Pymicra Documentation

Release 0.2.1

Tomás Chor

2016-08-06

Pymicra is a Python package designed to make it easier to work with micrometeorological datasets. It is aimed at improving the productivity and allowing us to focus on the micrometeorological issues, rather than programming ones.

Please check out the Github page and the documentation.

Here's a quick (incomplete!) list 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 measurements (air density, dry air density, etc.).
- Calculation of spectra and cross-spectra.
- Calculation fluxes and characteristic scales with or without WPL correction.
- Provide common constants generally used in atmospheric sciences.
- Plus all native features of Pandas (interpolation, resampling, grouping, statistical tests, slicing, handling of missing data and etc.).

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

INTRODUCTION

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. By doing that I hope to have made every function, class etc. pretty self-explanatory, both within the auto-generated docs (thanks to Sphinx) and by importing the package and running `help(pymicra.timeSeries)`, for example.

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 its Github page and any contact can be made through there (possibly creating an Github issue) or via email (tomaschor [at] gmail.com).

Quick notes

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 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 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.

Contributing

Currently, this project has only one part-time developer. This makes it hard to feed Pymicra with up-to-date information/routines as the state-of-the-art micrometeorology evolves quickly. The ideal scenario is one where micrometeorologists in the community not only use Pymicra, but also contribute to it, whether it's by adding to the code, finding bugs, investing ideas and etc. Thus, I have made an effort to not only document the routines/functions in the docstring, but also detail them with comments throughout the code. This was done specially to make it easier for other people to contribute.

If you want to contribute with code you can either create a Github account and contact me via the Github page by creating an issue/pull request, or you can contact me directly by email (tomaschor [at] gmail.com). Furthermore, if you have some useful routines laying around (or even some routines with a procedure you recently created and published) but aren't sure how to make them Pymicra-compatible, let me know by email and we'll work together to adapt it.

Cheers

Tomás Chor.

CHAPTER

TWO

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 OS.

Most of the required packages already come with python, such as `datetime` or `os`. The packages that may not come as default are:

- Pandas (recommended 0.17.1)
- Pint (0.7.2)
- Numpy
- Scipy
- setuptools (for installation only)

Note: Version 0.17.1 of Pandas is suggested, but it should work fine with any distribution from 0.13 up to 0.17.1. However, version 0.18 upwards is not currently supported because of a change in the rolling functions API.

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

Once `setuptools` is installed, download the package from the Github page, unpack it somewhere, then run `setup.py` on a terminal with the `install` directive. Assuming the file is downloaded into the Downloads directory:

```
cd ~/Downloads
unzip pymicra-master.zip
cp pymicra-master
sudo python setup.py install
```

This should successfully install Pymicra. Note that the `pymicra-master` may be different depending on whether you download directly from the master branch, the dev branch or a specific release on Github.

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, please contact me through email or creating a Github issue detailing your problem and I will try to improve the setup file accordingly. Alternatively, you may also try to manually install versions 1.11.0 of Numpy and 0.17.0 of Scipy and then running `sudo python setup.py install` again.

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

CHAPTER

THREE

GETTING STARTED

This "Getting started" tutorial is a brief introduction to Pymicra. This is in no way supposed to be a complete representation of everything that can be done with Pymicra.

In this tutorial we use some example data and refer to some example python scripts that can be downloaded here. These data and scripts are from a measurement campaign in a very small island (about 20 meters across) in a large artificial lake. At the time of these measurements the island was almost completely immersed into about 5 cm of water. Please feel free to explore both the example data and the example programs, as well as modify the programs for your own learning process!

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. You can check the default notation with

```
In [1]: %%capture
   ...: import pymicra as pm
   ...: print(pm.notation)
   ...:
```

The output is too long to be reproduced here, but on the left you'll see the full name of the variables (which corresponds to a notation namespace/attribute) and on the right you'll see the default notation for that variable.

We recommend to use the default notation for the sake of simplicity, however, you can change Pymicra's notation at any time by altering the attributes of `pm.notation`. For example, by default the notation for the mean is `'%s_mean'`, and every variable follows this base notation:

```
In [2]: pm.notation.mean
Out[2]: '%s_mean'

In [3]: pm.notation.mean_u
Out[3]: 'u_mean'

In [4]: pm.notation.mean_h2o_mass_concentration
Out[4]: 'conc_h2o_mean'
```

To change this, you have to change the `mean` notation and then re-build the whole notation with the `build` method:

```
In [5]: pm.notation.mean = 'm_%s'

In [6]: pm.notation.build()

In [7]: pm.notation.mean_u
Out[7]: 'm_u'

In [8]: pm.notation.mean_h2o_mass_concentration
Out[8]: 'm_conc_h2o'
```

```
In [9]: pm.notation.h2o='v'

In [10]: pm.notation.build()

In [11]: pm.notation.mean_h2o_mass_concentration
Out[11]: 'm_conc_v'
```

If you just want to change the notation of one variable, but not the full notation, just don't re-build. For example:

```
In [12]: pm.notation.mean_co2_mass_concentration = 'c_m'

In [13]: pm.notation.mean_co2_mass_concentration
Out[13]: 'c_m'

In [14]: pm.notation.mean_h2o_mass_concentration
Out[14]: 'm_conc_v'
```

It is important to note that this changes the notation used throughout every Pymicra function. If, however, you want to use a different notation in a specific part of the program (in one specific function for example) you can create a Notation object and pass it to the function, such as

```
In [15]: mynotation = pm.Notation()

In [16]: mynotation.co2='c'

In [17]: mynotation.build()

In [18]: fluxes = pm.eddyCovariance(data, units, notation=mynotation) # For example
```

In the example above the default Pymicra notation is left untouched, and a separate notation is defined which is then used in a Pymicra function separately.

Creating file configurations file

The easiest way to read data files is using a `fileConfig` object. This object holds the configuration of the data files so you can just call this object when reading these files. To make it easier, Pymicra prefers to read this configurations from a file. That way you can write the configurations for some data files once, store it into a configuration file and then use it from then on every time you want to read those data files. That is what Pymicra calls a "file configuration file", or "config file" for short. From that file, Pymicra can create a `pymicra.fileConfig` object. Consider, for example, the config file below

```
description='datalogger configuration file for a lake. Located at examples/lake.
↪config'

variables={
0:'%Y-%m-%d',
1:'%H:%M:%S.%f',
2:'u',
3:'v',
4:'w',
5:'theta_v',
6:'mrho_h2o',
7:'mrho_co2',
8:'p',
9:'theta'}

units={
'u':'m/s',
'v':'m/s',
```

```
'w':'m/s',
'theta_v':'celsius',
'mrho_co2':'mmol/m**3',
'mrho_h2o':'mmol/m**3',
'p':'kPa',
'theta':'celsius'
}

columns_separator=','
frequency=20
header_lines=None

filename_format='%Y%m%d-%H%M.csv'
date_cols = [0, 1]
```

First of all, note that the `.config` file is written in Python syntax, so it has to be able to actually be run on python. This has to be true for all `.config` files.

Furthermore, the extension of the file does not matter. We adopt the `.config` extension for clarity, but it could be anything else.

The previous config file describes the data files in the directory `../examples/ex_data/`. Here's an example of one such file for comparison:

```
2013-11-08,10:00:00.000000,2.375,-5.206,-0.103,27.06,1238.0,14.675,99.19,30.43,-0.
↪303,-0.274,-0.269,-0.261
2013-11-08,10:00:00.050000,2.4930000000000003,-5.098,-0.018000000000000002,27.
↪12,1196.0,14.409,99.199,30.43,-0.308,-0.275,-0.271,-0.263
2013-11-08,10:00:00.100000,2.263,-5.114,0.014,27.11,1220.0,14.636,99.102,30.43,-0.
↪306,-0.277,-0.273,-0.263
2013-11-08,10:00:00.150000,2.21,-5.235,-0.012,27.11,1238.0,14.688,99.154,30.43,-0.
↪308,-0.277,-0.273,-0.264
2013-11-08,10:00:00.200000,2.158,-5.174,-0.112,27.12,1174.0,14.476,99.154,30.44,-0.
↪31,-0.277,-0.273,-0.264
2013-11-08,10:00:00.250000,2.334,-5.279,-0.092,27.1,1195.0,14.671,99.154,30.43,-0.
↪308,-0.278,-0.273,-0.265
2013-11-08,10:00:00.300000,2.396,-5.2970000000000015,0.005,27.15,1198.0,14.669,99.
↪154,30.43,-0.309,-0.279,-0.272,-0.264
2013-11-08,10:00:00.350000,2.494,-5.246,0.039,27.13,1197.0,14.722,99.154,30.44,-0.
↪311,-0.279,-0.273,-0.264
2013-11-08,10:00:00.400000,2.263,-5.317,-0.079,27.12,1202.0,14.709,99.154,30.43,-0.
↪311,-0.279,-0.275,-0.265
2013-11-08,10:00:00.450000,2.135,-5.176,-0.036000000000000004,27.08,1202.0,14.
↪731,99.154,30.44,-0.314,-0.279,-0.275,-0.267
```

Note that not all columns of this file are described. Columns that are not described are also read but are discarded by default. You can change that using `only_named_columns=False` in the `timeSeries` function.

We obtain the config object with

```
In [19]: fconfig = pm.fileConfig('../examples/lake.config')

In [20]: print(fconfig)
<pymicra.fileConfig>
datalogger configuration file for a lake. Located at examples/lake.config
```

Each variable defined in this file works as a keyword, since it can also be input manually when calling `pymicra.fileConfig()`. Thus, for more information, you can also use `help(pymicra.fileConfig)`. Now we explain the keywords one by one. In the next section we will explain how to use this object for reading a data file.

description

The description is optional. It's a string that serves only to better identify the config file you're dealing with. It might useful for storage purposes and useful when printing the config object.

variables

The most important keyword is `variables`. This is a python dictionary where each key is a column and its corresponding value is the variable in that column. Note that we are using here the default notation to indicate which variable is in which column. If a different notation is to be used here, then you will have to define a new notation in your program (refer back to *Notation* for that).

Note: From this point on, for simplicity, we will assume that the default notation is used.

It is imperative that the columns be named accordingly. For example, measuring H_2O contents in mmol/m^3 is different from measuring it in g/m^3 or mg/g. The first is a molar density (moles per volume), the second is a mass density (mass per volume) and the third is a mass concentration (mass per mass). In the default notation these are indicated by the names `'mrho_h2o'`, `'rho_h2o'` and `'conc_h2o'`, respectively, and Pymicra needs to know which one is which.

Columns that contain parts of the timestamp have to have their name matching Python's date format string directive, which themselves are the 1989 version default C standard format dates, which is common in many platforms.

This is useful only 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 timestamp-indexed data) but Pymicra can also work well without this. If you don't wish to work with timestamps and want to work only by line number in each file, you can ignore these columns and indicate that you don't want to parse dates. In fact, parsing of dates makes Pymicra a lot slower. Reading a file parsing its dates is about 5.5 times slower than reading the same file without parsing any dates!

units

The `units` keyword is also very important. It tells Pymicra in which units each variable is being measured. Units are handled by Pint, so for more details on how to define the units please refer to their documentation. Suffices to say here that the format of the units are pretty intuitive. Some quick remarks are

- prefer to define units unambiguously (`'g/(m*(s**2))'` is generally preferred to `'g/m/s**2'`, although both will work).
- to define that a unit is dimensionless, `'1'` will not work. Define it as `'dimensionless'` or `'g/g'` and so on.
- if one variable does not have a unit (such as a sensor flag), you don't have to include that variable.
- the keys of `units` **should exactly match** the values of `variables`.

columns_separator

The `columns_separator` keyword is what it sounds: what separates one column from the other. Generally it is one character, such as a comma. A special case happens is if the columns are separated by whitespaces of varying length, or tabs. In that case it should be `"whitespace"`.

frequency

The `frequency` keyword is the frequency of the data collection in Hertz.

header_lines

The keyword `header_lines` tells us which of the first lines are part of the file header. If there is no header then is should be `None`. If there are header lines than it should be a list or int. For example, if the first two lines of the file are part of a header, it should be `[0,1]`. If it were the 4 first lines, `[0,1,2,3]` (`range(4)` would also be acceptable).

Header lines are not used by Pymicra and are therefore skipped.

filename_format

The `filename_format` keyword tells Pymicra how the data files are named.

date_cols

The `date_cols` keyword is optional. It is a list of integers that indicates which of the columns are a part of the timestamp. If it's not provided, then Pymicra will assume that columns whose names have the character "%" in them are part of the date and will try to parse them. If the default notation is used, this should always be true.

