pyLife Documentation

Release 2.0.0

pyLife Developer Team

Feb 22, 2022

ABOUT

CHAPTER

ONE

PYLIFE – A GENERAL LIBRARY FOR FATIGUE AND RELIABILITY

pyLife is an Open Source Python library for state of the art algorithms used in lifetime assessment of mechanical components subject to fatigue load.

1.1 Purpose of the project

This library was originally compiled at Bosch Research to collect algorithms needed by different in house software projects, that deal with lifetime prediction and material fatigue on a component level. In order to further extent and scrutinize it we decided to release it as Open Source. Read this article about pyLife's origin.

So we are welcoming collaboration not only from science and education but also from other commercial companies dealing with the topic. We commend this library to university teachers to use it for education purposes.

1.2 Status

pyLife-2.0.0 has been released. That means that for the time being we hope that we will not introduce *breaking* changes. That does not mean that the release is stable finished and perfect. We will do small improvements, especially with respect to documentation in the upcoming months and release them frequently as 2.0.x releases. Once we have noticeable feature additions we will come up with a 2.x.0 release. No ETA about that.

1.3 Contents

There are/will be the following subpackages:

- `stress` everything related to stress calculation
  - equivalent stress
  - stress gradient calculation
  - rainflow counting
  - …
- `strength` everything related to strength calculation
  - failure probability estimation
  - S-N-calculations

  - …
- `mesh` FEM mesh related stuff
  - stress gradients
  - FEM-mapping
  - hotspot detection
- `util` all the more general utilities
  - …
- `materialdata` analysis of material testing data
  - Wöhler (SN-curve) data analysis
- `materiallaws` modeling material behavior
  - Ramberg Osgood
  - Wöhler curves
- `vmap` a interface to VMAP

1.4 License

pyLife is open-sourced under the Apache-2.0 license. See the *LICENSE* file for details.

For a list of other open source components included in pyLife, see the file *3rd-party-licenses.txt*.

CHAPTER

TWO

INSTALLATION / GETTING STARTED

2.1 Just a glimpse

If you just want to check out pyLife's demos, you can use the our notebooks at mybinder. We will add new notebooks as soon as we have new functionality.

2.2 Installation to use pyLife

2.2.1 Prerequisites

You need a python installation e.g. a virtual environment with `pip` a recent (brand new ones might not work) python versions installed. There are several ways to achieve that.

Using anaconda

Install anaconda or miniconda [http://anaconda.com] on your computer and create a virtual environment with the package `pip` installed. See the conda documentation on how to do that. The newly created environment must be activated.

The following command lines should do it

```
conda create -n pylife-env python=3.9 pip --yes
conda activate pylife-env
```

Using virtualenv

Setup a python virtual environment containing pip according to these instructions and activate it.

Using the python installation of your Linux distribution

That's not recommended. If you really want to do that, you probably know how to do it.

2.2.2 pip install

The simplest way to install pyLife is just using the pip package

```
pip install pylife[all]
```

That installs pyLife with all the dependencies to use pyLife in python programs. You might want to install some further packages like `jupyter` in order to work with jupyter notebooks.

There is no conda package as of now, unfortunately.

2.3 Installation to develop pyLife

For general contribution guidelines please read *CONTRIBUTING.md*

2.3.1 Clone the git repository

Depending on your tools. From the command line

```
git clone https://github.com/boschresearch/pylife.git
```

will do it.

2.3.2 Install the dependencies

Install anaconda or miniconda [http://anaconda.com]. Create an anaconda environment with all the requirements by running

Create an environment – usually a good idea to use a prefixed environment in your pyLife working directory and activate it.

```
conda create -p .venv python=3.9 pip --yes
conda activate ./.venv
```

Then install the pyLife into that environment.

```
pip install -e .[testing,all]
```

2.3.3 Test the installation

You can run the test suite by the command

```
pytest
```

If it creates an output ending like below, the installation was successful.

```
================ 228 passed, 1 deselected, 13 warnings in 30.45s ===============
```

There might be some `DeprecationWarnings`. Ignore them for now.

