pymicra Documentation

Release 0.2.0

Tomás Chor

Jul 21, 2016

This package was designed to make it easier to work with micrometeorological data. Pymicra is currently fully written in python and it's aimed towards aggregating all of functionality that is commonly needed to work with micrometeorological data into one productivity-enhancing tool.

The package is extensively (almost entirely) based on Pandas, mostly the `pandas.DataFrame` class. We use Pint for units control and (generally) Numpy or Scipy for some numerical functions not contained in Pandas.

CHAPTER

ONE

REQUIRED PACKAGES

Most required packages already come with python. However, packages that generally have to be manually installed beforehand are:

- Pandas
- Pint
- Numpy
- Scipy
- setuptools (for installation only)

CHAPTER

TWO

MAIN FEATURES

Currently, this is a sum up of what Pymicra does:

- Reading, separating and understanding micrometeorological data in virtually any column-separated ASCII format (thanks to pandas).
- Quality control methods (max and min values check, spikes, reverse-arrangement test and etc..
- Rotation of coordinates (2D).
- Detrending of data in the most common ways (block averages, moving averages and polynomial detrending).
- Correction of sensor drift.
- Automatic calculation of most auxiliary variables based on actual measurements (air density, dry air density, etc.).
- Calculation of spectra and cross-spectra.
- Calculation fluxes and characteristic scales.
- WPL correction.
- Provide all the common constants generally used in micrometeorology.
- Plus all native features of Pandas (interpolation, resampling, grouping, statistical tests, slicing, handling of missing data and etc.)

Contents:

Introduction

section{Presentation}

This material is meant to be a quick introduction to the Python tool for Micrometeorological Analyses (Pymicra). Since the package is currently in an experimental state, it is impossible for me to include 100% of the functions contained in Pymicra in this guide because new functions are constantly being created. However, the most important classes, functions and capabilities are described here and the user is encouraged to interactively use the Pymicra and refer to the docstrings in the code to learn more.

Pymicra is a Python package that was created to condense many of the knowledge of the micrometeorological community in one single fast-to-implement software that is freely available to anyone. Because of that, I made an effort to include a detailed docstring along with every function and class, so if you import the package and run texttt{help(pymicra.timeSeries)}, for example, you will see a detailed description of what the function texttt{timeSeries} does and so on for any other functions. It is a good idea to import Pymicra using the IPython terminal and use its autocomplete function by hitting tab after typing texttt{pymicra.} to see and explore the available functions.

Since Pymicra is meant to be a community package, improvements, suggestions of improvement, and any kind of feedback are highly appreciated. The code is available at href{https://github.com/tomchor/pymicra}{its github page} and any contact can be made through there (possibly creating an issue) or via e-mail.

section{Description and suggestions}

In order for the user to program fast and effectively (and to reduce the time it takes me to write its code), Pymicra was written on top of the href{http://pandas.pydata.org/}{Pandas package}, so that it is faster to run the same code using Pymicra than it is running pure Python. As a consequence, Pymicra makes extensive use of Pandas' DataFrame class, which is a very useful 2-D data structure optimized for performance and for timestamp-indexed data.

It is possible to use Pymicra without having to be familiar with Pandas, but because Pymicra depends on Pandas, I suggest at the very least that the user take a quick look at a Pandas tutorial (href{http://pandas.pydata.org/pandas-docs/stable/10min.html}{this one for example}) so that one can be familiarized with the many functionalities that Pandas offers in order to take full advantage of Pymicra.

Installation

> **Warning:** The commands written here assume you are running a Ubuntu-based distribution of Linux. Although the basic steps should be similar for all Linux distributions, you should adapt the specific commands to your system in case you are using any other Linux distro.

In order to install Pymicra the `setuptools` Python package should be installed. If you don't have it installed already you can install it with `sudo apt install python-setuptools` or `sudo pip install setuptools`.

Download the package an unpack in somewhere. Then open a terminal and move to the directory created, whose name should be `pymicra`. Then run `sudo python setup.py install`. This should be enough to install the package.

