HydroBox Documentation

Release 0.1.0

Mirko Mälicke

May 17, 2018

Contents:

CHAPTER 1

Stable branch

CHAPTER 2

Development branch

Warning: This documentation is by no means finished and in development. Kind of everything here might be subject to change.

CHAPTER 3

About

The HydroBox package is a toolbox for hydrological data analysis developed at the Chair of Hydrology at the Karlsruhe Institute of Technology (KIT). The HydroBox has a submodule called toolbox, which is a collection of functions and classes that accept common numpy and pandas input formats and wrap around scipy functionality. Its purpose is:

- to speed up common hydrological data analysis tasks
- to integrate fully with custom numpy/pandas/scipy code

Jump directly to the *installation section* or *get started*.

3.1 Installation Guide

3.1.1 PyPi

Install the Hydrobox using pip. The latest version on PyPI can be installed using pip:

```
pip install hydrobox
```

3.1.2 GitHub

There might be a more recent version on GitHub available. It can be installed as follows:

```
git clone https://github.com/mmaelicke/hydrobox.git
cd hydrobox
pip install -r requirements.txt
pip install -e .
```

3.2 Getting Started

3.2.1 Under Development

Warning: This site is under construction

3.3 Examples

Important: These examples should help you tp get started with most of the functionality. However, some examples and tools might need a specific database backend or service running on the machine. In this case you have to install the requirements. The *reference section* should guide you to the correct function with more detailed information on the setup.

3.3.1 Discharge Tools

FDC from random data

Workflow

The workflow in this example will generate some random data and applies two processing steps to illustrate the general idea. All tools are designed to fit seamlessly into automated processing environments like WPS servers or other workflow engines.

The workflow in this example:

1. generates a ten year random discharge time series from a gamma distribution
2. aggregates the data to daily maximum values
3. creates a flow duration curve
4. uses python to visualize the flow duration curve

Generate the data

```
# use the ggplot plotting style
In [1]: import matplotlib as mpl

In [2]: mpl.style.use('ggplot')

In [3]: from hydrobox import toolbox

# Step 1:
In [4]: series = toolbox.io.timeseries_from_distribution(
   ...:     distribution='gamma',
   ...:     distribution_args=[2, 0.5],  # [location, scale]
   ...:     start='200001010000',        # start date
   ...:     end='201001010000',          # end date
```

(continues on next page)

(continued from previous page)

```
   ...:     freq='15min',                # temporal resolution
   ...:     size=None,                   # set to None, for inferring
   ...:     seed=42                      # set a random seed
   ...: )
   ...:

In [5]: print(series.head())
2000-01-01 00:00:00    1.196840
2000-01-01 00:15:00    0.747232
2000-01-01 00:30:00    0.691142
2000-01-01 00:45:00    0.691151
2000-01-01 01:00:00    2.324857
Freq: 15T, dtype: float64
```

Apply the aggregation

```
In [6]: import numpy as np

In [7]: series_daily = toolbox.aggregate(series, by='1D', func=np.max)

In [8]: print(series_daily.head())
2000-01-01    3.648999
2000-01-02    3.398266
2000-01-03    3.196676
2000-01-04    3.842573
2000-01-05    2.578654
Freq: D, dtype: float64
```

Calculate the flow duration curve (FDC)

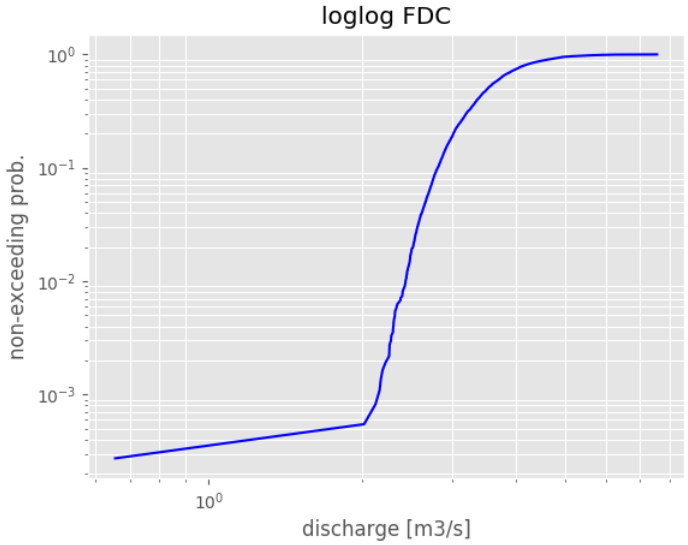
```
# the FDC is calculated on the values only
In [9]: fdc = toolbox.flow_duration_curve(x=series_daily.values,     # an FDC does␣
↪not need a DatetimeIndex
   ...:                                   plot=False                # return values, not␣
↪a plot
   ...:                                  )
   ...:

In [10]: print(fdc[:5])
[0.0002736  0.0005472  0.00082079 0.00109439 0.00136799]

In [11]: print(fdc[-5:])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\[0.99863201 0.99890561 0.
↪99917921 0.9994528  0.9997264 ]
```

The first output line shows the first five exceeding probabilities, while the second line shows the last five values. The output as `numpy.ndarray` is especially useful when the output is directed into another analysis function or is used inside a workflow engine. This way the plotting and styling can be adapted to the use-case.

However, in case you are using hydrobox in a pure Python environment, most tools can be directly used for plotting. At the current stage `matplotlib` is the only plotting possibility.

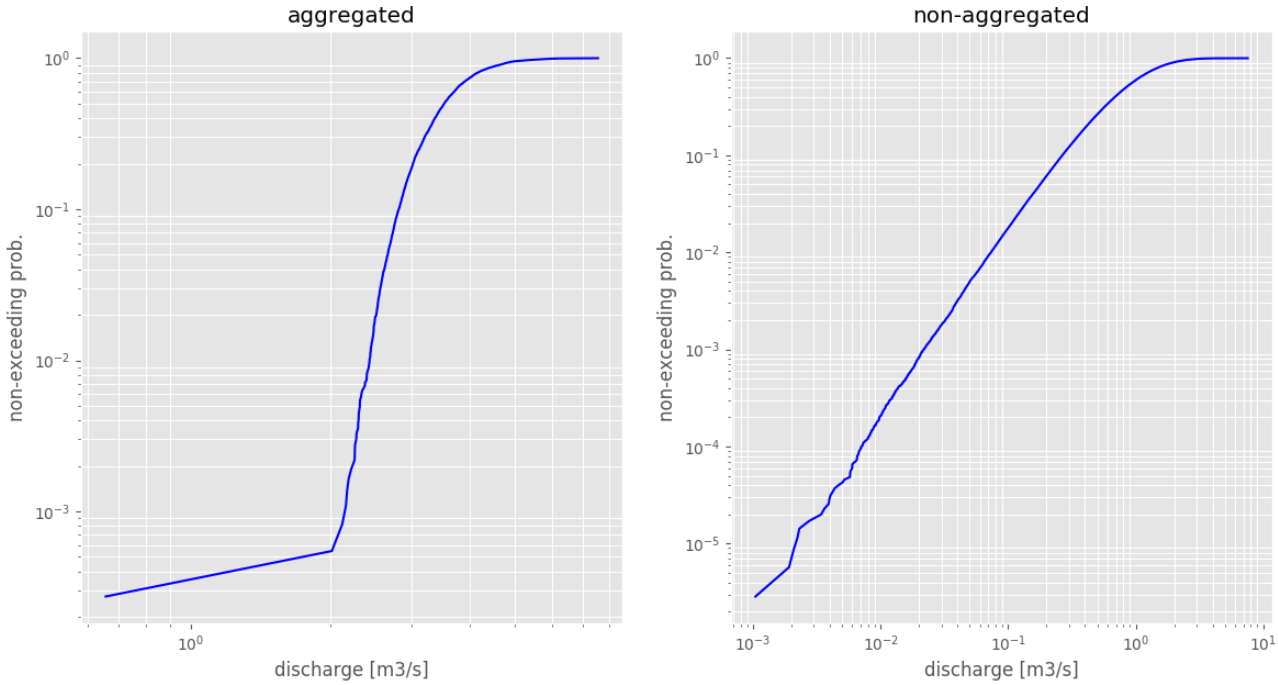
Plot the result

```
# If not encapsulated in a WPS server, the tool can also plot
In [12]: toolbox.flow_duration_curve(series_daily.values);
```



