pycomlink

Release 0.3.0

Christian Chwala

Nov 12, 2023

USER GUIDE

A Python library to process commercial microwave link data.

Anaconda Version .. image:: https://anaconda.org/conda-forge/pycomlink/badges/version.svg

target
https://anaconda.org/conda-forge/pycomlink

alt
Anaconda Version

CHAPTER

ONE

PYCOMLINK

A python toolbox for deriving rainfall information from commercial microwave link (CML) data.

1.1 Installation

`pycomlink` is tested with Python 3.9, 3.10 and 3.11. There have been problems with Python 3.8, see https://github.com/pycomlink/pycomlink/pull/120. Many things might work with older version, but there is no support for this.

It can be installed via ``*conda-forge*`<https://conda-forge.org/>`_:

```
$ conda install -c conda-forge pycomlink
```

If you are new to `conda` or if you are unsure, it is recommended to create a new conda environment, activate it, add the conda-forge channel and then install.

Installation via `pip` is also possible:

```
$ pip install pycomlink
```

If you install via `pip`, there might be problems with some dependencies, though. E.g. the dependency `pykrige` may only install if `scipy`, `numpy` and `matplotlib` have been installed before.

To run the example notebooks you will also need the Jupyter Notebook and `ipython`, both also available via `conda` or `pip`.

If you want to clone the repository for developing purposes follow these steps (installation of Jupyter Notebook included):

```
$ git clone https://github.com/pycomlink/pycomlink.git
$ cd pycomlink
$ conda env create environment_dev.yml
$ conda activate pycomlink-dev
$ cd ..
$ pip install -e pycomlink
```

1.2 Usage

The following jupyter notebooks showcase some use cases of `pycomlink`

- Basic example CML processing workflow
- Compare interpolation methods
- Get radar data along CML paths
- Nearby-link approach for rain event detection from RAINLINK
- Compare different WAA methods
- Detect data gaps stemming from heavy rainfall events that cause a loss of connection along a CML

Note that the links point to static versions of the example notebooks. You can run all these notebook online via mybinder if you click on the "launch binder" buttom at the top.

1.3 Features

- Perform all required CML data processing steps to derive rainfall information from raw signal levels:
  - data sanity checks
  - anomaly detection
  - wet/dry classification
  - baseline calculation
  - wet antenna correction
  - transformation from attenuation to rain rate
- Generate rainfall maps from the data of a CML network
- Validate you results against gridded rainfall data or rain gauges networks

Documentation

The documentation is hosted by readthedocs.org: https://pycomlink.readthedocs.io/en/latest/

1.4 Usage

1.4.1 pycomlink

IO

cmlh5_to_xarray

pycomlink.io.cmlh5_to_xarray.**read_cmlh5_file_to_xarray**(*filename*)

read a cmlh5 file and parse data from each cml_id to a xarray dataset

Parameters

filename (*string*) – filename of a cmlh5 file

Returns
list of xarray datasets

Return type
list

csv

Processing

wet_dry

cnn

pycomlink.processing.wet_dry.cnn.**cnn_wet_dry**(*trsl_channel_1*, *trsl_channel_2*, *threshold=None*, *batch_size=100*, *verbose=0*)

Wet dry classification using the CNN based on channel 1 and channel 2 of a CML

Parameters

- **trsl_channel_1** (*iterable of float*) – Time series of received signal level of channel 1
- **trsl_channel_2** (*iterable of float*) – Time series of received signal level of channel 2
- **threshold** (*float or None*) – Threshold between 0 and 1 which has to be surpassed to classify a period as 'wet'. If None, then no threshold is applied and raw wet probabilities in the range of [0,1] are returned.
- **batch_size** (*int*) – Batch size for parallel computing. Set to 1 when using a CPU!
- **verbose** (*int*) – Toggles Keras text output during prediction. Default is off.

Returns
Time series of wet/dry classification

Return type
iterable of int

Note: Implementation of CNN method[1]

References

pycomlink.processing.wet_dry.cnn.**get_model_file_path**()

[1] Polz, J., Chwala, C., Graf, M., and Kunstmann, H.: Rain event detection in commercial microwave link attenuation data using convolutional neural networks, Atmos. Meas. Tech., 13, 3835–3853, https://doi.org/10.5194/amt-13-3835-2020, 2020.

stft

pycomlink.processing.wet_dry.stft.**find_lowest_std_dev_period**(*rsl*, *window_length=600*)

Find beginning and end of dry period

Parameters

- **rsl** (*iterable of float*) – Time series of received signal level
- **window_length** (*int, optional*) – Length of window for identifying dry period (Default is 600)

Returns

- *int* – Index of beginning of dry period
- *int* – Index of end of dry period

pycomlink.processing.wet_dry.stft.**nans**(*shape*, *dtype=<class 'float'>*)

Helper function for wet/dry classification

pycomlink.processing.wet_dry.stft.**stft_classification**(*rsl*, *window_length*, *threshold*, *f_divide*, *t_dry_start=None*, *t_dry_stop=None*, *dry_length=None*, *mirror=False*, *window=None*, *Pxx=None*, *f=None*, *f_sampling=0.016666666666666666*)