CHAPTER

THREE

TUTORIALS

This section contains tutorials that teach the use of pyLife and its principles. As opposed to the *pyLife Cookbook* it does not show actual workflows.

3.1 The WoehlerCurve data structure

The `WoehlerCurve <https://pylife.readthedocs.io/en/latest/materiallaws/woehlercurve.html>`__ is the basis of pyLife's fatigue assessment functionality. It handles pandas objects containing data describing a Wöhler curve.

```
[1]: import pandas as pd
     import numpy as np
     from pylife.materiallaws import WoehlerCurve
     import matplotlib.pyplot as plt
```

3.1.1 The very basic Wöhler curve data

The basic Wöhler curve is a `pandas.Series` that contains at least three keys, * `SD`: the load level of the endurance limit * `ND`: the cycle number of the endurance limit * `k_1`: the slope of the Wöhler Curve

```
[2]: woehler_curve_data = pd.Series({
         'SD': 300.,
         'ND': 1.5e6,
         'k_1': 6.2,
     })
     woehler_curve_data
```

```
[2]: SD          300.0
     ND      1500000.0
     k_1           6.2
     dtype: float64
```

```
[3]: wc = WoehlerCurve(woehler_curve_data)
     #wc = woehler_curve_data.woehler (alternative way of writing it)
     wc.SD, wc.ND, wc.k_1
```

```
[3]: (300.0, 1500000.0, 6.2)
```

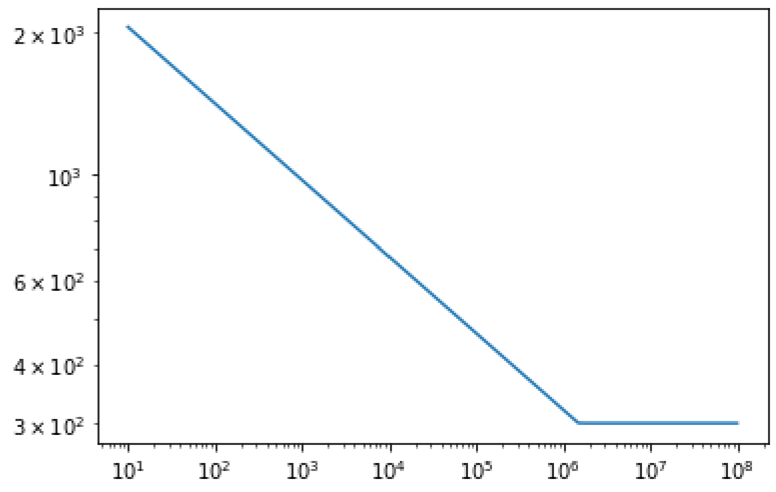
```
[4]: cycles = np.logspace(1., 8., 70)
     load = wc.basquin_load(cycles)
```

(continues on next page)

(continued from previous page)

```
plt.loglog()
plt.plot(cycles, load)
```

```
[4]: [<matplotlib.lines.Line2D at 0x7f750be36700>]
```