With most plotting functions, it is also possible to embed the plots into existing figures in order to fit seamlessly into reports etc.

```
In [13]: import matplotlib.pyplot as plt

# build the figure as needed
In [14]: fig, axes = plt.subplots(1,2, figsize=(14,7))

In [15]: toolbox.flow_duration_curve(series_daily.values, ax=axes[0]);

In [16]: toolbox.flow_duration_curve(series.values, ax=axes[1]);

In [17]: axes[0].set_title('aggregated');

In [18]: axes[1].set_title('non-aggregated');

In [19]: plt.show();
```



Reference

See also:

- *flow_duration_curve reference*
- *aggregate reference*

Hydrological Regime

Workflow

The workflow for the *regime function* is very similar to the one presented in the *flow duration curve* section.

In this example, we will use real world data. As the hydrobox is build on top of numpy and pandas, we can easily use the great input tools provided by pandas. This example will load a discharge time series from Hofkirchen in Germany, gauging the Danube river. The data is provided by Gewässerkundlicher Dienst Bayern under a CC BY 4.0 license. Therefore, this example will also illustrate how you can combine pandas and hydrobox to produce nice regime plots with just a few lines of code.

Note: In order to make use of the plotting, you need to run the tools in a Python environment. If you are using e.g. a WPS server calling the tools, be sure to capture the output.

Load the data using pandas

```
# some imports
In [20]: from hydrobox import toolbox

In [21]: import pandas as pd

# Step 1:
In [22]: df = pd.read_csv('./data/discharge_hofkirchen.csv',
   ....:     skiprows=10,            # meta data header, skip this
   ....:     sep=';',                # the cell separator
   ....:     decimal=',',             # german-style decimal sign
   ....:     index_col='Datum',      # the 'date' column
   ....:     parse_dates=[0]         # transform to DatetimeIndex
   ....: )
   ....:

# use only the 'Mittelwert' (mean) column
In [23]: series = df.Mittelwert

In [24]: print(series.head())
Datum
1900-11-01    328.0
1900-11-02    385.0
1900-11-03    422.0
1900-11-04    388.0
1900-11-05    381.0
Name: Mittelwert, dtype: float64
```

Note: The data was downloaded from: Datendownload GKD and is published under CC BY 4.0 license. If you are not using a german OS, note that the file encoding is ISO 8859-1 and you might have to remove the special german signs from the header before converting to UTF-8.

See also:

More information on the `read_csv function` is provided in the pandas documentation.

Output the regime

In order to calculate the regime, without a plot, we can set `plot` to *None*.

```
In [25]: regime = toolbox.regime(series, plot=False)

In [26]: print(regime)
[[534.]
 [558.]
 [639.]
 [698.]
 [671.]
 [677.]
 [604.]
 [538.]
 [482.]
 [437.]
```

(continues on next page)

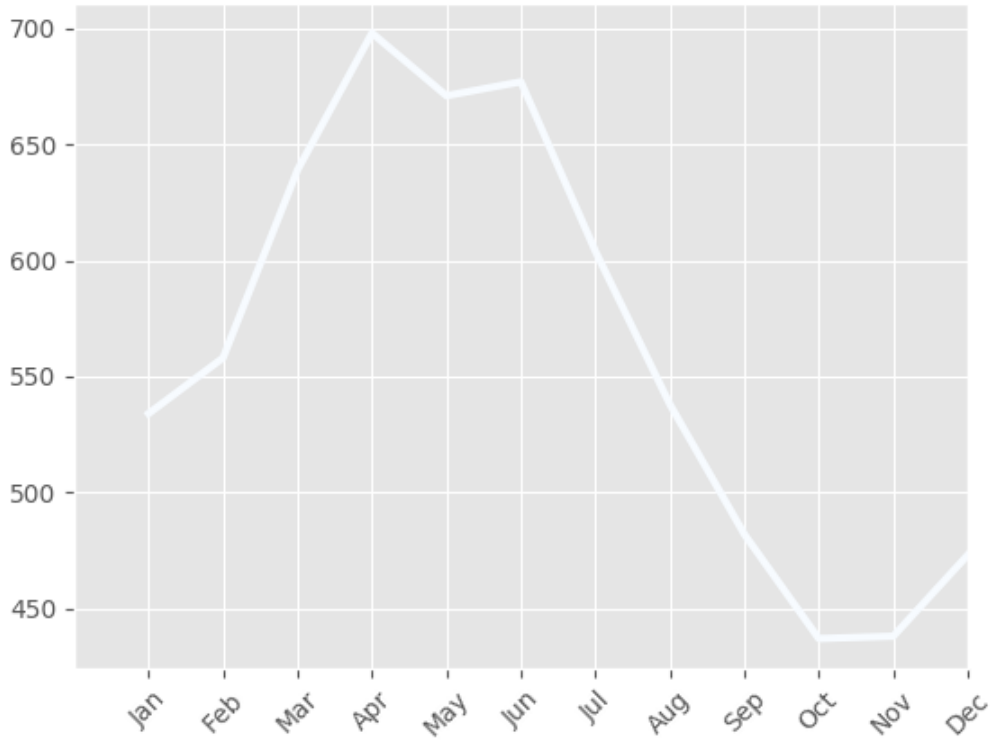
(continued from previous page)

```
 [438.]
 [473.]]
```

These plain numpy arrays can be used in any further custom workflow or plotting.