Perform wet/dry classification with Rolling Fourier-transform method

Parameters

- **rsl** (*iterable of float*) – Time series of received signal level
- **window_length** (*int*) – Length of the sliding window
- **threshold** (*int*) – Threshold which has to be surpassed to classifiy a period as 'wet'
- **f_divide** (*float*) – Parameter for classification with method Fourier transformation
- **t_dry_start** (*int*) – Index of starting point dry period
- **t_dry_stop** (*int*) – Index of end of dry period
- **dry_length** (*int*) – Length of dry period that will be automatically identified in the provided rsl time series
- **mirror** (*bool (defaults to False)*) – Mirroring values in window at end of time series
- **window** (*array of float, optional*) – Values of window function. If not given a Hamming window function is applied (Default is None)
- **Pxx** (*2-D array of float, optional*) – Spectrogram used for the wet/dry classification. Gets computed if not given (Default is None)
- **f** (*array of float, optional*) – Frequencies corresponding to the rows in Pxx. Gets computed if not given. (Default is None)
- **f_sampling** (*float, optional*) – Sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. (Default is 1/60.0)
- **mirror** (*bool*)

Returns

- *iterable of int* – Time series of wet/dry classification

- *dict* – Dictionary holding information about the classification

Note: Implementation of Rolling Fourier-transform method[2]

References

baseline

pycomlink.processing.baseline.**baseline_constant**(*trsl*, *wet*, *n_average_last_dry=1*)

Build baseline with constant level during a *wet* period

Parameters

- **trsl** (*numpy.array or pandas.Series*) – Transmitted signal level minus received signal level (TRSL) or received signal level or t
- **wet** (*numpy.array or pandas.Series*) – Information if classified index of times series is wet (True) or dry (False). Note that *NaN`s in `wet* will lead to *NaN`s in `baseline* also after the *NaN* period since it is then not clear whether or not there was a change of wet/dry within the *NaN* period.
- **n_average_last_dry** (*int, default = 1*) – Number of last baseline values before start of wet event that should be averaged to get the value of the baseline during the wet event. Note that this values should not be too large because the baseline might be at an expected level, e.g. if another wet event is ending shortly before.

Returns
baseline – Baseline during wet period

Return type
numpy.array

pycomlink.processing.baseline.**baseline_linear**(*rsl*, *wet*, *ignore_nan=False*)

Build baseline with linear interpolation from start till end of *wet* period

Parameters

- **rsl** (*numpy.array or pandas.Series*) –

  Received signal level or transmitted signal level minus received
  signal level
- **wet** (*numpy.array or pandas.Series*) – Information if classified index of times series is wet (True) or dry (False). Note that *NaN`s in `wet* will lead to *NaN`s in `baseline* also after the *NaN* period since it is then not clear wheter there was a change of wet/dry within the *NaN* period.
- **ignore_nan** (*bool*) – If set to True the last wet/dry state before a NaN will be used for deriving the baseline. If set to False, the baseline for any wet period which contains a NaN will be set to NaN for the duration of the wet period. Default is False.

Returns
baseline – Baseline during wet period

[2] Chwala, C., Gmeiner, A., Qiu, W., Hipp, S., Nienaber, D., Siart, U., Eibert, T., Pohl, M., Seltmann, J., Fritz, J. and Kunstmann, H.: "Precipitation observation using microwave backhaul links in the alpine and pre-alpine region of Southern Germany", Hydrology and Earth System Sciences, 16, 2647-2661, 2012

Return type
numpy.array

k_R_relation

pycomlink.processing.k_R_relation.**a_b**(*f_GHz*, *pol*, *approx_type='ITU_2005'*)

Approximation of parameters a and b for k-R power law

Parameters

- **f_GHz** (*int, float, np.array or xr.DataArray*) – Frequency of the microwave link(s) in GHz.
- **pol** (*str, np.array or xr.DataArray*) – Polarization, that is either 'horizontal' for horizontal or 'vertical' for vertical. 'H', 'h' and 'Horizontal' as well as 'V', 'v' and 'Vertical' are also allowed. Must have same shape as f_GHz or be a str. If it is a str, it will be expanded to the shape of f_GHz.
- **approx_type** (*str, optional*) – Approximation type (the default is 'ITU_2005', which implies parameter approximation using a table recommanded by ITU in 2005. An older version of 2003 is available via 'ITU_2003'.)

Returns
a,b – Parameters of A-R relationship

Return type
float

Note: The frequency value must be between 1 Ghz and 100 GHz.

The polarization has to be indicated by 'h' or 'H' for horizontal and 'v' or 'V' for vertical polarization respectively.

Currently only 'ITU' for approx_type is accepted. The approximation makes use of a table recommanded by ITU [4]. There are two versions available, P.838-2 (04/2003) and P.838-3 (03/2005).

