IPART

Release 2.0

Nov 06, 2020

The automated AR detect/tracking workflow:

Table of Contents

CHAPTER 1

Introduction

IPART (Image-Processing based Atmospheric River Tracking) is a Python package for automated Atmospheric River (AR) detection, axis finding and AR tracking from gridded Integrated Vapor Transport (IVT) data, for instance Reanalysis datasets.

An overview of what ARs are can be found in this review paper: Atmospheric rivers: a mini-review.

IPART is intended for researchers and students who are interested in the field of atmospheric river studies in the present day climate or future projections. Different from commonly used AR detection methods that rely on thresholding on the IVT magnitude, this package includes a method inspired by an image processing technique – Top-hap by reconstruction (THR).

Below is an example output figure:

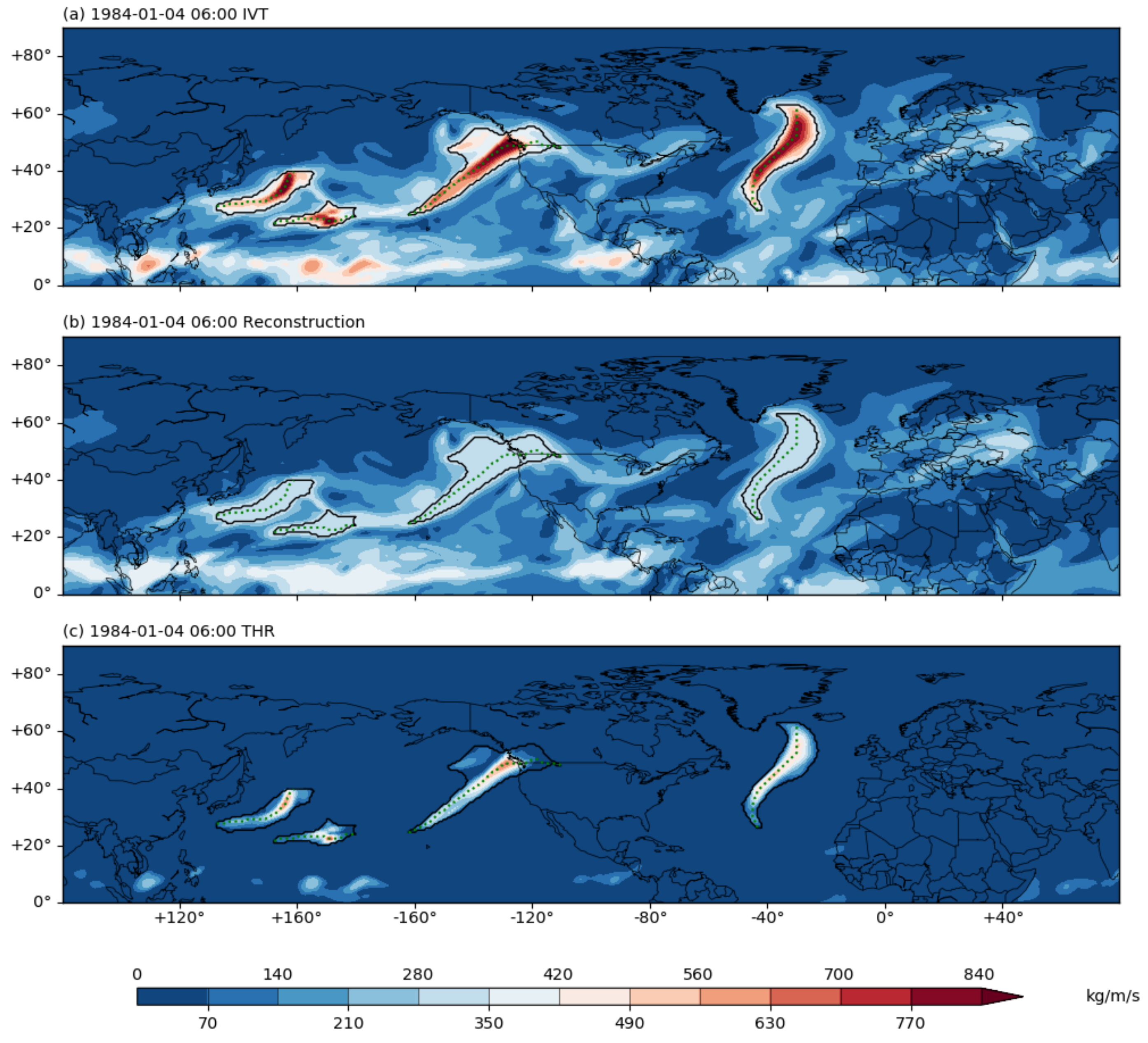


Fig. 1.1: (a) The IVT field in kg/(m*s) at 1984-01-04 06:00 UTC over the North Hemisphere. (b) the IVT reconstruction field at the same time point. (c) the IVT anomaly field from the THR process at the same time point. In all three subplots, the detected ARs are outlined in black contour. The AR axes are drawn in green dashed lines.

CHAPTER 2

Installation

2.1 Install from conda-forge

ipart can be installed in an existing conda environment:

```
conda install -c conda-forge ipart
```

will install ipart and its dependencies for Python 3.

2.2 Create a conda environment using the environment file

Alternatively, users can obtain the code of this pacakge from the github page, and create a new conda environment using the environment files provided. This way will install the optional cartopy package and allow you to run the notebook examples.

```
git clone https://github.com/ihesp/IPART
cd IPART
conda env create -f environment_py3.yml
```

This creates a new environment named ipartpy3. Activate the environment using:

```
conda activate ipartpy3
```

After that, you can check the list of packages installed by:

```
conda list
```

Similarly for Python 2, use:

```
conda env create -f environment_py2.yml
```

CHAPTER 3

Dependencies

- OS: Linux or MacOS. Windows is not tested.
- Python2.7 or Python3.7.
- netCDF4 (tested 1.4.2, 1.5.3 in py2, tested 1.5.3 in py3)
- numpy (developed in 1.16.5 in py2, tested 1.18.1, 1.19.0 in py3)
- scipy (developed in 1.2.1 in py2, tested 1.4.1, 1.5.1 in py3)
- matplotlib (tested 2.2.5 in py2, tested 3.3.1 in py3)
- pandas (developed in 0.23.4, 0.24.2 in py2, tested 1.0.3, 1.0.5 in py3)
- networkx (developed in 1.11 and 2.2 in py2, tested 2.4 in py3)
- scikit-image (developed in 0.14.2, 0.14.3 in py2, tested 0.16.2, 0.17.2 in py3)
- cartopy (optional, only used for plotting. Tested 0.17.0 in py2, tested 1.18.0 in py3)

CHAPTER 4

Main functionalities

There are four main functionalities provided by the package that collectively constitute a specific workflow for the automated AR detection/tracking task:

1. Perform THR computation on input data.
2. Detect ARs from the outputs from the previous step, and at the same time,
3. Identify the AR axis.
4. Track ARs detected at individual time steps to form tracks.

More details regarding these steps are provided in separate pages below.

Applications on example data can be found in a series of example notebooks at github repository.

4.1 Data preparation

4.1.1 netcdf data

Source data are the u- and v- components of the vertically integrated vapor fluxes, in a rectangular grid.

The u-component of the vertically integrated vapor fluxes have a *standard_name* of eastward_atmosphere_water_transport_across_unit_distance.

The v-component of the vertically integrated vapor fluxes have a *standard_name* of northward_atmosphere_water_transport_across_unit_distance.

These are usually computed as:

$$\begin{cases} F_u & = \int_{TOA}^{P_s} \frac{uq}{g} dP \\ F_v & = \int_{TOA}^{P_s} \frac{vq}{g} dP \end{cases}$$

where

- F_u (F_v) is the zonal (meridional) component of the integrated flux, both in $kg/(m * s)$.

- u (v): is the zonal (meridional) wind speed (in m/s) at a given level, and q is the specific humidity (in *kg/kg*) at the same level.
- dP is the pressure increment, in Pa, and g is acceleration by gravity.
- TOA can be substituted with a sensibly high level, e.g. $300hPa$.

No strict requirement for the spatial or temporal resolution of input data is imposed, however, for better results, one should use something greater than $\sim 1.5°$ latitude/longitude. 6-hourly temporal resolution is a standard for Reanalysis datasets, daily would probably work, but one should do some parameter adjustments in such a case.

Data are supposed to saved in netcdf files.

4.1.2 Metadata

Note: the user is responsible for making sure that the data are saved in the following rank order:

```
(time, level, latitude, longitude)
```

or:

```
(time, latitude, longitude)
```

The `level` dimension is optional. As data are vertical integrals, the length of the level dimension, if present, should be 1.

The user also needs to provide the time, latitude and longitude axes values. This is because these temporal and geographical information is used in the computation.

To test the sanity of your input data, run this script against the netcdf data file:

```
cd /path/to/IPART/folder/you/cloned
python test_data.py /path/to/your/netcdf/file 'id_of_variable'
```

For instance

```
cd ~/Downloads/IPART
python test_data.py ~/datasets/erai/uflux_file.nc 'uflux'
```

The expected outputs would look like this:

```
### <test_data>: Read in file:
 /home/guangzhi/scripts/IPART/notebooks/uflux_s_6_1984_Jan.nc
##################################################
Variable time axis:
##################################################
### Description of slab ###
  id: time
  shape: (124,)
  filename: None
  missing_value: None
  comments: None
  grid_name: None
  grid_type: None
  long_name: time
  units: hours since 1900-01-01 00:00:00.0
```

(continues on next page)

(continued from previous page)

```
  standard_name: None
  Order: []

### End of description ###

None
Time axis values:
[datetime.datetime(1984, 1, 1, 0, 0) datetime.datetime(1984, 1, 1, 6, 0)
 datetime.datetime(1984, 1, 1, 12, 0) datetime.datetime(1984, 1, 1, 18, 0)
 datetime.datetime(1984, 1, 2, 0, 0) datetime.datetime(1984, 1, 2, 6, 0)
 datetime.datetime(1984, 1, 2, 12, 0) datetime.datetime(1984, 1, 2, 18, 0)
 datetime.datetime(1984, 1, 29, 0, 0) datetime.datetime(1984, 1, 29, 6, 0)
 ...
 datetime.datetime(1984, 1, 31, 12, 0)
 datetime.datetime(1984, 1, 31, 18, 0)]

##################################################
Variable latitude axis:
##################################################
### Description of slab ###
  id: latitude
  shape: (94,)
  filename: None
  missing_value: None
  comments: None
  grid_name: None
  grid_type: None
  long_name: latitude
  units: degrees_north
  standard_name: None
  Order: []

### End of description ###

None
Latitude axis values:
[10.   10.75 11.5  12.25 13.   13.75 14.5  15.25 16.   16.75 17.5  18.25
 19.   19.75 20.5  21.25 22.   22.75 23.5  24.25 25.   25.75 26.5  27.25
 28.   28.75 29.5  30.25 31.   31.75 32.5  33.25 34.   34.75 35.5  36.25
 37.   37.75 38.5  39.25 40.   40.75 41.5  42.25 43.   43.75 44.5  45.25
 46.   46.75 47.5  48.25 49.   49.75 50.5  51.25 52.   52.75 53.5  54.25
 55.   55.75 56.5  57.25 58.   58.75 59.5  60.25 61.   61.75 62.5  63.25
 64.   64.75 65.5  66.25 67.   67.75 68.5  69.25 70.   70.75 71.5  72.25
 73.   73.75 74.5  75.25 76.   76.75 77.5  78.25 79.   79.75]

##################################################
Variable longitude axis:
##################################################
### Description of slab ###
  id: longitude
  shape: (480,)
  filename: None
  missing_value: None
  comments: None
  grid_name: None
  grid_type: None
  long_name: longitude
```

(continues on next page)

(continued from previous page)

```
  units: degrees_east
  standard_name: None
  Order: []

### End of description ###

None
Longitude axis values:
[-180.   -179.25 -178.5  -177.75 -177.   -176.25 -175.5  -174.75 -174.
 -173.25 -172.5  -171.75 -171.   -170.25 -169.5  -168.75 -168.   -167.25
 -166.5  -165.75 -165.   -164.25 -163.5  -162.75 -162.   -161.25 -160.5
 ...
  164.25  165.    165.75  166.5   167.25  168.    168.75  169.5   170.25
  171.    171.75  172.5   173.25  174.    174.75  175.5   176.25  177.
  177.75  178.5   179.25]

Data have unit of "kg m**-1 s**-1"
```

Pay some attention to the values listed in the **latitude** and **longitude** axes blocks, to make sure the values make physical sense. For high resolution data, the input variable may have a fairly large size, e.g. a longitude axis of length 720 (if your data have a resolution of $0.5 \times 0.5°$). If the longitude axis reports a largest value of 720, it is probably reporting the size of the longitude dimension, rather than the actual longitude label (as the maximum possible longitude label should be 360). In such cases, the user should take some extra steps to make sure that the data have proper metadata associated with them.

4.1.3 Compute IVT

With F_u and F_v, compute the IVT as

$$IVT = \sqrt{F_u^2 + F_v^2}$$

This is trivial to achieve, you can use the `compute_ivt.py` script provided in the package for this computation.

4.2 Perform the THR computation on IVT data

4.2.1 The Top-hat by Reconstruction (THR) algorithm

The AR detection method is inspired by the image processing technique **top-hat by reconstruction (THR)**, which consists of subtracting from the original image a **greyscale reconstruction by dilation** image. Some more details of the THR algorithm and its applications can be found in this work of [Vincent1993].

In the context of AR detection, the greyscale image in question is the non-negative IVT distribution, denoted as I.

The greyscale reconstruction by dilation component (hereafter reconstruction) corresponds to the background IVT component, denoted as $\delta(I)$.

The difference between I and $\delta(I)$ gives the transient IVT component, from which AR candidates are searched.

Note: we made a modification based on the THR algorithm as descripted in [Vincent1993]. The **marker** image used in this package is obtained by a grey scale erosion[1] with a structuring element E, while in a standard THR process

[1] Greyscale erosion (also known as minimum filtering) can be understood by analogy with a moving average. Instead of the average within a neighborhood, erosion replaces the central value with the neighborhood minimum. Similarly, dilation replaces with the maximum. And the neighborhood is defined by the structuring element E.

as in [Vincent1993], the **marker** image is obtained by a global substraction $I - \delta h$, where δh is the pixel intensity subtracted globally from the original input image I.

The introduction of the grey scale erosion process allows us to have a control over on what spatio-temporal scales ARs are to be detected. An important parameter in this erosion process is the size of the structuring element E.

We then extend the processes of erosion and reconstruction to 3D (i.e. time, x- and y- dimensions), measuring "spatio-temporal spikiness". The structuring element used for 3D erosion is a 3D ellipsoid:

$$E = \left\{(z, x, y) \in \mathbb{Z}^3 \mid (z/t)^2 + (x/s)^2 + (y/s)^2 \leq 1\right\}$$

with the axis length along the time dimension being t, and the axes for the x- and y- dimensions sharing the same length s. Both t and s are measured in pixels/grids.