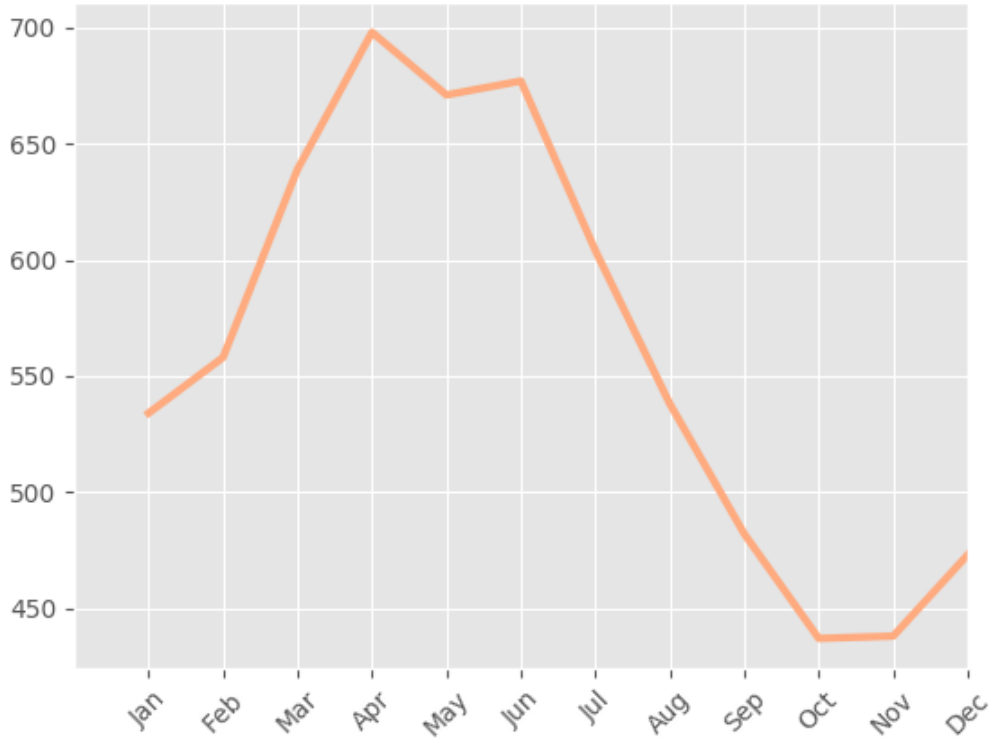
Plotting

```
In [27]: toolbox.regime(series)
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe90b63dd30>
```



Note: As stated in the *function reference*, the default plotting will choose the first color of the specified color map for the main aggregate line. As this defaults to the ‘‘Blue‘‘s, the first color is white. Therefore, when no percentiles are used (which would make use of the colormap), it is a good idea to overwrite the color for the main line.

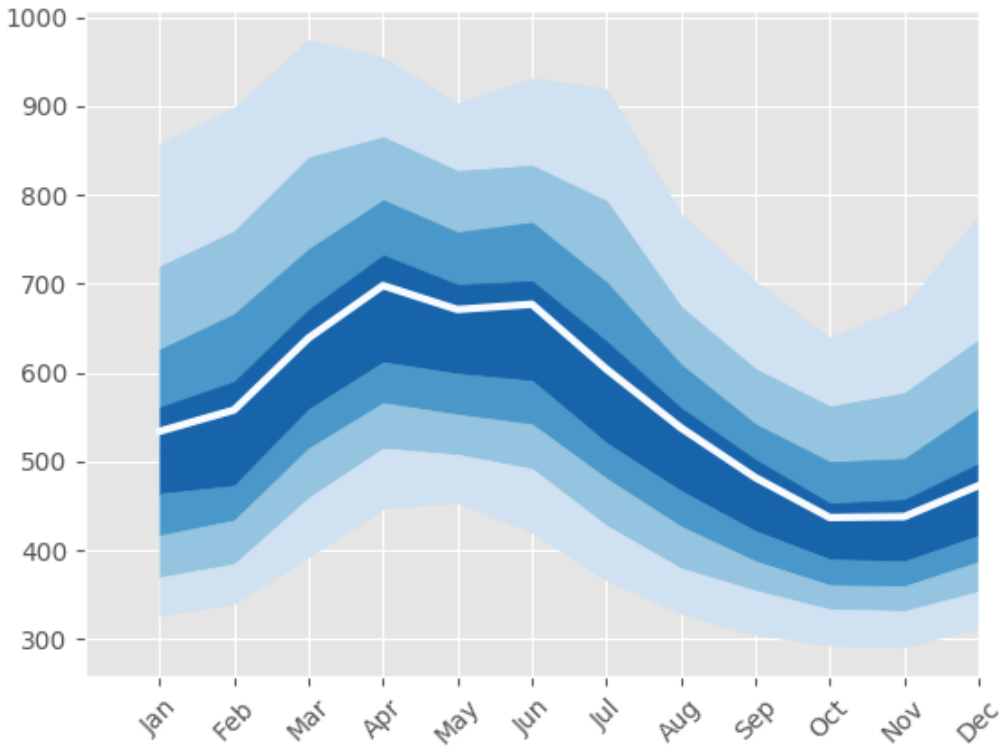
```
In [28]: toolbox.regime(series, color='#ffab7f')
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe90b5402e8>
```



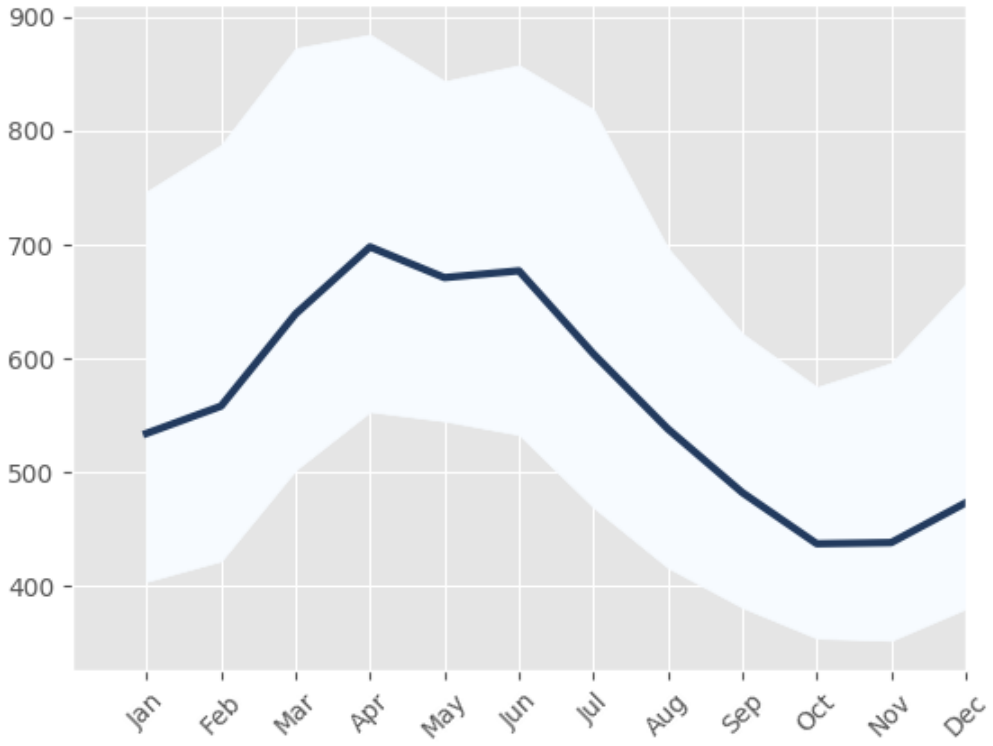
Using percentiles

The plot shown above is nice, but the tool is way more powerful. Using the `percentiles` keyword, we can either specify a number of percentiles or pass custom percentile edges.

```
In [29]: toolbox.regime(series, percentiles=10);
```