Reading data

To read a data file or a list of data files we use the function `timeSeries` along with a config file. Let us use the config file defined in the previous subsection with one of the data file it describes:

```
In [21]: fname = '../examples/ex_data/20131108-1000.csv'

In [22]: fconfig = pm.fileConfig('../examples/lake.config')

In [23]: data, units = pm.timeSeries(fname, fconfig, parse_dates=True)

In [24]: print(data)
                            u      v      w  theta_v  mrho_h2o  mrho_co2       p␣
↪theta
Timestamp                                                                        ␣
↪
2013-11-08 10:00:00.000  2.375 -5.206 -0.103    27.06      1238    14.675  99.190␣
↪30.43
2013-11-08 10:00:00.050  2.493 -5.098 -0.018    27.12      1196    14.409  99.199␣
↪30.43
2013-11-08 10:00:00.100  2.263 -5.114  0.014    27.11      1220    14.636  99.102␣
↪30.43
2013-11-08 10:00:00.150  2.210 -5.235 -0.012    27.11      1238    14.688  99.154␣
↪30.43
2013-11-08 10:00:00.200  2.158 -5.174 -0.112    27.12      1174    14.476  99.154␣
↪30.44
2013-11-08 10:00:00.250  2.334 -5.279 -0.092    27.10      1195    14.671  99.154␣
↪30.43
2013-11-08 10:00:00.300  2.396 -5.297  0.005    27.15      1198    14.669  99.154␣
↪30.43
2013-11-08 10:00:00.350  2.494 -5.246  0.039    27.13      1197    14.722  99.154␣
↪30.44
2013-11-08 10:00:00.400  2.263 -5.317 -0.079    27.12      1202    14.709  99.154␣
↪30.43
2013-11-08 10:00:00.450  2.135 -5.176 -0.036    27.08      1202    14.731  99.154␣
↪30.44
...                        ...    ...    ...      ...       ...       ...     ...␣
↪  ...
2013-11-08 10:59:59.500  4.951 -4.584  0.420    28.03      1261    14.772  99.102␣
↪32.08
```

```
2013-11-08 10:59:59.550  5.057 -4.436  0.492    28.00      1181    14.718  99.138 ␣
↪32.07
2013-11-08 10:59:59.600  5.145 -4.424  0.409    28.10      1216    14.889  99.112 ␣
↪32.08
2013-11-08 10:59:59.650  5.282 -4.038  0.448    28.03      1198    14.485  99.112 ␣
↪32.06
2013-11-08 10:59:59.700  5.065 -4.453  0.424    28.11      1184    14.578  99.138 ␣
↪32.07
2013-11-08 10:59:59.750  5.262 -4.703  0.126    27.98      1264    14.929  99.138 ␣
↪32.08
2013-11-08 10:59:59.800  5.323 -4.882  0.242    27.95      1229    14.258  99.138 ␣
↪32.07
2013-11-08 10:59:59.850  5.344 -5.119  0.457    27.96      1198    14.962  99.102 ␣
↪32.07
2013-11-08 10:59:59.900  5.281 -5.261  0.599    28.09      1231    14.615  99.112 ␣
↪32.07
2013-11-08 10:59:59.950  5.235 -4.801  0.362    28.02      1211    14.682  99.164 ␣
↪32.08

[72000 rows x 8 columns]
```

Note that `data` is a `pandas.DataFrame` object which contains the whole data available in the datafile with each column being a variable. Since we indicated that we wanted to parse the dates with the option `parse_dates=True`, each row has its respective timestamp. If, otherwise, we were to ignore the dates, the result would be a integer-indexed dataset:

```
In [25]: data2, units = pm.timeSeries(fname, fconfig, parse_dates=False)

In [26]: print(data2)
           u      v      w  theta_v  mrho_h2o  mrho_co2       p  theta
0      2.375 -5.206 -0.103    27.06      1238    14.675  99.190  30.43
1      2.493 -5.098 -0.018    27.12      1196    14.409  99.199  30.43
2      2.263 -5.114  0.014    27.11      1220    14.636  99.102  30.43
3      2.210 -5.235 -0.012    27.11      1238    14.688  99.154  30.43
4      2.158 -5.174 -0.112    27.12      1174    14.476  99.154  30.44
5      2.334 -5.279 -0.092    27.10      1195    14.671  99.154  30.43
6      2.396 -5.297  0.005    27.15      1198    14.669  99.154  30.43
7      2.494 -5.246  0.039    27.13      1197    14.722  99.154  30.44
8      2.263 -5.317 -0.079    27.12      1202    14.709  99.154  30.43
9      2.135 -5.176 -0.036    27.08      1202    14.731  99.154  30.44
...      ...    ...    ...      ...       ...       ...     ...    ...
71990  4.951 -4.584  0.420    28.03      1261    14.772  99.102  32.08
71991  5.057 -4.436  0.492    28.00      1181    14.718  99.138  32.07
71992  5.145 -4.424  0.409    28.10      1216    14.889  99.112  32.08
71993  5.282 -4.038  0.448    28.03      1198    14.485  99.112  32.06
71994  5.065 -4.453  0.424    28.11      1184    14.578  99.138  32.07
71995  5.262 -4.703  0.126    27.98      1264    14.929  99.138  32.08
71996  5.323 -4.882  0.242    27.95      1229    14.258  99.138  32.07
71997  5.344 -5.119  0.457    27.96      1198    14.962  99.102  32.07
71998  5.281 -5.261  0.599    28.09      1231    14.615  99.112  32.07
71999  5.235 -4.801  0.362    28.02      1211    14.682  99.164  32.08

[72000 rows x 8 columns]
```

And, as mentioned, the latter way is a lot faster:

```
In [27]: %timeit pm.timeSeries(fname, fconfig, parse_dates=False)
   ....: %timeit pm.timeSeries(fname, fconfig, parse_dates=True)
   ....:
1 loops, best of 3: 139 ms per loop
1 loops, best of 3: 713 ms per loop
```

Viewing and manipulating data

To view and manipulate data, mostly 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 and introduce some few ideas specific for Pymicra that don't exist for general Pandas DataFrames.

Printing and plotting

First, for viewing raw data on screen there's printing. Slicing and indexing are supported by Pandas, but without support for units:

```
In [29]: print(data['theta_v'])
Timestamp
2013-11-08 10:00:00.000    27.06
2013-11-08 10:00:00.050    27.12
2013-11-08 10:00:00.100    27.11
2013-11-08 10:00:00.150    27.11
2013-11-08 10:00:00.200    27.12
2013-11-08 10:00:00.250    27.10
                           ...
2013-11-08 10:59:59.700    28.11
2013-11-08 10:59:59.750    27.98
2013-11-08 10:59:59.800    27.95
2013-11-08 10:59:59.850    27.96
2013-11-08 10:59:59.900    28.09
2013-11-08 10:59:59.950    28.02
Name: theta_v, dtype: float64

In [30]: print(data[['u', 'v', 'w']])
                             u      v      w
Timestamp
2013-11-08 10:00:00.000  2.375 -5.206 -0.103
2013-11-08 10:00:00.050  2.493 -5.098 -0.018
2013-11-08 10:00:00.100  2.263 -5.114  0.014
2013-11-08 10:00:00.150  2.210 -5.235 -0.012
2013-11-08 10:00:00.200  2.158 -5.174 -0.112
2013-11-08 10:00:00.250  2.334 -5.279 -0.092
...                        ...    ...    ...
2013-11-08 10:59:59.700  5.065 -4.453  0.424
2013-11-08 10:59:59.750  5.262 -4.703  0.126
2013-11-08 10:59:59.800  5.323 -4.882  0.242
2013-11-08 10:59:59.850  5.344 -5.119  0.457
2013-11-08 10:59:59.900  5.281 -5.261  0.599
2013-11-08 10:59:59.950  5.235 -4.801  0.362

[72000 rows x 3 columns]

In [31]: print(data['20131108 10:15:00.000':'20131108 10:17:00.000'])
                             u      v      w  theta_v  mrho_h2o  mrho_co2       p␣
↪theta
Timestamp                                                                        ␣
↪
2013-11-08 10:15:00.000  2.634 -4.351  0.107    27.30      1229    15.002  99.128␣
↪30.80
2013-11-08 10:15:00.050  2.869 -4.249  0.040    27.44      1175    14.751  99.164␣
↪30.80
2013-11-08 10:15:00.100  3.320 -4.326 -0.079    27.26      1159    14.689  99.138␣
↪30.80
2013-11-08 10:15:00.150  2.759 -4.339 -0.007    27.24      1170    14.715  99.190␣
↪30.80
2013-11-08 10:15:00.200  2.748 -4.128 -0.038    27.21      1174    14.681  99.190␣
↪30.80
```

```
2013-11-08 10:15:00.250  3.149 -4.074 -0.387    27.20      1190    14.662  99.173 ␣
↪30.80
...                        ...    ...    ...      ...       ...       ...     ... ␣
↪  ...
2013-11-08 10:17:00.700  3.910 -4.698 -0.366    27.27      1170    14.592  99.128 ␣
↪30.85
2013-11-08 10:17:00.750  3.824 -4.535 -0.313    27.33      1165    14.492  99.164 ␣
↪30.85
2013-11-08 10:17:00.800  3.758 -4.353 -0.116    27.28      1103    14.495  99.164 ␣
↪30.85
2013-11-08 10:17:00.850  3.761 -4.454 -0.010    27.28      1128    14.611  99.164 ␣
↪30.85
2013-11-08 10:17:00.900  3.546 -4.766 -0.433    27.28      1131    14.709  99.147 ␣
↪30.85
2013-11-08 10:17:00.950  3.238 -4.601 -0.378    27.29      1130    14.809  99.147 ␣
↪30.85

[2420 rows x 8 columns]
```

Note that Pandas "guesses" if the argument you pass (`'theta_v'` or `'2013-11-08 10:15:00'` etc.) is a column indexer or a row indexer. To use these unambiguously, use the `.loc` method as

```
In [32]: print(data.loc['2013-11-08 10:15:00':'2013-11-08 10:17:00', ['u','v','w
↪']])
                             u      v      w
Timestamp
2013-11-08 10:15:00.000  2.634 -4.351  0.107
2013-11-08 10:15:00.050  2.869 -4.249  0.040
2013-11-08 10:15:00.100  3.320 -4.326 -0.079
2013-11-08 10:15:00.150  2.759 -4.339 -0.007
2013-11-08 10:15:00.200  2.748 -4.128 -0.038
2013-11-08 10:15:00.250  3.149 -4.074 -0.387
...                        ...    ...    ...
2013-11-08 10:17:00.700  3.910 -4.698 -0.366
2013-11-08 10:17:00.750  3.824 -4.535 -0.313
2013-11-08 10:17:00.800  3.758 -4.353 -0.116
2013-11-08 10:17:00.850  3.761 -4.454 -0.010
2013-11-08 10:17:00.900  3.546 -4.766 -0.433
2013-11-08 10:17:00.950  3.238 -4.601 -0.378

[2420 rows x 3 columns]
```

This method is actually preferred and you can find more information on this topic here.

To view these data with units, you can use the `.with_units()` method. The previous output would look like this using units:

```
In [33]: print(data.with_units(units)['theta_v'])
                          <degC>
Timestamp
2013-11-08 10:00:00.000   27.06
2013-11-08 10:00:00.050   27.12
2013-11-08 10:00:00.100   27.11
2013-11-08 10:00:00.150   27.11
2013-11-08 10:00:00.200   27.12
2013-11-08 10:00:00.250   27.10
...                         ...
2013-11-08 10:59:59.700   28.11
2013-11-08 10:59:59.750   27.98
2013-11-08 10:59:59.800   27.95
2013-11-08 10:59:59.850   27.96
2013-11-08 10:59:59.900   28.09
2013-11-08 10:59:59.950   28.02
```

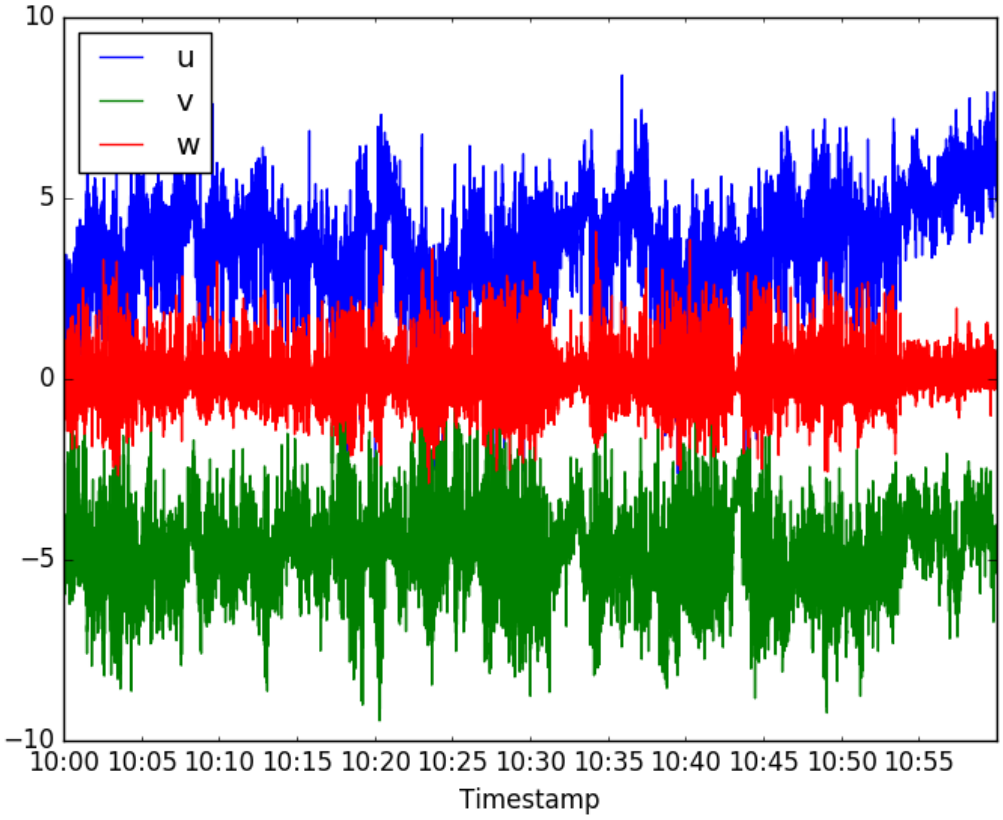
```
[72000 rows x 1 columns]

In [34]: print(data.with_units(units)[['u', 'v', 'w']])
                                        u                 v                 w
                        <meter / second> <meter / second> <meter / second>
Timestamp
2013-11-08 10:00:00.000            2.375            -5.206            -0.103
2013-11-08 10:00:00.050            2.493            -5.098            -0.018
2013-11-08 10:00:00.100            2.263            -5.114             0.014
2013-11-08 10:00:00.150            2.210            -5.235            -0.012
2013-11-08 10:00:00.200            2.158            -5.174            -0.112
2013-11-08 10:00:00.250            2.334            -5.279            -0.092
...                                  ...               ...               ...
2013-11-08 10:59:59.700            5.065            -4.453             0.424
2013-11-08 10:59:59.750            5.262            -4.703             0.126
2013-11-08 10:59:59.800            5.323            -4.882             0.242
2013-11-08 10:59:59.850            5.344            -5.119             0.457
2013-11-08 10:59:59.900            5.281            -5.261             0.599
2013-11-08 10:59:59.950            5.235            -4.801             0.362

[72000 rows x 3 columns]

In [35]: print(data.with_units(units)['2013-11-08 10:15:00'])
                                        u                 v                 w theta_v␣
↪ \
                        <meter / second> <meter / second> <meter / second>  <degC>␣
↪
Timestamp                                                                          ␣
↪
2013-11-08 10:15:00.000            2.634            -4.351             0.107   27.30␣
↪
2013-11-08 10:15:00.050            2.869            -4.249             0.040   27.44␣
↪
2013-11-08 10:15:00.100            3.320            -4.326            -0.079   27.26␣
↪
2013-11-08 10:15:00.150            2.759            -4.339            -0.007   27.24␣
↪
2013-11-08 10:15:00.200            2.748            -4.128            -0.038   27.21␣
↪
2013-11-08 10:15:00.250            3.149            -4.074            -0.387   27.20␣
↪
...                                  ...               ...               ...     ...␣
↪
2013-11-08 10:15:00.700            3.057            -4.090            -0.230   27.28␣
↪
2013-11-08 10:15:00.750            3.386            -4.169            -0.082   27.21␣
↪
2013-11-08 10:15:00.800            3.731            -4.180             0.291   27.42␣
↪
2013-11-08 10:15:00.850            3.676            -4.100             0.021   27.29␣
↪
2013-11-08 10:15:00.900            3.796            -4.390             0.170   27.24␣
↪
2013-11-08 10:15:00.950            3.294            -3.322             0.560   27.50␣
↪

                                        mrho_h2o                  mrho_co2          ␣
↪  p  theta
                        <millimole / meter ** 3> <millimole / meter ** 3>
↪<kilopascal> <degC>
Timestamp                                                                          ␣
↪
```

```
2013-11-08 10:15:00.000                    1229                    15.002       99.
↪128   30.8
2013-11-08 10:15:00.050                    1175                    14.751       99.
↪164   30.8
2013-11-08 10:15:00.100                    1159                    14.689       99.
↪138   30.8
2013-11-08 10:15:00.150                    1170                    14.715       99.
↪190   30.8
2013-11-08 10:15:00.200                    1174                    14.681       99.
↪190   30.8
2013-11-08 10:15:00.250                    1190                    14.662       99.
↪173   30.8
...                                         ...                       ...        ⌐
↪...    ...
2013-11-08 10:15:00.700                    1151                    14.758       99.
↪164   30.8
2013-11-08 10:15:00.750                    1195                    14.318       99.
↪164   30.8
2013-11-08 10:15:00.800                    1172                    14.369       99.
↪164   30.8
2013-11-08 10:15:00.850                    1173                    14.687       99.
↪164   30.8
2013-11-08 10:15:00.900                    1153                    14.442       99.
↪164   30.8
2013-11-08 10:15:00.950                    1176                    14.724       99.
↪208   30.8

[20 rows x 8 columns]
```