First download the zipped file at the href{github.com/tomchor/pymicra}{Pymicra github page} and unzip it somewhere. Then, using a terminal, move to the unzipped directory and run the setup file (texttt{setup.py}) with root access: texttt{sudo python setup.py install}. For this to work you must have setuptools installed, which can be promptly done using texttt{sudo apt-get install python-setuptools}. This should successfully install Pymicra. The only requirements are Pandas and Pint. Version 0.17 of Pandas is suggested, but it should work fine will any distribution from 0.13 up to 0.17.1 (0.18 is not supported because of changed in the rolling functions API).

Although fairly general, I have tested the setup program on a limited number of computers so far, so it is possible that an error occurs depending on the version of some auxiliary packages you have installed. If that happens and the installation fails for some reason, please contact me through email or creating a github issue detailing your problem and I will improve the setup file.

Alternatively, while the installation cannot be done, you can add the package manually to your Python path by adding the following lines to the beginning of your Python code:

begin{lstlisting} import sys sys.path.append('/full/path/to/pymicra') end{lstlisting}

and then proceed to import Pymicra normally.

To remove Pymicra, the easiest way is to use pip (`sudo apt-get install python-pip`) with the command `sudo pip uninstall pymicra`.

Notation

Pymicra uses a specific notation to name each one of its columns. This notation is extremely important, because it is by these labels that Pymicra knows which variable is in each column, so we advise at the very least a quick look at this chapter to understand the basics of naming.

The notation can be checked by opening the IPython terminal (because of its autocomplete function) and entering

begin{terminal} begin{alltt} import pymicra dir(pymicra.notation) end{alltt} end{terminal}

You can change Pymicra's notation at any time by altering the attributes of texttt{pymicra.notation}. Some examples are

Getting started

Reading datafiles

The easiest way to read data files is using a Pymicra class called texttt{dataloggerConfig} along with the function texttt{timeSeries}. First, let's explain what the texttt{dataloggerConfig} class does. This class holds most of the configurations inherent to the datalogger that actually influence the file output (such as columns separator), which variable is in each column, variables units, frequency, file headers, etc..

You can create a texttt{dataloggerConfig} object manually by passing each of these informations as a keyword to the texttt{dataloggerConfig} function, but the easiest method is to create a texttt{.dlc} file (meaning datalogger configuration), store it somewhere and create a texttt{dataloggerConfig} object by reading it every time you work with files with that same configuration.

Consider the example texttt{.dlc} file below, which will be explained next.

lstinputlisting{ex_itaipu.dlc}

First of all, every .dlc file is writen in Python syntax, so it has to be able to actually be run on python. The description is optional and the texttt{first_time_skip} and texttt{date_connector} keywords are generally not necessary, so you'll rarely have to use them. They can be omitted from the file.

The most important keyword is texttt{varNames}. Since Pymicra works with labels for its data, its best if all the names of the variables are properly writen, preferably following the default Pymicra notation. Let look at how to write parts of the timestamp first.

The columns that contain parts of the date have to have their name matching Python's href{https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior}{date format string directive}. This is useful in case you want to index your data by timestamp, which is a huge advantage in some cases (check out what Pandas can do with href{http://pandas.pydata.org/pandas-docs/stable/timeseries.html}{timestamp-indexed data}). If you don't wish to work with timestamps and want to work only by line number in each file, you can ignore there columns.

As for the physical quantities, it is strongly advised to follow Pymicra's notation, which is described in Pymicra's README.md file and explained in Chapter ref{chap:notation}. In the above example, which follows the Pymicra notation, u, v and w are the three wind components, texttt{theta_v} stands for the virtual temperature, and texttt{mrho_h2o} stands for the molar density of H2O. If it were the mass density the name would have been texttt{rho_h2o}, according to the notation.

The texttt{date_cols}

The texttt{frequency}, in Hertz, is the frequency of the data collection. Useful mostly when taking the spectrum or cospectrum.