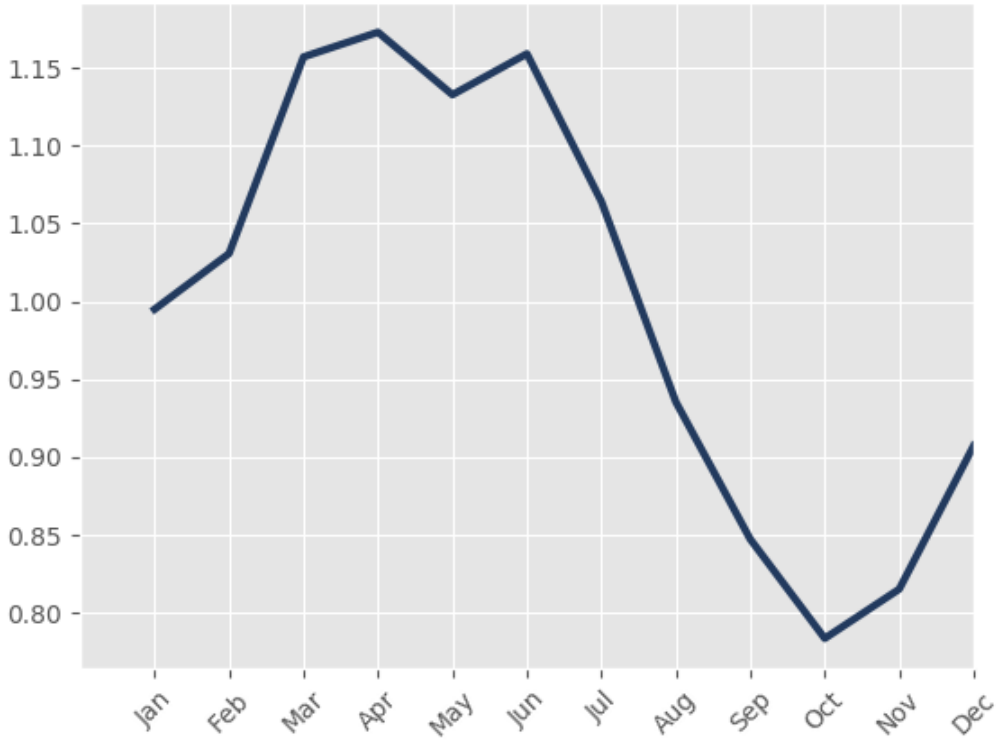

```
In [30]: toolbox.regime(series, percentiles=[25, 75, 100], color='#223a5e');
```



Adjusting the plot

Furthermore, the regime function can normalize the monthly aggregates to the overall aggregate. The function used for aggregation can also be changed. The following example will output monthly mean values over median values and normalize them to the MQ (overall mean).

```
In [31]: toolbox.regime(series, agg='nanmean', normalize=True, color='#223a5e')
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe90b5a7fd0>
```



Reference

See also:

- *regime reference*

3.3.2 Signal Processing

Simplifying a Signal

Whenever you seek to apply a tool on your data that will operate on each value and this tool is time and / or resource consuming, it might be a good idea to operate on as few values as possible. Simply removing duplicated values is not always the best approach. Think of a discharge time series where you want to calculate an index that depends on a previous state.

Set up a test case

The example below will show the idea behind the *simplify* method of the `signal` submodule. At first some imports.

```
In [1]: import numpy as np

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

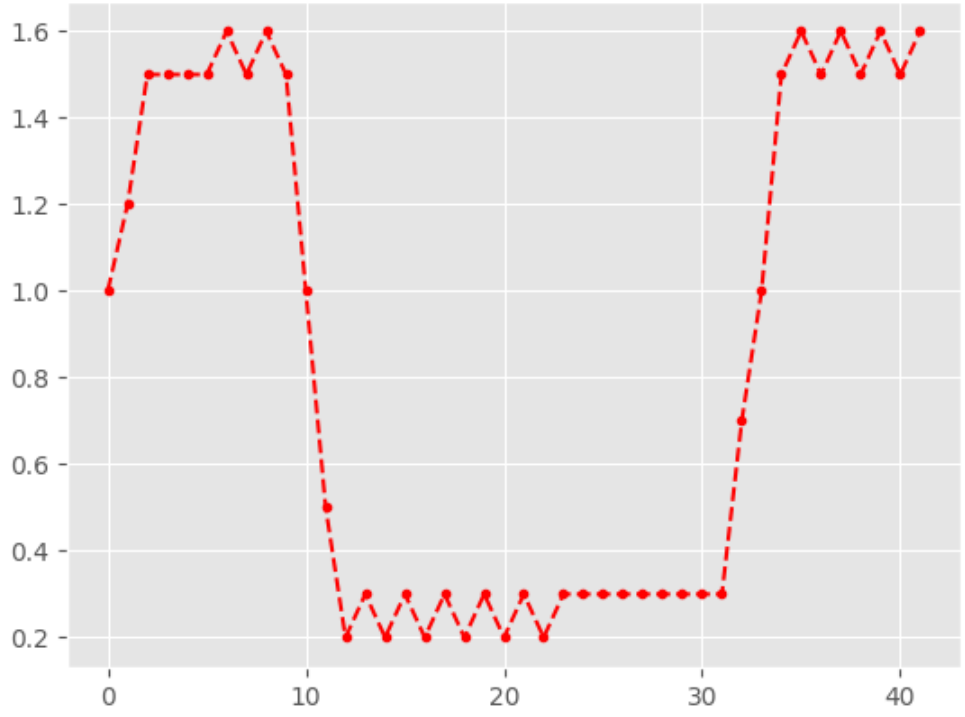
(continues on next page)

(continued from previous page)