Warning: Note that, although this method returns a Pandas DataFrame, it is not meant for calculations. Currently the DataFrame it returns is meant for visualization purposes only!

We can also plot the data on screen so we can view it interactively. This can be done directly from the DataFrame with

```
In [36]: from matplotlib import pyplot as plt

In [37]: data[['u', 'v', 'w']].plot()
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7f9b13847c50>

In [38]: plt.show()
```



Using the `plt.show()` command, the plot above would plot interactively. If we had used `plt.savefig('figure.png')` instead, it would have saved the figure as png. For more on plotting, you can checkout Pandas's visualization guide and find out ways to make this plot look nicer, how to render it with LaTeX and some more tricks.

Pymicra also has an `.xplot` method, which brings a little more options to Pandas's `.plot()` method.

Todo

give xplot examples

Converting units

You can manually convert between units using the contents from *Manipulating* and the Pint package. But Pymicra has a very useful method to do this called `.convert_cols` (more exist, but let's focus on this one).

Let's, for example, convert some units:

```
In [39]: conversions = {'p':'pascal', 'mrho_h2o':'mole/m^3', 'theta_v':'kelvin'}

In [40]: print(data.convert_cols(conversions, units, inplace_units=False))
(                           u      v      w  theta_v  mrho_h2o  mrho_co2      p␣
↪theta
Timestamp                                                                       ␣
↪
2013-11-08 10:00:00.000  2.375 -5.206 -0.103   300.21     1.238    14.675  99190␣
↪30.43
2013-11-08 10:00:00.050  2.493 -5.098 -0.018   300.27     1.196    14.409  99199␣
↪30.43
```

```
2013-11-08 10:00:00.100  2.263 -5.114  0.014   300.26     1.220    14.636  99102 ␣
↪30.43
2013-11-08 10:00:00.150  2.210 -5.235 -0.012   300.26     1.238    14.688  99154 ␣
↪30.43
2013-11-08 10:00:00.200  2.158 -5.174 -0.112   300.27     1.174    14.476  99154 ␣
↪30.44
2013-11-08 10:00:00.250  2.334 -5.279 -0.092   300.25     1.195    14.671  99154 ␣
↪30.43
...                        ...    ...    ...      ...       ...       ...    ...  ␣
↪ ...
2013-11-08 10:59:59.700  5.065 -4.453  0.424   301.26     1.184    14.578  99138 ␣
↪32.07
2013-11-08 10:59:59.750  5.262 -4.703  0.126   301.13     1.264    14.929  99138 ␣
↪32.08
2013-11-08 10:59:59.800  5.323 -4.882  0.242   301.10     1.229    14.258  99138 ␣
↪32.07
2013-11-08 10:59:59.850  5.344 -5.119  0.457   301.11     1.198    14.962  99102 ␣
↪32.07
2013-11-08 10:59:59.900  5.281 -5.261  0.599   301.24     1.231    14.615  99112 ␣
↪32.07
2013-11-08 10:59:59.950  5.235 -4.801  0.362   301.17     1.211    14.682  99164 ␣
↪32.08

[72000 rows x 8 columns], {'theta_v': <Unit('kelvin')>, 'p': <Unit('pascal')>,
↪'mrho_h2o': <Unit('mole / meter ** 3')>})
```

Note that the units dictionary is updated automatically if the `inplace_units` keyword is true. The default is false for safety reasons, but passing this keyword as true is much simpler and compact:

```
In [41]: conversions = {'theta':'kelvin', 'theta_v':'kelvin'}

In [42]: data = data.convert_cols(conversions, units, inplace_units=True)

In [43]: print(data.with_units(units))
                                     u                 v                 w  theta_
↪v  \
                        <meter / second> <meter / second> <meter / second> <kelvin>
↪
Timestamp                                                                          ␣
↪
2013-11-08 10:00:00.000            2.375            -5.206            -0.103   300.
↪21
2013-11-08 10:00:00.050            2.493            -5.098            -0.018   300.
↪27
2013-11-08 10:00:00.100            2.263            -5.114             0.014   300.
↪26
2013-11-08 10:00:00.150            2.210            -5.235            -0.012   300.
↪26
2013-11-08 10:00:00.200            2.158            -5.174            -0.112   300.
↪27
2013-11-08 10:00:00.250            2.334            -5.279            -0.092   300.
↪25
...                                  ...               ...               ...      ...
↪
2013-11-08 10:59:59.700            5.065            -4.453             0.424   301.
↪26
2013-11-08 10:59:59.750            5.262            -4.703             0.126   301.
↪13
2013-11-08 10:59:59.800            5.323            -4.882             0.242   301.
↪10
2013-11-08 10:59:59.850            5.344            -5.119             0.457   301.
↪11
```

```
2013-11-08 10:59:59.900            5.281           -5.261            0.599   301.
↪24
2013-11-08 10:59:59.950            5.235           -4.801            0.362   301.
↪17

                                      mrho_h2o                  mrho_co2          ⌊
↪  p    theta
                        <millimole / meter ** 3> <millimole / meter ** 3>
↪<kilopascal> <kelvin>
Timestamp                                                                       ⌊
↪
2013-11-08 10:00:00.000                    1238                    14.675       99.
↪190   303.58
2013-11-08 10:00:00.050                    1196                    14.409       99.
↪199   303.58
2013-11-08 10:00:00.100                    1220                    14.636       99.
↪102   303.58
2013-11-08 10:00:00.150                    1238                    14.688       99.
↪154   303.58
2013-11-08 10:00:00.200                    1174                    14.476       99.
↪154   303.59
2013-11-08 10:00:00.250                    1195                    14.671       99.
↪154   303.58
...                                         ...                       ...        ⌊
↪...      ...
2013-11-08 10:59:59.700                    1184                    14.578       99.
↪138   305.22
2013-11-08 10:59:59.750                    1264                    14.929       99.
↪138   305.23
2013-11-08 10:59:59.800                    1229                    14.258       99.
↪138   305.22
2013-11-08 10:59:59.850                    1198                    14.962       99.
↪102   305.22
2013-11-08 10:59:59.900                    1231                    14.615       99.
↪112   305.22
2013-11-08 10:59:59.950                    1211                    14.682       99.
↪164   305.23

[72000 rows x 8 columns]
```

Manipulating

Manipulating data is pretty intuitive with Pandas. For example

```
In [44]: data['rho_air'] = data['p']/(287.058*data['theta_v'])

In [45]: print(data['rho_air'])
Timestamp
2013-11-08 10:00:00.000    0.001151
2013-11-08 10:00:00.050    0.001151
2013-11-08 10:00:00.100    0.001150
2013-11-08 10:00:00.150    0.001150
2013-11-08 10:00:00.200    0.001150
2013-11-08 10:00:00.250    0.001150
                             ...
2013-11-08 10:59:59.700    0.001146
2013-11-08 10:59:59.750    0.001147
2013-11-08 10:59:59.800    0.001147
2013-11-08 10:59:59.850    0.001147
2013-11-08 10:59:59.900    0.001146
2013-11-08 10:59:59.950    0.001147
```

```
Name: rho_air, dtype: float64
```

If, however, you're not familiar with Pandas and prefer to just stick with what you know, you can get Numpy arrays from columns using the `.values` attribute:

```
In [46]: P = data['p'].values

In [47]: Tv = data['theta_v'].values

In [48]: print(type(Tv))
<type 'numpy.ndarray'>

In [49]: rho_air = P/(287.058*Tv)

In [50]: print(rho_air)
[ 0.00115099  0.00115087  0.00114978 ...,  0.00114654  0.00114616
  0.00114702]

In [51]: print(type(rho_air))
<type 'numpy.ndarray'>
```

Doing that you can step out of Pandas and do your own calculations using your own Python or Numpy code. This is pretty advantageous if you have a lot of routines that are already written in your own way.

CHAPTER

FOUR

HANDS-ON TUTORIAL

Here we give some examples of generally-used steps to obtain fluxes, spectra and Ogives. This is just the commonly done procedures, but more can (and sometimes should) be done depending on the data.

Calculating auxiliary variables and units

After reading the data it's easy to rotate the coordinates. Currently, only the 2D rotation is implemented. But in the future more will come (you can contribute with another one yourself!). To rotate, use the `pm.rotateCoor` function:

```
In [1]: import pymicra as pm

In [2]: fname = '../examples/ex_data/20131108-1000.csv'

In [3]: fconfig = pm.fileConfig('../examples/lake.config')

In [4]: data, units = pm.timeSeries(fname, fconfig, parse_dates=True)

In [5]: data = data.rotateCoor(how='2d')

In [6]: print(data.with_units(units).mean())
u          <meter / second>            6.021737e+00
v          <meter / second>            2.271683e-14
w          <meter / second>            1.178001e-16
theta_v    <degC>                      2.764434e+01
mrho_h2o   <millimole / meter ** 3>    1.169865e+03
mrho_co2   <millimole / meter ** 3>    1.466232e+01
p          <kilopascal>                9.913677e+01
theta      <degC>                      3.127128e+01
dtype: float64
```

As you can see, the u, v, w components have been rotated and `v` and `w` mean are zero.

Pymicra has a very useful function called `preProcess` which "expands" the variables in the dataset by using the original variables to calculate new ones, such as mixing ratios, moist and dry air densities etc. In the process some units are also converted, such as temperature units, which are converted to Kelvin, if they are in Celsius. The `preProcess` function has some options to calculate some variables in determined ways and is very "verbose", so as to show the user exactly what it is doing:

```
In [7]: data = pm.preProcess(data, units, expand_temperature=True,
   ...:     use_means=False, rho_air_from_theta_v=True, solutes=['co2'])
   ...:
Beginning of pre-processing ...
Converting theta_v and theta to kelvin ... Done!
Didn't locate mass density of h2o. Trying to calculate it ... Done!
Moist air density not present in dataset
Calculating rho_air = p/(Rdry * theta_v) ... Done!
Calculating dry_air mass_density = rho_air - rho_h2o ... Done!
```

```
Dry air molar density not in dataset
Calculating dry_air molar_density = rho_dry / dry_air_molar_mass ... Done!
Calculating specific humidity = rho_h2o / rho_air ... Done!
Calculating h2o mass mixing ratio = rho_h2o / rho_dry ... Done!
Calculating h2o molar mixing ratio = rho_h2o / rho_dry ... Done!
Didn't locate mass density of co2. Trying to calculate it ... Done!
Calculating co2 mass concentration (g/g) = rho_co2 / rho_air ... Done!
Calculating co2 mass mixing ratio = rho_co2 / rho_dry ... Done!
Calculating co2 molar mixing ratio = mrho_co2 / mrho_dry ... Done!
Pre-processing complete.


In [8]: print(data.with_units(units).mean())
u          <meter / second>            6.021737e+00
v          <meter / second>            2.271683e-14
w          <meter / second>            1.178001e-16
theta_v    <kelvin>                    3.007943e+02
mrho_h2o   <millimole / meter ** 3>    1.169865e+03
mrho_co2   <millimole / meter ** 3>    1.466232e+01
p          <kilopascal>                9.913677e+01
theta      <kelvin>                    3.044213e+02
rho_h2o    <kilogram / meter ** 3>     2.107547e-02
rho_air    <kilogram / meter ** 3>     1.148149e+00
rho_dry    <kilogram / meter ** 3>     1.127073e+00
mrho_dry   <mole / meter ** 3>         3.891223e+01
q          <dimensionless>             1.835686e-02
r_h2o      <dimensionless>             1.870102e-02
mr_h2o     <dimensionless>             3.006698e-02
rho_co2    <kilogram / meter ** 3>     6.452812e-04
conc_co2   <dimensionless>             5.620179e-04
r_co2      <dimensionless>             5.725280e-04
mr_co2     <dimensionless>             3.768047e-04
dtype: float64
```

Note that our dataset now has many other variables and that the temperatures are now in Kelvin. Note also that you must, at this point, specify which solutes you wish to consider. The only solute that Pymicra looks for by default is water vapor.