The texttt{columns} separator is what separates one column from the other. Generaly it is one character, such as a comma or a whitespace. A special case happens is if the columns are separated by whitespaces of varying length. In that case just put ‘‘whitespace’‘.

The keyword texttt{header_lines} is an int, or list of ints saying which rows are headers, starting from zero. So if the first two rows of the file are a header, the texttt{header_lines} in this case should be texttt{[0, 1]}.

The tt{filename_format}

The tt{date_connector}

The tt{first_time_skip}

With this file ready, it is easy to read any datafile. Consider the example below

EXAMPLE

In it we passed the path of a file to read and the tt{dataloggerConfig} object containing the configuration of the file. The function tt{timeSeries} reads the file according to the options provided and returns a DataFrame that is put into the tt{data} variable. Printing tt{data}, in this case, would print the following on screen.

DATA PRINT

Notice that every variable defined in our datalogger configuration file appears in the data variable.

chapter{Viewing and manipulating data}

To view and manipulate data you have to follow Pandas's DataFrame rules. For that we suggest that the user visit a Pandas tutorial. However, I'll explain some main ideas here for the sake of completeness.

chapter{Extracting fluxes}

Pymicra's auto-generated docs

Subpackages

Submodules

pymicra.constants module

Defines some useful constants

pymicra.core module

class `pymicra.core.`**`Notation`**`(`**args*, ***kwargs*`)`

Bases: `object`

This creates an object that holds the default notation for pymicra. Example of usage:

notation = notation()

fluctuations_of_co2_conc = notation.concentration % notation.fluctuations % notation.co2

You should be careful with the order. The last argument should not have any '%' symbols or you'll get a "TypeError: not all arguments converted during string formatting" message.

Methods

`build``()`

This useful method builds the full notation based on the base notation

`concentration` = **'conc_%s'**

`cospectrum` = **'Co_%s_%s'**

`cross_spectrum` = **'X_%s_%s'**

`density` = **'rho_%s'**

`fluctuations` = **"%s'"**

`flux_of` = **'F_%s'**

`mass_concentration` = **'conc_%s'**

`mass_density` = **'rho_%s'**

`mass_mixing_ratio` = **'r_%s'**

`mean` = **'%s_mean'**

mixing_ratio = 'r_%s'

molar_concentration = 'mconc_%s'

molar_density = 'mrho_%s'

molar_mixing_ratio = 'mr_%s'

quadrature = 'Qu_%s_%s'

spectrum = 'Sp_%s'

star = '%s_star'

std = '%s_std'

class pymicra.core.**dataloggerConfig**(**args*, ***kwargs*)

Bases: object

This class defines a specific configuration of a datalogger output file

from_file: str path of .dlc file (datalogger configuration file) to read from. This will ignore all other keywords.

varNames: list of strings or dict If a list: should be a list of strings with the names of the variables. If the variable is part if the date, then it should be provided as a datetime directive, so if the columns is only the year, its name must be *%Y* and so forth. While if it is the date in YYYY/MM/DD format, it should be *%Y/%m/%d*. For more info see https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior If a dict: the keys should be the numbers of the columns and the items should follow the rules for a list.

date_cols: list of ints should be indexes of the subset of varNames that corresponds to the variables that compose the timestamp. If it is not provided the program will try to guess by getting all variable names that have a percentage sign (%).

date_connector: string generally not really necessary. It is used to join and then parse the date_cols.

columns_separator: string used to assemble the date. Should only be used if the default char creates conflict. If the file is tabular-separated then this should be "whitespace".

header_lines: int up to which line of the file is a header. See pandas.read_csv header option.

first_time_skip: int how many units of frequency the first line of the file is offset (generally zero).

filename_format: string tells the format of the file with the standard notation for date and time and with variable parts as "?". E.g. if the files are 56_20150101.csv, 57_20150102.csv etc filename_format should be:

??_%Y%m%d.csv

this is useful primarily for the quality control feature.

