tobac.utils.decorators.**irispandas_to_xarray**(*save_iris_info: bool = False*)

tobac.utils.decorators.**njit_if_available**(*func*, ***kwargs*)

Decorator to wrap a function with numba.njit if available. If numba isn't available, it just returns the function.

Parameters

- **func** (*function object*) – Function to wrap with njit
- **kwargs** – Keyword arguments to pass to numba njit

tobac.utils.decorators.**xarray_to_iris**()

tobac.utils.decorators.**xarray_to_irispandas**()

21.16 tobac.utils.general module

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_coord=None*, *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 conaining 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*, *average_method='arithmetic'*)

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.
- **average_method** (*string, optional*) – Defines how spacings in x- and y-direction are combined.
  - 'arithmetic' : standard arithmetic mean like (dx+dy)/2
  - 'geometric' : geometric mean; conserves gridbox area

Default is 'arithmetic'.

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).

21.16.1 Parameters:

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.

21.16.2 Returns:

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.

tobac.utils.general.**standardize_track_dataset**(*TrackedFeatures*, *Mask*, *Projection=None*)

CAUTION: this function is experimental. No data structures output are guaranteed to be supported in future versions of tobac. Combine a feature mask with the feature data table into a common dataset. returned by tobac.segmentation with the TrackedFeatures dataset returned by tobac.linking_trackpy. Also rename the variables to be more descriptive and comply with cf-tree. Convert the default cell parent ID to an integer table. Add a cell dimension to reflect Projection is an xarray DataArray TODO: Add metadata attributes :param TrackedFeatures: xarray dataset of tobac Track information, the xarray dataset returned by tobac.tracking.linking_trackpy :type TrackedFeatures: xarray.core.dataset.Dataset :param Mask: xarray dataset of tobac segmentation mask information, the xarray dataset returned

by tobac.segmentation.segmentation

Parameters

Projection (*xarray.core.dataarray.DataArray, default = None*) – array.DataArray of the original input dataset (gridded nexrad data for example). If using gridded nexrad data, this can be input as: data['ProjectionCoordinateSystem'] An example of the type of information in the dataarray includes the following attributes: latitude_of_projection_origin :29.471900939941406 longitude_of_projection_origin :-95.0787353515625 _CoordinateTransformType :Projection _CoordinateAxes :x y z time _CoordinateAxesTypes :GeoX GeoY Height Time grid_mapping_name :azimuthal_equidistant semi_major_axis :6370997.0 inverse_flattening :298.25 longitude_of_prime_meridian :0.0 false_easting :0.0 false_northing :0.0

Returns

ds – xarray dataset of merged Track and Segmentation Mask datasets with renamed variables.

Return type

xarray.core.dataset.Dataset

tobac.utils.general.**transform_feature_points**(*features*, *new_dataset*, *latitude_name=None*, *longitude_name=None*, *altitude_name=None*, *max_time_away=None*, *max_space_away=None*, *max_vspace_away=None*, *warn_dropped_features=True*)

Function to transform input feature dataset horizontal grid points to a different grid. The typical use case for this function is to transform detected features to perform segmentation on a different grid.

The existing feature dataset must have some latitude/longitude coordinates associated with each feature, and the new_dataset must have latitude/longitude available with the same name. Note that due to xarray/iris incompatibilities, we suggest that the input coordinates match the standard_name from Iris.

Parameters

- **features** (*pd.DataFrame*) – Input feature dataframe
- **new_dataset** (*iris.cube.Cube or xarray*) – The dataset to transform the
- **latitude_name** (*str*) – The name of the latitude coordinate. If None, tries to auto-detect.
- **longitude_name** (*str*) – The name of the longitude coordinate. If None, tries to auto-detect.
- **altitude_name** (*str*) – The name of the altitude coordinate. If None, tries to auto-detect.
- **max_time_away** (*datetime.timedelta*) – The maximum time delta to associate feature points away from.
- **max_space_away** (*float*) – The maximum horizontal distance (in meters) to transform features to.
- **max_vspace_away** (*float*) – The maximum vertical distance (in meters) to transform features to.
- **warn_dropped_features** (*bool*) – Whether or not to print a warning message if one of the max_* options is going to result in features that are dropped.

Returns

transformed_features – A new feature dataframe, with the coordinates transformed to the new grid, suitable for use in segmentation

Return type

pd.DataFrame

21.17 tobac.utils.mask module

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 tacked 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 tacked 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 tacked 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 tacked 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 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.

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

21.18 tobac.utils.periodic_boundaries module

tobac.utils.periodic_boundaries.**adjust_pbc_point**(*in_dim: int*, *dim_min: int*, *dim_max: int*) → int

Function to adjust a point to the other boundary for PBCs

Parameters

- **in_dim** (*int*) – Input coordinate to adjust
- **dim_min** (*int*) – Minimum point for the dimension
- **dim_max** (*int*) – Maximum point for the dimension (inclusive)

Returns
The adjusted point on the opposite boundary

Return type
int

Raises
ValueError – If in_dim isn't on one of the boundary points

tobac.utils.periodic_boundaries.**calc_distance_coords_pbc**(*coords_1*, *coords_2*, *min_h1*, *max_h1*, *min_h2*, *max_h2*, *PBC_flag*)

Function to calculate the distance between cartesian coordinate set 1 and coordinate set 2. Note that we assume both coordinates are within their min/max already.