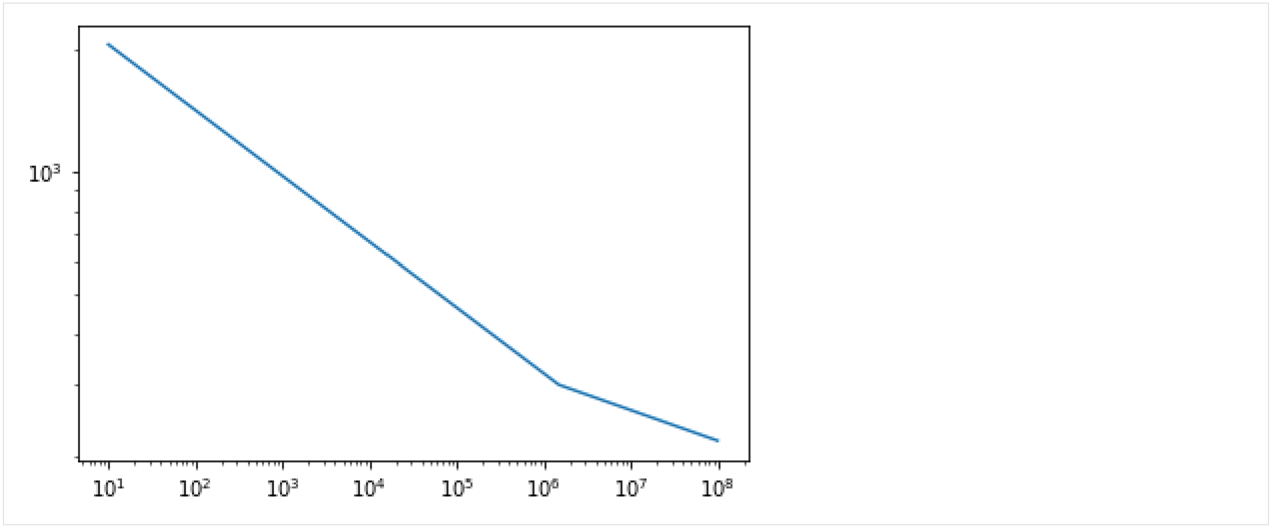
3.1.2 Optional parameters

The second slope `k_2`

You can optinally add a second slope `k_2` to the Wöhler curve data which is valid beyond `ND`.

```
[5]: woehler_curve_data = pd.Series({
         'SD': 300.,
         'ND': 1.5e6,
         'k_1': 6.2,
         'k_2': 13.3
     })
     plt.loglog()
     plt.plot(cycles, woehler_curve_data.woehler.basquin_load(cycles))
```

```
[5]: [<matplotlib.lines.Line2D at 0x7f7509a2eeb0>]
```



The failure probability and the scatter values `TN` and `TS`.

As everyone knows, material fatigue is a statistical phenomenon. That means that the cycles calculated for a certain load are the cycles at which the specimen fails with a certain probability. By default the failure probability is 50%.

```
[6]: woehler_curve_data.woehler.failure_probability
```

```
[6]: 0.5
```

You can provide values for the scattering of the Wöhler curve:

```
[7]: woehler_curve_data = pd.Series({
         'SD': 300.,
         'ND': 1.5e6,
         'k_1': 6.2,
         'TS': 1.25,
         'TN': 4.0
     })
```

Now you can then transform this Wöhlercurve to another failure probability:

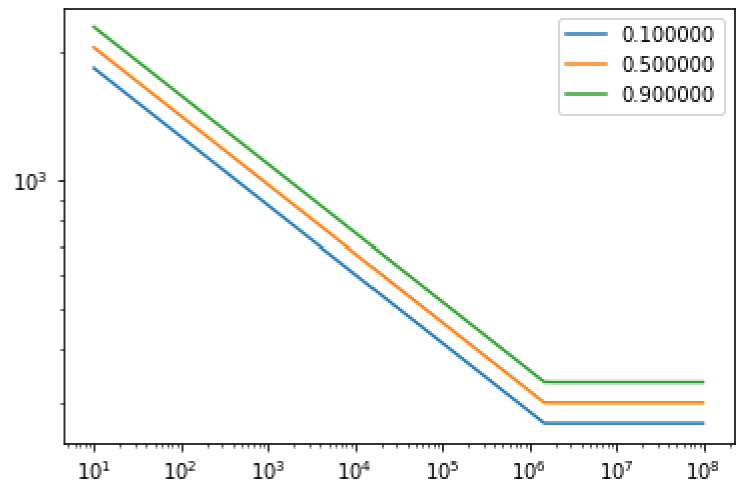
```
[8]: woehler_curve_data.woehler.transform_to_failure_probability(0.9).to_pandas()
```

```
[8]: SD                     3.354102e+02
     ND                     1.502105e+06
     k_1                    6.200000e+00
     TS                     1.250000e+00
     TN                     4.000000e+00
     k_2                             inf
     failure_probability    9.000000e-01
     dtype: float64
```