units: dictionary very important: a dictionary whose keys are the columns of the file and whose items are the units in which they appear.

description: string brief description of the datalogger configuration file.

class pymicra.core.**fileConfig**(**args*, ***kwargs*)

Bases: object

This class defines a specific configuration of a data file

from_file: str path of .cfg file (configuration file) to read from. This will ignore all other keywords.

variables: list of strings or dict If a list: should be a list of strings with the names of the variables. If the variable is part if the date, then it should be provided as a datetime directive, so if the columns is only the year, its name must be *%Y* and so forth. While if it is the date in YYYY/MM/DD format, it should be *%Y/%m/%d*. For more info see https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior If a dict: the keys should be the numbers of the columns and the items should follow the rules for a list.

date_cols: list of ints should be indexes of the subset of varNames that corresponds to the variables that compose the timestamp. If it is not provided the program will try to guess by getting all variable names that have a percentage sign (%).

date_connector: string generally not really necessary. It is used to join and then parse the date_cols.

columns_separator: string used to assemble the date. If the file is tabular-separated then this should be "whitespace".

header_lines: int or list up to which line of the file is a header. See pandas.read_csv header option.

filename_format: string tells the format of the file with the standard notation for date and time and with variable parts as "?". E.g. if the files are 56_20150101.csv, 57_20150102.csv etc filename_format should be:

??_%Y%m%d.csv

this is useful primarily for the quality control feature.

units: dictionary very important: a dictionary whose keys are the columns of the file and whose items are the units in which they appear.

description: string brief description of the datalogger configuration file.

varNames: DEPRECATED use variables now.

Methods

check_consistency()
Checks consistency of fileConfig Currently only checks every key of self.units dictionary agaisnt values of variables dict.

get_date_cols()
Guesses what are the columns that contain the dates by searching for percentage signs in them

class pymicra.core.**myData**(*df*, *dic*)
Bases: object

Attempt to create a myData object

class pymicra.core.**siteConfig**(**args*, ***kwargs*)
Bases: object

Keeper of the configurations and constants of an experiment. (such as height of instruments, location, canopy height and etc)

pymicra.data module

pymicra.data.**bulkCorr**(*data*)
Bulk correlation coefficient according to Cancelli, Dias, Chamecki. Dimensionless criteria for the production of... doi:10.1029/2012WR012127

pymicra.data.**crossSpectra**(*data*, *frequency=10*, *notation=None*, *anti_aliasing=True*)
Calculates the spectrum for a set of data

Parameters data: pandas.DataFrame or pandas.Series

dataframe with one (will return the spectrum) or two (will return to cross-spectrum) columns

frequency: float

frequency of measurement of signal to pass to numpy.fft.rfftfreq

anti_aliasing: bool

whether or not to apply anti-aliasing according to Gobbi, Chamecki & Dias, 2006 (doi:10.1029/2005WR004374)

notation: notation object

notation to be used

`pymicra.data.`**`detrend`**`(`**args*, ***kwargs*`)`

Returns the detrended fluctuations of a given dataset

Parameters data: pandas.DataFrame, pandas.Series

dataset to be detrended

how: string

how of average to apply. Currently {'movingmean', 'movingmedian', 'block', 'linear', 'poly'}.

rule: pandas offset string

the blocks for which the trends should be calculated in the block and linear type

window: pandas date offset string or int

if moving mean/median is chosen, this tells us the window size to pass to pandas. If int, this is the number of points used in the window. If string we will to guess the number of points from the index. Small windows (equivalent to 1min approx) work better when using rollingmedian.

block_func: str, function

how to resample in block type. Default is mean but it can be any numpy function that returns a float. E.g, median.

degree: int

degree of polynomial fit (only if how=='linear' or how=='polynomial')

Returns:

out: pandas.DataFrame or pandas.Series

`pymicra.data.`**`reverse_arrangement`**`(`*array*, *points_number=None*, *alpha=0.05*, *verbose=False*`)`

Performs the reverse arrangement test according to Bendat and Piersol - Random Data - 4th edition, page 96

Parameters array: np.array, list, tuple, generator

array which to test for the reverse arrangement test

points_number: integer

number of chunks to consider to the test. Maximum is the length of the array. If it is less, then the number of points will be reduced by application of a mean

alpha: float

Significance level for which to apply the test

WARNING! This fuction approximates table A.6 from Bendat&Piersol as a normal distribution.