```
In [3]: import matplotlib as mpl

In [4]: mpl.style.use('ggplot')
```

And now setup and plot the test signal.

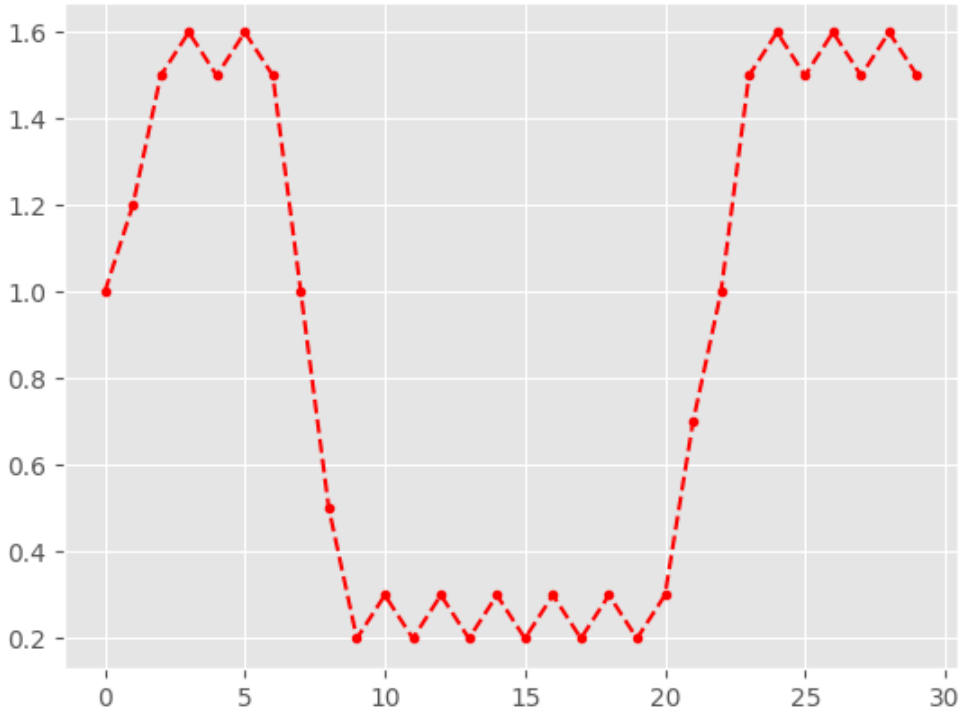
```
In [5]: x = np.array([1., 1.2, 1.5, 1.5, 1.5, 1.5, 1.6, 1.5, 1.6, 1.5, 1., 0.5,
   ...:               0.2, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2, 0.3,
   ...:               0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.7, 1., 1.5, 1.6,
   ...:               1.5, 1.6, 1.5, 1.6, 1.5, 1.6])
   ...:

In [6]: plt.plot(x, '.--r');
```



Handling replicas

There are a number of replications. We want to get rid of those.

```
In [7]: from hydrobox.toolbox import simplify

In [8]: plt.plot(simplify(x, flatten=False), '.--r');
```

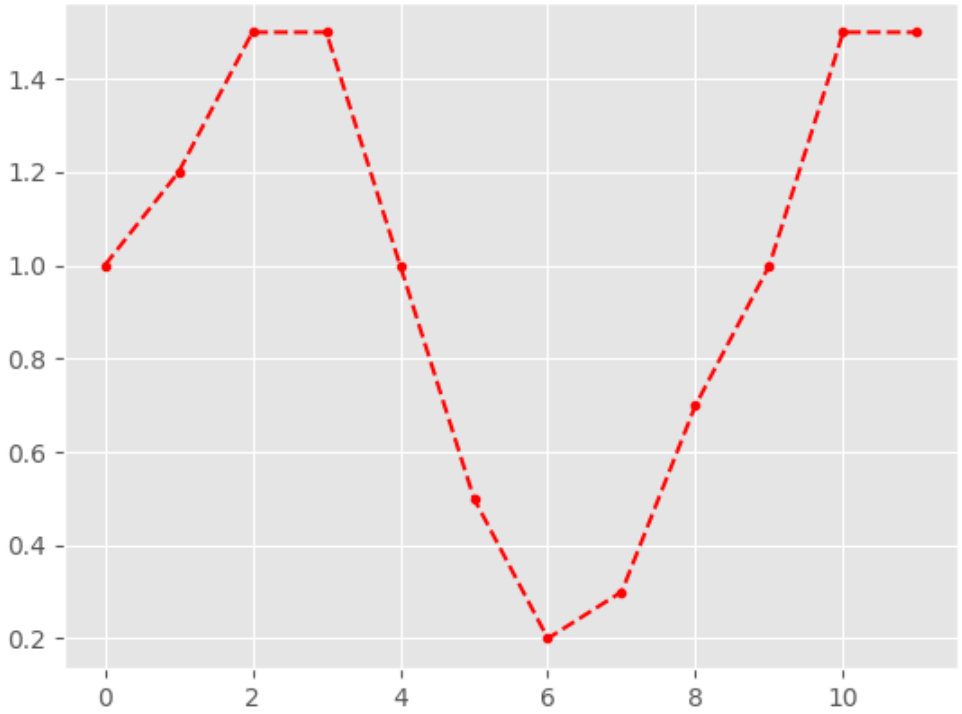


Look at the amount of markers in both plots, where the signal gave a constant value. The replicas got dropped from the signal.

Handling sensor precision noise

So far, this only removed subsequent value duplicates. The other signal information this method can simplify is the constant repetition of two values. This usually happens in environmental sensors either in constant conditions or during really slow state changes. In these cases the signal can alternate between two states within the sensor resolution. These recordings can be evened out by setting the `flatten` attribute to True.

```
In [9]: plt.plot(simplify(x, flatten=True), '.--r');
```



Of course, the index information was completely lost. In this example the x-axis is just counting the occurrences of values. In case you need the index information for further analysis, you have to extract the index and preserve it, before calling the simplify method.

Important: The preservation of indices, whenever the data is of type `pandas.Series` is planned for a future release.

Warning: Keep in mind that two very strong assumptions are underlying this method. It does change the signal dramatically. Ensuring that the sensor noise assumption is correct is completely up to you.

3.4 Contribution Guide

3.4.1 How to Contribute

There are several ways how you can contribute to hydrobox. All contributions should make use of the Fork / Pull request workflow in the GitHub repository. More information on pull requests can be found on the GitHub About pull requests page.

1. Add new tools to the toolbox
2. Improve / Add unit tests to increase code coverage
3. Improve / Add docstrings on existing functions

4. Add more examples to the documentation

Add Tools to the Toolbox

Important:

In a nutshell:

1. Fork the repository on GitHub
2. Commit your method to your fork
3. Add documentation and unittests for your method
4. Make sure your fork is building correctly
5. Pull request your fork back into the main repository

The idea behind hydrobox is to be used on top of numpy, scipy and pandas. This implies using the data types defined in these libraries whenever possible. The main purpose of hydrobox is to save hydrologists from reproducing their codes in every single project. Therefore a hydrobox tool should:

1. combine analysis steps belonging together into one function, while
2. separating preprocessing from analysis
3. be helpful to other hydrologists
4. output common python, numpy or pandas datatypes

Important: For this guide, we will add a function `from_csv` to the `io` submodule. This should illustrate how you can add your stuff.

Fork and structure

Once you forked the project, place a new file in the appropriate module or add a new one. Once your function has been added, import your function in the `hydrobox.toolbox` file. Please use an meaningful name for your function. It should be clear what the function does. In some cases tool functions are tool specific to make them available at the global `hydrobox.toolbox` scope. Then the submodule itself will be imported in the toolbox and you do not need to adjust the imports. One example is the `io` submodule.

Here, we pretend to add a `from_csv` file to the toolbox. This function will go into a file `text.py` in the `hydrobox.io` submodule:

```
def from_csv(path, sep=','):
    """
    numpydoc docstring here
    """
    return pd.from_csv(path, sep=sep)
```

Now, import this function in the `__init__` of `hydrobox.io`. If you want your method to be available in the global scope, import it in `hydrobox.toolbox` as well.

Important: Please do only use numpydoc docstring conventions and make sure to properly style and comment the Parameters section.

Decorating your tool

Hydrobox includes two helpful decorators in the `hydrobox.util` submodule: `accept` and `enforce`. We encourage you to use the `accept` decorator whenever possible. This will help to produce way cleaner code. This decorator will check the input data for their data type and raise a `TypeError` in case the passed data does not have the correct type. If more than one type is accepted, simply pass a tuple. In case a argument can be on `NoneType` or a `callable`, use the two literals `'None'` and `'callable'` and pass them as strings.

```
from io import TextIOWrapper

@accept(path=(str, TextIOWrapper), sep=str)
def from_csv(path, sep=','):
    ...
```

In this example, the `path` argument can be a string or a file pointer, `sep` has to be a string. Thus, there is no need to check the user input in your tool as the decorator already did this for you. We encourage you to use this decorator especially for checking the input data to be of type `numpy.ndarray` `pandas.DataFrame` and `pandas.Series`.

Test your tool

Although the code coverage of this project is not yet really good, it would be nice not to drop it any further. A good code coverage needs unit tests. Beyond code coverage, unit tests will help us to detect whenever our contribution breaks existing code. And last but not least a unit test will help you to build more reliable code. In a nutshell, it would be really helpful if you produce unit tests for your code. More information on unit tests is given in the *Add / Improve unit tests* section. Some useful links to get you stated with unit tests in Python can be found below.

See also:

- **'unit tests module reference'_**
- Unit Test Wikipedia page
- *Add / Improve unit tests*

Document your tool

In order to make it possible for others to use your tool, a good, comprehensive documentation is needed. As a first step, you should always add a docstring to your function. For hydrobox, please use the numpydoc docstring format. More information can also be found in the *Add / Improve docstrings* section.

See also:

- numpydoc reference site
- *Add / Improve docstrings*

Produce examples

Sometimes a docstring is not enough to understand a tool. Although short examples, references and formulas can go into numpydoc docstring formats, you might want to offer different examples covering the whole bandwidth of your tool. Then you should produce some examples for this documentation. You can refer to the *Examples* section for more information.

See also:

- *Examples*

Pull Request

Once your have finished with your implementations, create a pull request on GitHub. More info in the *Pull Request* section.

3.4.2 Add / improve unit tests

Important: If you are not familiar with unit testing in general, please refer to https://en.wikipedia.org/wiki/Unit_testing. If you are not familiar with the `unittests` module. please refer to https://docs.python.org/3/library/unittest.html

Unit tests are important as they make your code much more reliable and reusable for other users. The basic idea behind a unit test is to test any possible input and output to your tool against the expected behavior. For this you have to set up a test case, run the scenario and compare it to what you expected. When some modules and packages which you rely on change and break your code, the unit test will notice and fail. I am personally using unit tests whenever I try to improve my code, this way I can be sure that I did not optimize any functionality away (and that happens a lot...).

For creating a unit test you need to define a class. Each method of this class represents a test. There are different ways of implementing unit tests, either one test method to test a whole tool or one test method per single check you want to perform. In hydrobox, we decided to use one TestCase class for each method and try to break down each check into a single test method. The example below illustrates this.

```
import unittest
import pandas as pd
import from_csv          # import your tool here

class TestFromCsv(unittest.TestCase):
    def test_row_count(self):
        df = from_csv('file_of_known_size.txt')
        self.assertEqual(len(df), 450)

    def test_col_count(self):
        df = from_csv('file_of_known_size.txt')
        self.assertEqual(len(df.columns), 5)

    def test_change_sep(self):
        """
        change the separator to a sign that does not appear
        in the file. then there sould be only one column.
        """
        df = from_csv('file_of_known_size.txt', sep="|")
        self.assertEqual(len(df.columns)), 1)
```

(continues on next page)

(continued from previous page)

```

if __name__=='__main__':
    unittest.main()
```

This is a very basic example that checks three different things. It uses our new tool to load a file of known content into the variable `df`.

3.4.3 Add / improve docstrings

Important: If you are not familiar with the numpydoc docstring format, please refer to http://numpydoc.readthedocs.io/en/latest/format.html.

The most important parts of a numpydoc docstring are shown in the example below. Please make sure, that your docstring always contains the *main description*, *parameters* and *returns*.

```
@accept(path=(str, TextIOWrapper), sep=str)
def from_csv(path, sep=','):
r"""short descriptive tile

After a short title, give a few sentences of explanation. What does this
Method do and how is it intended to be used?

Parameters
----------
path : str, TextIOWrapper
    The parameters can also get a full description about their meaning
    possible values. Please be as extensive as necessary here.
    Note the whitespace between the parameter name and the (list) of
    accepted types.
sep : str, optional
    In case an argument is optional, you can indicate this by the
    optional keyword after the type.

Returns
-------
pandas.DataFrame

Notes
-----

The first description at the top should be a rather technical description.
The optional Notes section can be added and used to inform the user about
the background of the function or further readings.
For this purpose you can also include references[1]_ into your Notes.
In the documentations, these will be rendered in the Reference section.

And last but not least you can also input some math:

.. math:: a^2 + b^2 = c^2

References
----------

```

(continues on next page)

(continued from previous page)

```
..  [1] Python, M., Chapman, G., Cleese, J., Gilliam, T., Jones, T.,
    Idle, E., & Palin, M. (2000). the Holy Grail. EMI Records.