As convenience you can provide the failure probability as a optional parameter to the `basquin_load()` and `basquin_cycles()` methods.

```
[9]: wc = WoehlerCurve(woehler_curve_data)
     plt.loglog()
     for fp in [0.1, 0.5, 0.9]:
         plt.plot(cycles, wc.basquin_load(cycles, failure_probability=fp), label="%f" % fp)

     plt.legend()
```

```
[9]: <matplotlib.legend.Legend at 0x7f7509ad29d0>
```



3.2 Load Collectives and Load Histograms

From the load (stress) side pyLife provides the classes `LoadCollective` and `LoadHistogram` to deal with load collectives. `LoadCollective` contains individal hysteresis loops whereas `LoadHistogram` contains a 2D-histogram of classes of hysteresis loops and the number of cycles with which they occur.

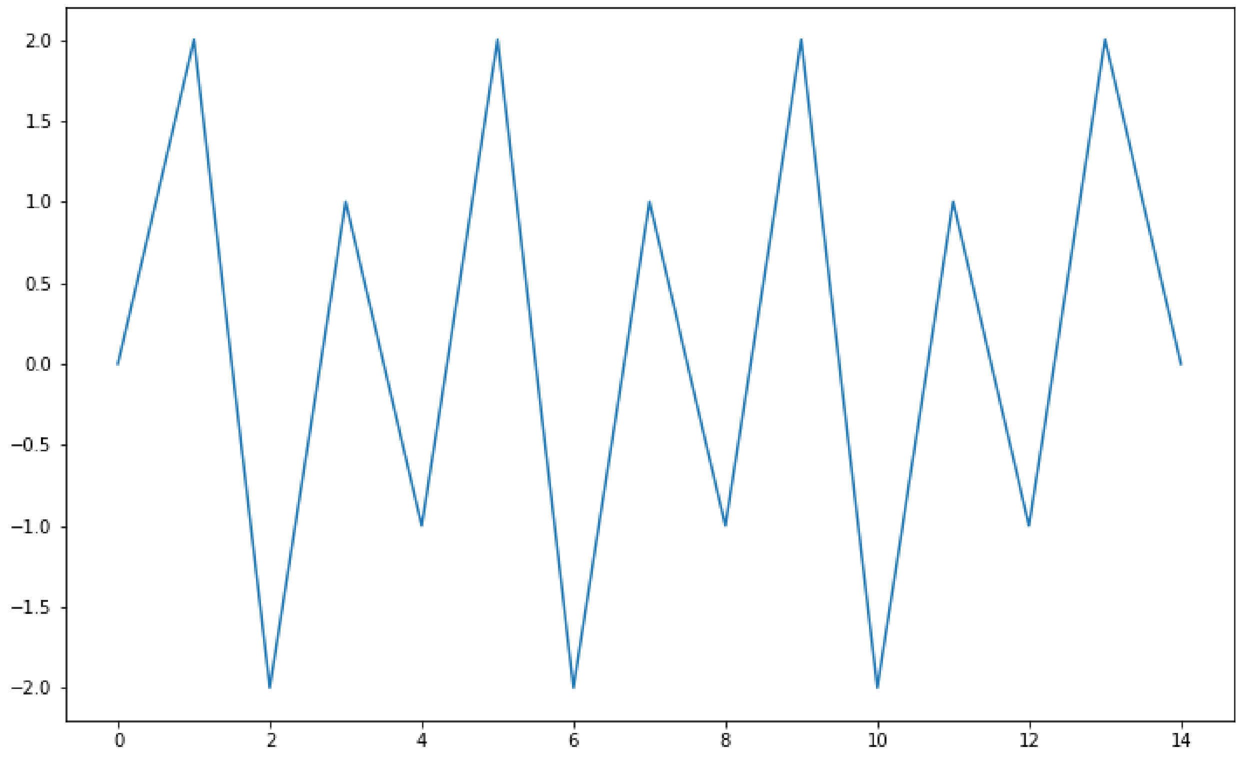
```
[1]: import numpy as np
     import pandas as pd

     import matplotlib.pyplot as plt

     import pylife.stress.timesignal as TS
     import pylife.stress.rainflow as RF
     import pylife.strength.meanstress as MS
     import pylife.strength.fatigue
     plt.rcParams['figure.figsize'] = [12, 7.5]
```