This may no be true, since they do not express which distribution they use to construct

their table. However, in the range 9<N<101, this approximation is as good as 5% at N=10

and 0.1% at N=100.

Still not adapted for dataframes

`pymicra.data.`**`rotate2D`**(*data*, *notation=None*)

Parameters data: pandas DataFrame

the dataFrame to be rotated

notation: notation object

a notation object to know which are the wind variables

`pymicra.data.`**`spectra`**(**args*, ***kwargs*)

Calculates the cross-spectra for a set of data

Parameters data: pandas.DataFrame or pandas.Series

dataframe with more than one columns

frequency: float

frequency of measurement of signal to pass to numpy.fft.rfftfreq

anti_aliasing: bool

whether or not to apply anti-aliasing according to Gobbi, Chamecki & Dias, 2006 (doi:10.1029/2005WR004374)

`pymicra.data.`**`spectrum`**(*data*, *frequency=10*, *anti_aliasing=False*, *outname=None*)

Calculates the spectrum for a set of data

Parameters data: pandas.DataFrame

dataframe with one (will return the spectrum) or two (will return to cross-spectrum) columns

frequency: float

frequency of measurement of signal to pass to numpy.fft.rfftfreq

anti_aliasing: bool

whether or not to apply anti-aliasing according to Gobbi, Chamecki & Dias, 2006 (doi:10.1029/2005WR004374)

outname: str

name of the output column

`pymicra.data.`**`trend`**(**args*, ***kwargs*)

Wrapper to return the trend given data. Can be achieved using a moving avg, block avg or polynomial fitting

Parameters data: pandas.DataFrame or pandas.Series

the data whose trend wee seek.

how: string

how of average to apply. Currently {'movingmean', 'movingmedian', 'block', 'linear'}.

rule: string

pandas offset string to define the block in the block average. Default is "10min".

window: pandas date offset string or int

if moving mean/median is chosen, this tells us the window size to pass to pandas. If int, this is the number of points used in the window. If string we will to guess the number of points from the index. Small windows (equivalent to 1min approx) work better when using rollingmedian.

block_func: str, function

how to resample in block type. Default is mean but it can be any numpy function that returns a float. E.g, median.

degree: int

degree of polynomial fit (only if how=='linear' or how=='polynomial')

Returns:

out: pandas.DataFrame or pandas.Series

pymicra.decorators module

`pymicra.decorators.`**`autoassign`**(*function*) → method

autoassign(*argnames) -> decorator autoassign(exclude=argnames) -> decorator

allow a method to assign (some of) its arguments as attributes of 'self' automatically. E.g.

```
>>> class Foo(object):
...     @autoassign
...     def __init__(self, foo, bar): pass
...
>>> breakfast = Foo('spam', 'eggs')
>>> breakfast.foo, breakfast.bar
('spam', 'eggs')
```

To restrict autoassignment to 'bar' and 'baz', write:

@autoassign('bar', 'baz') def method(self, foo, bar, baz): ...

To prevent 'foo' and 'baz' from being autoassigned, use:

@autoassign(exclude=('foo', 'baz')) def method(self, foo, bar, baz): ...

`pymicra.decorators.`**`pdgeneral`**(*convert_out=True*)

`pymicra.decorators.`**`pdgeneral_in`**(*func*)

`pymicra.decorators.`**`pdgeneral_io`**(*func*)

If the input is a series transform it to a dtaframe then transform the output from dataframe back into a series. If the input is a series and the output is a one-element series, transform it to a float.