Note: the axis length of an ellipsoid is *half* the size of the ellipsoid in that dimension. For relatively large sized E, the difference in the THR results using an ellipsoid structuring element and a 3D cube with size $(2t + 1, 2s + 1, 2s + 1)$ is fairly small.

Considering the close physical correspondences between ARs and extra-tropical storm systems, the "correct" THR parameter choices of t and s should be centered around the spatio-temporal scale of ARs.

Let's assume the data we are working with is 6-hourly in time, and $0.75 \times 0.75°$ in space.

The typical synoptic time scale is about a week, giving $t = 4\,days$ (recall that t is only *half* the size of the time dimension).

The typical width of ARs is within $1000\,km$, therefore $s = 6\,grids$ is chosen. Given the $0.75\,°$ resolution of data, this corresponds to a distance of about $80km/grid \times (6 \times 2 + 1)grids = 1040\,km$. An extra grid is added to ensure an odd numbered grid length, same for the t parameter: the number of time steps is $4\,steps/day \times 4days \times 2 + 1\,step = 33\,steps$.

4.2.2 Compute THR

Using the above setup, the THR process is computed using following code:

```
from ipart import thr
ivt, ivtrec, ivtano = thr.THR(ivt_input, [16, 6, 6])
```

where `ivt_input` is the input IVT data, `ivtrec` is the reconstruction component, and `ivtano` is the anomalous component.

Note: the *thr.THR()* function accepts an optional argument *oro*, which is to provide the algorithm with some surface elevation information, with the help of which detection sensitivity of landfalling ARs can be enhanced.

See also:

`thr.THR()`.

4.2.3 Dedicated Python script

The package provides two script to help doing this computation:

- `compute_thr_singlefile`: when your IVT data are saved in a single file.

- `compute_thr_multifile`: when your IVT data are too large to fit in a single file, e.g. data spanning multiple decades and saved into one-file-per year. Like in the case of a simple moving average, discontinuity at the end of one year and the beginning of the next may introduce some errors. When the data are too large to fit into RAM, one possible solution is to read in 2 years at a time, concatenate them then perform the filtering/THR process to achieve a smooth year-to-year transition. Then read in the 3rd year to form another 2-year concatenation with the 2nd year. Then the process rotates on untill all years are processed.

4.2.4 Example output

4.2.5 Notebook example

An example of this process is given in this notebook.

4.2.6 References

4.3 Detect AR appearances from THR output

4.3.1 Definition of AR occurrence

An AR occurrence at a given time point is defined using these following rules:

1. A connected region in the IVT anomaly field ($I - \delta(I)$, computed in the section "*The Top-hat by Reconstruction (THR) algorithm*") where its values is greater than 0.
2. The centroid (weighted by IVT values of the grid cells) of the region is north of $20°N$ (or south of $20°S$ for the Southern Hemisphere), and south of $80°N$ (or north of $80°S$ for the Southern Hemisphere), i.e. we are only interested in mid-latitude systems.
3. The region's area has to be within $50 - 1800 \times 10^4 km^2$.
4. After the computation of this AR candidates axis (see *Find the axis from detected AR*) and the effective width (defined as area/length ratio), the length has to be $\geq 1500km$, and length/width ratio has to be ≥ 2 if length is below $2000\,km$.

Note: API is provided to control all these parameters.

4.3.2 Input data

These are the input data required for AR occurrence detection:

- u- and v- components of the integrated vapor fluxes (F_u and F_v).
- IVT (as $\sqrt{F_u^2 + F_v^2}$), see *Compute IVT* for more.
- The output from the THR process: the reconstruction component ($\delta(I)$) and the anomaly component ($I - \delta(I)$). See *The Top-hat by Reconstruction (THR) algorithm* for more.

Additional inputs:

- latitude, longitude and time axis. See *Metadata* for more.
- detection parameters, see below.

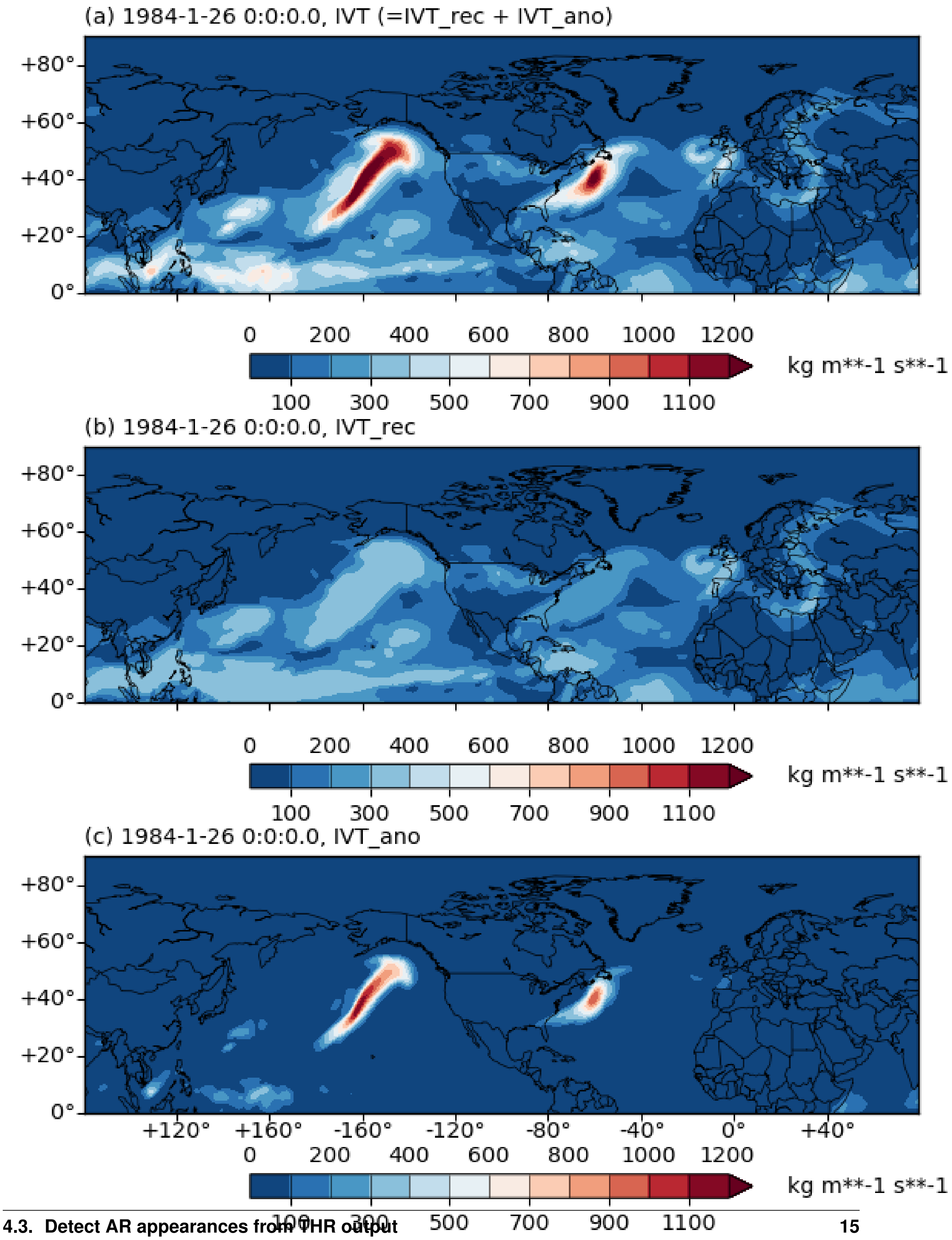


Fig. 4.1: (a) The IVT field in kg/(m*s) at 1984-01-26 00:00 UTC over the North Hemisphere. (b) the IVT reconstruction field ($\delta(I)$) at the same time point. (c) the IVT anomaly field ($I - \delta(I)$) from the THR process at the same time

```
PARAM_DICT={
    # kg/(m*s), define AR candidates as regions >= than this anomalous ivt.
    'thres_low' : 1,

    # km^2, drop AR candidates smaller than this area.
    'min_area': 50*1e4,

    # km^2, drop AR candidates larger than this area.
    'max_area': 1800*1e4,

    # float, min length/width ratio, applied only when length<min_length.
    'min_LW': 2,

    # degree, exclude systems whose centroids are lower than this latitude.
    'min_lat': 20,

    # degree, exclude systems whose centroids are higher than this latitude.
    'max_lat': 80,

    # km, ARs shorter than this length is treated as relaxed.
    'min_length': 2000,

    # km, ARs shorter than this length is discarded.
    'min_length_hard': 1500,

    # degree lat/lon, error when simplifying axis using rdp algorithm.
    'rdp_thres': 2,

    # grids. Remove small holes in AR contour.
    'fill_radius': None,

    # do peak partition or not, used to separate systems that are merged
    # together with an outer contour.
    'single_dome': False,

    # max prominence/height ratio of a local peak. Only used when SINGLE_DOME=True
    'max_ph_ratio': 0.6,

    # minimal proportion of flux component in a direction to total flux to
    # allow edge building in that direction
    'edge_eps': 0.4
    }
```

4.3.3 Usage in Python scripts

The following snippet shows the detection function calls:

```
from ipart.AR_detector import findARs
time_idx, labelsNV, anglesNV, crossfluxesNV, result_df = findARs(ivtNV.data,
    ivtrecNV.data, ivtanoNV.data, quNV.data, qvNV.data, latax, lonax,
    times=timeax, **PARAM_DICT)
```

where these input arguments are:

- `ivtNV` is a `ipart.utils.NCVAR` data object, which is a rudimentary wrapper object designed to achieve a tighter bound between data values and metadata. Same for the other variables with an `NV` suffix.

- `ivtNV.data` is the IVT data values in `numpy.ndarray` format, with dimensions of `(time, level, latitude, longitude)` or `(time, latitude, longitude)`.
- `ivtrecNV` is $\delta(I)$, and `ivtanoNV` is $I - \delta(I)$, see *The Top-hat by Reconstruction (THR) algorithm* for more details.
- `quNV`: is F_u, and `qvNV` is F_v.
- `latax`: is an 1d array storing the latitude coordinates of `ivtNV` and others.
- `lonax`: is an 1d array storing the longitude coordinates of `ivtNV` and others.
- `timeax` is a list of python `datetime` objects storing time stamps of the data in `ivtNV` and others.
- `PARAM_DICT` is the parameter dictionary as defined above.

The return values are:

- `time_idx` is a list of indices of the time dimension when any AR is found.
- `labelsNV` is a `ipart.utils.NCVAR` object, whose `data` attribute is an ndarray variable saving the numerical labels of all found ARs in each time step. It has shape of `(time, lat, lon)`.
- `anglesNV` is a `ipart.utils.NCVAR` object storing an ndarray variable saving the difference in the orientation of IVT vectors in all found ARs, wrt the AR axis.
- `crossfluxesNV` is a `ipart.utils.NCVAR` object storing an ndarray variable saving the cross-sectional IVT flux, computed as the projection of IVT vectors onto the AR axis, using angles in angles.
- The `result_df` return value is a `pandas.DataFrame` object saving in a table the various attributes of all detected ARs at this time point.

See also:

`AR_detector.findARs()`, *`AR_detector.findARsGen()`*, *`AR_detector.getARData()`*.

AR records DataFrame

The rows of `ardf` are different AR records, the columns of `ardf` are listed below:

- `id` : integer numeric id for this AR at this particular time point. ARs at different time points can share the same id, and an AR can be uniquely identified with the combination of time stamp + id.
- `time` : time stamp in the YYYY-MM-DD HH:mm:ss format.
- `contour_y` : list of floats, the y-coordinates (latitudes) of the AR contour in degrees North.
- `contour_x` : list of floats, the x-coordinates (longitude) of the AR contour in degrees North.
- `centroid_y` : float, latitude of the AR centroid, weighted by the IVT value.
- `centroid_x` : float, longitude of the AR centroid, weighted by the IVT value.
- `axis_y` : list of floats, latitudes of the AR axis.
- `axis_x` : list of floats, longitude of the AR axis.
- `axis_rdp_y` : list of floats, latitude of the simplified AR axis.
- `axis_rdp_x` : list of floats, longitude of the simplified AR axis.
- `area` : float, area of the AR in km^2.
- `length` : float, length of the AR in km.
- `width` : float, effective width in km, as area/length.

- `LW_ratio` : float, length/width ratio.
- `strength` : float, spatially averaged IVT value within the AR region, in $kg/(m \cdot s)$.
- `strength_ano` : float, spatially averaged anomalous IVT value within the AR region, in $kg/(m \cdot s)$.
- `strength_std` : float, standard deviation of IVT within the AR region, in $kg/(m \cdot s)$.
- `max_strength` : float, maximum IVT value within the AR region, in $kg/(m \cdot s)$.
- `mean_angle` : float, spatially averaged angle between the IVT vector and the AR axis, in degrees.
- `is_relaxed` : True or False, whether the AR is flagged as "relaxed".
- `qv_mean` : float, spatially averaged meridional integrated vapor flux, in $kg/(m \cdot s)$.

4.3.4 Dedicated Python script

You can use the `scripts/detect_ARs.py` or `scripts/detect_ARs_generator_version.py` script (check them out in the github repo). for AR detection process in production. The former does the computation and returns all outpus in one go, and the latter yields results at each time step separately, so the outputs can be saved to disk rather than accumulating in RAM. Note that this process is essentially time-independent, i.e. the computation of one time point does not rely on another, therefore you can potentially parallelize this process to achieve greater efficiency.

4.3.5 Example output

The resultant detected ARs can be visualized using the following snippet:

```python
import matplotlib.pyplot as plt
from ipart.utils import plot
import cartopy.crs as ccrs

plot_vars=[slab,slabrec,slabano]
titles=['IVT', 'Reconstruction', 'THR']
iso=plot.Isofill(plot_vars,12,1,1,min_level=0,max_level=800)

figure=plt.figure(figsize=(12,10),dpi=100)

for jj in range(len(plot_vars)):
    ax=figure.add_subplot(3,1,jj+1,projection=ccrs.PlateCarree())
    pobj=plot.plot2(plot_vars[jj],iso,ax,
            xarray=lonax, yarray=latax,
            title='%s %s' %(timett_str, titles[jj]),
            fix_aspect=False)

plot.plotAR(ardf,ax,lonax)
figure.show()
```

See also:

`utils.plot.Isofill`, *`utils.plot.plot2()`*.

One example output figure is shown below:

4.3.6 Notebook example

An example of this process is given in this notebook.