References

pycomlink.processing.k_R_relation.**calc_R_from_A**(*A*, *L_km*, *f_GHz=None*, *pol=None*, *a=None*, *b=None*, *a_b_approximation='ITU_2005'*, *R_min=0.1*)

Calculate rain rate from path-integrated attenuation using the k-R power law

Note that either *f_GHz* and *pol* or *a* and *b* have to be provided. The former option calculates the parameters *a* and *b* for the k-R power law internally based on frequency and polarization.

Parameters

- **A** (*float or iterable of float*) – Path-integrated attenuation of microwave link signal
- **L_km** (*float*) – Length of the link in km
- **f_GHz** (*float, np.array, or xr.DataArray optional*) – Frequency in GHz. If provided together with *pol*, it will be used to derive the parameters a and b for the k-R power law.
- **pol** (*string, np.array or xr.DataArray optional*) – Polarization, that is either 'horizontal' for horizontal or 'vertical' for vertical. 'H', 'h' and 'Horizontal' as well as 'V', 'v' and 'Vertical' are also allowed. Has to be provided together with *f_GHz*. It will be used to derive the parameters a and b for the k-R power law. Must have same shape as f_GHz or be a str. If it is a str, it will be expanded to the shape of f_GHz.

- **a** (*float, optional*) – Parameter of A-R relationship
- **b** (*float, optional*) – Parameter of A-R relationship
- **a_b_approximation** (*string*) – Specifies which approximation for the k-R power law shall be used. See the function *a_b* for details.
- **R_min** (*float*) – Minimal rain rate in mm/h. Everything below will be set to zero.

Returns
Rain rate

Return type
float or iterable of float

Note: The A-R and k-R relation are defined as

$$A = kL_{km} = aR^bL_{km}$$

where *A* is the path-integrated attenuation in dB and *k* is the specific attenuation in dB/km.

---

pycomlink.processing.k_R_relation.**calc_R_from_A_min_max**(*Ar_max*, *L*, *f_GHz=None*, *a=None*, *b=None*, *pol='H'*, *R_min=0.1*, *k=90*)

Calculate rain rate from attenuation using the A-R Relationship

Parameters

- **Ar_max** (*float or iterable of float*) – Attenuation of microwave signal (with min/max measurements of RSL/TSL)
- **f_GHz** (*float, optional*) – Frequency in GHz
- **pol** (*string*) – Polarization, default is 'H'
- **a** (*float, optional*) – Parameter of A-R relationship
- **b** (*float, optional*) – Parameter of A-R relationship
- **L** (*float*) – length of the link
- **R_min** (*float*) – Minimal rain rate in mm/h. Everything below will be set to zero.
- **k** (*int, optional*) – number of measurements between two consecutive measurement of rx/tx

Returns
Rain rate

Return type
float or iterable of float

Note: Based on: "Empirical Study of the Quantization Bias Effects in Commercial Microwave Links Min/Max Attenuation Measurements for Rain Monitoring" by OSTROMETZKY J., ESHEL A.

---

min_max

wet_antenna

pycomlink.processing.wet_antenna.**eps_water**(*f_Hz*, *T_K*)

Calculate the dielectric permitiviy of water

Formulas taken from dielectric permittivity of liquid water without salt according to Liebe et al. 1991 Int. J. IR+mm Waves 12(12), 659-675

Based on MATLAB code by Christian Mätzler, June 2002 Cosmetic changes by Christian Chwala, August 2012

Parameters

- **f_Hz** (*array-like*) – Frequency in Hz
- **T_K** (*float*) – Temperature in Kelvin

Returns
eps

Return type
np.complex

pycomlink.processing.wet_antenna.**waa_leijnse_2008**(*R*, *f_Hz*, *T_K=293.0*, *gamma=2.06e-05*, *delta=0.24*, *n_antenna=1.73 + 0.014j*, *l_antenna=0.001*)

Calculate wet antenna attenuation according to Leijnse et al. 2008

Calculate the wet antenna attenuation assuming a rain rate dependent thin flat water film on the antenna following the results from [3]_.

Water film thickness:
l = gamma * R ** delta

Parameters

- **R** (*array-like or scalar*) – Rain rate in mm/h
- **f_Hz** (array-like or scalar (but only either *R* or *f_Hz* can be array)) – Frequency of CML in Hz
- **gamma** (*float*) – Parameter that determines the magnitutde of the water film thickness
- **delta** (*float*) – Parameter that determines the non-linearity of the relation between water film thickness and rain rates
- **n_antenna** (*float*) – Refractive index of antenna material
- **l_antenna** (*float*) – Thickness of antenna cover

Returns
waa – Wet antenna attenuation in dB

Return type
array-like

References

pycomlink.processing.wet_antenna.**waa_leijnse_2008_from_A_obs**(*A_obs*, *f_Hz*, *pol*, *L_km*, *T_K=293.0*, *gamma=2.06e-05*, *delta=0.24*, *n_antenna=1.73 + 0.014j*, *l_antenna=0.001*)

Calculate wet antenna attenuation according to Leijnse et al. 2008

Calculate the wet antenna attenuation from observed attenuation, using the method proposed in[2], assuming a rain rate dependent thin flat water film on the antenna.