It is recommended that you carefully observe the output of `preProcess` to see if it's doing what you think it's doing and to analyse its calculation logic. By doing that you can have a better idea of what it does without having to check the code, and how to prevent it from doing some calculations you don't want. For example, if you're not happy with the way it calculates moist air density, you can calculate it yourself by extracting **Numpy_** arrays using the `.values` method:

```
T = data['theta'].values
P = data['p'].values
water_density = data['mrho_h2o'].values
data['rho_air'] = my_rho_air_calculation(T, P, water_density)
```

Next we calculate the fluctuations. The way to do that is with the `detrend()` function/method.

```
In [9]: ddata = data.detrend(how='linear', units=units, ignore=['p', 'theta'])

In [10]: print(ddata.with_units(units).mean())
u'         <meter / second>            -3.217650e-05
v'         <meter / second>            -4.224175e-05
w'         <meter / second>            -4.156934e-06
theta_v'   <kelvin>                    -3.421680e-05
mrho_h2o'  <millimole / meter ** 3>    -7.917622e-04
mrho_co2'  <millimole / meter ** 3>     2.996904e-06
rho_h2o'   <kilogram / meter ** 3>     -1.426702e-08
rho_air'   <kilogram / meter ** 3>      1.506704e-07
```

```
rho_dry'    <kilogram / meter ** 3>     1.648373e-07
mrho_dry'   <mole / meter ** 3>         5.689934e-06
q'          <dimensionless>            -1.480115e-08
r_h2o'      <dimensionless>            -1.535592e-08
mr_h2o'     <dimensionless>            -2.468290e-08
rho_co2'    <kilogram / meter ** 3>     1.318839e-10
conc_co2'   <dimensionless>             4.107355e-11
r_co2'      <dimensionless>             3.319263e-11
mr_co2'     <dimensionless>             2.184087e-11
dtype: float64
```

Note the `ignore` keyword with which you can pass which variables you don't want fluctuations of. In our case both the pressure fluctuations and the thermodynamic temperature (measured with the LI7500 and the datalogger's internal thermometer) can't be properly measured with the sensors used. Passing these variables as ignored will prevent Pymicra from calculating these fluctuations, which later on prevents it from inadvertently using the fluctuations of these variables with these sensors when calculating fluxes. `theta` fluctuations will instead be calculated with virtual temperature fluctuations and pressure fluctuations are generally not used.

The `how` keyword supports `'linear'`, `'movingmean'`, `'movingmedian'`, `'block'` and `'poly'`. Each of those (with the exception of `'block'` and `'linear'`) are passed to Pandas or Numpy (`pandas.rolling_mean`, `pandas.rolling_median` and `numpy.polyfit`) and support their respective keywords, such as `data.detrend(how='movingmedian',units=units,ignore=['theta','p']),min_periods=1)` to determine the minimum number of observations in window required to have a value.

Now we must expand our dataset with the fluctuations by joining DataFrames:

```
In [11]: data = data.join(ddata)

In [12]: print(data.with_units(units).mean())
u           <meter / second>            6.021737e+00
v           <meter / second>            2.271683e-14
w           <meter / second>            1.178001e-16
theta_v     <kelvin>                    3.007943e+02
mrho_h2o    <millimole / meter ** 3>    1.169865e+03
mrho_co2    <millimole / meter ** 3>    1.466232e+01
p           <kilopascal>                9.913677e+01
theta       <kelvin>                    3.044213e+02
rho_h2o     <kilogram / meter ** 3>     2.107547e-02
rho_air     <kilogram / meter ** 3>     1.148149e+00
rho_dry     <kilogram / meter ** 3>     1.127073e+00
mrho_dry    <mole / meter ** 3>         3.891223e+01
q           <dimensionless>             1.835686e-02
r_h2o       <dimensionless>             1.870102e-02
mr_h2o      <dimensionless>             3.006698e-02
                                            ...
w'          <meter / second>           -4.156934e-06
theta_v'    <kelvin>                   -3.421680e-05
mrho_h2o'   <millimole / meter ** 3>   -7.917622e-04
mrho_co2'   <millimole / meter ** 3>    2.996904e-06
rho_h2o'    <kilogram / meter ** 3>    -1.426702e-08
rho_air'    <kilogram / meter ** 3>     1.506704e-07
rho_dry'    <kilogram / meter ** 3>     1.648373e-07
mrho_dry'   <mole / meter ** 3>         5.689934e-06
q'          <dimensionless>            -1.480115e-08
r_h2o'      <dimensionless>            -1.535592e-08
mr_h2o'     <dimensionless>            -2.468290e-08
rho_co2'    <kilogram / meter ** 3>     1.318839e-10
conc_co2'   <dimensionless>             4.107355e-11
r_co2'      <dimensionless>             3.319263e-11
mr_co2'     <dimensionless>             2.184087e-11
dtype: float64
```

Creating a site configuration file

In order to use some micrometeorological function, we need a site configuration object, or site object. This object tells Pymicra how the instruments are organized and how is experimental site (vegetation, roughness length etc.). This object is created with the `siteConfig()` call and the easiest way is to use it with a site config file. This is like a `fileConfig` file, but simpler and for the micrometeorological aspects of the site. Consider this example of a site config file, saved at `../examples/lake.site`:

```
description          = "Site configurations for the lake island"

measurement_height   = 2.75  # m
canopy_height        = 0.1   # m
displacement_height  = 0.05  # m
roughness_length     = .01   # m
```

As in the case of `fileConfig`, the description is optional, but is handy for organization purposes and printing on screen.

The `measurement_height` keyword is the general measurement height to be used in the calculation, generally taken as the sonic anemometer height (in meters).

The `canopy_height` keyword is what it sounds like. Should be the mean canopy height in meters. The `displacement_height` is the zero-plane displacement height. If it is not given, it'll be calculated as 2/3 of the mean canopy height.

The `roughness_length` is the roughness length in meters.

The creation of the site config object is done as

```
In [13]: siteconf = pm.siteConfig('../examples/lake.site')

In [14]: print(siteconf)
<pymicra.siteConfig> object
Site configurations for the lake island
----
altitude               None
canopy_height           0.1
displacement_height    0.05
from_file              None
instruments_height     None
latitude               None
longitude              None
measurement_height     2.75
roughness_length       0.01
dtype: object
```

Extracting fluxes

Finally, we are able to calculate the fluxes and turbulent scales. For that we use the `eddyCovariance` function along with the `get_turbulent_scales` keyword (which is true by default). Again, it is recommended that you carefully check the output of this function before moving on:

```
In [15]: results = pm.eddyCovariance(data, units, site_config=siteconf,
   ....:     get_turbulent_scales=True, wpl=True, solutes=['co2'])
   ....:
Beginning Eddy Covariance method...
Fluctuations of theta not found. Will try to calculate it with theta' = (theta_v' -
↪ 0.61 theta_mean q')/(1 + 0.61 q_mean ... done!
Calculating fluxes from covariances ... done!
Applying WPL correction for water vapor flux ... done!
```

```
Applying WPL correction for latent heat flux using result for water vapor flux ...␣
↪done!
Re-calculating cov(mrho_h2o', w') according to WPL correction ... done!
Applying WPL correction for F_co2 ... done!
Re-calculating cov(mrho_co2', w') according to WPL correction ... done!
Beginning to extract turbulent scales...
Data seems to be covariances. Will it use as covariances ...
Calculating the turbulent scales of wind, temperature and humidity ... done!
Calculating the turbulent scale of co2 ... done!
Calculating Obukhov length and stability parameter ... done!
Calculating turbulent scales of mass concentration ... done!
Done with Eddy Covariance.


In [16]: print(results.with_units(units).mean())
tau             <kilogram / meter / second ** 2>     1.957307e-01
H               <watt / meter ** 2>                  5.083720e+01
Hv              <watt / meter ** 2>                  7.453761e+01
E               <millimole / meter ** 2 / second>    7.007486e+00
LE              <watt / meter ** 2>                  3.063926e+02
F_co2           <millimole / meter ** 2 / second>    2.648931e-04
u_star          <meter / second>                     4.128863e-01
theta_v_star    <kelvin>                             1.566857e-01
theta_star      <kelvin>                             1.068650e-01
mrho_h2o_star   <millimole / meter ** 3>             1.697195e+01
mrho_co2_star   <millimole / meter ** 3>             6.415643e-04
Lo              <meter>                             -8.342964e+01
zeta            <dimensionless>                     -3.236260e-02
q_star          <dimensionless>                      2.663024e-04
conc_co2_star   <dimensionless>                      2.459169e-08
dtype: float64
```

Note that once more we must list the solutes available.

Note also that the `units` dictionary is automatically updated at with every function and method with the new variables created and their units! That way if you're in doubt of which unit the outputs are coming, just check `units` directly or with the `.with_units()` method.

Check out the example to get fluxes of many files here and download the example data here.

The output of this file is

```
                                                 tau                     H                  ␣
↪       Hv                                     E                 LE  \
                      <kilogram / meter / second ** 2> <watt / meter ** 2> <watt /␣
↪meter ** 2> <millimole / meter ** 2 / second> <watt / meter ** 2>
2013-11-08 10:00:00                             0.195731           50.837195                ␣
↪74.537609                         7.007486          306.392623
2013-11-08 12:00:00                             0.070182           -7.116972                ␣
↪14.737946                         6.647450          290.331014
2013-11-08 13:00:00                             0.059115           -3.325774                ␣
↪12.743194                         4.875838          212.906974
2013-11-08 16:00:00                             0.017075          -13.847040                ␣
↪-4.131275                         2.961474          128.982980
2013-11-08 17:00:00                             0.003284           -5.557620                ␣
↪-2.525949                         0.931213           40.576805
2013-11-08 18:00:00                             0.026564           -6.632330                ␣
↪-2.782190                         1.184226           51.654208
2013-11-08 19:00:00                             0.044550          -14.860873                -
↪11.991508                         0.925637           40.476446

                              u_star theta_v_star theta_star            mrho_h2o_
↪star            Lo            zeta            q_star
```

```
                     <meter / second>     <kelvin>   <kelvin> <millimole / meter **␣
↪3>     <meter> <dimensionless> <dimensionless>
2013-11-08 10:00:00         0.412886     0.156686   0.106865                16.
↪971952  -83.429641       -0.032363        0.000266
2013-11-08 12:00:00         0.247697     0.051834  -0.025031                26.
↪837060  -90.992476       -0.029673        0.000423
2013-11-08 13:00:00         0.227652     0.048903  -0.012763                21.
↪417958  -81.621514       -0.033080        0.000338
2013-11-08 16:00:00         0.122980    -0.029651  -0.099383                24.
↪080946   39.583979        0.068209        0.000384
2013-11-08 17:00:00         0.053951    -0.041353  -0.090986                17.
↪260443    5.463569        0.494182        0.000276
2013-11-08 18:00:00         0.153328    -0.016003  -0.038149                 7.
↪723494  113.854892        0.023714        0.000123
2013-11-08 19:00:00         0.198083    -0.053132  -0.065846                 4.
↪672974   56.961515        0.047400        0.000074
```

Note that the date parsing when reading the file comes is handy now, although there are computationally faster ways to do it other than setting `parse_dates=True` in the `timeSeries()` call.

Obtaining the spectra

Using Numpy's fast Fourier transform implementation, Pymicra is also able to extract spectra, co-spectra and quadratures.