Parameters

- **coords_1** (*2D or 3D array-like*) – Set of coordinates passed in from trackpy of either (vdim, hdim_1, hdim_2) coordinates or (hdim_1, hdim_2) coordinates.

- **coords_2** (*2D or 3D array-like*) – Similar to coords_1, but for the second pair of coordinates
- **min_h1** (*int*) – Minimum point in hdim_1
- **max_h1** (*int*) – Maximum point in hdim_1, exclusive. max_h1-min_h1 should be the size.
- **min_h2** (*int*) – Minimum point in hdim_2
- **max_h2** (*int*) – Maximum point in hdim_2, exclusive. max_h2-min_h2 should be the size.
- **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
Distance between coords_1 and coords_2 in cartesian space.

Return type
float

tobac.utils.periodic_boundaries.**get_pbc_coordinates**(*h1_min: int*, *h1_max: int*, *h2_min: int*, *h2_max: int*, *h1_start_coord: int*, *h1_end_coord: int*, *h2_start_coord: int*, *h2_end_coord: int*, *PBC_flag: str = 'none'*) → list[tuple[int, int, int, int]]

Function to get the real (i.e., shifted away from periodic boundaries) coordinate boxes of interest given a set of coordinates that may cross periodic boundaries. This computes, for example, multiple bounding boxes to encompass the real coordinates when given periodic coordinates that loop around to the other boundary.

For example, if you pass in [as h1_start_coord, h1_end_coord, h2_start_coord, h2_end_coord] (-3, 5, 2,6) with PBC_flag of 'both' or 'hdim_1', h1_max of 10, and h1_min of 0 this function will return: [(0,5,2,6), (7,10,2,6)].

If you pass in something outside the bounds of the array, this will truncate your requested box. For example, if you pass in [as h1_start_coord, h1_end_coord, h2_start_coord, h2_end_coord] (-3, 5, 2,6) with PBC_flag of 'none' or 'hdim_2', this function will return: [(0,5,2,6)], assuming h1_min is 0.

Parameters

- **h1_min** (*int*) – Minimum array value in hdim_1, typically 0.
- **h1_max** (*int*) – Maximum array value in hdim_1 (exclusive). h1_max - h1_min should be the size in h1.
- **h2_min** (*int*) – Minimum array value in hdim_2, typically 0.
- **h2_max** (*int*) – Maximum array value in hdim_2 (exclusive). h2_max - h2_min should be the size in h2.
- **h1_start_coord** (*int*) – Start coordinate in hdim_1. Can be < h1_min if dealing with PBCs.
- **h1_end_coord** (*int*) – End coordinate in hdim_1. Can be >= h1_max if dealing with PBCs.
- **h2_start_coord** (*int*) – Start coordinate in hdim_2. Can be < h2_min if dealing with PBCs.
- **h2_end_coord** (*int*) – End coordinate in hdim_2. Can be >= h2_max if dealing with PBCs.
- **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 list of tuples containing (h1_start, h1_end, h2_start, h2_end) of each of the boxes needed to encompass the coordinates.

Return type
list of tuples

tobac.utils.periodic_boundaries.**transfm_pbc_point**(*in_dim*, *dim_min*, *dim_max*)

Function to transform a PBC-feature point for contiguity

Parameters

- **in_dim** (*int*) – Input coordinate to adjust
- **dim_min** (*int*) – Minimum point for the dimension
- **dim_max** (*int*) – Maximum point for the dimension (inclusive)

Returns
The transformed point

Return type
int

tobac.utils.periodic_boundaries.**weighted_circmean**(*values: np.ndarray*, *weights: np.ndarray*, *high: float = 6.283185307179586*, *low: float = 0*, *axis: int | None = None*) → np.ndarray

Calculate the weighted circular mean over a set of values. If all the weights are equal, this function is equivalent to scipy.stats.circmean

Parameters

- **values** (*array-like*) – Array of values to calculate the mean over
- **weights** (*array-like*) – Array of weights corresponding to each value
- **high** (*float, optional*) – Upper bound of the range of values. Defaults to 2*pi
- **low** (*float, optional*) – Lower bound of the range of values. Defaults to 0
- **axis** (*int | None, optional*) – Axis over which to take the average. If None, the average will be taken over the entire array. Defaults to None

Returns
rescaled_average – The weighted, circular mean over the given values

Return type
numpy.ndarray

21.19 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*)

21.20 Module contents

PYTHON MODULE INDEX

t

INDEX

H

I

L

M

N

P

R

S

T

V

W

X