The equations proposed in[2] calculate the WAA from the rain rate R. With CML data the rain rates is not directly available. We need to use the observed attenuation to derive the WAA. This is done here by building a look-up-table for the relation between A_obs and WAA, where A_obs is calculated as A_obs = A_rain + WAA. A_rain is derived from the A-R relation for the given CML frequency and length.

Parameters

- **A_obs** (*array-like or scalar*) – Observed attenuation
- **f_Hz** (array-like or scalar (but only either *R* or *f_Hz* can be array)) – Frequency of CML in Hz
- **pol** (*string*) – Polarization of CML. Has to be either 'H' or 'V'.
- **L_km** (*float*) – Lenght of CML in kilometer
- **gamma** (*float*) – Parameter that determines the magnitutde of the water film thickness
- **delta** (*float*) – Parameter that determines the non-linearity of the relation between water film thickness and rain rates
- **n_antenna** (*float*) – Refractive index of antenna material
- **l_antenna** (*float*) – Thickness of antenna cover

Returns
waa – Wet antenna attenuation in dB

Return type
array-like

References

pycomlink.processing.wet_antenna.**waa_pastorek_2021**(*R*, *A_max=14*, *zeta=0.55*, *d=0.1*)

Calculate wet antenna attenuation according to Pastorek et al. 2021 [3] (model denoted "KR-alt" in their study, i.e. a variation of the WAA model suggested by Kharadly and Ross 2001 [4])

Calculate the wet antenna from rain rate explicitly assuming an upper limit A_max.

Parameters

- **A_max** (*upper bound of WAA ("C" in [3])*)
- **R** (*array-like or scalar*) – Rain rate in mm/h
- **zeta** (*power-law parameters*)
- **d** (*power-law parameters*)

[2] H. Leijnse, R. Uijlenhoet, J.N.M. Stricker: "Microwave link rainfall estimation: Effects of link length and frequency, temporal sampling, power resolution, and wet antenna attenuation", Advances in Water Resources, Volume 31, Issue 11, 2008, Pages 1481-1493, https://doi.org/10.1016/j.advwatres.2008.03.004.

Returns
waa – Wet antenna attenuation in dB

Return type
array-like

References

pycomlink.processing.wet_antenna.**waa_pastorek_2021_from_A_obs**(*A_obs*, *f_Hz*, *pol*, *L_km*, *A_max=14*, *zeta=0.55*, *d=0.1*)

Calculate wet antenna attenuation according to Pastorek et al. 2021 [3] (model denoted "KR-alt" in their study, i.e. a variation of the WAA model suggested by Kharadly and Ross 2001 [4])

Calculate the wet antenna from rain rate explicitly assuming an upper limit A_max.

The equation proposed in [3] calculates the WAA from the rain rate R. With CML data the rain rates is not directly available. We need to use the observed attenuation to derive the WAA. This is done here by building a look-up-table for the relation between A_obs and WAA, where A_obs is calculated as A_obs = A_rain + WAA. A_rain is derived from the A-R relation for the given CML frequency and length.

Parameters

- **A_max** (*upper bound of WAA ("C" in [3])*)
- **R** (*array-like or scalar*) – Rain rate in mm/h
- **f_Hz** (array-like or scalar (but only either *R* or *f_Hz* can be array)) – Frequency of CML in Hz
- **pol** (*string*) – Polarisation of CML. Must be either 'H' or 'V'.
- **L_km** (*float*) – Lenght of CML in kilometer
- **zeta** (*power-law parameters*)
- **d** (*power-law parameters*)

Returns
waa – Wet antenna attenuation in dB

Return type
array-like

References

pycomlink.processing.wet_antenna.**waa_schleiss_2013**(*rsl*, *baseline*, *wet*, *waa_max*, *delta_t*, *tau*)

Calculate WAA according to Schleiss et al 2013

Parameters

- **rsl** (*iterable of float*) – Time series of received signal level
- **baseline** (*iterable of float*) – Time series of baseline for rsl
- **wet** (*iterable of int or iterable of float*) – Time series with wet/dry classification information.
- **waa_max** (*float*) – Maximum value of wet antenna attenuation
- **delta_t** (*float*) – Parameter for wet antenna attention model
- **tau** (*float*) – Parameter for wet antenna attenuation model

Returns
Time series of wet antenna attenuation

Return type
iterable of float

Note: The wet antenna adjusting is based on a peer-reviewed publication[1]

References

xarray_wrapper

pycomlink.processing.xarray_wrapper.**xarray_apply_along_time_dim**()

A decorator to apply CML processing function along the time dimension of DataArrays

This will work if the decorated function takes 1D numpy arrays, which hold the CML time series data, as arguments. Additional argument are also handled.