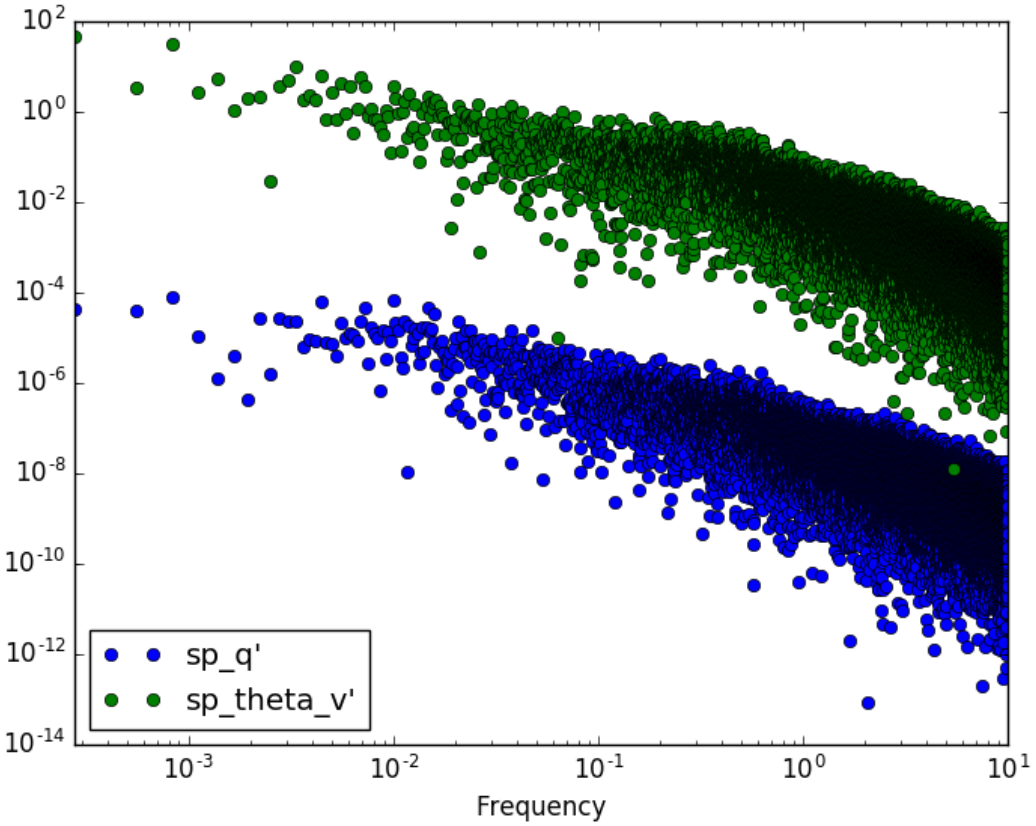
We begin normally with out data:

```
In [17]: import pymicra as pm

In [18]: fname = '../examples/ex_data/20131108-1000.csv'

In [19]: fconfig = pm.fileConfig('../examples/lake.config')

In [20]: data, units = pm.timeSeries(fname, fconfig, parse_dates=True)

In [21]: data = data.rotateCoor(how='2d')

In [22]: data = pm.preProcess(data, units, expand_temperature=True,
   ....:     use_means=False, rho_air_from_theta_v=True, solutes=['co2'])
   ....:
Beginning of pre-processing ...
Converting theta_v and theta to kelvin ... Done!
Didn't locate mass density of h2o. Trying to calculate it ... Done!
Moist air density not present in dataset
Calculating rho_air = p/(Rdry * theta_v) ... Done!
Calculating dry_air mass_density = rho_air - rho_h2o ... Done!
Dry air molar density not in dataset
Calculating dry_air molar_density = rho_dry / dry_air_molar_mass ... Done!
Calculating specific humidity = rho_h2o / rho_air ... Done!
Calculating h2o mass mixing ratio = rho_h2o / rho_dry ... Done!
Calculating h2o molar mixing ratio = rho_h2o / rho_dry ... Done!
Didn't locate mass density of co2. Trying to calculate it ... Done!
Calculating co2 mass concentration (g/g) = rho_co2 / rho_air ... Done!
Calculating co2 mass mixing ratio = rho_co2 / rho_dry ... Done!
Calculating co2 molar mixing ratio = mrho_co2 / mrho_dry ... Done!
Pre-processing complete.


In [23]: ddata = data.detrend(how='linear', units=units, ignore=['p', 'theta'])

In [24]: spectra = pm.spectra(ddata[["q'", "theta_v'"]], frequency=20, anti_
↪aliasing=True)
```

```
In [25]: print(spectra)
                  sp_q'  sp_theta_v'
Frequency
0.000000   1.577334e-12     0.000008
0.000278   4.144727e-05    45.943783
0.000556   3.865083e-05     3.334596
0.000833   7.980502e-05    31.672784
0.001111   1.073580e-05     2.803636
0.001389   1.241483e-06     5.217900
0.001667   3.835368e-06     1.047473
0.001944   4.227195e-07     1.946339
0.002222   2.814149e-05     2.102988
0.002500   1.583606e-06     0.028766
0.002778   2.811464e-05     3.580177
0.003056   2.370466e-05     5.149250
0.003333   2.293237e-05    10.265385
0.003611   6.403187e-06     1.859855
0.003889   9.423586e-06     2.375210
...                 ...          ...
9.996111   1.084777e-09     0.000124
9.996389   1.158869e-08     0.000737
9.996667   4.004893e-09     0.000586
9.996944   1.888643e-09     0.000200
9.997222   3.102855e-10     0.000402
9.997500   3.321963e-10     0.000106
9.997778   3.111850e-10     0.000146
9.998056   2.923886e-10     0.000850
9.998333   2.343816e-10     0.000508
9.998611   7.671555e-10     0.000464
9.998889   3.931099e-09     0.000459
9.999167   9.530669e-10     0.000081
9.999444   3.370132e-09     0.000122
9.999722   2.259948e-10     0.000505
10.000000  4.645530e-10     0.001140

[36001 rows x 2 columns]
```

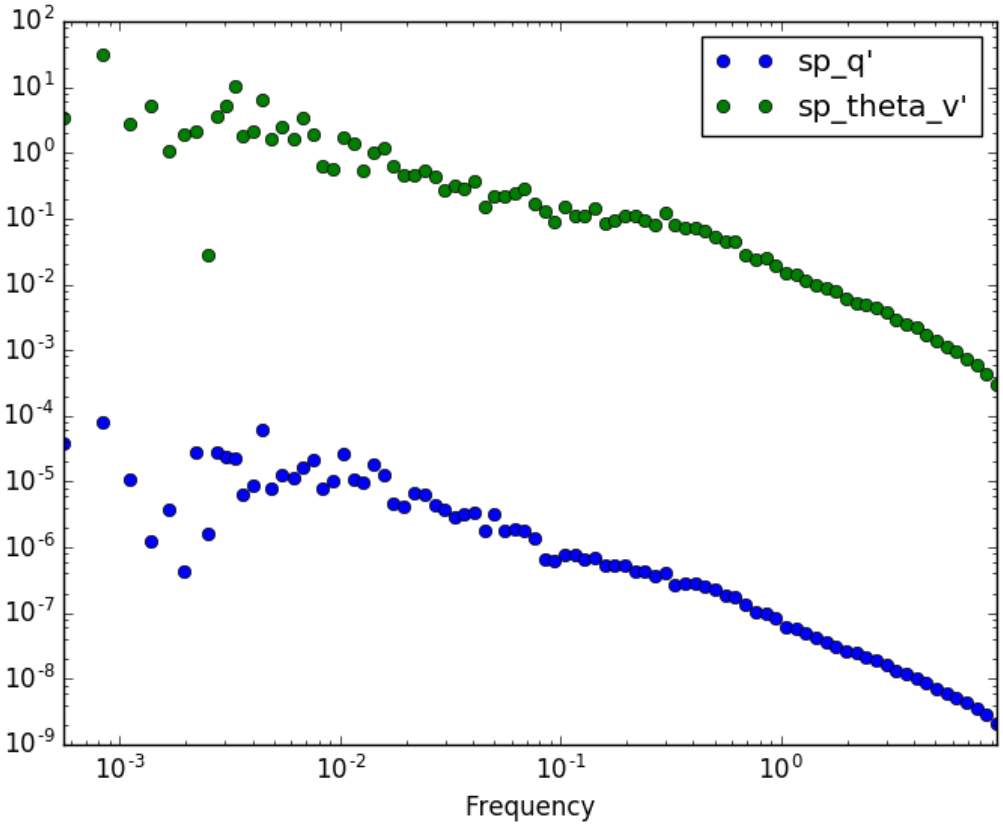
We can plot it with the raw points, but it's hard to see anything

```
In [26]: spectra.plot(loglog=True, style='o')
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7d8e748290>

In [27]: plt.show()
```



The best option it to apply a binning procedure before plotting it:

```
In [28]: spectra.binned(bins_number=100).plot(loglog=True, style='o')
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7d8e31bad0>

In [29]: plt.show()
```



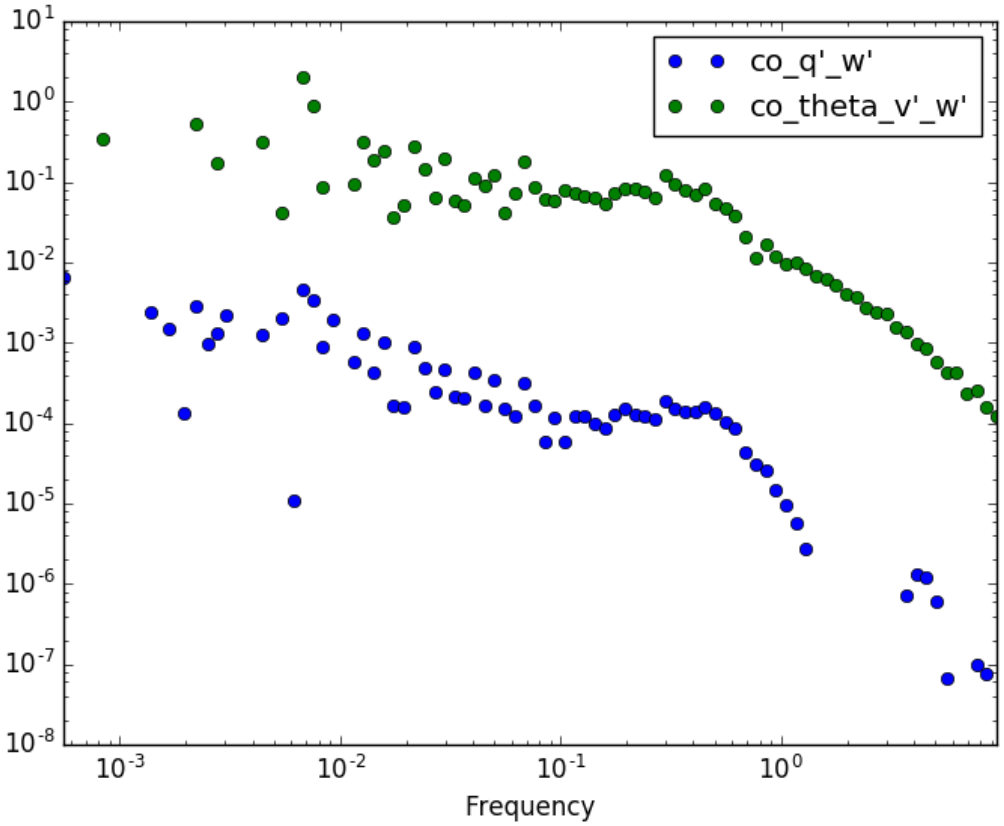
We can also calculate the cross-spectra

```
In [30]: crspectra = pm.crossSpectra(ddata[["q'", "theta_v'", "w'"]],␣
↪frequency=20, anti_aliasing=True)

In [31]: print(crspectra)
                                    X_q'_theta_v'
↪ X_q'_w'                            X_theta_v'_w'
Frequency
↪
0.000000                (3.64642595549e-09+0j)                    (4.
↪42997418716e-10+0j)                  (1.02410628187e-06+0j)
0.000278       (0.0431404707606-0.00656843186362j)    (-0.00601159315561+0.
↪007643444018O3j)          (-7.46848648027+7.00299465568j)
0.000556       (0.00909424097384+0.0067955652924j)       (0.00662015331717-0.
↪012703253092j)         (-0.675807329418-4.152928054l4j)
0.000833       (0.0496067630364+0.008174128079770j)   (-0.000691711708511+0.
↪007552422795640j)          (0.343598570049+4.76543160858j)
0.001111      (0.00484104482788+0.002581390131390j)        (-0.002261508595+0.
↪002958157248j)         (-0.308491706877+1.87768027935j)
0.001389    (-0.00252062835328+0.000352653072163j)    (0.00237400427977+0.
↪000536610998917j)          (-4.66760091614-1.76385631957j)
0.001667     (0.000581530584633-0.001918141433060j)     (0.00147491034621+0.
↪001525818251060j)        (-0.539460590207+0.968980525892j)
0.001944   (-0.000488827031498+0.000764070372007j)   (0.000136221918826+6.
↪77467074179e-05j)         (-0.0350721992609-0.324564014820j)
0.002222      (0.00687873261507+0.003444450842070j)      (0.0029147646648-0.
↪001475667644880j)         (0.531848998543-0.717463373178j)
0.002500   (-3.20032027175e-05+0.000211020463093j)    (0.000959520700828-0.
↪000865870499310j)         (-0.134770968914-0.110360696950j)
0.002778      (0.00914484576834+0.004126398087280j)     (0.00130655515532-0.
↪001705617424820j)         (0.174648858167-0.746549794061j)
```

```
0.003056       (0.00297280641473-0.0106406599447j)      (0.00220624778578+0.
↪00567444669839j)           (-2.27048661708+1.70198460404j)
0.003333       (0.0149860509294+0.00329056839565j)    (-0.00259518294404-0.
↪000725287626379j)           (-1.7999953421-0.101584385679j)
0.003611      (0.00339758185817+0.000604514115291j)    (-0.000827847258403+0.
↪00297355512566j)           (-0.158533980555+1.65594753145j)
0.003889      (0.00466685976134+0.000776798595543j)     (-0.00282762037168-0.
↪00222452651306j)           (-1.58369821003-0.868571842093j)
...                                              ...                            ␣
↪      ...                                          ...
9.996111   (-3.00966155446e-07-2.09964207749e-07j)  (-7.46261459459e-07+1.
↪34881230785e-06j)  (-5.40229561034e-05-0.000518664207694j)
9.996389    (9.86724075233e-07-2.75125913256e-06j)     (-1.47663445849e-06-2.
↪71710633e-07j)  (-6.12221317164e-05-0.000373701153871j)
9.996667   (-1.20756421484e-06+9.41878022718e-07j)  (-1.60728034798e-06-1.
↪61932245965e-07j)     (0.00044654723968+0.000426829311903j)
9.996944   (-1.85202902306e-08-6.15032063274e-07j)   (1.28907886822e-06+2.
↪79374256215e-07j)  (-0.000103618431941+0.000417045830281j)
9.997222   (-3.02777208846e-07+1.81456590353e-07j)  (-4.11844312729e-07-4.
↪47580373048e-08j)   (0.000375703825302+0.000284523744964j)
9.997500    (1.86261188261e-07+2.21641762612e-08j)   (1.17248477329e-06-7.
↪01784908781e-08j)   (0.000652725368031-0.000117577128686j)
9.997778    (8.74534442357e-08-1.94132607794e-07j)    (3.5863011104e-07-1.
↪03520888491e-06j)      (0.00074660164755-6.7197272793e-05j)
9.998056    (4.80097791895e-07-1.34733818884e-07j)   (5.52736731063e-07-3.
↪98018175528e-07j)     (0.00109099384549-0.000398836704681j)
9.998333     (1.6826867691e-07+3.01393754479e-07j)  (-1.86557075023e-07+9.
↪81927620647e-08j)  (-7.66709753361e-06+0.000310390782787j)
9.998611   (-4.96402326019e-07-3.31429269153e-07j)  (-6.41016999642e-07-4.
↪06693785042e-07j)   (0.000590483357437-1.37756359355e-05j)
9.998889   (-7.93045719726e-07+1.08504506495e-06j)      (5.2007159701e-07+7.
↪169547693e-07j)   (9.29734160856e-05-0.000288183813551j)
9.999167    (2.70012075766e-07-6.29925739168e-08j)     (-1.2874391849e-06-4.
↪174494327e-07j)  (-0.000337151474966-0.000203359800886j)
9.999444   (-6.28929385556e-07+1.26376757642e-07j)  (-2.21335092292e-06+1.
↪02519591583e-06j)     (0.00045149636163-0.000108322082259j)
9.999722    (3.37062730718e-07-2.34723066532e-08j)  (-4.19929617984e-08+1.
↪65355300419e-07j)  (-7.98050818792e-05+0.000242259733087j)
10.000000                   (7.27769346786e-07+0j)                    (3.
↪52033674659e-07+0j)                  (0.000551496430587+0j)

[36001 rows x 3 columns]
```