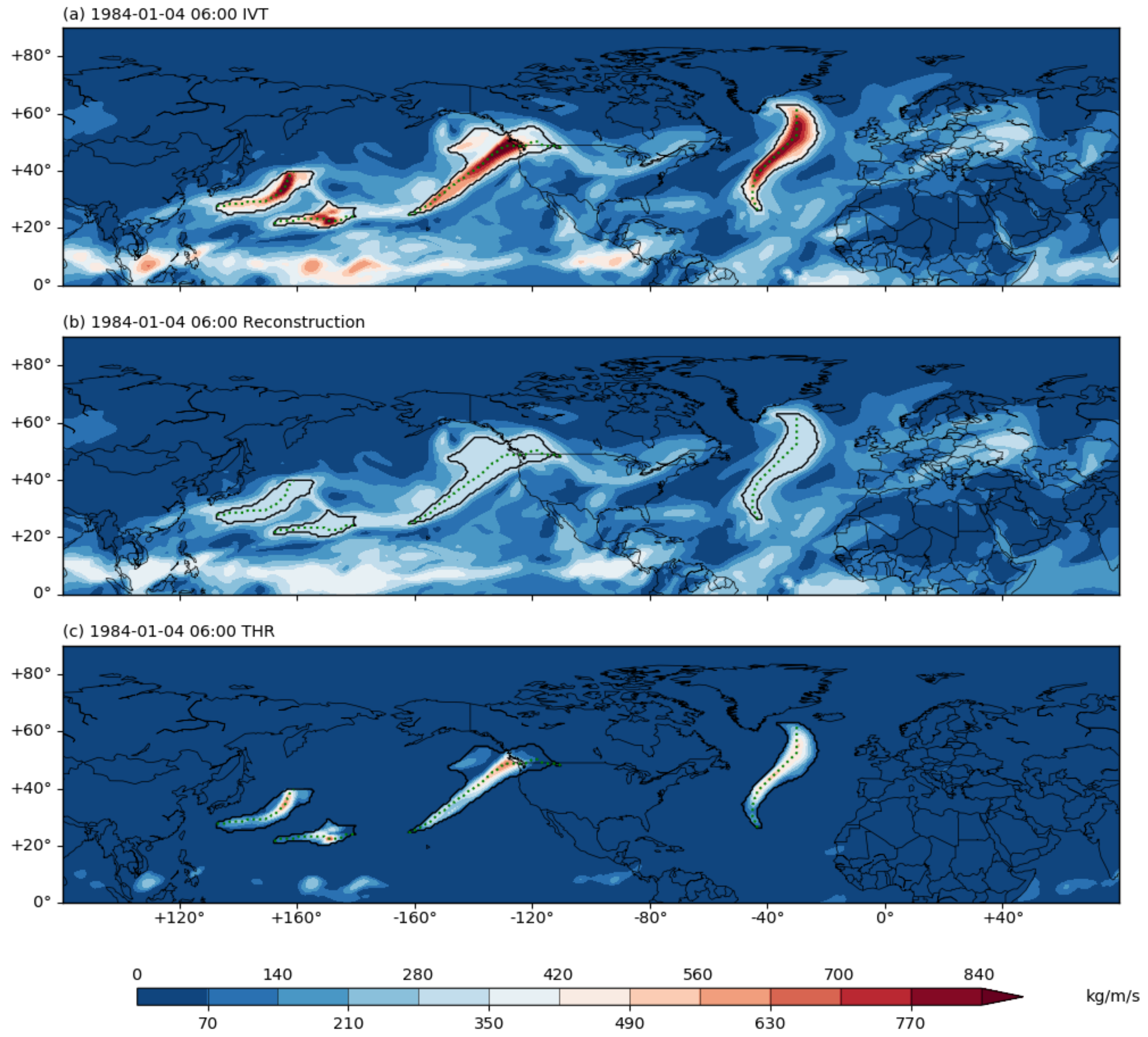


Fig. 4.2: (a) The IVT field in kg/(m*s) at 1984-01-04 06:00 UTC over the North Hemisphere. (b) the IVT reconstruction field at the same time point. (c) the IVT anomaly field from the THR process at the same time point. In all three subplots, the detected ARs are outlined in black contour. The AR axes are drawn in green dashed lines.

4.4 Find the axis from detected AR

4.4.1 Axis-finding in a planar graph framework

After *Detect AR appearances from THR output*, a binary mask I_k representing the spatial region of each candidate AR is obtained. An axis is sought from this region that summarizes the shape and orientation of the AR. A solution in a planar graph framework is proposed here.

A directed planar graph is build using the coordinate pairs (λ_k, ϕ_k) as nodes (see Figure Fig. 4.3 below). At each node, directed edges to its eight neighbors are created, so long as the moisture flux component along the direction of the edge exceeds a user-defined fraction (ϵ, see the `PARAM_DICT` in *Input data*) to the total flux. The along-edge flux is defined as:

$f_{ij} = u_i \sin(\alpha) + v_i \cos(\alpha)(4.1)$

where f_{ij} is the flux along the edge e_{ij} that points from node n_i to node n_j, and α is the azimuth angle of e_{ij}.

Therefore an edge can be created if $f_{ij}/\sqrt{u_i^2 + v_i^2} \geq \epsilon$. It is advised to use a relatively small $\epsilon = 0.4$ is used, as the orientation of an AR can deviate considerably from its moisture fluxes, and denser edges in the graph allows the axis to capture the full extent of the AR.

The boundary pixels of the AR region are then found, labeled L_k. The trans-boundary moisture fluxes are compute as the dot product of the gradients of I_k and (u_k, v_k): $\nabla I_k \cdot (u_k, v_k)$.

Then the boundary pixels with net input moisture fluxes can be defined as:

$$L_{k,in} = \{p \in L_k \mid (\nabla I_k \cdot (u_k, v_k))(p) > 0\}$$

Similarly, boundary pixels with net output moisture fluxes is the set

$$L_{k,out} = \{p \in L_k \mid (\nabla I_k \cdot (u_k, v_k))(p) < 0\}$$

These boundary pixels are colored in green and black, respectively, in Fig. 4.4.

For each pair of boundary nodes $\{(n_i, n_j) \mid n_i \in L_{k,in},\, n_j \in L_{k,out}\}$, a simple path (a path with no repeated nodes) is sought that, among all possible paths that connect the entry node n_i and the exit node n_j, is the **shortest** in the sense that its path-integral of weights is the lowest.

The weight for edge e_{ij} is defined as:

$$w_{ij} = e^{-f_{ij}/A}$$

where $f_{i,j}$ is the projected moisture flux along edge $e_{i,j}$ and $A = 100\, kg/(m \cdot s)$ is a scaling factor.

This formulation ensures a non-negative weight for each edge, and penalizes the inclusion of weak edges when a weighted shortest path search is performed.

The Dijkstra path-finding algorithm is used to find this shortest path p_{ij}^*.

Then among all p_{ij}^* that connect all entry-exit pairs, the one with the largest path-integral of along-edge fluxes is chosen as the AR axis, as highlighted in yellow in Fig. 4.4.

It could be seen that various aspects of the physical processes of ARs are encoded. The shortest path design gives a natural looking axis that is free from discontinuities and redundant curvatures, and never shoots out of the AR boundary. The weight formulation assigns smaller weights to edges with larger moisture fluxes, "urging: the shortest path to pass through nodes with greater intensity. The found axis is by design directed, which in certain applications can provide the necessary information to orient the AR with respect to its ambiance, such as the horizontal temperature gradient, which relates to the low level jet by the thermal wind relation.

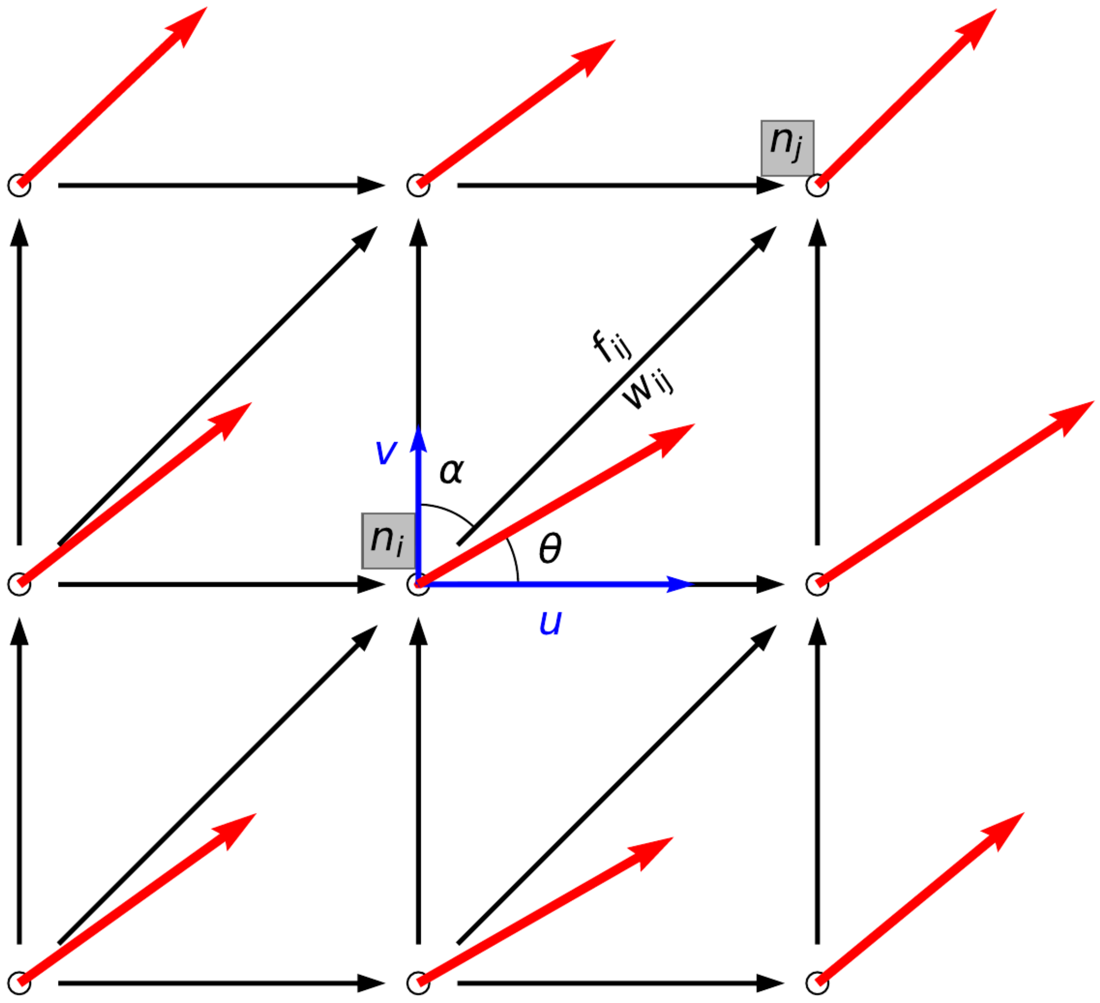


Fig. 4.3: Schematic diagram illustrating the planar graph build from the AR pixels and horizontal moisture fluxes. Nodes are taken from pixels within region I_k, and are represented as circles. Red vectors denote IVT vectors. The one at node n_i forms an angle θ with the x-axis, and has components (u, v). Black arrows denote directed edges between nodes, using an 8-connectivity neighborhood scheme. The edge between node n_i and n_j is e_{ij}, and forms an azimuth angle α with the y-axis. w_{ij} is the weight attribute assigned to edge e_{ij}, and f_{ij} is along-edge moisture flux.

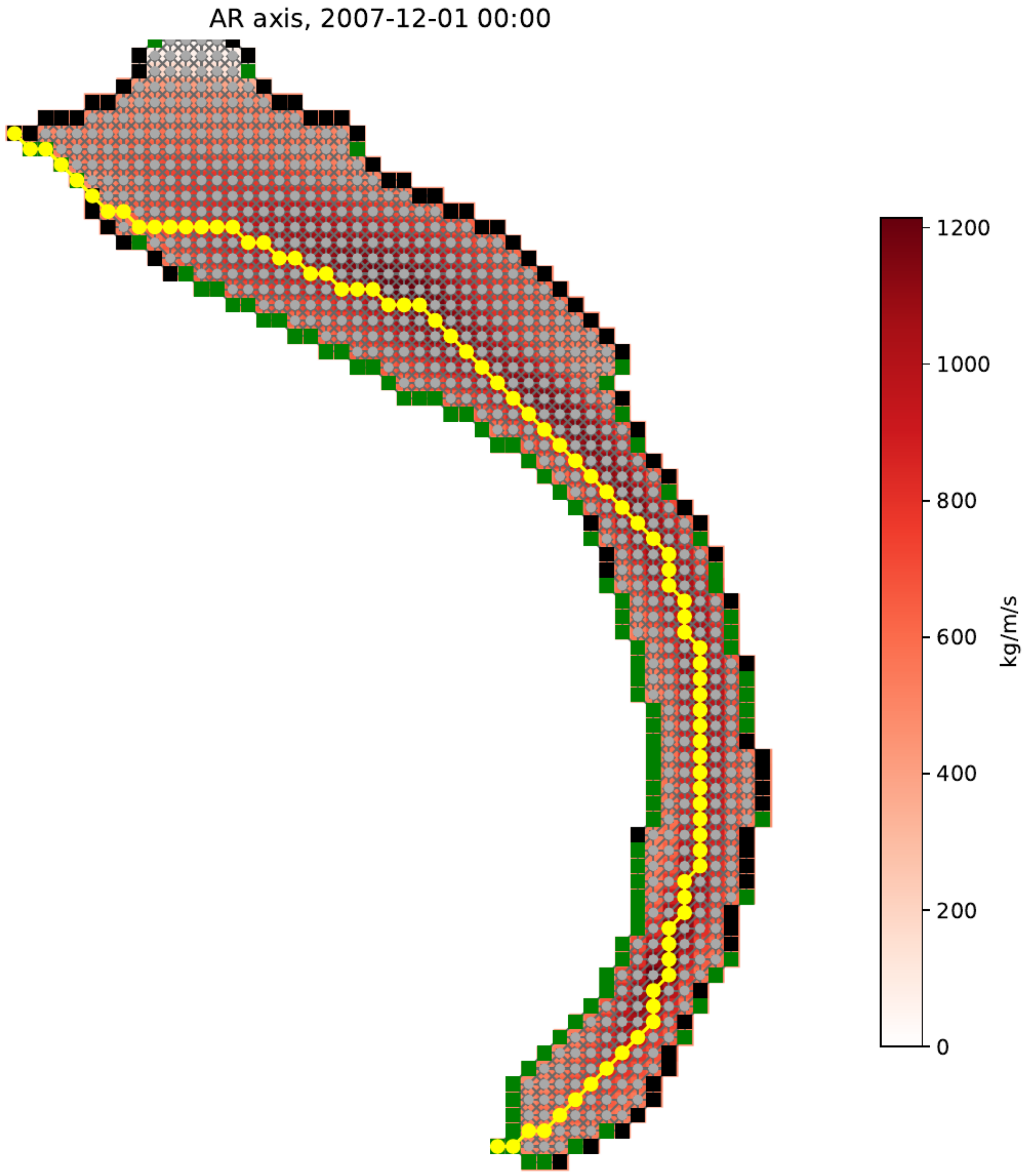


Fig. 4.4: Application of the axis finding algorithm on the AR in the North Pacific, 2007-Dec-1, 00 UTC. IVT within the AR is shown as colors, in $kg/(m \cdot s)$. The region of the AR (I_k) is shown as a collection of gray dots, which constitute nodes of the directed graph. Edges among neighboring nodes are created. A square marker is drawn at each boundary node, and is filled with green if the boundary node has net input moisture fluxes ($n_i \in L_{k,in}$), and black if it has net output moisture fluxes ($n_i \in L_{k,out}$). The found axis is highlighted in yellow.