3.2.1 A simple load signal

Let's take a look at a really simple load signal:

```
[2]: load_signal = np.array([0., 2.0, -2.0, 1.0, -1.0, 2.0, -2.0, 1.0, -1.0, 2.0, -2.0, 1.0, -
     ↪1.0, 2.0, 0.])
     plt.plot(load_signal)
```

```
[2]: [<matplotlib.lines.Line2D at 0x7f1f1b662eb0>]
```



Now let's perform a rainflow analysis.

```
[3]: detector = RF.FourPointDetector(recorder=RF.LoopValueRecorder())
     detector.process(load_signal)
```

```
[3]: <pylife.stress.rainflow.fourpoint.FourPointDetector at 0x7f1f195ef790>
```

The detector now contains the recorder which recorded the hysteresis loops for us. The simple load collective comes as a attribute of the detector:

```
[4]: collective = detector.recorder.collective
     collective
```

```
[4]:    from   to
     0   1.0 -1.0
     1  -2.0  2.0
     2   1.0 -1.0
     3  -2.0  2.0
     4   1.0 -1.0
```

As you can see, the rainflow analysis found five histresis loops, three from 1.0 to -1.0 and two from -2.0 to 2.0. Alternatively you can ask the recorder for a load histogram:

```
[5]: histogram = detector.recorder.histogram(bins=6)
     histogram
```

```
[5]: from          to
     (-2.0, -1.5]  (-1.0, -0.5]    0.0
                   (-0.5, 0.0]     0.0
                   (0.0, 0.5]      0.0
                   (0.5, 1.0]      0.0
                   (1.0, 1.5]      0.0
                   (1.5, 2.0]      2.0
     (-1.5, -1.0]  (-1.0, -0.5]    0.0
                   (-0.5, 0.0]     0.0
                   (0.0, 0.5]      0.0
                   (0.5, 1.0]      0.0
                   (1.0, 1.5]      0.0
                   (1.5, 2.0]      0.0
     (-1.0, -0.5]  (-1.0, -0.5]    0.0
                   (-0.5, 0.0]     0.0
                   (0.0, 0.5]      0.0
                   (0.5, 1.0]      0.0
                   (1.0, 1.5]      0.0
                   (1.5, 2.0]      0.0
     (-0.5, 0.0]   (-1.0, -0.5]    0.0
                   (-0.5, 0.0]     0.0
                   (0.0, 0.5]      0.0
                   (0.5, 1.0]      0.0
                   (1.0, 1.5]      0.0
                   (1.5, 2.0]      0.0
     (0.0, 0.5]    (-1.0, -0.5]    0.0
                   (-0.5, 0.0]     0.0
                   (0.0, 0.5]      0.0
                   (0.5, 1.0]      0.0
                   (1.0, 1.5]      0.0
                   (1.5, 2.0]      0.0
     (0.5, 1.0]    (-1.0, -0.5]    3.0
                   (-0.5, 0.0]     0.0
                   (0.0, 0.5]      0.0
                   (0.5, 1.0]      0.0
                   (1.0, 1.5]      0.0
                   (1.5, 2.0]      0.0
     dtype: float64
```

This is a bit hard to read. What you see is a `pands.Series` that has a two dimensional `IntervalIndex` as index. The histogram is all empty except the two classes `from`: (-2.0, 1.5] `to`: (1.5, 2.0] has 2.0 cycles and `from`: (0.5, 1.0] `to`: (-1.0, -1.5] has 3.0 cycles. Tose correspond to the two loops from -2.0 to 2.0 and the three loops 1.0 to -1.0.