Spatial

coverage

pycomlink.spatial.coverage.**calc_coverage_mask**(*cml_list*, *xgrid*, *ygrid*, *max_dist_from_cml*)

Generate a coverage mask with a certain area around all CMLs

Parameters

- **cml_list** (*list*) – List of Comlink objects
- **xgrid** (*array*) – 2D matrix of x locations
- **ygrid** (*array*) – 2D matrix of y locations
- **max_dist_from_cml** (*float*) – Maximum distance from a CML path that should be considered as covered. The units must be the same as for the coordinates of the CMLs. Hence, if lat-lon is used in decimal degrees, this unit has also to be used here. Note that the different scaling of lat-lon degrees for higher latitudes is not accounted for.

Returns
grid_points_covered_by_cmls – 2D array with size of *xgrid* and *ygrid* with True values where the grid point is within the area considered covered.

Return type
array of bool

[1] Schleiss, M., Rieckermann, J. and Berne, A.: "Quantification and modeling of wet-antenna attenuation for commercial microwave links", IEEE Geoscience and Remote Sensing Letters, 10, 2013

helper

pycomlink.spatial.helper.**haversine**(*lon1*, *lat1*, *lon2*, *lat2*)

Calculate the great circle distance between two points on the earth (specified in decimal degrees)

idw

class pycomlink.spatial.idw.**Invdisttree**(*X*, *leafsize=10*, *stat=0*)

Bases: object

inverse-distance-weighted interpolation using KDTree:

Copied from http://stackoverflow.com/questions/3104781/ inverse-distance-weighted-idw-interpolation-with-python

Usage granted by original author here: https://github.com/scipy/scipy/issues/2022#issuecomment-296373506

invdisttree = Invdisttree(X, z) – data points, values interpol = invdisttree(q, nnear=3, eps=0, p=1, weights=None, stat=0)

interpolates z from the 3 points nearest each query point q; For example, interpol[a query point q] finds the 3 data points nearest q, at distances d1 d2 d3 and returns the IDW average of the values z1 z2 z3

(z1/d1 + z2/d2 + z3/d3) / (1/d1 + 1/d2 + 1/d3) = .55 z1 + .27 z2 + .18 z3 for distances 1 2 3

q may be one point, or a batch of points. eps: approximate nearest, dist <= (1 + eps) * true nearest p: use 1 / distance**p weights: optional multipliers for 1 / distance**p, of the same shape as q stat: accumulate wsum, wn for average weights

How many nearest neighbors should one take ? a) start with 8 11 14 .. 28 in 2d 3d 4d .. 10d; see Wendel's formula b) make 3 runs with nnear= e.g. 6 8 10, and look at the results –

|interpol 6 - interpol 8| etc., or **|f - interpol*|** if you have f(q). I find that runtimes don't increase much at all with nnear – ymmv.

p=1, p=2 ?

p=2 weights nearer points more, farther points less. In 2d, the circles around query points have areas ~ distance**2, so p=2 is inverse-area weighting. For example,

(z1/area1 + z2/area2 + z3/area3) / (1/area1 + 1/area2 + 1/area3) = .74 z1 + .18 z2 + .08 z3 for distances 1 2 3

Similarly, in 3d, p=3 is inverse-volume weighting.

Scaling:

if different X coordinates measure different things, Euclidean distance can be way off. For example, if X0 is in the range 0 to 1 but X1 0 to 1000, the X1 distances will swamp X0; rescale the data, i.e. make X0.std() ~= X1.std() .

A nice property of IDW is that it's scale-free around query points: if I have values z1 z2 z3 from 3 points at distances d1 d2 d3, the IDW average

(z1/d1 + z2/d2 + z3/d3) / (1/d1 + 1/d2 + 1/d3)

is the same for distances 1 2 3, or 10 20 30 – only the ratios matter. In contrast, the commonly-used Gaussian kernel exp(- (distance/h)**2) is exceedingly sensitive to distance and to h.

interpolator

class pycomlink.spatial.interpolator.**IdwKdtreeInterpolator**(*nnear=8*, *p=2*, *exclude_nan=True*, *max_distance=None*)

Bases: *PointsToGridInterpolator*

class pycomlink.spatial.interpolator.**OrdinaryKrigingInterpolator**(*nlags=100*, *variogram_model='spherical'*, *variogram_parameters={'nugget': 0.1, 'range': 1, 'sill': 0.9}*, *weight=True*, *n_closest_points=None*, *exclude_nan=True*, *backend='C'*)

Bases: *PointsToGridInterpolator*

class pycomlink.spatial.interpolator.**PointsToGridInterpolator**

Bases: `object`

PointsToGridInterpolator class docstring

Util

maintenance

exception pycomlink.util.maintenance.**DeprecatedWarning**

Bases: `UserWarning`

pycomlink.util.maintenance.**deprecated**(*instructions*)

Flags a method as deprecated. Args:

instructions: A human-friendly string of instructions, such
as: 'Please migrate to add_proxy() ASAP.'