4.4.2 Usage in Python scripts

The following snippet shows the axis finding process:

```
from ipart.AR_detector import findARAxis
axis_list, axismask=findARAxis(quslab, qvslab, mask_list, costhetas,
    sinthetas, param_dict)
```

where:

- `quslab`, `qvslab` are the u- and v- component of integrated vapor fluxes at a given time point.
- `mask_list` is a list of binary masks denoting the region of an each found AR.
- `sinthetas` and `costhetas` are used to compute the azimuthal angles for each grid cell.
- `param_dict` is the parameter dictionary as defined in *Input data*.

See also:

`AR_detector.findARAxis()`, *`AR_detector.maskToGraph()`*, *`AR_detector.getARAxis()`*.

4.4.3 Dedicated Python script

No detected Python script is offered for this process, as it is performed in the *`AR_detector.findARsGen()`* function.

4.4.4 Notebook example

An example of this process is given in this notebook.

4.5 Track ARs at individual time steps to form tracks

4.5.1 The modified Hausdorff distance definition

After the *Detect AR appearances from THR output* process, assume we have detected

- n ARs at time t, and
- m ARs at time $t + 1$.

There are theoretically $n \times m$ possible associations to link these two groups of ARs. Of cause not all of them are meaningful. The rules that are applied in the association process are:

1. *The nearest neighbor link method*: for any AR at time t, the nearest AR at time $t + 1$ "wins" and is associated with it, subject to that:
2. the **inter-AR distance (H)** is $\leq 1200\, km$.
3. no merging or splitting is allowed, any AR at time t can only be linked to one AR at time $t + 1$, similarly, any AR at time $t + 1$ can only be linked to one AR at time t.
4. after all associations at any give time point have been created, any left-over AR at time $t + 1$ forms a track on their own, and waits to be associated in the next iteration between $t + 1$ and $t + 2$.
5. any track that does not get updated during the $t - (t + 1)$ process terminates. This assumes that no gap in the track is allowed.

The remaining important question is how to define that **inter-AR distance (H)**. Here we adopt a modified **Hausdorff distance** definition:

$$H(A,B) \equiv min\{h_f(A,B), h_b(A,B)\}$$

where $H(A,B)$ is the **modified Hausdorff distance** from track **A** to track **B**, $h_f(A,B)$ is the **forward Hausdorff distance** from **A** to **B**, and $h_b(A,B)$ the **backward Hausdorff distance** from **A** to **B**. They are defined, respectively, as:

$$h_f(A,B) \equiv \max_{a \in A}\{\min_{b \in B}\{d_g(a,b)\}\}$$

namely, the largest great circle distance of all distances from a point in **A** to the closest point in **B**. And the backward Hausdorff distance is:

$$h_b(A,B) \equiv \max_{b \in B}\{\min_{a \in A}\{d_g(a,b)\}\}$$

Note that in general $h_f \neq h_b$. Unlike the standard definition of Hausdorff distance that takes the maximum of h_f and h_b, we take the minimum of the two.

The rationale behind this modification is that merging/splitting of ARs mostly happen in an end-to-end manner, during which a sudden increase/decrease in the length of the AR induce misalignment among the anchor points. Specifically, merging (splitting) tends to induce large backward (forward) Hausdorff distance. Therefore $min\{h_f(A,B), h_b(A,B)\}$ offers a more faithful description of the spatial closeness of ARs. For merging/splitting events in a side-to-side manner, this definition works just as well.

4.5.2 The nearest neighbor link method

To link AR records to form a track, a nearest neighbor method is used that the two AR axes found in consecutive time steps with a Hausdorff distance $\leq 1200\,km$ are linked, with an exclusive preference to the smallest Hausdorff distance.

Formally, suppose n tracks have been found at $t = t\ \ : A = \{a_1, a_2, \cdots, a_n\}$, and $t = t + 1$ has m new records: $B = \{b_1, b_2, \cdots, b_m\}$. The Hausdorff distances between all pairs of possible associations form a distance matrix:

$$M = \begin{bmatrix} H(a_1,b_1) & H(a_1,b_2) & \cdots & H(a_1,b_m) \\ H(a_2,b_1) & H(a_2,b_2) & \cdots & H(a_2,b_m) \\ \vdots & \vdots & \vdots & \vdots \\ H(a_n,b_1) & H(a_n,b_2) & \cdots & H(a_n,b_m) \end{bmatrix}$$

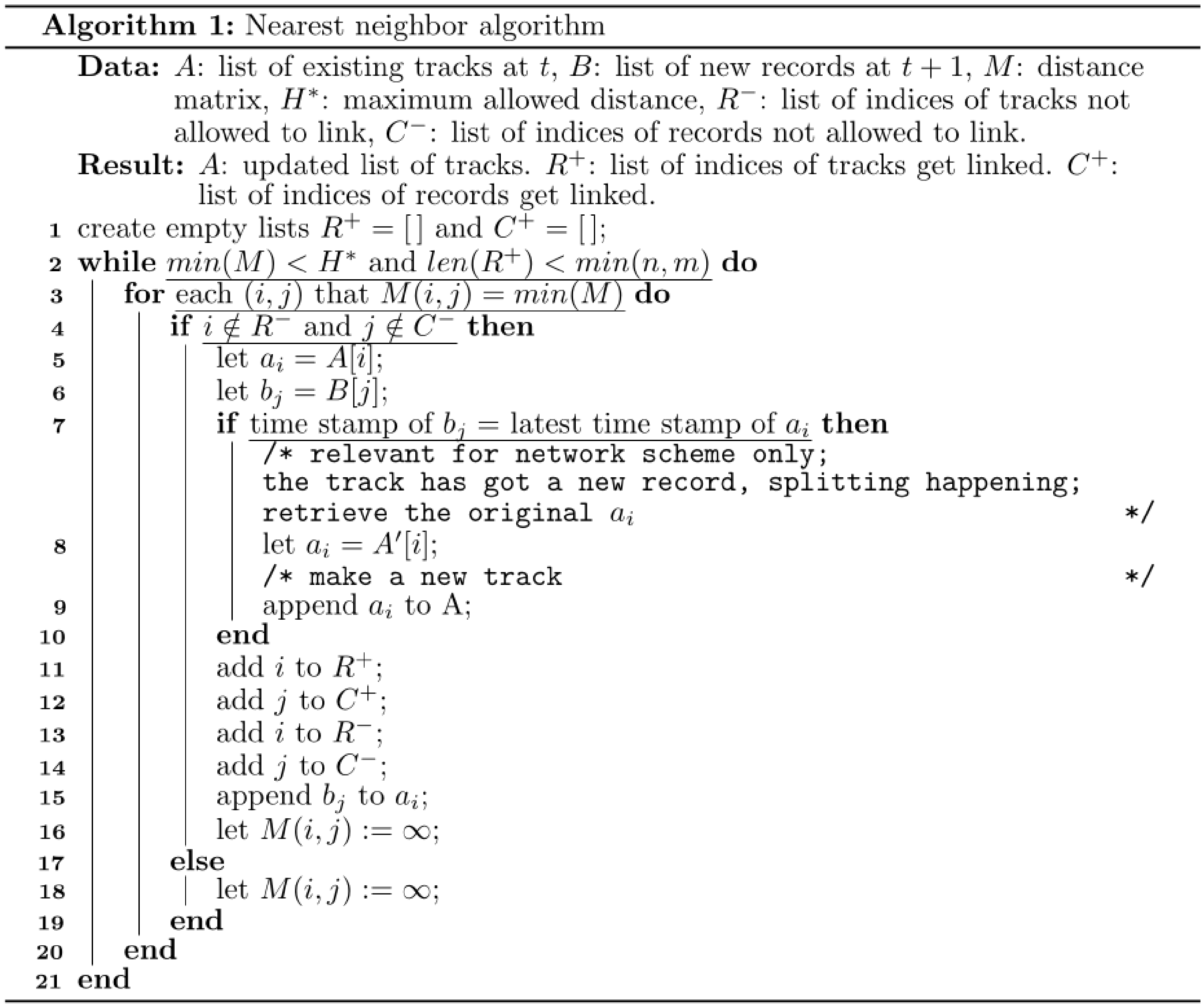
Then Algorithm shown in Fig. 4.5 is called with these arguments:

$$(A = A, B = B, M = M, H^* = 1200\,km, R^- = [\,], C^- = [\,])$$

The algorithm iteratively links two AR records with the smallest distance, so long as the distance does not exceed a given threshold H^*.

It ensures that no existing track connects to more than one new records, and no new record connects to more than one existing tracks. After this, any left-over records in B form a new track on their own. Then the same procedure repeats with updated time $t := t + 1$. Tracks that do not get any new record can be removed from the stack list, which only maintains a few active tracks at any given time. Therefore the complexity does not scale with time.

Note: One can use 3 consecutive calls of the above algorithm, with different input arguments, to achieve merging and splitting in the tracks.



```
Algorithm 1: Nearest neighbor algorithm
  Data: $A$: list of existing tracks at $t$, $B$: list of new records at $t+1$, $M$: distance
        matrix, $H^*$: maximum allowed distance, $R^-$: list of indices of tracks not
        allowed to link, $C^-$: list of indices of records not allowed to link.
  Result: $A$: updated list of tracks. $R^+$: list of indices of tracks get linked. $C^+$:
          list of indices of records get linked.
  create empty lists $R^+ = [\,]$ and $C^+ = [\,]$;
  while $min(M) < H^*$ and $len(R^+) < min(n, m)$ do
      for each $(i, j)$ that $M(i, j) = min(M)$ do
          if $i \notin R^-$ and $j \notin C^-$ then
              let $a_i = A[i]$;
              let $b_j = B[j]$;
              if time stamp of $b_j$ = latest time stamp of $a_i$ then
                  /* relevant for network scheme only;
                  the track has got a new record, splitting happening;
                  retrieve the original $a_i$                              */
                  let $a_i = A'[i]$;
                  /* make a new track                                      */
                  append $a_i$ to A;
              end
              add $i$ to $R^+$;
              add $j$ to $C^+$;
              add $i$ to $R^-$;
              add $j$ to $C^-$;
              append $b_j$ to $a_i$;
              let $M(i, j) := \infty$;
          else
              let $M(i, j) := \infty$;
          end
      end
  end
```

Fig. 4.5: Algorithm for the nearest neighbor link method.

4.5.3 Input data

This step takes as inputs the AR records detected at individual time steps as computed in *Detect AR appearances from THR output*.

The tracker parameters used:

```
# Int, hours, gap allowed to link 2 records. Should be the time resolution of
# the data.
TIME_GAP_ALLOW=6

# tracking scheme. 'simple': all tracks are simple paths.
# 'full': use the network scheme, tracks are connected by their joint points.
TRACK_SCHEME='simple'  # 'simple' | 'full'

# int, max Hausdorff distance in km to define a neighborhood relationship
MAX_DIST_ALLOW=1200  # km

# int, min duration in hrs to keep a track.
MIN_DURATION=24

# int, min number of non-relaxed records in a track to keep a track.
MIN_NONRELAX=1

# whether to plot linkage schematic plots or not
SCHEMATIC=True
```

4.5.4 Usage in Python scripts

The tracking process is handled by the `AR_tracer.trackARs()` function:

```
from ipart.AR_tracer import trackARs
from ipart.AR_tracer import readCSVRecord

ardf=readCSVRecord(RECORD_FILE)
track_list=trackARs(ardf, TIME_GAP_ALLOW, MAX_DIST_ALLOW,
    track_scheme=TRACK_SCHEME, isplot=SCHEMATIC, plot_dir=plot_dir)
```

where

- `RECORD_FILE` is the path to the `csv` file saving the individual AR records. Refer to this notebook for more information on the creation of this file.
- `ardf` is a `pandas.DataFrame` object containing the AR records at individual time points.
- `track_list` is a list of `AR objects`, each stores a sequence of AR records that form a single track. The `data` attribute of the `AR object` is a `pandas.DataFrame` object, with the same columns as shown in *AR records DataFrame*.

After this, one can optionally perform a filtering on the obtained tracks, using `AR_tracer.filterTracks()`, to remove, for instance, tracks that do not last for more than 24 hours:

```
from ipart.AR_tracer import filterTracks
track_list=filterTracks(track_list, MIN_DURATION, MIN_NONRELAX)
```

4.5.5 Example output

The resultant AR track can be visualized using the following snippet:

```
from ipart.utils import plot
import cartopy.crs as ccrs

latax=np.arange(0, 90)
lonax=np.arange(80, 440)   # degree east, shifted by 80 to ensure monotonically␣
↪increasing axis

plot_ar=track_list[6]   # plot the 7th track in list

figure=plt.figure(figsize=(12,6),dpi=100)
ax=figure.add_subplot(111, projection=ccrs.PlateCarree())
plotplotARTrack(plot_ar,latax,lonax,ax,full=True)
```

See also:

utils.plot.plotARTrack().

The output figure looks like Fig. 4.6 below.