Examples
--------

>>> from_csv('file.txt').size
(220201, 5)

"""
...
```

Note: You should only add short and descriptive examples into the docstring itself. Make use of the *Examples section* of this documentation.

3.4.4 Enhance the Examples

Todo: write this section

3.4.5 Create a Pull Request

Important: If you are not familiar with Pull Requests, please refer to https://help.github.com/articles/about-pull-requests.

The best scenario for a pull request would be one that includes the new tool / enhancement, a proper docstring, unit tests and a new example. However, we will also accept a pull request including only a tool and a docstring. In these cases, please provide a proper description in the pull request message in order to make it possible for others to add missing content.

Beside a good description, a descriptive title is vital. Please state what you actually want to contribute in the pull request title. For the examples produced in this guide a descriptive title would be something like: *Added from_csv tool for reading files*.

Note: If your contribution does only contain minor changes like PEP8 fixes, typos and small bugfixes, you can of course pull request these changes without examples and unittests.

And finally, I am really looking forward to your contributions and thanks in advance!

3.5 Reference

Note: The reference section is still under development, as is the toolbox itself. It may not reflect the toolbox structure correctly at any time.

3.5.1 Input / Output

Random

timeseries_from_distribution

hydrobox.io.random.**timeseries_from_distribution**(*distribution='gamma', distribution_args=[10, 2], size=10, seed=None, start='now', end=None, freq='D'*)

Generate a random time series

This function will return a `pandas.Series` indexed by a `pandas.DatetimeIndex` holding random data that is generated by the given distribution. The distribution name has to be importable from `numpy.random` and the *distribution_args* list will be passed as **args*. The *seed* parameter will be directed to `np.random.seed` in order to return reproducable pseudo-random results.

Parameters

distribution [string, default='gamma'] Any distribution density function from `numpy.random` can be chosen. The distribution properties (like location or scale) can be passed with the parameter *distribution_args*.

distribution_args [list, None, default=[10,2]] This list will be passed as `*distribution_args` into the given density function. If no areguments shall be passed, distribution_args can be set to *None*.

size [int, default=10] Specifies the length of the produced time series.

seed [int, default=None] Will be passed to `numpy.random.seed`.

start [string, datetime, default='now'] Starting point for the `pandas.DatetimeIndex`. Can be either a `datetime` or string. The string has either to be *'now'* for using the current time step, or a Datetime string of format *YYYYMMDDHHmmss*, where the time (*HHmmss*) can be omitted. If *end* is used, start or *size* should be set to *None*.

end [string, datetime, defualt=None] see start.

freq [string, default='D'] Specify the temporal resulution of the time series. This can either be used in case *size* is omitted, but *start* and *end* are given, or in case either *start* or *end* is omitted but *size* is given. Any string accepted by the *freq* attribute of `pandas.Grouper` is accepted.

Returns

pandas.Series

See also:

`pandas.Grouper` further information of *freq* settings

3.5.2 Preprocessing

Scaling

aggregation

hydrobox.preprocessing.scale.**aggregate**(*x*, *by*, *func='mean'*)

Time series aggregation

This function version will only operate on a single `pandas.Series` or `pandas.DataFrame` instance. It has to be indexed by a *pandas.DatetimeIndex*. The input data will be aggregated to the given frequency by passing a *pandas.Grouper* conform string argument specifying the desired period like: '1M' for one month or '3Y-Sep' for three years starting at the first of October.

Parameters

x: ''pandas.Series'', ''pandas.DataFrame'' The input data, will be aggregated over the index.

by [string] Specifies the desired temporal resolution. Will be passed as `freq` argument of a `pandas.Grouper` object for grouping the data into the new resolution. If by is `None`, the whole Series will be aggregated to only one value. The same applies to `by='all'`.

func [string] Function identifier used for aggregation. Has to be importable from `numpy`. The function must accept n input values and aggregate them to only a single one.

Returns

pandas.Series : if x was of type `pandas.Series`

pandas.DataFrame : if c was of type `pandas.DataFrame`

cut_period

hydrobox.preprocessing.scale.**cut_period**(*x*, *start*, *stop*)

Truncate Time series

Truncates a `pandas.Series` or `pandas.DataFrame` to the given period. The start and stop parameter need to be either a string or a `datetime.datetime`, which will then be converted. Returns the truncated time series.

Parameters

x [`pandas.Series`, `pandas.DataFrame`] The input data, will be truncated

start [string, datetime] Begin of truncation. Can be a `datetime.datetime` or a string. If a string is passed, it has to use the format 'YYYYMMDDhhmmss', where the time componen 'hhmmss' can be omitted.

stop [string, datetime,] End of truncation. Can be a `datetime.datetime` or a string. If a string is passed, it has to use the format 'YYYYMMDDhhmmss', where the time componen 'hhmmss' can be omitted.

3.5.3 Basic statistical tools

Moving window statistics

moving_window

hydrobox.toolbox.**moving_window**(*x*, *window_size=5*, *window_type=None*, *func='nanmean'*)

Moving window statistics

Applies a moving window function to the input data. Each of the grouped windows will be aggregated into a resulting time series.

Parameters

x [pandas.Series, pandas.DataFrame] Input data. The data should have a pandas.DatetimeIndex in order to produce meaningful results. However, this is not needed and will technically work on different indexed data.

window_size [int] The specified number of values will be grouped into a window. This parameter might have different behavior in case the window_type is not *None*.

window_type [str, default=None] If *None*, an even spaced window will be used and shifted by one for each group. Else, a window constructing class can be specified. Possible constructors are specified in pandas.DataFrame.rolling.

func [str] Aggregating function for calculating the new window value. It has to be importable from numpy, accept various input values and return only a single value like numpy.std or numpy.median.

Returns

pandas.Series

pandas.DataFrame

Notes

Be aware that most window types (if window_type is not None) do only work with either numpy.sum or numpy.mean.

Furthermore, most windows cannot work with the 'nan' versions of numpy aggregating function. Therefore in case window_type is None, any 'nan' will be removed from the func string. In case you want to force this behaviour, wrap the numpy function into a lambda.

Examples

This way, you can prevent the replacement of a np.nan* function:

```
>>> moving_window(x, func=lambda x: np.nanmean(x))
array([NaN, NaN, NaN, 4.7445, 4.784 ... 6.34532])
```

Linear Regression

linear_regression

hydrobox.toolbox.**linear_regression**(*x, df=None, plot=False, ax=None, notext=False)

Linear Regression tool

This tool can be used for a number of regression related tasks. It can calculate a linear regression between two observables and also return a scatter plot including the regression parameters and function.

In case more than two `Series` or `arrays` are passed, they will be merged into a `DataFrame` and a linear regression between all combinations will be calculated and potted if desired.

Parameters

x [pandas.Series, numpy.ndarray] If df is None, at least two Series or arrays have to be passed. If more are passed, a multi output will be produced.

df [pandas.DataFrame] If df is set, all x occurrences will be ignored. DataFrame of the input to be used for calculating the linear regression, This attribute can be useful, whenever a multi input to x does not get merged correctly. Note that linear_regression will only use the DataFrame.data array and ignore all other structural elements.

plot [bool] If True, the function will output a matplotlib Figure or plot into an existing instance. If False (default) the data used for the plots will be returned.

ax [matplotlib.Axes.Axessubplot] Has to be a single matplotlib Axes instance if two data sets are passed or a list of Axes if more than two data sets are passed.

notext [bool] If True, the output of the fitting parameters as a text into the plot will be suppressed. This setting is ignored, is plot is set to False.

Returns

matplotlib.Figure

numpy.ndarray

Notes

If plot is True and ax is not None, the number of passed Axes has to match the total combinations between the data sets. This is

$$N^2$$

where N is the length of x, or the length of df.columns.

> **Warning:** This function does just calculate a linear regression. It handles a multi input recursively and has some data wrangling overhead. If you are seeking a fast linear regression tool, use the scipy.stats.linregress function directly.

3.5.4 Discharge Tools

Catchment hydrology

Common tools for diagnosic tools frequently used in catchment hydrology.

flow_duration_curve

hydrobox.discharge.**flow_duration_curve**(*x*, *log=True*, *plot=True*, *non_exceeding=True*, *ax=None*, ***kwargs*)

Calculate a flow duration curve

Calculate flow duration curve from the discharge measurements. The function can either return a `matplotlib` plot or return the ordered (non)-exceeding probabilities of the observations. These values can then be used in any external plotting environment.

In case x.ndim > 1, the function will be called iteratively along axis 0.

Parameters

x [numpy.ndarray, pandas.Series] Series of prefereably discharge measurements

log [bool, default=True] if *True* plot on loglog axis, ignored when plot is *False*

plot [bool, default=True] if *False* plotting will be suppressed and the resulting array will be returned

non_exceeding [bool, default=True] if *True* use non-exceeding probabilities

ax [matplotlib.AxesSubplot, default=None] if not None, will plot into that AxesSubplot instance

kwargs [kwargs,] will be passed to the `matplotlib.pyplot.plot` function

Returns

matplotlib.AxesSubplot : if *plot* was *True*

numpy.ndarray : if *plot was 'False*

Notes

The probabilities are calculated using the Weibull empirical probability. Following[1], this probability can be calculated as:

$$p = m/(n+1)$$

where *m* is the rank of an observation in the ordered time series and *n* are the total observations. The increasion by one will prevent 0% and 100% probabilities.

References

regime

hydrobox.discharge.**regime**(*x*, *percentiles=None*, *normalize=False*, *agg='nanmedian'*, *plot=True*, *ax=None*, ***kwargs*)

Calculate hydrological regime

Calculate a hydrological regime from discharge measurements. A regime is a annual overview, where all observations are aggregated across the month. Therefore it does only make sense to calculate a regime over more than one year with a temporal resolution higher than monthly.

[1] Sloto, R. a., & Crouse, M. Y. (1996). Hysep: a computer program for streamflow hydrograph separation and analysis. U.S. Geological Survey Water-Resources Investigations Report, 96(4040), 54.

The regime can either be plotted or the calculated monthly aggreates can be returned (along with the quantiles, if any were calculated).

Parameters

x [pandas.Series] The `Series` has to be indexed by a `pandas.DatetimeIndex` and hold the preferably discharge measurements. However, the methods does also work for other observables, if *agg* is adjusted.

percentiles [int, list, numpy.ndarray, default=None] percentiles can be used to calculate percentiles along with the main aggregate. The percentiles can either be set by an integer or a list. If an integer is passed, that many percentiles will be evenly spreaded between the 0th and 100th percentiles. A list can set the desired percentiles directly.

normalize [bool, default=False] If *True*, the regime will be normalized by the aggregate over all months. Then the numbers do not give the discharge itself, but the ratio of the monthly discharge to the overall discharge.

agg [string, default='nanmedian'] Define the function used for aggregation. Usually this will be 'mean' or 'median'. If there might be *NaN* values in the observations, the 'nan' prefixed functions can be used. In general, any aggregating function, which can be imported from `numpy` can be used.

plot [bool, default=True] if *False* plotting will be suppressed and the resulting `pandas.DataFrame` will be returned. In case *quantiles* was None, only the regime values will be returned as *numpy.ndarray*

ax [matplotlib.AxesSubplot, default=None] if not None, will plot into that AxesSubplot instance

cmap [string, optional] Specify a colormap for generating the Percentile areas is a smooth color gradient. This has to be a valid colormap reference, see https://matplotlib.org/examples/color/colormaps_reference.html. Defaults to `'Blue'`.

color [string, optional] Define the color of the main aggregate. If `None`, the first color of the specified cmap will be used.

lw [int, optinal] linewidth parameter in pixel. Defaults to 3.

linestyle [string, optional] Any valid matplotlib linestyle definition is accepted.

`':'` - dotted

`'-.'` - dash-dotted

`'--'` - dashed

`'-'` - solid

Returns

matplotlib.AxesSubplot : if *plot* was *True*

pandas.DataFrame : if *plot* was *False* and *quantiles* are not None

numpy.ndarray : if *plot* was *False* and *quantiles* is None

Notes

In case the color argument is not passed it will default to the first color in the the specified colormap (cmap). You might want to overwrite this in case no percentiles are produced, as many colormaps range from light to dark colors and the first color might just default to while.

Discharge coefficients

Common indices frequently used to describe discharge measurements in a single coefficient.

Richards-Baker Flashiness Index

hydrobox.discharge.indices.**richards_baker**(*x*)

Richards-Baker Flashiness Index

Calculates the Richards-Baker Flashiness index (RB Index), which is a extension of the Richards Pathlengh index. In contrast to the Pathlength of a signal, the R-B Index is relative to the total discharge and independend of the chosen unit.

Parameters

x [numpy.ndarray, pd.Series] The discharge input values.

Returns

numpy.ndarray

Notes

The Richards-Baker Flashiness Index[2] is defined as:

$$RBI = \frac{\sum_{i=1}^{n} |q_i - q_{i-1}|}{\sum_{i=1}^{n} q_i}$$

References

3.5.5 Signal Processing

Optimize

simplify

hydrobox.toolbox.**simplify**(*x*, *flatten=True*, *threshold=0*)

Simplify signal

An given input is simplified by reducing the amount of nodes representing the signal. Whenever node[n+1] - node[n] <= threshold, no information gain is assumed between the two nodes. Thus, node[n+1] will be removed.

In case flatten is True, noise in the signal will be flattened as well. This is done by removing node[n + 1] in case node[n] and node[n + 1] hold the same value. In case the underlying frequency in the noise is higher than one time step or the amplitude is higher than the sensor precision, this method will not assume the value change as noise. In these cases a filter needs to be applied first.

Parameters

x [numpy.ndarray, pandas.Series, pandas.DataFrame] numpy.array of signal

flatten [bool] Specify if a 1 frequence 1 amplitude change in signal be flattened out as assumed noise.

[2] Baker D.B., P. Richards, T.T. Loftus, J.W. Kramer. A new flashiness index: characteristics and applications to midwestern rivers and streams. JAWRA Journal of the American Water Resources Association, 40(2), 503-522, 2004.

threshold [int, float] value threshold at which a difference in signal is assumed

Returns

numpy.ndarray

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index