Note:
Taken from https://gist.github.com/kgriffs/8202106

temporal

pycomlink.util.temporal.**aggregate_df_onto_DatetimeIndex**(*df*, *new_index*, *method*, *label='right'*, *new_index_tz='utc'*)

Aggregate a DataFrame or Series using a given DatetimeIndex

Parameters

- **df** (*pandas.DataFrame*) – The dataframe that should be reindexed
- **new_index** (*pandas.DatetimeIndex*) – The time stamp index on which *df* should be aggregated
- **method** (*numpy function*) – The function to be used for aggregation via *DataFrame.groupby('new_time_ix').agg(method)*

- **label** (*str {'right', 'left'}, optional*) – Which side of the aggregated period to take the label for the new index from
- **new_index_tz** (*str, optional*) – Defaults to 'utc'. Note that if *new_index* already has time zone information, this kwarg is ignored

Returns
df_reindexed

Return type
pandas.DataFrame

Validation

stats

class pycomlink.validation.stats.**RainError**(*pearson_correlation*, *coefficient_of_variation*, *root_mean_square_error*, *mean_absolute_error*, *R_sum_reference*, *R_sum_predicted*, *R_mean_reference*, *R_mean_predicted*, *false_wet_rate*, *missed_wet_rate*, *false_wet_precipitation_rate*, *missed_wet_precipitation_rate*, *rainfall_threshold_wet*, *N_all_pairs*, *N_nan_pairs*, *N_nan_reference_only*, *N_nan_predicted_only*)

Bases: *RainError*

namedtuple with the following rainfall performance measures:

pearson_correlation:
Pearson correlation coefficient

coefficient_of_variation:
Coefficient of variation following the definition in[1]

root_mean_square_error:
Root mean square error

mean_absolute_error:
Mean absolute error

R_sum_reference:
Precipitation sum of the reference array (mm)

R_sum_predicted:
Precipitation sum of the predicted array (mm)

R_mean_reference:
Precipitation mean of the reference array (mm)

R_mean_predicted:
Precipitation mean of the predicted array (mm)

false_wet_rate:
Rate of cml wet events when reference is dry

missed_wet_rate:
Rate of cml dry events when reference is wet

false_wet_precipitation_rate:
Mean precipitation rate of false wet events

missed_wet_precipitation_rate:
Mean precipitation rate of missed wet events

rainfall_threshold_wet:
Threshold separating wet/rain and dry/non-rain periods

N_all_pairs:
Number of all reference-predicted pairs

N_nan_pairs:
Number of reference-predicted pairs with at least one NaN

N_nan_reference_only:
Number of NaN values in the reference array

N_nan_predicted_only:
Number of NaN values in predicted array

References

class pycomlink.validation.stats.**WetDryError**(*false_wet_rate*, *missed_wet_rate*, *matthews_correlation*, *true_wet_rate*, *true_dry_rate*, *N_dry_reference*, *N_wet_reference*, *N_true_wet*, *N_true_dry*, *N_false_wet*, *N_missed_wet*, *N_all_pairs*, *N_nan_pairs*, *N_nan_reference_only*, *N_nan_predicted_only*)

Bases: *WetDryError*

namedtuple with the following wet-dry performance measures:

false_wet_rate:
Rate of cml wet events when reference is dry

missed_wet_rate:
Rate of cml dry events when reference is wet

matthews_correlation:
Matthews correlation coefficient

true_wet_rate:
Rate of cml wet events when the reference is also wet

true_dry_rate:
Rate of cml dry events when the reference is also dry

N_dry_reference:
Number of dry events in the reference

N_wet_reference:
Number of wet events in the reference

N_true_wet:
Number of cml wet events when the reference is also wet

N_true_dry:
Number of cml dry events when the reference is also dry

N_false_wet:
Number of cml wet events when the reference is dry

N_missed_wet:
Number of cml dry events when the reference is wet

N_all_pairs:
Number of all reference-predicted pairs

N_nan_pairs:
Number of reference-predicted pairs with at least one NaN

N_nan_reference_only:
Number of NaN values in reference array

N_nan_predicted_only:
Number of NaN values in predicted array

class pycomlink.validation.stats.**WetError**(*false*, *missed*)

Bases: tuple

false

Alias for field number 0

missed

Alias for field number 1

pycomlink.validation.stats.**calc_rain_error_performance_metrics**(*reference*, *predicted*, *rainfall_threshold_wet*)

Calculate performance metrics for rainfall estimation

This function calculates metrics and statistics relevant to judge the performance of rainfall estimation. The calculation is based on two arrays with rainfall values, which should contain rain rates or rainfall sums. Beware that the units of *R_sum...* und *R_mean...* will depend on your input. The calculation does not take any information on temporal resolution or aggregation into account!

Parameters

- **reference** (*float array-like*) – Rainfall reference
- **predicted** (*float array-like*) – Predicted rainfall
- **rainfall_threshold_wet** (*float*) – Rainfall threshold for which *reference* and *predicted* are considered *wet* if value >= threshold. This threshold only impacts the results of the performance metrics which are based on the differentiation between *wet* and *dry* periods.