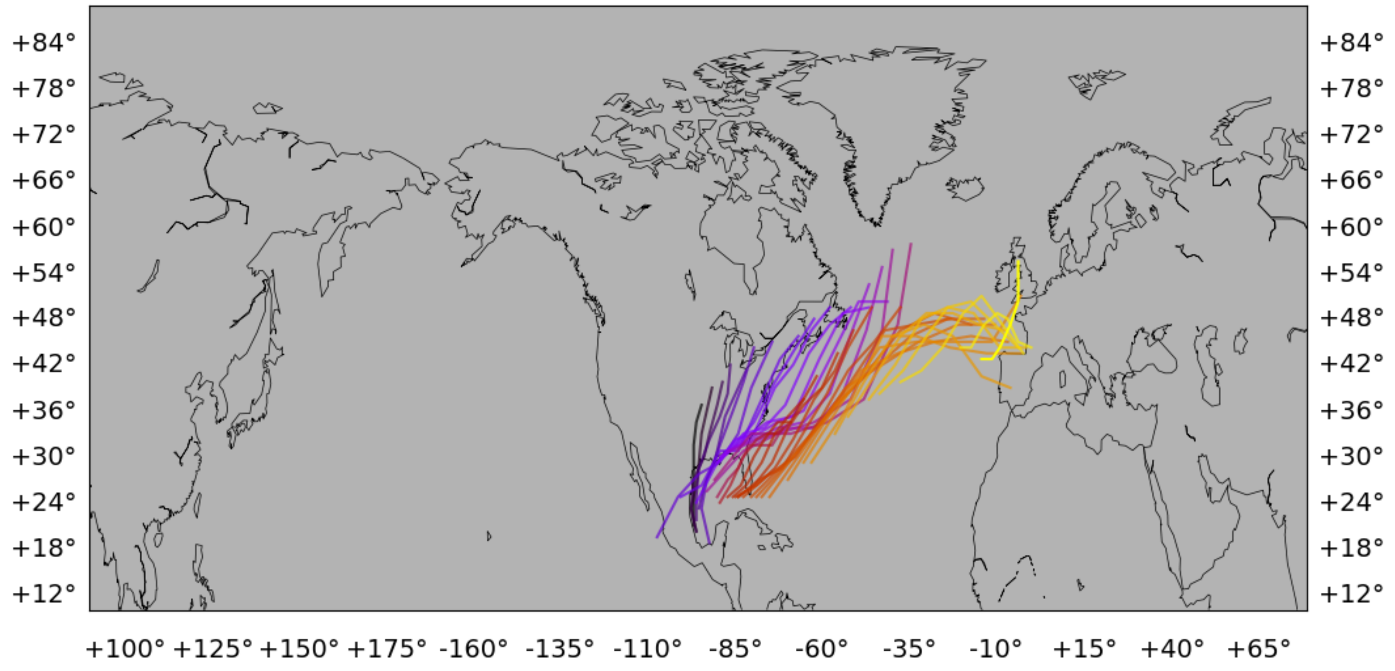


Fig. 4.6: Locations of a track labelled "198424" found in year 1984. Black to yellow color scheme indicates the evolution.

4.5.6 Dedicated Python script

You can use the `scripts/trace_ARs.py` script for AR tracking process in production.

Note: Unlike the AR occurrence detection process, this tracking process is time-dependent and therefore can not be paralleized. Also, if you divide the detection process into batches, e.g. one for each year, you may want to combine the output csv records into one big data file, and perform the tracking on this combined data file. This would prevent a track lasting from the end of one year into the next from being broken into 2.

4.5.7 Notebook example

An example of this process is given in this notebook.

CHAPTER 5

ipart module contents

5.1 Documentation page for thr.py

Perform THR computation on IVT data

Author: guangzhi XU (xugzhi1987@gmail.com) Update time: 2020-07-22 09:58:36.

`thr.`**`THR`**(*ivtNV*, *kernel*, *oroNV=None*, *high_terrain=600*, *verbose=True*)
Perform THR filtering process on 3d data

Parameters

- **`ivtNV`** (*`NCVAR`*) – 3D or 4D input IVT data, with dimensions (time, lat, lon) or (time, level, lat, lon).
- **`kernel`** (*`list or tuple`*) – list/tuple of integers specifying the shape of the kernel/structuring element used in the gray erosion process.

Keyword Arguments

- **`oroNV`** (*`NCVAR or None`*) – 2D array, surface orographic data in meters. This optional surface height info is used to perform a separate reconstruction computation for areas with high elevations, and the results can be used to enhance the continent-penetration ability of landfalling ARs. Sensitivity in landfalling ARs is enhanced, other areas are not affected. Needs to have compatible (lat, lon) shape as <ivt>. If None, omit this process and treat areas with different heights all equally.

  New in v2.0.

- **`high_terrain`** (*`float`*) – minimum orographic height (in m) to define as high terrain area, within which a separate reconstruction is performed. Only used if <oroNV> is not None.

  New in v2.0.

- **`verbose`** (*`bool`*) – print some messages or not.

Returns

ivtNV (NCVAR) – 3D or 4D array, input <ivt>. ivtrecNV (NCVAR): 3D or 4D array, the reconstruction component from the

THR process.

ivtanoNV (NCVAR): 3D or 4D array, the difference between input <ivt> and <ivtrecNV>.

`thr.`**`getAttrDict`**(*ref_var*, *component*)

Prepare a metadata dict from a reference variable

Parameters

- **`ref_var`** (*`ipart.utils.NCVAR`*) – NCVAR obj from which meta is obtained and modified.
- **`component`** (*`str`*) – 'reconstruction' or 'anomaly'. Strings as modifiers to modify attributes in <ref_var>.

Returns *attdict (dict)* – attribute dictionary.

`thr.`**`rotatingTHR`**(*filelist*, *varin*, *kernel*, *outputdir*, *oroNV=None*, *selector=None*, *high_terrain=600*, *verbose=True*)

Compute time filtering on data in different files.

Parameters

- **`filelist`** (*`list`*) – list of abs paths to data files. User is responsible to make sure files in list have correct chronological order. Note that time axis in data files should be the 1st axis.
- **`varin`** (*`str`*) – variable id in files.
- **`kernel`** (*`list or tuple`*) – list/tuple of integers specifying the shape of the kernel/structuring element used in the gray erosion process.
- **`selector`** (*`utils.funcs.Selector`*) – selector obj to select subset of data.
- **`outputdir`** (*`str`*) – path to folder to save outputs.

Keyword Arguments

- **`oroNV`** (*`NCVAR`*) – 2D array, surface orographic data in meters. This additional surface height info is used to perform a separate reconstruction computation for areas with high elevations, and the results can be used to enhance the continent-penetration ability of landfalling ARs. Sensitivity in landfalling ARs is enhanced, other areas are not affected. Needs to have compatible shape as <ivt>. If None, omit this process and treat areas with different heights all equally.

  New in v2.0.
- **`high_terrain`** (*`float`*) – minimum orographic height to define as high terrain area, within which a separate reconstruction is performed. Only used if <oroNV> is not None.

  New in v2.0.
- **`verbose`** (*`bool`*) – print some messages or not.

Designed to perform temporal filtering on data that are too large to fit into memory, e.g. high-resolution data across multiple decades.

Function will read in 2 files at once, call the filtering function on the concatenated data, and shift 1 step to the next 2 files. If at the begining, pad 0s to the left end. If in the mid, pad filtered data in the mid of the concatenated data in the previous step. If at the end, pad 0s to the right end.

The filtering function <func> is assumed to apply a filtering window with odd length n, and truncates (n-1)/2 points from both ends. If the function doesn't truncate data, will raise an exception.

5.2 Documentation page for AR_detector.py

AR detection functions.

Author: guangzhi XU (xugzhi1987@gmail.com) Update time: 2020-07-22 09:27:22.

`AR_detector.`**`applyCropIdx`**(*slab*, *cropidx*)

Cut out a bounding box from given 2d slab given corner indices

Parameters

- **`slab`** (`NCVAR or ndarray`) – 2D array to cut a box from.
- **`cropidx`** (`tuple`) – (y, x) coordinate indices, output from cropMask().

Returns

cropslab (NCVAR or ndarray) –

2D sub array cut from <slab> using <cropidx> as boundary indices.

`AR_detector.`**`areaFilt`**(*mask*, *area*, *min_area=None*, *max_area=None*, *zonal_cyclic=False*)

Filter AR binary masks by region areas

Parameters

- **`mask`** (`ndarray`) – 2D binary mask with detected objects shown as 1s.
- **`area`** (`ndarray`) – 2D map showing grid cell areas in km^2.

Keyword Arguments

- **`min_area`** (`float or None`) – if not None, minimum area to filter objects in <mask>.
- **`max_area`** (`float or None`) – if not None, maximum area to filter objects in <mask>.
- **`zonal_cyclic`** (`bool`) – if True, treat zonal boundary as cyclic.

Returns *result (ndarray)* – 2D binary mask with objects area-filtered.

`AR_detector.`**`cart2Spherical`**(*x*, *y*, *z*, *shift_lon*)

Convert Cartesian to spherical coordiantes :param x,y,z: x-, y- and z- coordiantes. :type x,y,z: float :param shift_lon: longitude to shift so the longitude dimension

starts from this number.

Returns

result (ndrray) –

1x3 array, the columns are the lat-, lon- and r- coordinates. r- coordinates are all 1s (unit sphere).

`AR_detector.`**`cart2Wind`**(*vs*, *lats*, *lons*)

Convert winds in Cartesian coordinates to u,v, inverse to wind2Cart. :param vs: Cartesian representation of the horizontal

winds.

Parameters **lats,lons** (*float or ndarray*) – latitude and longitude coordinates corresponding to the wind vectors given by <u> and <v>.

Returns *u,v (float or ndarray)* – u- and v- components of horizontal winds.

AR_detector.**checkCyclic**(*mask*)
Check binary mask is zonally cyclic or not

Parameters **mask** (*ndarray*) – 2d binary array, with 1s denoting existance of a feature.

Returns

result (bool) –

True if feature in <mask> is zonally cyclic, False otherwise.

AR_detector.**computeTheta**(*p1*, *p2*)
Tangent line to the arc |p1-p2|

Parameters **p1,p2** (*float*) – (lat,lon) coordinates

Returns

theta (float) –

unit vector tangent at point <p1> pointing at <p2> on unit sphere.

AR_detector.**cropMask**(*mask*, *edge=4*)
Cut out a bounding box around mask==1 areas

Parameters

- **mask** (*ndarray*) – 2D binary map showing the location of an AR with 1s.
- **edge** (*int*) – number of pixels as edge at 4 sides.

Returns

mask[y1 –

y2,x1:x2] (ndarray): a sub region cut from <mask> surrouding regions with value=1.

(yy,xx): y-, x- indices of the box of the cut region. Can later by used in applyCropIdx(new_slab, (yy,xx)) to crop out the same region from a new array <new_slab>.

AR_detector.**crossSectionFlux**(*mask*, *quslabNV*, *qvslabNV*, *axis_rdp*)
Compute setion-wise orientation differences and cross-section fluxes in an AR

Parameters

- **mask** (*ndarray*) – CROPPED (see cropMask and applyCropIdx) 2D binary map showing the location of an AR with 1s.
- **quslab** (*NCVAR*) – CROPPED (n * m) 2D array of u-flux, in kg/m/s.
- **qvslab** (*NCVAR*) – CROPPED (n * m) 2D array of v-flux, in kg/m/s.
- **axis_rdp** (*ndarray*) – Nx2 array storing the (lat, lon) coordinates of rdp-simplified AR axis.

Returns

angles (ndarray) –

2D map with the same shape as <mask>, showing section-wise orientation differences between horizontal flux (as in <quslab>, <qvslab>) and the AR axis of that section. In degrees. Regions outside of AR (0s in <mask>) are masked.

anglesmean (float): area-weighted averaged of <angles> inside <mask>. crossflux (ndarray): 2D map with the same shape as <mask>,

> the section-wise cross-section fluxes in the AR, defined as the projection of fluxes onto the AR axis, i.e. flux multiplied by the cos of <angles>.

seg_thetas (list): list of (x, y, z) Cartesian coordinates of the tangent vectors along section boundaries.

`AR_detector.`**`cyclicLabel`**(*mask*, *connectivity=1*, *iszonalcyclic=False*)

Label connected region, zonally cyclic version

Parameters **`mask`** (*`ndarray`*) – 2d binary mask with 1s denoting feature regions.

Keyword Arguments

- **`connectivity`** (*`int`*) – 1 or 2 connectivity. 2 probaby won't work for zonally cyclic labelling.
- **`iszonalcyclic`** (*`bool`*) – doing zonally cyclic labelling or not. If False, call skimage.measure.label(). If True, call skimage.measure.label() for initial labelling, then shift the map zonally by half the zonal length, do another measure.label() to find zonally linked regions. Then use this to update the original labels.

Returns

result (ndarray) –

2d array with same shape as <mask>. Each connected region in <mask> is given an int label.

`AR_detector.`**`determineThresLow`**(*anoslab*, *sill=0.8*)

Determine the threshold for anomalous IVT field, experimental

Parameters **`anoslab`** (*`ndarray`*) – (n * m) 2D anomalous IVT slab, in kg/m/s.

Keyword Arguments **`sill`** (*`float`*) – float in (0, 1). Fraction of max score to define as the 1st time the score is regarded as reaching stable level.

Returns *result (float)* – determined lower threshold.

Method of determining the threshold: 1. make a loop through an array of thresholds from 1 to the 99th percentile of <ivtano>. 2. at each level, record the number of pixels > threshold. 3. after looping, pixel number counts will have a curve with a

> lower-left elbow. Compute the product of number counts and thresholds P. P will have a peak value around the elbow.

4. choose the 1st time P reaches the 80% of max(P), and pick the corresponding threshold as result.

Largely empirical, but seems to work good on MERRA2 IVT results.

`AR_detector.`**`findARAxis`**(*quslab*, *qvslab*, *armask_list*, *costhetas*, *sinthetas*, *param_dict*, *verbose=True*)

Find AR axis

Parameters

- **`quslab`** (*`ndarray`*) – (n * m) 2D u-flux slab, in kg/m/s.
- **`qvslab`** (*`ndarray`*) – (n * m) 2D v-flux slab, in kg/m/s.

- **armask_list** (*list*) – list of 2D binary masks, each with the same shape as <quslab> etc., and with 1s denoting the location of a found AR.
- **costhetas** (*ndarray*) – (n * m) 2D slab of grid cell shape: cos=dx/sqrt(dx^2+dy^2).
- **sinthetas** (*ndarray*) – (n * m) 2D slab of grid cell shape: sin=dy/sqrt(dx^2+dy^2).
- **param_dict** (*dict*) – a dict containing parameters controlling the detection process. Keys of the dict: 'thres_low', 'min_area', 'max_area', 'max_isoq', 'max_isoq_hard', 'min_lat', 'max_lat', 'min_length', 'min_length_hard', 'rdp_thres', 'fill_radius', 'single_dome', 'max_ph_ratio', 'edge_eps'. See the doc string of findARs() for more.

Keyword Arguments **verbose** (*bool*) – print some messages or not.

Returns

axes (list) –

list of AR axis coordinates. Each coordinate is defined as a Nx2 ndarray storing (y, x) indices of the axis (indices defined in the matrix of corresponding mask in <armask_list>.)

axismask (ndarray): 2D binary mask showing all axes in <axes> merged into one map.

New in v2.0.

AR_detector.**findARs**(*ivt*, *ivtrec*, *ivtano*, *qu*, *qv*, *lats*, *lons*, *times=None*, *ref_time='days since 1900-01-01'*, *thres_low=1*, *min_area=500000.0*, *max_area=18000000.0*, *min_LW=2*, *min_lat=20*, *max_lat=80*, *min_length=2000*, *min_length_hard=1500*, *rdp_thres=2*, *fill_radius=None*, *single_dome=False*, *max_ph_ratio=0.6*, *edge_eps=0.4*, *zonal_cyclic=False*, *verbose=True*)

Find ARs from THR results, get all results in one go.