Currently the output functionality works only when the output is one variable, not an array of elements.

pymicra.io module

Author: Tomas Chor Date: 2015-08-07 ——————————-

`pymicra.io.`**`dataset`**(**args, **kwargs*)

`pymicra.io.`**`readDataFile`**(*fname, variables=None, only_named_cols=True, **kwargs*)

Author: Tomas Chor

Parameters kwargs: dict

dictionary with kwargs of pandas' read_csv function see http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html for more detail

variables: list or dict

list or dictionary containing the names of each variable in the file (if dict, the keys must be ints)

Returns dataFrame: pandas.DataFrame object

`pymicra.io.`**`readDataFiles`**(*flist, verbose=0, **kwargs*)

Author: Tomas Chor Reads data from a list of files

`pymicra.io.`**`readUnitsCsv`**(*filename, **kwargs*)

Reads a csv file in which the first line is the name of the variables and the second line contains the units

`pymicra.io.read_dlc(dlcfile)`
Reads datalogger configuration file

WARNING! When defining the .dlc note that by default columns that are enclosed between doublequotes will appear without the doublequotes. So if your file is of the form :

"2013-04-05 00:00:00", .345, .344, ...

Then the .dlc should have: varNames=['%Y-%m-%d %H:%M:%S','u','v']. This is the default csv format of CampbellSci dataloggers. To disable this feature, you should parse the file with read_csv using the kw: quoting=3.

`pymicra.io.read_fileConfig(dlcfile)`
Reads metadata configuration file

WARNING! When defining the .config file note that by default columns that are enclosed between doublequotes will appear without the doublequotes. So if your file is of the form :

"2013-04-05 00:00:00", .345, .344, ...

Then the .dlc should have: variables = {0:'%Y-%m-%d %H:%M:%S'}. This is the default csv format of CampbellSci dataloggers. To disable this feature, you should parse the file with read_csv using the kw: quoting=3.

`pymicra.io.read_site(sitefile)`
Reads .site configuration file, which holds siteConfig definitions

The .site should have definitions as regular python syntax (in meters!): measurement_height = 10 canopy_height = 5 displacement_height = 3 roughness_length = 1.0

sitedile: str path to .site file

`pymicra.io.timeSeries(flist, datalogger, parse_dates=True, verbose=False, read_data_kw={}, parse_dates_kw={}, clean_dates=True, return_units=True, only_named_cols=True)`
Creates a micrometeorological time series from a file or list of files.

pymicra.methods module

`pymicra.methods.binwrapper(self, clean_index=True, **kwargs)`
Method to return binned data from a dataframe using the function classbin

pymicra.physics module

Author: Tomas Chor

Module that contains physical functions for general use

TO DO LIST:

- ADD GENERAL SOLAR ZENITH CALCULATION
- ADD FOOTPRINT CALCULATION?

`pymicra.physics.CtoK(T)`
Return temp in Kelvin given temp T in Celsius

`pymicra.physics.R_moistAir(q)`
Calculates the gas constant for umid air from the specific humidity q

`pymicra.physics.airDensity_from_theta(data, units, notation=None, inplace=True, use_means=False, theta=None, theta_unit=None)`
Calculates moist air density using theta measurements

`pymicra.physics.`**`airDensity_from_theta_v`**(*data*, *units*, *notation=None*, *inplace=True*, *use_means=False*)
Calculates moist air density using p = rho R_dry T_virtual

`pymicra.physics.`**`dewPointTemp`**(*theta*, *e*)
Calculates the dew point temperature. theta has to be in Kelvin and e in kPa

`pymicra.physics.`**`dryAirDensity_from_p`**(*data*, *units*, *notation=None*, *inplace=True*)
Calculates dry air density NEEDS IMPROVEMENT REGARDING HANDLING OF UNITS

`pymicra.physics.`**`latent_heat_water`**(*T*)
Calculates the latent heat of evaporation for water

Receives T in Kelvin and returns the latent heat in J/g

`pymicra.physics.`**`perfGas`**(*p=None*, *rho=None*, *R=None*, *T=None*, *gas=None*)
Returns the only value that is not provided in the ideal gas law

P.S.: I'm using type to identify None objects because this way it works againt pandas objects

`pymicra.physics.`**`ppxv2density`**(*ser*, *T*, *p*, *units*, *solute=None*)
ser should be a series! concentration in ser should be ppmv, which will be transformed to g/m3 p should be in kPa!