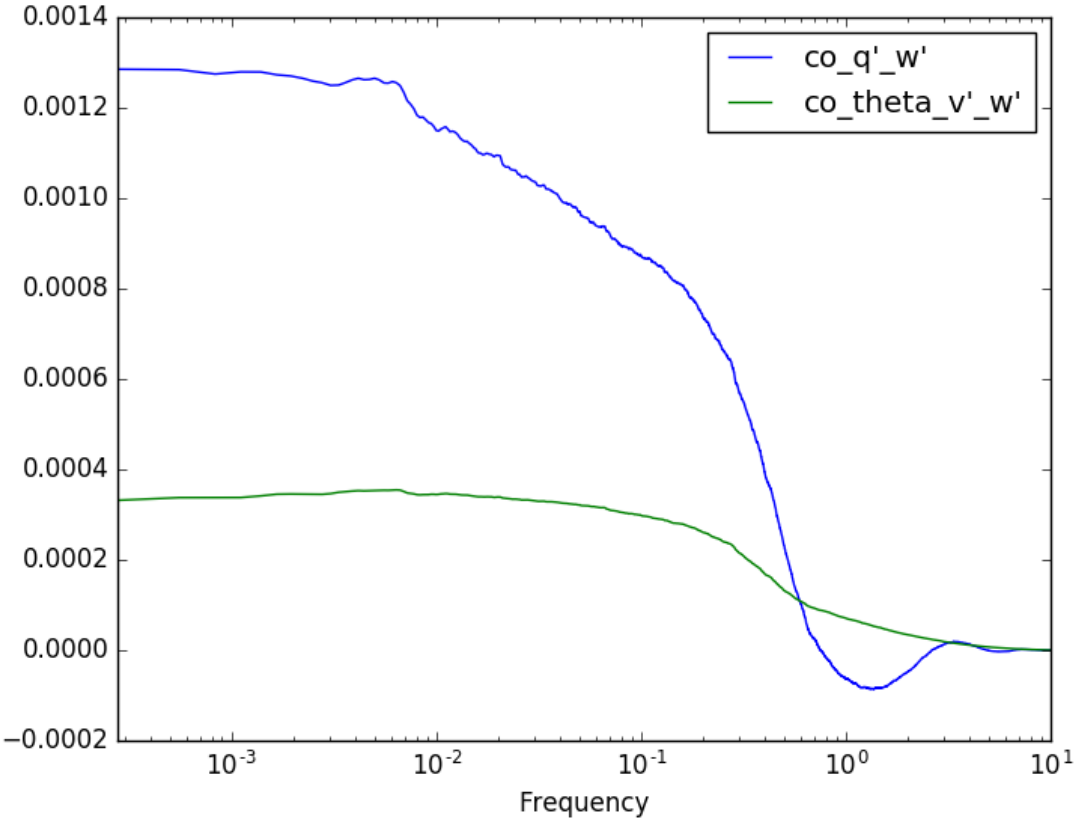
We can then get the cospectra and plot it's binned version (the same can be done with the `.quadrature()` method). It's important to note that, although to pass from cross-spectra to cospectra one can merely do `data.apply(np.real)`, it's recommended to use the `.cospectra()` method or the `pm.spectral.cospectra()` function, since this way the notation on the resulting DataFrame will be correctly passed on, as you can see by the legend in the resulting plot.

```
In [32]: cospectra = crspectra[[r"X_q'_w'", r"X_theta_v'_w'"]].cospectra()

In [33]: cospectra.binned(bins_number=100).plot(loglog=True, style='o')
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7da6d36290>

In [34]: plt.show()
```



Now we can finally obtain an Ogive with the `pm.spectral.Ogive()` function. Note that we plot `Og/Og.sum()` (i.e. we normalize the ogive) instead of `Og` merely for visualization purposes, as the scale of both ogives are too different.

```
In [35]: Og = pm.spectral.Ogive(cospectra)

In [36]: (Og/Og.sum()).plot(logx=True)
Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x7f7d8e5bd1d0>

In [37]: plt.show()
```



Note: Include some examples for spectral corrections.

CHAPTER

FIVE

PYMICRA'S AUTO-GENERATED DOCS

Subpackages

pymicra.algs package

Submodules

pymicra.algs.auxiliar

pymicra.algs.auxiliar.**applyResult**(*result*, *failed*, *df*, *control=None*, *testname=None*, *filename=None*, *falseshow=False*, *index_n=None*)

Auxiliar function to be used with util.qcontrol

Parameters

- **result** (*bool*) – whether the test failed and succeeded
- **failed** (*list*) – list of failed variables. None object if the test was successful
- **control** (*dictionary*) – dictionary whose keys are the names of the tests and items are lists
- **testname** (*string*) – name of the test (has to match control dict)
- **filename** (*string*) – name or path or identifier of the file tested
- **falseshow** (*bool*) – whether to show the failed variables or not

pymicra.algs.auxiliar.**completeHM**(*string*)

Completes %H%M strings for cases when 2hours 0 minutes appear as 020. Should be dropped eventually because this is pretty much a hack that corrects for file configuration

pymicra.algs.auxiliar.**first_last**(*fname*)

Returns first and last lines of a file

pymicra.algs.auxiliar.**lenYear**(*year*)

Calculates the length of a year in days Useful to figure out if a certain year is a leap year

pymicra.algs.auxiliar.**stripDown**(*str, final='', args=['_', '-']*)

Auxiliar function to strip down keywords from symbols

pymicra.algs.auxiliar.**testValid**(*df_valid*, *testname=''*, *falseverbose=True*, *trueverbose=True*, *filepath=None*)

Tests a boolean DataFrane obtained from the test and prints standard output

Parameters

- **df_valid** (*pandas.Series*) – series contaning only True or False values for each of the variables, which should be the indexes
- **testname** (*string*) – the name of the test that generated the True/False values
- **falseverbose** (*bool*) – whether to return which variables caused a false result

- **trueverbose** (*bool*) – whether to print something successful cases

Returns

- **result** (*bool*) – True if the run passed the passed
- **failed** (*list*) – list of failed variables if result==False. None otherwise.

pymicra.algs.general

pymicra.algs.general.**classbin**(*x*, *y*, *bins_number=100*, *function=<function mean>*, *xfunction=<function mean>*, *logscale=True*)

Separates x and y inputs into bins based on the x array. x and y do not have to be ordered.

Parameters

- **x** (*np.array*) – independent variable
- **y** (*np.array*) – dependent variable
- **bins_number** (*int*) – number of classes (or bins) desired
- **function** (*callable*) – funtion to be applied to both x and y-bins in order to smooth the data
- **logscale** (*boolean*) – whether or not to use a log-spaced scale to set the bins

Returns

- *np.array* – x binned
- *np.array* – y binned

pymicra.algs.general.**diff_central**(*x*, *y*)

Applies the central finite difference scheme

Parameters

- **x** (*array*) – independent variable
- **y** (*array*) – dependent variable

Returns **dydx** – the dependent variable differentiated

Return type array

pymicra.algs.general.**file_len**(*fname*)

Returns length of a file through piping bash's function wc

Parameters **fname** (*string*) – path of the file

pymicra.algs.general.**find_nearest**(*array*, *value*)

Smart and small function to find the index of the nearest value, in an array, of some other value

Parameters

- **array** (*array*) – list or array
- **value** (*float*) – value to look for in the array

pymicra.algs.general.**fitByDate**(*data*, *degree=1*, *rule=None*)

Given a pandas DataFrame with the index as datetime, this routine fit a n-degree polynomial to the dataset

Parameters

- **data** (*pd.DataFrame, pd.Series*) – dataframe whose columns have to be fitted
- **degree** (*int*) – degree of the polynomial. Default is 1.
- **rule** (*str*) – pandas offside string. Ex.: "10min".

pymicra.algs.general.**fitWrap**(*x*, *y*, *degree=1*)
A wrapper to numpy.polyfit and numpy.polyval that fits data given an x and y arrays. This is specifically designed to be used with by pandas.DataFrame.apply method

Parameters

- **x** (*array, list*) –
- **y** (*array, list*) –
- **degree** (*int*) –

pymicra.algs.general.**get_index**(*x*, *to_look_for*)
Just like the .index method of lists, except it works for multiple values

Parameters

- **x** (*list or array*) – the main array
- **to_look_for** (*list or array*) – the subset of the main whose indexes are desired

Returns **indexes** – array with the indexes of each element in y

Return type np.array

pymicra.algs.general.**get_notation**(*notation_def*)
Auxiliar function ro retrieve notation

pymicra.algs.general.**inverse_normal_cdf**(*mu*, *sigma*)
Applied the inverse normal cumulative distribution

mu: mean sigma: standard deviation

pymicra.algs.general.**latexify**(*variables*, *math_mode=True*)

pymicra.algs.general.**limitedSubs**(*data*, *max_interp=3*, *func=<function <lambda>>*)
Substitute elements for NaNs if a certain conditions given by fund is met at a maximum of max_interp times in a row. If there are more than that number in a row, then they are not substituted.

Parameters

- **data** (*pandas.dataframe*) – data to be interpolated
- **max_interp** (*int*) – number of maximum NaNs in a row to interpolate
- **func** (*function*) – function of x only that determines the which elements become NaNs. Should return only True or False.

Returns **df** – dataframe with the elements substituted

Return type pandas.dataframe

pymicra.algs.general.**limited_interpolation**(*data*, *maxcount=3*)
Interpolates linearly but only if gap is smaller of equal to maxcout

Parameters

- **data** (*pandas.DataFrame*) – dataset to interpolate
- **maxcount** (*int*) – maximum number of consecutive NaNs to interpolate. If the number is smaller than that, nothing is done with the points.

pymicra.algs.general.**line2date**(*line*, *dlconfig*)
Gets a date from a line of file according to dataloggerConfig object.

Parameters

- **line** (*string*) – line of file with date inside
- **dlconfig** (*pymicra.dataloggerConfig*) – configuration of the datalogger

Returns **timestamp**

Return type datetime object

pymicra.algs.general.**mad**(*data*, *axis=None*)

pymicra.algs.general.**name2date**(*filename*, *dlconfig*)
Gets a date from a the name of the file according to a datalogger config object

Parameters

- **filename** (*string*) – the (base) name of the file
- **dlconfig** (*pymicra.dataloggerConfig*) – configuration of the datalogger

Returns

- **cdate** (*datetime object*)
- *Warning: Needs to be optimized in order to read question markers also after the date*

pymicra.algs.general.**parseDates**(*data*, *dataloggerConfig=None*, *date_col_names=None*, *clean=True*, *verbose=False*, *connector=''*)
Author: Tomas Chor date: 2015-08-10 This routine parses the date from a pandas DataFrame when it is divided into several columns

Parameters

- **data** (*pandas DataFrame*) – dataFrame whose dates have to be parsed
- **date_col_names** (*list*) – A list of the names of the columns in which the date is divided the naming of the date columns must be in accordance with the datetime directives, so if the first column is only the year, its name must be *%Y* and so forth. see https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior
- **connector** (*string*) – should be used only when the default connector causes some conflit
- **first_time_skip** (*int*) – the offset (mostly because of the bad converting done by LBA
- **clean** (*bool*) – remove date columns from data after it is introduced as index

Returns data indexed by timestamp

Return type pandas.DataFrame

pymicra.algs.general.**resample**(*df*, *rule*, *how=None*, ***kwargs*)
Extends pandas resample methods to index made of integers

pymicra.algs.general.**splitData**(*data*, *rule='30min'*, *return_index=False*, ***kwargs*)
Splits a given pandas DataFrame into a series of "rule"-spaced DataFrames

Parameters

- **data** (*pandas dataframe*) – data to be split
- **rule** (*str or int*) –

  If it is a string, it should be a pandas string offset. Some possible values (that should be followed by an integer) are: D calendar day frequency W weekly frequency M month end frequency MS month start frequency Q quarter end frequency BQ business quarter endfrequency QS quarter start frequency A year end frequency AS year start frequency H hourly frequency T minutely frequency Min minutely frequency S secondly frequency L milliseconds U microseconds

  If it is a int, it should be the number of lines desired in each separated piece.

  If it is None, then the dataframe isn't separated and a list containing only the full dataframe is returned.

  check it complete at http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases

pymicra.algs.numeric

pymicra.algs.units

pymicra.algs.units.**add**(*elems*, *units*, *inplace_units=False*, *unitdict=None*, *key=None*)
Add elements considering their units

pymicra.algs.units.**convert_cols**(*data*, *guide*, *units*, *inplace_units=False*)
Converts data from one unit to the other

Parameters

- **data** (*pandas.DataFrame*) – to be chanhed from one unit to the other
- **guide** (*dict*) – {names of columns : units to converted to}
- **units** (*dict*) – units dictionary
- **inplace_units** (*bool*) – if inunit is a dict, the dict is update in place. "key" keyword must be provided

pymicra.algs.units.**convert_indexes**(*data*, *guide*, *units*, *inplace_units=False*)
Converts data from one unit to the other

Parameters

- **data** (*pandas.Series*) – to be chanhed from one unit to the other
- **guide** (*dict*) – {names of columns : units to converted to}
- **units** (*dict*) – units dictionary
- **inplace_units** (*bool*) – if inunit is a dict, the dict is update in place. "key" keyword must be provided

pymicra.algs.units.**convert_to**(*data*, *inunit*, *outunit*, *inplace_units=False*, *key=None*)
Converts data from one unit to the other

Parameters

- **data** (*pandas.series*) – to be chanhed from one unit to the other
- **inunit** (*pint.quantity or dict*) – unit(s) that the data is in
- **outunit** (*str*) – convert to this unit
- **inplace_units** (*bool*) – if inunit is a dict, the dict is update in place. "key" keyword must be provided
- **key** (*str*) – if inunit is a dict, it is the name of the variable to be changed

pymicra.algs.units.**divide**(*elems*, *units*, *inplace_units=False*, *unitdict=None*, *key=None*)
Divide elements considering their units

pymicra.algs.units.**multiply**(*elems*, *units*, *inplace_units=False*, *unitdict=None*, *key=None*)
Multiply elements considering their units

pymicra.algs.units.**operate**(*elems*, *units*, *inplace_units=False*, *unitdict=None*, *key=None*, *operation='+'*)
Operate on elements considering their units

Parameters

- **elems** (*list, tuple*) – list of pandas.Series
- **units** (*list, tuple*) – list of pint.units ordered as the elems list
- **inplace_units** (*bool*) – sets dictionary inplace_units
- **unitdict** (*dict*) – dict to be set inplace
- **key** (*str*) – name of variables to be set inplace as dict key

pymicra.algs.units.**parseUnits**(*unitstr*)
Gets unit from string, list of strings, or dict's values, using the UnitRegistry defined in __init__.py

Module contents

pymicra.micro package

Submodules

pymicra.micro.functions

pymicra.micro.functions.**Psi**(*zeta*, *x='tau'*, *zeta0=0.0*)
Integral Monin-Obukhov scale or deviation function, which is the deviation of a variable (x) in relation nto their logarithmic profiles due the stability zeta != 0

Taken from Simpson.ea1998–the.validity.of.similarity.theory;.in.the.roughness.sublayer

Parameters

- **zeta** (*float*) – the stability variable
- **x** (*string*) – the variable. Options are 'tau', 'H', 'E', 'F'
- **zeta0** (*float*) – value of zeta_{0 x}. Only used for the stable case

pymicra.micro.functions.**nondimensionalGrad**(*zeta*, *x=None*)
The nondimensional gradients, defined as:

phi_F(zeta) = kappa*(z-d)*dCdz/c_star phi_H(zeta) = kappa*(z-d)*dTdz/T_star phi_E(zeta) = kappa*(z-d)*dqdz/q_star

Currently using Businger-Dyer eqs.