Parameters

- **ivt** (*ndarray*) – 3D or 4D input IVT data, with dimensions (time, lat, lon) or (time, level, lat, lon).
- **ivtrec** (*ndarray*) – 3D or 4D array, the reconstruction component from the THR process.
- **ivtano** (*ndarray*) – 3D or 4D array, the difference between input <ivt> and <ivtrec>.
- **qu** (*ndarray*) – 3D or 4D array, zonal component of integrated moisture flux.
- **qv** (*ndarray*) – 3D or 4D array, meridional component of integrated moisture flux.
- **lats** (*ndarray*) – 1D, latitude coordinates, the length needs to be the same as the lat dimension of <ivt>.
- **lons** (*ndarray*) – 1D, longitude coordinates, the length needs to be the same as the lon dimension of <ivt>.

Keyword Arguments

- **times** (*list or array*) – time stamps of the input data as a list of strings, e.g. ['2007-01-01 06:00:00', '2007-01-01 12:00', ...]. Needs to have the same length as the time dimension of <ivt>. If None, default to create a dummy 6-hourly time axis, using <ref_time> as start, with a length as the time dimension of <ivt>.
- **ref_time** (*str*) – reference time point to create dummy time axis, if no time stamps are given in <times>.
- **thres_low** (*float or None*) – kg/m/s, define AR candidates as regions >= this anomalous ivt level. If None is given, compute a threshold based on anomalous ivt data in <ivtano> using an empirical method:

1. make a loop through an array of thresholds from 1 to the 99th percentile of <ivtano>. 2. at each level, record the number of pixels > threshold. 3. after looping, pixel number counts will have a curve with a

   lower-left elbow. Compute the product of number counts and thresholds P. P will have a peak value around the elbow.

4. choose the 1st time P reaches the 80% of max(P), and pick the corresponding threshold as result.

- **min_area** (*float*) – km^2, drop AR candidates smaller than this area.
- **max_area** (*float*) – km^2, drop AR candidates larger than this area.
- **min_LW** (*float*) – minimum length/width ratio.
- **min_lat** (*float*) – degree, exclude systems whose centroids are lower than this latitude. NOTE this is the absolute latitude for both NH and SH. For SH, systems with centroid latitude north of - min_lat will be excluded.
- **max_lat** (*float*) – degree, exclude systems whose centroids are higher than this latitude. NOTE this is the absolute latitude for both NH and SH. For SH, systems with centroid latitude south of - max_lat will be excluded.
- **min_length** (*float*) – km, ARs shorter than this length is treated as relaxed.
- **min_length_hard** (*float*) – km, ARs shorter than this length is discarded.
- **rdp_thres** (*float*) – degree lat/lon, error when simplifying axis using rdp algorithm.
- **fill_radius** (*int or None*) – number of grids as radius to fill small holes in AR contour. If None, computed as

  max(1,int(4*0.75/RESO))

  where RESO is the approximate resolution in degrees of lat/lon, estimated from <lat>, <lon>.
- **single_dome** (*bool*) – do peak partition or not, used to separate systems that are merged together with an outer contour.
- **max_ph_ratio** (*float*) – max prominence/height ratio of a local peak. Only used when single_dome=True
- **edge_eps** (*float*) – minimal proportion of flux component in a direction to total flux to allow edge building in that direction.
- **zonal_cyclic** (*bool*) – if True, treat the data as zonally cyclic (e.g. entire hemisphere or global). ARs covering regions across the longitude bounds will be correctly treated as one. If your data is not zonally cyclic, or a zonal shift of the data can put the domain of interest to the center, consider doing the shift and setting this to False, as it will save computations.
- **verbose** (*bool*) – print some messages or not.

Returns

time_idx (list) – indices of the time dimension when any AR is found. labels_allNV (NCVAR): 3D array, with dimension

(time, lat, lon). At each time slice, a unique int label is assign to each detected AR at that time, and the AR region is filled out with the label value in the (lat, lon) map.

angles_allNV (NCVAR): 3D array showing orientation differences between AR axes and fluxes, for all ARs. In degrees.

crossfluxes_allNV (NCVAR): 3D array showing cross- sectional fluxes in all ARs. In kg/m/s.

result_df (DataFrame): AR record table. Each row is an AR, see code in getARData() for columns.

See also:

findARsGen(): generator version, yields results at time points separately.

`AR_detector.`**`findARsGen`**(*ivt*, *ivtrec*, *ivtano*, *qu*, *qv*, *lats*, *lons*, *times=None*, *ref_time='days since 1900-01-01'*, *thres_low=1*, *min_area=500000.0*, *max_area=18000000.0*, *min_LW=2*, *min_lat=20*, *max_lat=80*, *min_length=2000*, *min_length_hard=1500*, *rdp_thres=2*, *fill_radius=None*, *single_dome=False*, *max_ph_ratio=0.6*, *edge_eps=0.4*, *zonal_cyclic=False*, *verbose=True*)

Find ARs from THR results, generator version

Parameters

- **`ivt`** (*`ndarray`*) – 3D or 4D input IVT data, with dimensions (time, lat, lon) or (time, level, lat, lon).
- **`ivtrec`** (*`ndarray`*) – 3D or 4D array, the reconstruction component from the THR process.
- **`ivtano`** (*`ndarray`*) – 3D or 4D array, the difference between input <ivt> and <ivtrec>.
- **`qu`** (*`ndarray`*) – 3D or 4D array, zonal component of integrated moisture flux.
- **`qv`** (*`ndarray`*) – 3D or 4D array, meridional component of integrated moisture flux.
- **`lats`** (*`ndarray`*) – 1D, latitude coordinates, the length needs to be the same as the lat dimension of <ivt>.
- **`lons`** (*`ndarray`*) – 1D, longitude coordinates, the length needs to be the same as the lon dimension of <ivt>.

Keyword Arguments

- **`times`** (*`list or array`*) – time stamps of the input data as a list of strings, e.g. ['2007-01-01 06:00:00', '2007-01-01 12:00', …]. Needs to have the same length as the time dimension of <ivt>. If None, default to create a dummy 6-hourly time axis, using <ref_time> as start, with a length as the time dimension of <ivt>.
- **`ref_time`** (*`str`*) – reference time point to create dummy time axis, if no time stamps are given in <times>.
- **`thres_low`** (*`float or None`*) – kg/m/s, define AR candidates as regions >= this anomalous ivt level. If None is given, compute a threshold based on anomalous ivt data in <ivtano> using an empirical method:

  1. make a loop through an array of thresholds from 1 to the 99th percentile of <ivtano>. 2. at each level, record the number of pixels > threshold. 3. after looping, pixel number counts will have a curve with a

     lower-left elbow. Compute the product of number counts and thresholds P. P will have a peak value around the elbow.

4. choose the 1st time P reaches the 80% of max(P), and pick the corresponding threshold as result.

- **min_area** (*float*) – km^2, drop AR candidates smaller than this area.
- **max_area** (*float*) – km^2, drop AR candidates larger than this area.
- **min_LW** (*float*) – minimum length/width ratio.
- **min_lat** (*float*) – degree, exclude systems whose centroids are lower than this latitude. NOTE this is the absolute latitude for both NH and SH. For SH, systems with centroid latitude north of - min_lat will be excluded.
- **max_lat** (*float*) – degree, exclude systems whose centroids are higher than this latitude. NOTE this is the absolute latitude for both NH and SH. For SH, systems with centroid latitude south of - max_lat will be excluded.
- **min_length** (*float*) – km, ARs shorter than this length is treated as relaxed.
- **min_length_hard** (*float*) – km, ARs shorter than this length is discarded.
- **rdp_thres** (*float*) – degree lat/lon, error when simplifying axis using rdp algorithm.
- **fill_radius** (*int or None*) – number of grids as radius to fill small holes in AR contour. If None, computed as

  max(1,int(4*0.75/RESO))

  where RESO is the approximate resolution in degrees of lat/lon, estimated from <lat>, <lon>.
- **single_dome** (*bool*) – do peak partition or not, used to separate systems that are merged together with an outer contour.
- **max_ph_ratio** (*float*) – max prominence/height ratio of a local peak. Only used when single_dome=True
- **edge_eps** (*float*) – minimal proportion of flux component in a direction to total flux to allow edge building in that direction.
- **zonal_cyclic** (*bool*) – if True, treat the data as zonally cyclic (e.g. entire hemisphere or global). ARs covering regions across the longitude bounds will be correctly treated as one. If your data is not zonally cyclic, or a zonal shift of the data can put the domain of interest to the center, consider doing the shift and setting this to False, as it will save computations.
- **verbose** (*bool*) – print some messages or not.

Returns

ii (int) – index of the time dimension when any AR is found. timett_str (str): time when any AR is found, in string format. labelsNV (NCVAR): 2D array, with dimension

(lat, lon). A unique int label is assign to each detected AR at the time, and the AR region is filled out with the label value in the (lat, lon) map.

anglesNV (NCVAR): 2D array showing orientation differences between AR axes and fluxes, for all ARs. In degrees.

crossfluxesNV (NCVAR): 2D array showing cross- sectional fluxes in all ARs. In kg/m/s.

ardf (DataFrame): AR record table. Each row is an AR, see code in getARData() for columns.

See also:

findARs(): collect and return all results in one go.

New in v2.0.

`AR_detector.`**`getARAxis`**(*g*, *quslab*, *qvslab*, *mask*)

Find AR axis from AR region mask

Parameters

- **g** (*networkx.DiGraph*) – directed planar graph constructed from AR mask and flows. See maskToGraph().
- **quslab** (*ndarray*) – 2D map of u-flux.
- **qvslab** (*ndarray*) – 2D map of v-flux.
- **mask** (*ndarray*) – 2D binary map showing the location of an AR with 1s.

Returns

path (ndarray) –

Nx2 array storing the AR axis coordinate indices in (y, x) format.

axismask (ndarray): 2D binary map with same shape as <mask>, with grid cells corresponding to coordinates in <path> set to 1s.

`AR_detector.`**`getARData`**(*slab*, *quslab*, *qvslab*, *anoslab*, *quano*, *qvano*, *areas*, *lats*, *lons*, *mask_list*, *axis_list*, *timestr*, *param_dict*)

Fetch AR related data

Parameters

- **slab** (*ndarray*) – (n * m) 2D array of IVT, in kg/m/s.
- **quslab** (*ndarray*) – (n * m) 2D array of u-flux, in kg/m/s.
- **qvslab** (*ndarray*) – (n * m) 2D array of v-flux, in kg/m/s.
- **anoslab** (*ndarray*) – (n * m) 2D array of IVT anomalies, in kg/m/s.
- **quano** (*ndarray*) – (n * m) 2D array of u-flux anomalies, in kg/m/s.
- **qvano** (*ndarray*) – (n * m) 2D array of v-flux anomalies, in kg/m/s.
- **areas** (*ndarray*) – (n * m) 2D grid cell area slab, in km^2.
- **lats** (*ndarray*) – 1d array, latitude coordinates.
- **lons** (*ndarray*) – 1d array, longitude coordinates.
- **mask_list** (*list*) – list of 2D binary masks, each with the same shape as <anoslab> etc., and with 1s denoting the location of a found AR.
- **axis_list** (*list*) – list of AR axis coordinates. Each coordinate is defined as a Nx2 ndarray storing (y, x) indices of the axis (indices defined in the matrix of corresponding mask in <masks>.)
- **timestr** (*str*) – string of time snap.
- **param_dict** (*dict*) – a dict containing parameters controlling the detection process. Keys of the dict: 'thres_low', 'min_area', 'max_area', 'max_isoq', 'max_isoq_hard', 'min_lat', 'max_lat', 'min_length', 'min_length_hard', 'rdp_thres', 'fill_radius', 'single_dome', 'max_ph_ratio', 'edge_eps'. See the doc string of findARs() for more.

Returns

labelsNV (NCVAR) –

(n * m) 2D int map showing all ARs at current time. Each AR is labeled by an int label, starting from 1. Background is filled with 0s.

anglesNV (NCVAR): (n * m) 2D map showing orientation differences between AR axes and fluxes, for all ARs. In degrees.

crossfluxesNV (NCVAR): (n * m) 2D map showing cross- sectional fluxes in all ARs. In kg/m/s.

df (pandas.DataFrame): AR record table. Each row is an AR, see code below for columns.

`AR_detector.`**`getNormalVectors`**(*point_list*, *idx*)

Get the normal vector and the tagent vector to the plane dividing 2 sections along the AR axis.

Parameters

- **`point_list`** (*`list`*) – list of (lat, lon) coordinates.
- **`idx`** (*`int`*) – index of the point in <point_list>, denoting the point in question.

Returns

normi (tuple) –

the (x, y, z) Cartesian coordinate of the unit normal vector, at the point denoted by <idx>, on the Earth surface. This is the normal vector to the plane spanned by the vector Theta and P. Where P is the vector pointing to the point in question (point_list[idx]), and Theta is the tangent vector evenly dividing the angle formed by <P,P1>, and <P,P2>. Where P1, P2 are 2 points on both side of P.

thetai (tuple): the (x, y, z) Cartesian coordinate of the tangent vector Theta above.

`AR_detector.`**`insertCropSlab`**(*shape*, *cropslab*, *cropidx*)

Insert the cropped sub-array back to a larger empty slab

Parameters

- **`shape`** (*`tuple`*) – (n, m) size of the larger slab.
- **`cropslab`** (*`ndarray`*) – 2D array to insert.
- **`cropidx`** (*`tuple`*) – (y, x) coordinate indices, output from cropMask(), defines where <cropslab> will be inserted into.

Returns

result (ndarray) –

2D slab with shape (n, m), an empty array with a box at <cropidx> replaced with data from <cropslab>.

`AR_detector.`**`maskToGraph`**(*mask*, *quslab*, *qvslab*, *costhetas*, *sinthetas*, *edge_eps*, *connectivity=2*)

Create graph from AR mask

Parameters

- **`mask`** (*`ndarray`*) – 2D binary map showing the location of an AR with 1s.
- **`quslab`** (*`ndarray`*) – 2D map of u-flux.
- **`qvslab`** (*`ndarray`*) – 2D map of v-flux.
- **`costhetas`** (*`ndarray`*) – (n * m) 2D slab of grid cell shape: cos=dx/sqrt(dx^2+dy^2).

- **sinthetas** (*ndarray*) – (n * m) 2D slab of grid cell shape: sin=dy/sqrt(dx^2+dy^2).
- **edge_eps** (*float*) – float in (0,1), minimal proportion of flux component in a direction to total flux to allow edge building in that direction. Defined in Global preamble.
- **connectivity** (*int*) – 1 or 2. 4- or 8- connectivity in defining neighbor- hood relationship in a 2D square grid.

Returns

g (networkx.DiGraph) –

directed planar graph constructed from AR mask and flows.

`AR_detector.`**`partPeaks`**(*cropmask*, *cropidx*, *orislab*, *area*, *min_area*, *max_ph_ratio*, *fill_radius*)

Separate local maxima by topographical prominence, watershed version

Parameters

- **cropmask** (*ndarray*) – 2D binary array, defines regions of local maxima.
- **cropidx** (*tuple*) – (y, x) coordinate indices, output from cropMask().
- **orislab** (*ndarray*) – 2D array, giving magnitude/height/intensity values defining the topography.
- **area** (*ndarray*) – (n * m) 2D grid cell area slab, in km^2.
- **min_area** (*float*) – km^2, drop AR candidates smaller than this area.
- **max_ph_ratio** (*float*) – maximum peak/height ratio. Local peaks with a peak/height ratio larger than this value is treated as an independent peak.
- **fill_radius** (*int*) – number of grids as radius to further separate peaks.