Returns
RainError

Return type
named tuple

References

https://en.wikipedia.org/wiki/Matthews_correlation_coefficient https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/metrics/regression.py#L184 https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/metrics/regression.py#L112 Overeem et al. 2013: www.pnas.org/cgi/doi/10.1073/pnas.1217961110

pycomlink.validation.stats.**calc_wet_dry_performance_metrics**(*reference*, *predicted*)

Calculate performance metrics for a wet-dry classification

This function calculates metrics and statistics relevant to judge the performance of a wet-dry classification. The calculation is based on two boolean arrays, where *wet* is True and *dry* is False.

Parameters

- **reference** (*boolean array-like*) – Reference values, with *wet* being True

- **predicted** (*boolean array-like*) – Predicted values, with *wet* being True

Returns
WetDryError

Return type
named tuple

pycomlink.validation.stats.**calc_wet_error_rates**(*df_wet_truth*, *df_wet*)

validator

class pycomlink.validation.validator.**GridValidator**(*lats=None*, *lons=None*, *values=None*, *xr_ds=None*)

Bases: *Validator*

get_time_series(*cml*, *values*)

plot_intersections(*cml*, *ax=None*)

resample_to_grid_time_series(*df*, *grid_time_index_label*, *grid_time_zone=None*)

class pycomlink.validation.validator.**PointValidator**(*lons*, *values*)

Bases: *Validator*

get_time_series(*cml*, *values*)

class pycomlink.validation.validator.**Validator**

Bases: object

calc_stats(*cml*, *time_series*)

pycomlink.validation.validator.**calc_wet_dry_error**(*df_wet_truth*, *df_wet*)

Visualisation

interactive_maps

maps

1.4.2 What's New

v0.3.10

Enhancements

- added more flexible handling of input for *a_b()* function (by maxmargraf in PR #141)
- updaetd WAA example notebook with WAA example with method from Pastorek (by cchwala in PR #136)

Maintenance

- Refactoring of nearby-link approach code (by maxmargraf in PR #139)

Bug fixes

- Fixed some errors in the nearby-link approach code (by maxmargraf in PR #139)
- Fixed bug in *read_cmlh5_file_to_xarray()* (by maxmargraf in PR #138)

v0.3.9

Enhancements

- Added IDW and Kriging interpolation comparison noteboook (by cchwala in PR #132)

Maintenance

- Updated README with current list of example notebooks
- Removed pinning of scipy and pandas versions (by cchwala in PR #132)

Bug fixes

- Added test for Kriging and fixed wrong naming of IDW test (by cchwala in PR #132)

v0.3.8

Enhancements

- Extended implementation of "nearby wet-dry approach" and added some fixes and more test (by maxmargraf in PR #129)

v0.3.7

Maintenance

- Change absolute imports of pycomlink to relative imports (by cchwala in PR #119)
- Drop Python 3.7 and Python 3.8 support in CI (by cchwala in PR #120)
- Replaced depreciated np.complex and np.bool (by maxmargraf in PR #122, #123 and #124)

Bug fixes

- Fix problems related to missing pol argument in example workflow (by cchwala in PR #116)

v0.3.6

Enhancements

- Implemented "nearby wet-dry approach" from RAINLINK (by maxmargraf in PR #113)
- Updated ITU recommendation for k-R power law to version from 2005 (by nblettner in PR #110)

Maintenance

- remove parameters from model.compile() in wet-dry CNN method (by cchwala in PR #112)

v0.3.5

Enhancements

- Added *bottelneck* as dependency to allow *max_gap* keyword in *xarray.DataArray.interpolate* (by maxmargraf in PR #99)
- Added WAA model after Pastorek et al. 2021 (by nblettern via direct commit to master branch)
- Added function and example notebook for blackout gap detection (by maxmargraf in PR #101)
- Refactore and extended grid intersction code, now using sparse matrices (by cchwala in PR #106)

Maintenance

- Pinned scipy to < 1.9 because of problem in pykrige

Bug fixes

- Fixed problems in IDW code (by cchwala in PR #105)

v0.3.4

Bug fixes

- Reference files are now included in conda-forge build (PR #97)

Maintenance

- *tensorflow-gpu* dependency (which seems to be obsolete) was removed from requirements (PR #97)

v0.3.3

Enhancements

- Added xarray-wrapper for WAA Leijnse and updated WAA example notebook (by cchwala in PR #82)
- Add CNN-based anomaly detection for CML data (by Glawion in PR#87)
- xarray wrapper now uses *xr.apply_ufunc* to apply processing functions along time dimension, instead of looping over the *channel_id* dimension. This should be a lot more flexible. (by cchwala in PR #89)

Bug fixes

- Fixed problem with xarray_wrapper for calc_R_from_A (by cchwala in PR #89)

Maintenance

- Move CI from Travis to Github Actions (by maxmargraf in PR #85)
- Add readthedocs and zenodo badge to README (by maxmargraaf in PR #85)

v0.3.2

- minor fix to include example NetCDF data in source distribution (by cchwala in PR #84)

v0.3.1

- small update to how the dependencies are defined
- testing for Python verions 3.7, 3.8 and 3.9

v0.3.0

Backward Incompatible Changes

- The old API using *pycomlink.core.Comlink* objects has been removed. All processing functions now work with *xarray.DataArrays* or pure *numpy.ndarray*. Most of the original functions and notebooks from v0.2.x do not work anymore, but the basic parts have already been refactored so that the full processing chain, from raw CML data to rainfall fields works in v0.3.0.