`pymicra.physics.`**`satWaterPressure`**(*T*, *unit='kelvin'*)
Returns the saturated water vapor pressure according eq (3.97) of Wallace and Hobbes, page 99.

e0, b, T1 and T2 are constants specific for water vapor

`pymicra.physics.`**`solarZenith`**(*date*, *lat=-3.13*, *lon=-60.016667*, *lon0=-63.0*, *negative=False*, *dr=None*)
Calculates the solar zenith angle at any given day

needs validation and needs to work without lon0

`pymicra.physics.`**`theta_from_theta_s`**(*data*, *units*, *notation=None*, *return_df=True*)
From Schotanus, Nieuwstadt, de Bruin; DOI 10.1007/BF00164332

theta_s = theta (1 + 0.51 q) (1 - (vn/c)**2)**0.5 theta_s ~ theta (1 + 0.51 q)

`pymicra.physics.`**`theta_from_theta_v`**(*data*, *units*, *notation=None*, *return_df=True*)
theta_v ~ theta (1 + 0.61 q)

`pymicra.physics.`**`theta_std_from_theta_v_fluc`**(*1 + 0.61 q*)

pymicra.tests module

`pymicra.tests.`**`check_RA`**(*data*, *detrend=True*, *detrend_kw={'how': 'linear'}*, *RAT_vars=None*, *RAT_points=50*, *RAT_significance=0.05*)
Performs the Reverse Arrangement Test in each column of data

`pymicra.tests.`**`check_limits`**(*data*, *tables*, *max_percent=1.0*, *replace_with='interpolation'*)
Checks dataframe for lower and upper limits. If found, they are substituted by the linear trend of the run. The number of faulty points is also checked for each column against the maximum percentage of accepted faults max_percent

`pymicra.tests.`**`check_maxdif`**(*data*, *tables*, *detrend=True*, *detrend_kw={'how': 'movingmean', 'window': 900}*)
Check the maximum and minimum differences between the fluctuations of a run.

`pymicra.tests.`**`check_nans`**(*data*, *max_percent=0.1*, *replace_with='interpolation'*)
Checks data for NaN values

`pymicra.tests.`**`check_numlines`**(*fname*, *numlines=18000*, *falseverbose=False*)
Checks length of file against a correct value. Returns False is length is wrong and True if length is right

fname: string path of the file to check

numlines: int correct number of lines that the file has to have pymicra.tests.**check_replaced**(*replaced*, *max_count=180*)

Sums and checks if the number of replaced points is larger than the maximum accepted

pymicra.tests.**check_spikes**(*data*, *chunk_size='2min'*, *detrend=True*, *detrend_kw={'how': 'linear'}*, *visualize=False*, *vis_col=1*, *max_consec_spikes=3*, *cut_func=<function <lambda>>*, *replace_with='interpolation'*, *max_percent=1.0*)

Applies spikes-check according to Vickers and Mahrt (1997)

pymicra.tests.**check_stationarity**(*data*, *tables*, *detrend=False*, *detrend_kw={'how': 'movingmean', 'window': 900}*, *trend=True*, *trend_kw={'how': 'movingmedian', 'window': '1min'}*)

Check difference between the maximum and minimum values of the run trend agaisnt an upper-limit. This aims to flag nonstationary runs

pymicra.tests.**check_std**(*data*, *tables*, *detrend=False*, *detrend_kw={'how': 'linear'}*, *chunk_size='2min'*, *falseverbose=False*)