Returns

result (ndarray) –

2D binary array, similar as the input <cropmask> but with connected peaks (if any) separated so that each connected region (with 1s) denotes an independent local maximum.

`AR_detector.`**`partPeaksOld`**(*cropmask*, *cropidx*, *orislab*, *max_ph_ratio*)

Separate local maxima by topographical prominence

Parameters

- **cropmask** (*ndarray*) – 2D binary array, defines regions of local maxima.
- **cropidx** (*tuple*) – (y, x) coordinate indices, output from cropMask().
- **orislab** (*ndarray*) – 2D array, giving magnitude/height/intensity values defining the topography.
- **max_ph_ratio** (*float*) – maximum peak/height ratio. Local peaks with a peak/height ratio larger than this value is treated as an independent peak.

Returns

result (ndarray) –

2D binary array, similar as the input <cropmask> but with connected peaks (if any) separated so that each connected region (with 1s) denotes an independent local maximum.

`AR_detector.`**`plotGraph`**(*graph*, *ax=None*, *show=True*)

Helper func to plot the graph of an AR coordinates. For debugging. :param graph: networkx Graph obj to visualize. :type graph: networkx.Graph

Keyword Arguments

- **`ax`** (*`matplotlib axis obj`*) – axis to plot on. If None, create a new one.
- **`show`** (*`bool`*) – whether to show the plot or not.

`AR_detector.`**`prepareMeta`**(*lats*, *lons*, *times*, *ntime*, *nlat*, *nlon*, *ref_time='days since 1900-01-01'*, *verbose=True*)

Prepare metadata for AR detection function calls

Parameters

- **`lats`** (*`ndarray`*) – 1D, latitude coordinates, the length needs to equal <nlat>.
- **`lons`** (*`ndarray`*) – 1D, longitude coordinates, the length needs to equal <nlon>.
- **`times`** (*`list or array`*) – time stamps of the input data as a list of strings, e.g. ['2007-01-01 06:00:00', '2007-01-01 12:00', …]. Needs to have the a length of <ntime>.
- **`ntime`** (*`int`*) – length of the time axis, should equal the length of <times>.
- **`nlat`** (*`int`*) – length of the latitude axis, should equal the length of <lats>.
- **`nlon`** (*`int`*) – length of the longitude axis, should equal the length of <lons>.

Keyword Arguments

- **`ref_time`** (*`str`*) – reference time point to create time axis.
- **`verbose`** (*`bool`*) – print some messages or not.

Returns

timeax (list) – a list of datetime objs. areamap (ndarray): grid cell areas in km^2, with shape (<nlat> x <nlon>). costhetas (ndarray): ratios of dx/sqrt(dx^2 + dy^2) for all grid cells.

with shape (<nlat> x <nlon>).

sinthetas (ndarray): ratios of dy/sqrt(dx^2 + dy^2) for all grid cells. with shape (<nlat> x <nlon>).

reso (float): (approximate) horizontal resolution in degrees of lat/lon estimate from <lats> and <lons>.

New in v2.0.

`AR_detector.`**`save2DF`**(*result_dict*)

Save AR records to a pandas DataFrame

Parameters **`result_dict`** (*`dict`*) – key: time str in 'yyyy-mm-dd hh:00' value: pandas dataframe. See getARData().

Returns

result_df (pandas.DataFrame) –

AR record table containing records from multiple time steps sorted by time.

`AR_detector.`**`spherical2Cart`**(*lat*, *lon*)

Convert spherical to Cartesian coordiantes :param lat,lon: latitude and longitude coordinates. :type lat,lon: float or ndarray

Returns

result (ndarray) –

Nx3 array, the columns are the x-, y- and z- coordinates.

AR_detector.**uvDecomp**(*u0*, *v0*, *i1*, *i2*)

Decompose background-transient components of u-, v- fluxes

Parameters

- **u0** (*ndarray*) – nd array of total u-flux.
- **v0** (*ndarray*) – nd array of total v-flux.
- **i1** (*ndarray*) – nd array of the reconstruction component of IVT.
- **i2** (*ndarray*) – nd array of the anomalous component of IVT (i2 = IVT - i1).

Returns

u1 (ndarray) –

nd array of the u-flux component corresponding to <i1>, i.e. the background component.

v1 (ndarray): nd array of the v-flux component corresponding to <i1>, i.e. the background component.

u2 (ndarray): nd array of the u-flux component corresponding to <i2>, i.e. the transient component.

v2 (ndarray): nd array of the v-flux component corresponding to <i2>, i.e. the transient component.

AR_detector.**wind2Cart**(*u*, *v*, *lats*, *lons*)

Convert u,v winds to Cartesian, consistent with spherical2Cart.

Parameters

- **u,v** (*float or ndarray*) – u- and v- components of horizontal winds.
- **lats,lons** (*float or ndarray*) – latitude and longitude coordinates corresponding to the wind vectors given by <u> and <v>.

Returns

vs (float or ndarray) –

Cartesian representation of the horizontal winds.

5.3 Documentation page for AR_tracer.py

Functions to compute Hausdorff distance between AR axes pairs and link ARs across time steps to form tracks.

Author: guangzhi XU (xugzhi1987@gmail.com) Update time: 2020-06-05 22:46:19.

class AR_tracer.**AR**(*id*, *data*)

Ojbect representing an AR entity

Hausdorff(*lats*, *lons*)

Compute modified Hausdorff distance from the lastest record to given axis

Parameters

- **lats** (*ndarray*) – 1d array, the target axis's latitude coordinates.
- **lons** (*ndarray*) – 1d array, the target axis's longitude coordinates.

Returns *float* – modified Hausdorff distance from this AR to the given axis.

__init__(*id*, *data*)

Parameters

- **`id`** (*`int`*) – a numeric id for each AR.
- **`data`** (*`pandas.DataFrame`*) – DataFrame storing an AR's records.

`anchor_lats`
1d array, get the latitude coordinates from roughly evenly spaced points from the AR axis.

Type ndarray

`anchor_lons`
1d array, get the longitude coordinates from roughly evenly spaced points from the AR axis.

Type ndarray

`append`(*ar*)
Add new records to the AR track

`backwardHausdorff`(*lats*, *lons*)
Compute backward Hausdorff distance from the lastest record to given axis

Parameters

- **`lats`** (*`ndarray`*) – 1d array, the target axis's latitude coordinates.
- **`lons`** (*`ndarray`*) – 1d array, the target axis's longitude coordinates.

Returns *float* – backward Hausdorff distance from this AR to the given axis.

`coor`
(Nx3) ndarray, (time, lat_centroid, lon_centroid) coordinates of an AR track.

Type ndarray

`duration`
track duration in hours.

Type int

`forwardHausdorff`(*lats*, *lons*)
Compute forward Hausdorff distance from the lastest record to given axis

Parameters

- **`lats`** (*`ndarray`*) – 1d array, the target axis's latitude coordinates.
- **`lons`** (*`ndarray`*) – 1d array, the target axis's longitude coordinates.

Returns *float* – forward Hausdorff distance from this AR to the given axis.

`latest`
the AR record of the latest time point in an AR track.

Type Series

`lats`
1d array, the latitude coordinates of the AR axis in the latest record in an AR's track.

Type ndarray

`lons`
1d array, the longitude coordinates of the AR axis in the latest record in an AR's track.

Type ndarray

`rdp_lats`
1d array, the latitude coordinates of the simplified AR axis in the latest record in an AR's track.

Type ndarray

rdp_lons
1d array, the longitude coordinates of the simplified AR axis in the latest record in an AR's track.

Type ndarray

times
sorted time stamps of an AR track.

Type Series

`AR_tracer.`**`filterTracks`**(*tr_list*, *min_duration*, *min_nonrelax*, *verbose=True*)
Filter tracks

Parameters

- **tr_list** (*list*) – list of AR objects, found tracks.
- **min_duration** (*int*) – min duration in hrs to keep a track.
- **min_nonrelax** (*int*) – min number of non-relaxed records in a track to keep a track.

Keyword Arguments **verbose** (*bool*) – print some messages or not.

Returns *tr_list (list)* – list of AR objects, filtered tracks.

Tracks that are filtered:

- tracks that are too short, controlled by 'min_duration'
- tracks that consist of solely relaxed records.

`AR_tracer.`**`forwardHausdorff`**(*lats1*, *lons1*, *lats2*, *lons2*)
Compute forward Hausdorff distance betweem 2 tracks

Parameters

- **lats1** (*list or 1D array*) – latitudes of track1.
- **lons1** (*list or 1D array*) – longitudes of track1.
- **lats2** (*list or 1D array*) – latitudes of track2.
- **lons2** (*list or 1D array*) – longitudes of track2.

Returns forward Hausdorff distance in km.

`AR_tracer.`**`getAnchors`**(*arr*, *num_anchors=7*)
Get anchor points along from an axis.

Parameters **arr** (*ndarray*) – 1D array from which to sample the anchor points.

Returns *(ndarray)* – 1D array of the sampled anchor points from <arr>.

`AR_tracer.`**`getDistMatrix`**(*tr_list*, *newlats*, *newlons*)
Compute distance matrix among track axis anchors

Parameters

- **tr_list** (*list*) – list of AR objs, existing systems at time t=t.
- **newlats** (*list or 1D array*) – latitudes at t=t+1.
- **newlons** (*list or 1D array*) – longitudes at t=t+1.

Returns

dists (ndarray) –

n*m matrix consisting distances between existing and new tracks. Rows as new records at tnow, columns as existing tracks.

`AR_tracer.`**`matchCenters`**(*tr_list*, *newrec*, *time_gap_allow*, *max_dist_allow*, *track_scheme='simple'*, *isplot=False*, *plot_dir=None*, *verbose=True*)

Match and link nearby centers at 2 consecutive time steps

Parameters

- **`tr_list`** (*`list`*) – list of AR objs, existing systems at time t=t.
- **`newrec`** (*`DataFrame`*) – new center data at time t=t+1.
- **`time_gap_allow`** (*`int`*) – max allowed gap between 2 records, in number of hours.
- **`max_dist_allow`** (*`float`*) – max allowed Hausdorff distance allowed between 2 records, in km.

Keyword Arguments

- **`track_scheme`** (*`str`*) – tracking scheme. 'simple': all tracks are simple
- **`'full'`** (*`paths.`*) – use the network scheme, tracks are connected by their
- **`points.`** (*`joint`*) –
- **`isplot`** (*`bool`*) – create schematic plot or not.
- **`plot_dir`** (*`str`*) – folder to save schematic plot. Only used if isplot=True.
- **`verbose`** (*`bool`*) – print some messages or not.

Returns

tr_list (list) –

list of AR objs, ARs with new matching records appended at the end.

allocated_recs (list): list of ints, ids of new records that are attributed to existing systems during the process.

Matching is based on geo-distances and uses nearest neighbour strategy.

`AR_tracer.`**`plotHD`**(*y1*, *x1*, *y2*, *x2*, *timelabel=None*, *linkflag=''*, *ax=None*, *show=True*)

Plot Hausdorff links

Parameters

- **`y1,x1`** (*`ndarray`*) – 1d array, y, x coordinates of AR axis A.
- **`y2,x2`** (*`ndarray`*) – 1d array, y, x coordinates of AR axis B.

Keyword Arguments

- **`timelabel`** (*`str or None`*) – string of the time stamp. If given, plot as subplot title.
- **`linkflag`** (*`str`*) – a single char to denote the type of linking, used in generated plot. '' for initial linking, 'M' for a merging, 'S' for a splitting.
- **`ax`** (*`plt axis obj`*) – if not give, create a new axis to plot with.
- **`show`** (*`bool`*) – whether to show the figure or not.

`AR_tracer.`**`readCSVRecord`**(*abpath_in*)

Read in individual AR records from .csv file

Parameters **`abpath_in`** (*`str`*) – absolute file path to AR record file.

Returns *ardf (pandas.DataFrame)* – record saved in DataFrame.

New in v2.0.

`AR_tracer.`**`trackARs`**(*record*, *time_gap_allow*, *max_dist_allow*, *track_scheme='simple'*, *isplot=False*, *plot_dir=None*, *verbose=True*)

Track ARs at consecutive time points to form tracks

Parameters

- **`record`** (*`DataFrame`*) – AR records at different time slices.
- **`time_gap_allow`** (*`int`*) – max allowed gap between 2 records, in number of hours.
- **`max_dist_allow`** (*`float`*) – max allowed Hausdorff distance allowed between 2 records, in km.

Keyword Arguments

- **`track_scheme`** (*`str`*) – tracking scheme. 'simple': all tracks are simple
- **`'full'`** (*`paths.`*) – use the network scheme, tracks are connected by their
- **`points.`** (*`joint`*) –
- **`isplot`** (*`bool`*) – whether to create schematic plots of linking.
- **`plot_dir`** (*`str`*) – folder to save schematic plot.
- **`verbose`** (*`bool`*) – print some messages or not.

Returns *finished_list (list)* – list of AR objs. Found tracks.

5.4 Documentation page for utils/funcs.py

Only some functions from this module are documented. The other undocumented functions are intended as private functions, not to be exposed to the user.

Utility functions

Author: guangzhi XU (xugzhi1987@gmail.com) Update time: 2020-07-22 09:27:36.

`utils.funcs.`**`get3DEllipse`**(*t*, *y*, *x*)

Get a binary 3D ellipse structuring element

Parameters

- **`t`** (*`int`*) – ellipse axis length in the t (1st) dimension.
- **`y`** (*`int`*) – ellipse axis length in the y (2nd) dimension.
- **`x`** (*`int`*) – ellipse axis length in the x (3rd) dimension.
- **`that the axis length is half the size of the ellipse`** (*`Note`*) –
- **`that dimension.`** (*`in`*) –

Returns

result (ndarray) –

3D binary array, with 1s side the ellipse defined as (T/t)^2 + (Y/y)^2 + (X/x)^2 <= 1.

`utils.funcs.`**`dLatitude`**(*lats*, *lons*, *R=6371000*, *verbose=True*)

Return a slab of latitudinal increment (meter) delta_y.

Parameters

- **`lats`** (*`ndarray`*) – 1d array, latitude coordinates in degrees.
- **`lons`** (*`ndarray`*) – 1d array, longitude coordinates in degrees.

Keyword Arguments **`R`** (*`float`*) – radius of Earth;

Returns

delta_x (ndarray) –

2d array, latitudinal increments. <var>.

`utils.funcs.`**`dLongitude`**(*lats*, *lons*, *side='c'*, *R=6371000*)

Return a slab of longitudinal increment (meter) delta_x.