TODO: generalize coefficients

pymicra.micro.functions.**nondimensionalSTD**(*zeta*, *x=None*)
The nondimensional standard deviation function, defined as:

phi_c(zeta) = sigma_c / c_star

From zahn.ea

pymicra.micro.functions.**rte**(*data*, *w_fluctuations="w"'*, *order=None*)
Returns the Relative Transfer Efficiency in the time domain, rte according to Cancelli, Dias, Chamecki. Dimensionless criteria for the production-dissipation equilibrium of scalar fluctuations and their implications for scalar similarity, Water Resources Research, 2012

Parameters

- **order** (*2-elements list*) – order of variables: if its rte_ab should be [a,b], if its rte_ba should [b,a]
- **TO BE VALIDATED!** (*NEEDS*) –

pymicra.micro.functions.**ste**(*data*, *w_fluctuations="w"'*)
Returns the Symmetric Transfer Efficiency in the time domain, ste according to Cancelli, Dias, Chamecki. Dimensionless criteria for the production-dissipation equilibrium of scalar fluctuations and their implications for scalar similarity, Water Resources Research, 2012

Parameters

- **data** (*pandas dataframe*) – a three-columns dataframe, where one of them should be the vertical velocity fluctuations
- **w_fluctuations** (*str*) – the name of the vertical velocity fluctuations

pymicra.micro.scales

pymicra.micro.scales.**MonObuLen**(*theta_v_star*, *theta_v_mean*, *u_star*, *g=None*)
Redirects to obukhovLen()

pymicra.micro.scales.**MonObuVar**(*L_m*, *siteConf*)
Redirects to stabilityParam()

pymicra.micro.scales.**obukhovLen**(*data*, *units*, *theta_v_mean=None*, *theta_v_mean_unit=None*, *notation=None*, *inplace_units=True*)
Calculates the Monin-Obukhov Length according to:

GARRAT, The atmospheric boundary layer, 1992 (eq. 1.11, p. 10) L = (u_star^2 * theta_v) / (kappa * g * theta_v_star)

KUNDU, Fluid mechanics, 1990 (eq 71, chap. 12, p. 462) L_M = - u_star^3 / (kappa * alpha * g * cov(w,T'))

ARYA, Introduction to micrometeorology (eq. 11.1, p. 214) L = - u_star^3 / (kappa * (g/T_0) * (H_0/(rho*c_p)))

STULL, An introduction to Boundary layer meteorology, 1988 (eq. 5.7b, p. 181) L = - (theta_v * u_star^3) / (kappa *g* cov(w',theta_v'))

pymicra.micro.scales.**stabilityParam**(*L_m*, *siteConf*)
Calculates the Monin-Obukhov Similarity Variable defined as

zeta = (z-d)/Lo where d is the displacement height or zero-plane displacement and L_m is the Monin-Obukhov Length.

pymicra.micro.scales.**turbulentScales**(*data*, *siteConf*, *units*, *notation=None*, *theta_v_mean=None*, *theta_v_mean_unit=None*, *theta_fluct_from_theta_v=True*, *solutes=[]*, *output_as_df=True*, *inplace_units=True*)
Calculates characteristic lengths for data

The names of the variables are retrived out the dictionary. You can update the dictionary and change the names by using the notation_defs keyworkd, which is a notation object

Parameters

- **data** (*pandas.DataFrame*) – dataset to be used. It must either be the raw and turbulent data, or the covariances of such data
- **siteConf** (*pymicra.siteConfig object*) – has the site configurations to calculate the obukhovLen
- **units** (*dict*) – dict units for the input data
- **output_as_df** (*boolean*) – True if you want the output to be a one-line pandas.DataFrame. A pd.Series will be output if False.
- **inplace_units** (*bool*) – whether or not to update the units dict in place

Returns depending on return_as_df

Return type pandas.Series or pandas.Dataframe

pymicra.micro.spectral

pymicra.micro.spectral.**Ogive**(*df*, *no_nan=True*)
Integrates the Ogive from Coespectra

Parameters **df** (*dataframe*) – cospectrum to be integrated

pymicra.micro.spectral.**correctLag**(**args*, ***kwargs*)
Identifies and correct lags between data, assuming the vertical wind velocity has lag 0.

pymicra.micro.spectral.**cospectra**(**args*, ***kwargs*)
Gets cospectra from cross-spectrum

pymicra.micro.spectral.**hfc_Dias_ea_16**(*cross_spec*, *T*)
Applies correction to high frequencies using the quadrature. Xab_ab = Co_ab - i Qu_ab

Co_recovered = Co_ab + 2*pi*n*T*Qu_ab

Parameters

- **cross_spec** (*series of dataframe*) – the cross spectrum whose coespectrum you'd like to correct
- **T** (*float*) – response time to use

pymicra.micro.spectral.**hfc_Massman_Ibrom_08**(*df*)

pymicra.micro.spectral.**hfc_zeroQuad**(*slow_spec*, *freqs*, *T*)
Applies a correction factor to the spectrum of a slow-measured variable based on the response-time T. Quadrature must be analytically zero!

Parameters

- **slow_spec** (*numpy.array or series*) – the spectrum to be corrected (not cross-spectrum!)
- **freqs** (*numpy.array*) – frequencies
- **T** (*float*) – the response-time

Returns **spec** – the recovered array

Return type numpy.array

pymicra.micro.spectral.**phaseCorrection**(*cross_spec*, *T*)

pymicra.micro.spectral.**quadrature**(**args*, ***kwargs*)
Gets quadrature from cross-spectrum

pymicra.micro.spectral.**recspeAux**(*df*, *T*)
Wrapper to make hfc_zeroQuad work in a pandas.DataFrame

pymicra.micro.spectral.**zeroQuadCorrection**(**args*, ***kwargs*)
Applies the correction assuming that the quadrature is zero to a dataframe with spectra

Wrapper to make hfc_zeroQuad work in a pandas.DataFrame

Parameters

- **df** (*pandas.DataFrame*) – dataframe with cospectra to correct
- **T** (*float*) – response time to be used

pymicra.micro.util

pymicra.micro.util.**eddyCovariance**(*data*, *units*, *wpl=True*, *get_turbulent_scales=True*, *site_config=None*, *output_as_df=True*, *notation=None*, *theta_fluct_from_theta_v=True*, *inplace_units=True*, *solutes=[]*)
Get fluxes from the turbulent fluctuations

Parameters

- **data** (*pandas.DataFrame*) – dataframe with the characteristic lengths calculated
- **units** (*dict*) – units dictionary
- **wpl** (*boolean*) – whether or not to apply WPL correction on the latent heat flux and solutes flux

- **get_turbulent_scales** (*bool*) – whether or not to use getScales to return turbulent scales
- **site_config** (*pymica.siteConfig*) – siteConfig object to pass to getScales if get_turbulent_scales==True
- **notation** (*pymicra.Notation*) – object that holds the notation used in the dataframe
- **inplace_units** (*bool*) – whether or not to treat the units inplace
- **solutes** (*list*) – list that holds every solute considered for flux

pymicra.micro.util.**preProcess**(*data*, *units*, *notation=None*, *use_means=False*, *expand_temperature=True*, *rho_air_from_theta_v=True*, *inplace_units=True*, *theta=None*, *theta_unit=None*, *solutes=[]*)

Calculates moist and dry air densities, specific humidity mass density and other important variables using the variables provided in the input DataFrame.

Parameters

- **data** (*pandas.DataFrame*) – dataframe with micrometeorological measurements
- **units** (*dict*) – units dictionary with the columns of data as keys
- **notation** (*pymicra.notation*) – defining notation used in data
- **rho_air_from_theta_v** (*bool*) – whether to use theta_v to calculate air density or theta
- **inplace_units** (*bool*) – treat units inplace or not
- **theta** (*pandas.Series*) – auxiliar theta measurement to be used if rho_air_from_theta_v==False
- **theta_unit** (*pint.quantity*) – auxiliar theta measurement's unit to be used if rho_air_from_theta_v==False
- **solutes** (*list*) – list of string where each string is a solute to be considered

Returns **data** – dataframe with original columns and new calculated ones

Return type pandas.DataFrame

pymicra.micro.util.**rotateCoor**(*data*, *notation=None*, *how='2d'*)

Module contents

Submodule that holds the specifically micrometeorological functions and variables

pymicra.constants

Defines some useful constants

pymicra.core

Defines classes that are the basis of Pymicra

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

Bases: object

Holds the notation used in every function of pymicra except when told otherwise.

build(*from_level=0*)

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

Given notation for means, fluctuations, and etc, along with names of variables, this method builds the notation for mean h2o concentration, virtual temperature fluctuations and so on.

Parameters

- **self** (*pymicra.Notation*) – notation to be built
- **from_level** (*int*) – level from which to build. If 0, build everything from scratch and higher notations will be overwritten. If 1, skip one step in building process. Still to be implemented!

Returns Notation object with built notation

Return type pymicra.Notation

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

Bases: object

This class defines a specific configuration of a data file

Parameters

- **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.

get_date_cols()

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

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

Bases: object

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

Check help(pm.siteConfig.__init__) for other parameters

Parameters **from_file** (*str*) – path to .site file which contais other keywords

pymicra.data

Defines functions useful to generic signal data

pymicra.data.**bulkCorr**(*data*)
Bulk correlation coefficient according

Bulk correlation coefficient according to Cancelli, Dias, Chamecki. Dimensionless criteria for the production of... doi:10.1029/2012WR012127

Parameters **data** (*pandas.dataframe*) – a two-columns dataframe

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

Returns **spectrum** – whose column is the spectrum or coespectrum of the input dataframe

Return type pandas.DataFrame

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 fluctuations of the input data

Return type 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
- **This fuction approximates table A.6 from Bendat&Piersol as a normal distribution.** (*WARNING!*) –
- **may no be true, since they do not express which distribution they use to construct** (*This*) –
- **table. However, in the range 9<N<101, this approximation is as good as 5% at N=10** (*their*) –
- **0.1% at N=100.** (*and*) –
- **not adapted for dataframes** (*Still*) –

pymicra.data.**rotate2D**(*data*, *notation=None*)
Rotates the coordinates of wind data

Parameters

- **data** (*pandas DataFrame*) – the dataFrame to be rotated
- **notation** (*notation object*) – a notation object to know which are the wind variables

Returns the complete data input with the wind components rotated

Return type pandas.DataFrame

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)

Returns **spectra** – whose column is the spectrum or coespectrum of the input dataframe

Return type pandas.DataFrame

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 trends of data input

Return type pandas.DataFrame or pandas.Series

pymicra.decorators

Defines useful decorators for Pymicra

pymicra.decorators.**autoassign**(**names*, ***kwargs*)

Decorator that automatically assigns keywords as atributes

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

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*)

Defines a decorator to make functions work on both pandas.Series and DataFrames

Parameters **convert_out** (*bool*) – if True, also converts output back to Series if input is Series

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

Defines a decorator that transforms Series into DataFrame

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

Defines some useful functions to aid on the input/output of data

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

Reads one datafile using pandas.read_csv()

Parameters

- **variables** (*list or dict*) – keys are columns and values are names of variable
- **only_named_columns** (*bool*) – if True, don't read columns that don't appear on variables' keys
- **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 dictionary containing the names of each variable in the file (if dict, the keys must be ints)

Returns pandas.DataFrame object

Return type pandas.DataFrame

pymicra.io.**readDataFiles**(*flist*, *verbose=0*, ***kwargs*)
Reads data from a list of files by calling readDataFile individually for each entry

Parameters

- **flist** (*sequence of strings*) – files to be parsed
- **verbose** (*bool*) – whether to print
- ****kwargs** – readDataFile kwargs

Returns data

Return type pandas.DataFrame

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

Parameters

- **filename** (*string*) – path of the csv file to read
- ****kwargs** – to be passed to pandas.read_csv

Returns

- **df** (*pandas.DataFrame*) – dataframe with the data
- **unitsdic** (*dictionary*) – dictionary with the variable names as keys and the units as values

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

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

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

Then the .config should have: variables={0:'%Y-%m-%d %H:%M:%S',1:'u',2:'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_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

Parameters **sitefile** (*str*) – path to the site configuration file

Returns pymicra site configuration object

Return type pymicra.siteConfig

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.

Parameters

- **flist** (*list or string*) – either list or names of files (dataFrame will be one concatenated dataframe) or the name of one file

- **datalogger** (*pymicra.dataloggerConfig object*) – configuration of the datalogger which is from where all the configurations of the file will be taken
- **parse_date** (*bool*) – whether or not to index the data by date. Note that if this is False many of the functionalities of pymicra will be lost. (i.d. there are repeated timestamps)
- **verbose** (*int, bool*) – verbose level

Returns

- *pandas.DataFrame* – data contained in the files in flist
- *dict (optional)* – units of the data

pymicra.methods

Defines some methods. Some have functions defined here but most use functions defined elsewhere. This is done by monkey-patching Pandas.

pymicra.methods.**binwrapper**(*self*, *clean_index=True*, ***kwargs*)

Method to return binned data from a dataframe using the function classbin

pymicra.physics

Module that contains physical functions. They are all general use, but most are specially frequent in micrometeorology.