Checks dataframe for columns with too small of a standard deviation

pymicra.util module

pymicra.util.**correctDrift**(*drifted*, *correct_drifted_vars=None*, *correct=None*, *get_fit=True*, *write_fit=True*, *fit_file='correctDrift_linfit.params'*, *apply_fit=True*, *show_plot=False*, *return_plot=False*, *units={}*, *return_index=False*)

pymicra.util.**qcontrol**(*files*, *datalogger_config*, *read_files_kw={'parse_dates': False, 'only_named_cols': False, 'clean_dates': False}*, *accepted_nans_percent=1.0*, *accepted_spikes_percent=1.0*, *accepted_bound_percent=1.0*, *max_replacement_count=180*, *file_lines=None*, *begin_date=None*, *end_date=None*, *nans_test=True*, *maxdif_detrend=True*, *maxdif_detrend_kw={'how': 'movingmean', 'window': 900}*, *maxdif_trend=True*, *maxdif_trend_kw={'how': 'movingmedian', 'window': 600}*, *std_detrend=True*, *std_detrend_kw={'how': 'movingmean', 'window': 900}*, *RAT_detrend=True*, *RAT_detrend_kw={'how': 'linear'}*, *spikes_detrend=True*, *spikes_detrend_kw={'how': 'linear'}*, *lower_limits={}*, *upper_limits={}*, *spikes_test=True*, *visualize_spikes=False*, *spikes_vis_col='u'*, *spikes_func=<function <lambda>>*, *replace_with='interpolation'*, *max_consec_spikes=3*, *chunk_size=1200*, *std_limits={}*, *dif_limits={}*, *RAT=False*, *RAT_vars=None*, *RAT_points=50*, *RAT_significance=0.05*, *trueverbose=False*, *falseverbose=True*, *falseshow=False*, *trueshow=False*, *trueshow_vars=None*, *outdir='quality_controlled'*, *summary_file='qcontrol_summary.csv'*, *replaced_report=None*, *full_report=None*)

Function that applies various tests quality control to a set of datafiles and re-writes the successful files in another directory. A list of currently-applied tests is found below in order of application. The only test available by default is the spikes test. All others depend on their respective keywords.

date check: files outside a date_range are left out (end_date and begin_date keywords)

lines test: checks each file to see if they have a certain number of lines. Files with a different number of lines fail this test. Active this test by passing the file_lines keyword.

NaNs test: checks for any NaN values. NaNs are replaced with interpolation or linear trend. If the percentage of NaNs is greater than accepted_nans_percent, run is discarded. Activate it by passing nans_test=True.

boundaries test: runs with values in any column lower than a pre-determined lower limit or higher than a upper limits are left out. Activate it by passing a lower_limits or upper_limits keyword.

spikes test: runs with more than a certain percetage of spikes are left out. Activate it by passing a spikes_test keyword. Adjust the test with the spikes_func visualize_spikes, spikes_vis_col, max_consec_spikes, accepted_spikes_percent and chunk_size keywords.

replacement count test: checks the total amount of points that were replaced (including NaN, boundaries and spikes test) against the max_replacement_count keyword. Fails if any columns has more replacements than that.

standard deviation (STD) check: runs with a standard deviation lower than a pre-determined value (generally close to the sensor precision) are left out. Activate it by passing a std_limits keyword.

maximum difference test: runs whose trend have a maximum difference greater than a certain value are left out. This excludes non-stationary runs. Activate it by passing a dif_limits keyword.

reverse arrangement test (RAT): runs that fail the reverse arrangement test for any variable are left out. Activate it by passing a RAT keyword.

`pymicra.util.`**`separateFiles`**(*files, dlconfig, outformat='out_%Y-%m-%d_%H:%M.csv', outdir='', verbose=False, firstflag='.first', lastflag='.last', save_ram=False, frequency='30min', quoting=0, use_edges=False*)

Separates files into (default) 30-minute smaller files. Useful for output files such as the ones by Campbell Sci, that can have days of data in one single file.

Module contents

Pymicra - Python Micrometeorology Analysis tool

Author Tomas Chor

Date of start 2015-08-15

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

S

T