Parameters

- **`lats`** (*`ndarray`*) – 1d array, latitude coordinates in degrees.
- **`lons`** (*`ndarray`*) – 1d array, longitude coordinates in degrees.

Keyword Arguments

- **`side`** (*`str`*) –

  'n': northern boundary of each latitudinal band;

  's': southern boundary of each latitudinal band; 'c': central line of latitudinal band;

  —— 'n'

  /—— 'c'

  /_______ 's'

- **`R`** (*`float`*) – radius of Earth.

Returns *delta_x (ndarray)* – 2d array, longitudinal increments.

`utils.funcs.`**`readVar`**(*abpath_in*, *varid*)

Read in netcdf variable

Parameters

- **`abpath_in`** (*`str`*) – absolute file path to nc file.
- **`varid`** (*`str`*) – id of variable to read.

Returns *var (TransientVariable)* – 4d TransientVariable.

NOTE: deprecated, use netCDF4 instead of CDAT.

`utils.funcs.`**`getTimeAxis`**(*times*, *ntime*, *ref_time='days since 1900-01-01'*)

Create a time axis

Parameters

- **`times`** (*`list or tuple or array`*) – array of datetime objs, or strings, giving the time stamps in the format of 'yyyy-mm-dd HH:MM'. It is assumed to be in chronological order. If None, default to create a dummy time axis, with 6-hourly time step, starting from <ref_time>, with a length of <ntime>.

- **ntime** (*int*) – length of the time axis. If <times> is not None, it is checked to insure that length of <times> equals <ntime>. If <times> is None, <ntime> is used to create a dummy time axis with a length of <ntime>.

Keyword Arguments **ref_time** (*str*) – reference time point. If <times> is not None, used to create the numerical values of the resulant time axis with this reference time. If <times> is None, used to create a dummy time axis with this reference time.

Returns

result (list) –

a list of datetime objs. If <times> is not None, the datetime objs are using the provided time stamps. Otherwise, it is a new time series, with 6-hourly time step, starting from <ref_time>, with a length of <ntime>.

New in v2.0.

5.5 Documentation page for utils/plot.py

Only some functions from this module are documented. The other undocumented functions are intended as private functions, not to be exposed to the user.

Plotting Functions.

Author: guangzhi XU (xugzhi1987@gmail.com) Update time: 2020-07-22 09:27:30.

utils.plot.**plot2**(*var*, *method*, *ax*, *legend='global'*, *xarray=None*, *yarray=None*, *title=None*, *latlon=True*, *latlongrid=False*, *fill_color='0.8'*, *legend_ori='horizontal'*, *clean=False*, *iscartopy=True*, *fix_aspect=True*, *verbose=True*)

A helper function for quickly create 2D plots

Parameters

- **var** (*NCVAR or ndarray*) – variable to plot. At least 2D.
- **method** – plotting method, could be an instance of Boxfill, Isofill.
- **ax** – matplotlib axis obj.

Keyword Arguments

- **legend** (*str*) – location of colorbar. Could be: 'global': all subplots share the colorbar of the 1st subplot in figure. or 'local': each subplot in figure uses its own colorbar.
- **xarray** (*ndarray*) – 1d array, the array values for the x-axis. If None, use the int indices for the x-dimension.
- **yarray** (*ndarray*) – 1d array, the array values for the y-axis. If None, use the int indices for the y-dimension.
- **title** (*str*) – title to plot at subtitle. If None, plot only an alphabetical index.
- **latlon** (*bool*) – plot lat/lon axis labels or not.
- **latlongrid** (*bool*) – plot lat/lon grid lines or not.
- **fill_color** – color to fill continent or masked regions.
- **legend_ori** (*str*) – 'horizontal' or 'vertical', colorbar orientation.
- **clean** (*bool*) – if True, omit axis labels, colorbar, subtitle, continents, boundaries etc.. Useful to overlay plots.

- **iscartopy** (*bool*) – plot using cartopy or not. Usually used to force plot as a normal 2d plot instead of geographical plot using cartopy.
- **fix_aspect** (*bool*) – passed to the cartopy plotting function (e.g. contourf()) for control of aspect ratio. NOTE: needs to be deprecated.

`utils.plot.`**`plotAR`**(*ardf*, *ax*, *lonax*)

Helper function to plot the regions and axes of ARs

Parameters

- **ardf** (*pandas.DataFrame*) – table containing AR records.
- **ax** (*matplotlib axis*) – axis to plot onto.
- **lonax** (*ndarray*) – 1d array of the longitude axis the plot is using.

`utils.plot.`**`plotARTrack`**(*arlist*, *latax*, *lonax*, *ax*, *full=False*, *label=None*, *linestyle='solid'*, *marker=None*)

Plot AR tracks

Parameters

- **arlist** (*list*) – list of AR objects to plot.
- **latax,lonax** (*ndarray*) – 1darrays giving latitude- and longitude- coordinates of the plotting domain.
- **ax** (*matplotlib.axis*) – axis to plot onto.

Keyword Arguments

- **full** (*bool*) – if True, plot tracks of an AR from its entire lifecycle. if False, plot only the track of the last time step.
- **label** (*str or None*) – type of label to label tracks. 'id': label with AR id. 'time': label with time stamp. None: don't put label.
- **linestyle** (*str*) – line style to plot the tracks.
- **marker** (*str*) – marker to plot track axes.

class `utils.plot.`**`Boxfill`**(*vars*, *zero=1*, *split=2*, *max_level=None*, *min_level=None*, *ql=None*, *qr=None*, *cmap=None*, *verbose=True*)

`__init__`(*vars*, *zero=1*, *split=2*, *max_level=None*, *min_level=None*, *ql=None*, *qr=None*, *cmap=None*, *verbose=True*)

Return an isofill object with specified color scheme.

Parameters **vars** – one or a list of variables, from which the <minlevel> and <maxlevel> is obtained. If <vars> has more than 1 variables, then function calculates the minimum and maximum of all variables, thus the color legend would be unified for all subplots. Note that if <max_level> and/or <min_level> are given, they will override the maximum/minimal levels derived from <vars>.

Keyword Arguments

- **zero** (*int*) – controls 0 in created <levels>: -1: 0 is NOT allowed to be a level; 1: 0 is permitted to be a level; 2: 0 forced to be a level.
- **split** (*int*) – int, control behavior of negative and positive values 0: Do not split negatives and positives, map onto entire range of [0,1]; 1: split only when vmin<0 and vmax>0, otherwise map onto entire range of [0,1];

  If split is to be applied, negative values are mapped onto first half [0,0.5], and postives onto second half (0.5,1].

2: force split, if vmin<0 and vmax>0, same as <split>==1; If vmin<0 and vmax<0, map onto 1st half [0,0.5]; If vmin>0 and vmax>0, map onto 2nd half (0.5,1].

- **`max_level,min_level`** (*`float`*) – the max/min limits to be plotted out, values outside the range will be grouped into the last level intervals on both ends.
- **`ql,qr`** (*`float`*) – extreme percentiles for the lower and upper boundaries. Could be one of the values in the list:

  percents=[0.001,0.005,0.01,0.025,0.05,0.1]

  E.g. ql=0.001; qr=0.005 means that the 0.1% percentile will be set to the minimum level; 0.005% (from the right most, or 99.95% from the left most) percentil will be set to the maximum level.

  If both <ql> and <min_level> are given, use <min_level>. If both <qr> and <max_level> are given, use <max_level>.
- **`cmap`** – specify a color map. Could be: 1) None, a default blue-white-red (bwr) color map will be created. 2) a matplotlib cmap obj. 3) a string name of a matplotlib cmap obj, to list of few:

  'bwr': blue-white-red 'RdBu': darkred-white-blue 'RdYlBu': red-yellow-white-blue 'RdYlGn': red-yellow-white-green 'spectral': purple-yellow-cyan-blue 'seismic': darkblue-white-darkred 'jet': rainbow darkblue-darkred 'rainbow': rainbow purple-red

  Append '_r' to get the reversed colormap.

Note: <minlevel> and <maxlevel> are better derived using numpy.min() and numpy.max(). MV.min() and MV.max() may have problems. Iso levels are computed using vcs function (mkscale()), and a matplotlib colormap is created (if not given), and the colormap will be changed so positive/negative splits (if required) is achieved.

Update time: 2015-04-27 14:55:33

class `utils.plot.`**`Isofill`**(*vars*, *num=15*, *zero=1*, *split=2*, *max_level=None*, *min_level=None*, *ql=None*, *qr=None*, *cmap=None*, *verbose=True*)

`__init__`(*vars*, *num=15*, *zero=1*, *split=2*, *max_level=None*, *min_level=None*, *ql=None*, *qr=None*, *cmap=None*, *verbose=True*)
Return an isofill object with specified color scheme.

Parameters **`vars`** – one or a list of variables, from which the <minlevel> and <maxlevel> is obtained. If <vars> has more than 1 variables, then function calculates the minimum and maximum of all variables, thus the color legend would be unified for all subplots. Note that if <max_level> and/or <min_level> are given, they will override the maximum/minimal levels derived from <vars>.

Keyword Arguments

- **`Num`** (*`int`*) – is the (maximum) number of isoline levels;
- **`zero`** (*`int`*) – controls 0 in created <levels>: -1: 0 is NOT allowed to be a level; 1: 0 is permitted to be a level; 2: 0 forced to be a level.
- **`split`** (*`int`*) – int, control behavior of negative and positive values 0: Do not split negatives and positives, map onto entire range of [0,1]; 1: split only when vmin<0 and vmax>0, otherwise map onto entire range of [0,1];

If split is to be applied, negative values are mapped onto first half [0,0.5], and postives onto second half (0.5,1].

2: force split, if vmin<0 and vmax>0, same as <split>==1; If vmin<0 and vmax<0, map onto 1st half [0,0.5]; If vmin>0 and vmax>0, map onto 2nd half (0.5,1].

- **`max_level,min_level`** (*`float`*) – the max/min limits to be plotted out, values outside the range will be grouped into the last level intervals on both ends.
- **`ql,qr`** (*`float`*) – extreme percentiles for the lower and upper boundaries. Could be one of the values in the list:

  percents=[0.001,0.005,0.01,0.025,0.05,0.1]

  E.g. ql=0.001; qr=0.005 means that the 0.1% percentile will be set to the minimum level; 0.005% (from the right most, or 99.95% from the left most) percentil will be set to the maximum level.

  If both <ql> and <min_level> are given, use <min_level>. If both <qr> and <max_level> are given, use <max_level>.
- **`cmap`** – specify a color map. Could be: 1) None, a default blue-white-red (bwr) color map will be created. 2) a matplotlib cmap obj. 3) a string name of a matplotlib cmap obj, to list a few:

  'bwr': blue-white-red 'RdBu': darkred-white-blue 'RdYlBu': red-yellow-white-blue 'RdYlGn': red-yellow-white-green 'spectral': purple-yellow-cyan-blue 'seismic': darkblue-white-darkred 'jet': rainbow darkblue-darkred 'rainbow': rainbow purple-red

  Append '_r' to get the reversed colormap.

Note: <minlevel> and <maxlevel> are better derived using numpy.min() and numpy.max(). MV.min() and MV.max() may have problems. Iso levels are computed using vcs function (mkscale()), and a matplotlib colormap is created (if not given), and the colormap will be changed so positive/negative splits (if required) is achieved.

Update time: 2015-04-27 14:55:33

class `utils.plot.`**`Plot2D`**(*var*, *method*, *ax=None*, *xarray=None*, *yarray=None*, *title=None*, *latlon=True*, *latlongrid=False*, *legend='global'*, *legend_ori='horizontal'*, *clean=False*)

`__init__`(*var*, *method*, *ax=None*, *xarray=None*, *yarray=None*, *title=None*, *latlon=True*, *latlongrid=False*, *legend='global'*, *legend_ori='horizontal'*, *clean=False*)

Utility class for 2D plots

Parameters

- **`var`** (*`NCVAR or ndarray`*) – variable to plot. At least 2D.
- **`method`** – plotting method, could be an instance of Boxfill, Isofill.

Keyword Arguments

- **`ax`** – matplotlib axis obj. If None, create a new.
- **`xarray`** (*`ndarray`*) – 1d array, the array values for the x-axis. If None, use the int indices for the x-dimension.

- **yarray** (*ndarray*) – 1d array, the array values for the y-axis. If None, use the int indices for the y-dimension.
- **title** (*str*) – title to plot at subtitle. If None, plot only an alphabetical index.
- **latlon** (*bool*) – plot lat/lon axis labels or not.
- **latlongrid** (*bool*) – plot lat/lon grid lines or not.
- **legend** (*str*) – location of colorbar. Could be: 'global': all subplots share the colorbar of the 1st subplot in figure. or 'local': each subplot in figure uses its own colorbar.
- **legend_ori** (*str*) – 'horizontal' or 'vertical', colorbar orientation. clean (bool): if True, omit axis labels, colorbar, subtitle, continents, boundaries etc.. Useful to overlay plots.

classmethod checkBasemap(*var*, *xarray*, *yarray*)
Check variable should be plotted using basemap or not

getGrid()
Get lat/lon grid info from data

classmethod getSlab(*var*)
Get a 2D slab from variable

Parameters

- **var** (*NCVAR or ndarray*) – nd variable. If is transient variable, try
- **slice its 1st time point. If numpy.ndarray, try to take a slab** (*to*) –
- **its last 2 dimensions.** (*from*) –

Returns *result (ndarray)* – a 2d slab from input <var> to plot.

class utils.plot.**Plot2Cartopy**(*var*, *method*, *ax*, *legend='global'*, *title=None*, *xarray=None*, *yarray=None*, *latlon=True*, *latlongrid=False*, *fill_color='0.8'*, *legend_ori='horizontal'*, *clean=False*, *fix_aspect=False*)

classmethod checkBasemap(*var*, *xarray*, *yarray*)
Check variable should be plotted using basemap or not

getGrid()
Get lat/lon grid info from data

classmethod getSlab(*var*)
Get a 2D slab from variable

Parameters

- **var** (*NCVAR or ndarray*) – nd variable. If is transient variable, try
- **slice its 1st time point. If numpy.ndarray, try to take a slab** (*to*) –
- **its last 2 dimensions.** (*from*) –

Returns *result (ndarray)* – a 2d slab from input <var> to plot.

CHAPTER 6

Github and Contact

The code of this package is hosted at https://github.com/ihesp/IPART.

For any queries, please contact xugzhi1987@gmail.com.

CHAPTER 7

Contributing and getting help

We welcome contributions from the community. Please create a fork of the project on GitHub and use a pull request to propose your changes. We strongly encourage creating an issue before starting to work on major changes, to discuss these changes first.

For help using the package, please post issues on the project GitHub page.

CHAPTER 8

License

license

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[Vincent1993] L. Vincent, "Morphological grayscale reconstruction in image analysis: applications and efficient algorithms," in IEEE Transactions on Image Processing, vol. 2, no. 2, pp. 176-201, April 1993.

Python Module Index

Index