TO DO LIST:

- ADD GENERAL SOLAR ZENITH CALCULATION
- ADD FOOTPRINT CALCULATION?

pymicra.physics.**R_moistAir**(*q*)

Calculates the gas constant for umid air from the specific humidity q

Parameters **q** (*float*) – the specific humidity in g(water)/g(air)

Returns **R_air** – the specific gas constant for humid air in J/(g*K)

Return type float

pymicra.physics.**airDensity_from_theta**(*data*, *units*, *notation=None*, *inplace_units=True*, *use_means=False*, *theta=None*, *theta_unit=None*)

Calculates moist air density using theta measurements

Parameters

- **data** (*pandas.DataFrame*) – dataset to add rho_air
- **units** (*dict*) – units dictionary
- **notation** (*pymicra.notation*) – notation to be used
- **inplace_units** (*bool*) – whether or not to treat units inplace
- **use_means** (*bool*) – use the mean of theta or not when calculating
- **theta** (*pandas.Series*) – auxiliar theta measurement
- **theta_unit** (*pint.quantity*) – auxiliar theta's unit

pymicra.physics.**airDensity_from_theta_v**(*data*, *units*, *notation=None*, *inplace_units=True*, *use_means=False*, *return_full_df=True*)

Calculates moist air density using p = rho R_dry T_virtual

Parameters

- **data** (*pandas.DataFrame*) – data to use to calculate air density
- **units** (*dict*) – dictionary of units
- **notation** (*pymicra.Notation*) – notation to be used
- **inplace_units** (*bool*) – whether or not to update the units inplace. If False, units are returns too
- **use_means** (*bool*) – whether or not to use averages of pressure and virtual temperature, instead of the means plus fluctuations

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_units=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**(*data*, *units*, *notation=None*, *inplace_units=True*, *solutes=[]*)

Calculates density of solutes based on their molar concentration (ppmv, ppbv and etc), not to be confused with mass concentration (ppm, ppb and etc).

Uses the relation $\rho_x = \frac{Cp}{\theta R_x}$

Parameters

- **data** (*pandas.DataFrame*) – dataset of micromet variables
- **units** (*dict*) – dict of pint units
- **notation** (*pymicra.Notation*) – notation to be used here
- **inplace_units** (*bool*) – whether or not to treat the dict units in place
- **solutes** (*list or tuple*) – solutes to consider when doing this conversion

Returns input data plus calculated density columns

Return type pandas.DataFrame

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

Parameters **T** (*float*) – thermodynamic temperature

Returns

Return type saturated vapor pressure of water (in kPa)

pymicra.physics.**theta_from_theta_s**(*data*, *units*, *notation=None*, *return_full_df=True*, *inplace_units=True*)

Calculates thermodynamic temperature using sonic temperature measurements

From Schotanus, Nieuwstadt, de Bruin; DOI 10.1007/BF00164332

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

$theta_s \approx theta(1 + 0.51q)$

Parameters

- **data** (*pandas.dataframe*) – dataset
- **units** (*dict*) – units dictionary
- **notation** (*pymicra.Notation*) –

Returns thermodynamic temperature

Return type pandas.Series

pymicra.physics.**theta_from_theta_v**(*data*, *units*, *notation=None*, *return_full_df=True*, *inplace_units=True*)

Calculates thermodynamic temperature from virtual temperature measurements

$theta_v \approx theta(1 + 0.61q)$

Parameters

- **data** (*pandas.dataframe*) – dataset
- **units** (*dict*) – units dictionary
- **notation** (*pymicra.Notation*) –

Returns virtual temperature

Return type pandas.DataFrame or Series

pymicra.physics.**theta_std_from_theta_v_fluc**(*data*, *units*, *notation=None*)

Derived from theta_v = theta(1 + 0.61 q)

Parameters

- **data** (*pandas.dataframe*) – dataframe with q, q', theta, theta_v'
- **units** (*dict*) – units dictionary
- **notation** (*pymicra.Notation*) – Notation object or None

Returns standard deviation of the thermodynamic temperature

Return type float

pymicra.tests

This module contains functions that test certain conditions on pandas.dataframes to be used with the qcontrol().

They all return True for the columns that pass the test and False for the columns that fail the test.

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

Parameters

- **data** (*pandas.DataFrame*) – to apply RAT to each column
- **detrend_kw** (*dict*) – keywords to pass to pymicra.detrend
- **RAT_vars** – list of variables to which apply the RAT
- **RAT_points** (*int*) – if it's an int N, then reduce each column to N points by averaging. If None, then the whole columns are used
- **RAT_significance** (*float*) – significance with which to apply the RAT

Returns **valid** – True or False for each column. If True, column passed the test

Return type pd.Series

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

Parameters

- **data** (*pandas dataframe*) – dataframe to be checked
- **tables** (*pandas.dataframe*) – dataframe with the lower and upper limits for variables
- **max_percent** (*float*) – number from 0 to 100 that represents the maximum percentage of faulty runs accepted by this test.

Returns

- **df** (*pandas.DataFrame*) – input data but with the faulty points substituted by the linear trend of the run.
- **valid** (*pandas.Series*) – True for the columns that passed this test, False for the columns that didn't.

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

Parameters

- **fname** (*string*) – path of the file to check
- **numlines** (*int*) – correct number of lines that the file has to have

Returns Either with True or False

Return type pandas.Series

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)

Parameters

- **data** (*pandas.dataframe*) – data to de-spike
- **chunk_size** (*str, int*) – size of chunks to consider. If str should be pandas offset string. If int, number of lines.
- **detrend** (*bool*) – whether to detrend the data and work with the fluctuations or to work with the absolute series.
- **detrend_kw** (*dict*) – dict of keywords to pass to pymicra.trend in order to detrend data (if detrend==True).
- **visualize** (*bool*) – whether of not to visualize the interpolation ocurring

- **vis_col** (*str, int or list*) – the column(s) to visualize when seeing the interpolation (only effective if visualize==True)
- **max_consec_spikes** (*int*) – maximum number of consecutive spikes to actually be considered spikes and substituted
- **cut_func** (*function*) – function used to define spikes
- **replace_with** (*str*) – method to use when replacing spikes. Options are 'interpolation' or 'trend'.
- **max_percent** (*float*) – maximum percentage of spikes to allow.

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

Parameters

- **data** (*pandas.DataFrame*) – dataset whose standard deviation to check
- **tables** (*pandas.DataFrame*) – dataset containing the standard deviation limits for each column
- **detrend** (*bool*) – whether to work with the absolute series and the fluctuations
- **detrend_kw** (*dict*) – keywords to pass to pymicra.detrend with detrend==True
- **chunk_size** (*str*) – pandas datetime offset string

Returns **valid** – contatining True of False for each column. True means passed the test.

Return type pandas.Series

pymicra.util

Module for general utilities

- INCLUDE DECODIFICAION OF DATA?
- INCLUDE UPPER LIMIT TO STD TEST?
- INCLUDE DROPOUT TEST
- INCLUDE THIRD MOMENT TEST
- CHANGE NOTATION IN QCONTROL'S SUMMARY

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*)

Parameters

- **correct** (*pandas.DataFrame*) – dataset with the correct averages
- **drifted** (*pandas.DataFrame*) – dataset with the averages that need to be corrected
- **correct_drifted_vars** (*dict*) – dictionary where every key is a var in the right dataset and its value is its correspondent in the drifted dataset

- **get_fit** (*bool*) – whether ot not to fit a linear relation between both datasets. Generally slow. Should only be done once
- **write_fit** (*bool*) – if get_fit == True, whether or not to write the linear fit to a file (recommended)
- **fit_file** (*string*) – where to write the linear fit (if one is written) or from where to read the linear fit (if no fit is written)
- **apply_fit** (*bool*) – whether of not to apply the lineat fit and correct the data (at least get_fit and fit_file must be true)
- **show_plot** (*bool*) – whether or not to show drifted vs correct plot, to see if it's a good fit
- **units** (*dict*) – if given, it creates a {file_file}.units file, to tell write down in which units data has to be in order to be correctly corrected
- **return_index** (*bool*) – whether to return the indexes of the used points for the calculation. Serves to check the regression

Returns **outdf** – drifted dataset corrected with right dataset

Return type pandas.DataFrame

pymicra.util.**qcontrol**(*files*, *datalogger_config*, *read_files_kw={'return_units': False, '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.
- **NaN's 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 predetermined 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.

Parameters

- **files** (*list*) – list of filepaths
- **datalogger_config** (*pymicra.datalogerConf object or str*) – datalogger configuration object used for all files in the list of files or path to a dlc file.
- **read_files_kw** (*dict*) – keywords to pass to pymicra.timeSeries. Default is {'parse_dates':False} because parsing dates at every file is slow, so this makes the whole process faster. However, {'parse_dates':True, 'clean_dates':False} is recommended if time is not a problem because the window and chunk_size keywords may be used as, for example '2min', instead of 1200, which is the equivalent number of points.
- **file_lines** (*int*) – number of line a "good" file must have. Fails if the run has any other number of lines.
- **begin_date** (*str*) – dates before this automatically fail.
- **end_date** (*str*) – dates after this automatically fail.
- **std_limits** (*dict*) – keys must be names of variables and values must be upper limits for the standard deviation.
- **std_detrend** (*bool*) – whether or not to work with the fluctations of the data on the spikes and standard deviation test.
- **std_detrend_kw** – keywords to be passed to pymicra.detrend specifically to be used on the STD test.
- **lower_limits** (*dict*) – keys must be names of variables and values must be lower absolute limits for the values of each var.
- **upper_limits** (*dict*) – keys must be names of variables and values must be upper absolute limits for the values of each var.
- **dif_limits** (*dict*) – keys must be names of variables and values must be upper limits for the maximum difference of values that the linear trend of the run must have.
- **maxdif_detrend** (*bool*) – whether to detrend data before checking for differences.
- **maxdif_detrend_kw** (*dict*) – keywords to pass to pymicra.detrend when detrending for max difference test.
- **maxdif_trend** (*bool*) – whether to check for differences using the trend, instead of raw points (which can be the fluctuations or the original absolute values of data, depending if maxdif_detrend==True or False).
- **maxdif_trend_kw** (*dict*) – keywords to pass to pymicra.detrend when trending for max difference test. dictionary of keywords to pass to pymicra.data.trend. This is used in the max difference test, since the difference is taken between the max and min values of the trend, not of the series. Default = {'how':'linear'}.
- **spikes_test** (*bool*) – whether or not to check for spikes.

- **spikes_detrend** (*bool*) – whether or not to work with the fluctations of the data on the spikes test.
- **spikes_detrend_kw** (*dict*) – keywords to pass to pymicra.detrend when detrending for spikes.
- **visualize_spikes** (*bool*) – whether or not to plot the spikes identification and interpolation (useful for calibration of spikes_func). Only one column is visualized at each time. This is set with the spikes_vis_col keyword.
- **spikes_vis_col** (*str*) – column to use to visualize spikes.
- **spikes_func** (*function*) – function used to look for spikes. Can be defined used numpy/pandas notation for methods with lambda functions. Default is: lambda x: (abs(x - x.mean()) > abs(x.std()*4.))
- **replace_with** (*str*) – method to use when replacing the spikes. Options are 'interpolation' and 'trend'.
- **max_consec_spikes** (*int*) – limit of consecutive spike points to be interpolated. After this spikes are left as they are in the output.
- **accepted_percent** (*float*) – limit percentage of spike points in the data. If spike points represent a higher percentage than the run fails the spikes check.
- **chunk_size** (*str*) – string representing time length of chunks used in the spikes and standard deviation check. Default is "2Min". Putting None will not separate in chunks. It's recommended to use rolling functions in this case (might be slow).
- **RAT** (*bool*) – whether or not to perform the reverse arrangement test on data.
- **RAT_vars** (*list*) – list containing the name of variables to go through the reverse arrangement test. If None, all variables are tested.
- **RAT_points** (*int*) – number of final points to apply the RAT. If 50, the run will be averaged to a 50-points run.
- **RAT_significance** – significance level to apply the RAT.
- **RAT_detrend_kw** – keywords to be passed to pymicra.detrend specifically to be used on the RA test. {"how":"linear"} is strongly recommended for this case.
- **trueverbose** (*bool*) – whether or not to show details on the successful runs.
- **falseverbose** (*bool*) – whether or not to show details on the failed runs.
- **trueshow** (*bool*) – whether of not to plot the successful runs on screen.
- **trueshow_vars** (*list*) – list of columns to plot if run is successfull.
- **falseshow** (*bool*) – whether of not to plot the failed runs on screen.
- **outdir** (*str*) – name of directory in which to write the successful runs. Directory must already exist.
- **summary_file** (*str*) – path of file to be created with the summary of the runs. Will be overwriten if already exists.

Returns **ext_summary** – dict with the extended summary, which has the path of the files that got "stuck" in each test along with the successful ones

Return type pandas.DataFrame

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.

Parameters

- **files** (*list*) – list of file paths to be separated
- **dlconfig** (*pymicra datalogger configuration file*) – to tell how the dates are displayed inside the file
- **outformat** (*str*) – the format of the file names to output
- **outdir** (*str*) – the path to the directory in which to output the files
- **verbose** (*bool*) – whether to print to the screen
- **firstflag** (*str*) – flag to put after the name of the file for the first file to be created
- **lastflag** (*str*) – flag to put after the name of the fle for the last file to be created
- **save_ram** (*bool*) – if you have an amount of files that are to big for pandas to load on your ram this should be set to true
- **frequency** – the frequency in which to separate
- **quoting** (*int*) – for pandas (see read_csv documentation)
- **edges** (*use*) – use this carefully. This concatenates the last few lines of a file to the first few lines of the next file in case they don't finish on a nice round time with respect to the frequency

Returns

Return type None

- genindex
- modindex
- search

p

O

P

Q

R

S

T

Z