Enhancements

- Added new example notebook for basic processing workflow (by cchwala in PR #77)
- Added new example data (by maxmargraf in PR #75)
- started docs from scratch with working integration to readthedocs (by jpolz in PR #74)
- read data from cmlh5 files to *xarray.Dataset* (by maxmargraf in PR #68)
- Added functions to perform wet-dry classification with trained CNN (by jpolz in PR #67)
- applied black formatting to codebase (by nblettner in PR #66)
- make repo runnable via mybinder (by jpolz in PR #64)

v0.2.4

- Added WAA calculation and test for method proposed by Leijnse et al 2008
- Added function to calculate WAA directly from A_obs for Leijnse et al 2008 method.
- Added WAA example notebook
- Added function to derive attenuation value *A_min_max* from min/max CML measurements (these measurements periodically provide the min and max value over a defined time period, typically 15 minutes). (by DanSereb in PR #37 and #45)
- Added function to derive rain rate *R* from *A_min_max* (by DanSereb in PR #37 and #45)
- Added example notebook with simple comparison of processing of "instantaneous" and "min-max" CML data (by DanSereb in PR #37 and #45)

v0.2.3

Bug fixes

- Added missing kwarg for polarization in *calc_A* in *Processor*. Before, *calc_A* always used the default polarization for the A-R relation which leads to rain rate overestimation!
- Changed reference values in test for Ordinary Kriging interpolator, because *pykrige v1.4.0* seems to produce slightly different results than *v1.3.1*

v0.2.2

Enhancements

- Codebase is Python 3 now, keeping backwards compatibility to Python 2.7 via using the *future* module.
- min-max CML data can now be written to and read from cmlh5. Standard column names are *tx_min*, *tx_max*, *rx_min* and *rx_max*. When reading from cmlh5 without specifying dedicated column names, the function tries out the standard column names for min-max and instantaneous. If it does not find any match it will print an error message.
- Added example file with min-max data for 75 CMLs. This dataset is derived from the existing example dataset of 75 CMLs with instantaneous measurements.
- Added example notebook comparing min-max and instantaneous CML data

- Added TravisCI and Codecov and increased the test coverage a little
- Extended functionality for *append_data*. A maximum length or maximum allowed age for the data can be specified
- More options for interpolation. Added option to pass *max_distance* for IDW and Added option for resampling in *Interpolator* (instead of just doing hourly means of variable *R*)
- Interpolated fields are now always transformed into an *xarray.Dataset*. The *Dataset* is also stored as attribute if the *Interpolator* object
- Improved grid intersection calculation in validator

Bug fixes

- *t_start* and *t_stop* have not been taken into account in the main interpolation loop
- Fix: Catching *LinAlgError* in Kriging interpolation

v0.2.1

Minor update

- removing geopandas dependecy
- update MANIFEST.in to include notebooks and example data in pypi releases

v0.2.0

Backward Incompatible Changes

- Complete rewrite of interpolator classes. The old interpolator class *spatial.interpol.Interpolator()* is depreciated. New interpolator base classes for IDW and Kriging have been added together with a convenience inteprolator for CML data. Usage is showcased in a new example notebook.
- **Some old functionality has moved to separate files.**
  - resampling to a given *DatetimeIndex* is now availabel in *util.temporal* and will be removed from *validatoin.validator.Validation()* class soon.
  - calculation of wet-dry error is now in module *validation.stats*
  - calculation of spatial coverage with CMLs was moved to function *spatial.coverage.calc_coverage_mask()*.
  - error metric for performance evaluation of wet-dry classification is now in *validation.stats*. Errors are now returned with meaningful names as namedtuples. *validation.validator.calc_wet_dry_error()* is depreciated now.

Enhancements

- Read and write to and from multiple cmlh5 files (#12)
- Improved *NaN* handling in *wet* indicator for baseline determination
- Speed up of KDtreeIDW using numba and by reusing previously calculated variables
- Added example notebook for baseline determination
- Added data set of 75 CMLs (with fake locations)
- Added example notebook to show usage of new interpolator classes
- Added decorator to mark depreciated code

Bug fixes

- *setup.py* now reads all packages subdirectories correctly
- Force integers for shape in *nans* helper function in *stft* module
- Always use first value of *dry_stop* timestamp list in *stft* module. The old code did not work anyway for a list with length = 1 and would have failed if *dry_stop* would have been a scalar value. Now we assume that we always get a list of values (which should be true for *mlab.find*.

v0.1.1

No info for older version…

PYTHON MODULE INDEX

p

INDEX

N

O

P

R

S

V

W

X