3.2.2 Working with load collectives and load histograms

A load collective and a load histogram can be processed by the two classes `LoadCollective` and `LoadHistogram`. Both inherit from the common base class `AbstractLoadCollective`. There is the common accessor attribute `load_collective` that convert a pandas object with the load collective resp. load histogram data into the corresponding class.

First let's look at a load collective. You can easily calculate the amplitude of each hysteresis loop:

[6]:
```
cl = collective.load_collective
cl.amplitude
```

[6]:
```
0    1.0
1    2.0
2    1.0
3    2.0
4    1.0
Name: amplitude, dtype: float64
```

Same for the mean stress and the R-value:

[7]:
```
cl.meanstress, cl.R
```

[7]:
```
(0    0.0
 1    0.0
 2    0.0
 3    0.0
 4    0.0
 Name: meanstress, dtype: float64,
 0   -1.0
 1   -1.0
 2   -1.0
 3   -1.0
 4   -1.0
 Name: R, dtype: float64)
```

There is also the attribute `cycles`:

[8]:
```
cl.cycles
```

[8]:
```
0    1.0
1    1.0
2    1.0
3    1.0
4    1.0
Name: cycles, dtype: float64
```

As you can see, the cycles are all 1.0 because we have an entry for each indivudual hysteresis loop which by definition occurs only once.

Now let's take a look at the histogram:

[9]:
```
hi = histogram.load_collective
hi.amplitude, hi.meanstress, hi.R
```

[9]:
```
(from          to
 (-2.0, -1.5]  (-1.0, -0.5]    0.50
```

(continues on next page)

(continued from previous page)

```
              (-0.5, 0.0]     0.75
              (0.0, 0.5]      1.00
              (0.5, 1.0]      1.25
              (1.0, 1.5]      1.50
              (1.5, 2.0]      1.75
(-1.5, -1.0]  (-1.0, -0.5]    0.25
              (-0.5, 0.0]     0.50
              (0.0, 0.5]      0.75
              (0.5, 1.0]      1.00
              (1.0, 1.5]      1.25
              (1.5, 2.0]      1.50
(-1.0, -0.5]  (-1.0, -0.5]    0.00
              (-0.5, 0.0]     0.25
              (0.0, 0.5]      0.50
              (0.5, 1.0]      0.75
              (1.0, 1.5]      1.00
              (1.5, 2.0]      1.25
(-0.5, 0.0]   (-1.0, -0.5]    0.25
              (-0.5, 0.0]     0.00
              (0.0, 0.5]      0.25
              (0.5, 1.0]      0.50
              (1.0, 1.5]      0.75
              (1.5, 2.0]      1.00
(0.0, 0.5]    (-1.0, -0.5]    0.50
              (-0.5, 0.0]     0.25
              (0.0, 0.5]      0.00
              (0.5, 1.0]      0.25
              (1.0, 1.5]      0.50
              (1.5, 2.0]      0.75
(0.5, 1.0]    (-1.0, -0.5]    0.75
              (-0.5, 0.0]     0.50
              (0.0, 0.5]      0.25
              (0.5, 1.0]      0.00
              (1.0, 1.5]      0.25
              (1.5, 2.0]      0.50
Name: amplitude, dtype: float64,
from          to
(-2.0, -1.5]  (-1.0, -0.5]   -1.25
              (-0.5, 0.0]    -1.00
              (0.0, 0.5]     -0.75
              (0.5, 1.0]     -0.50
              (1.0, 1.5]     -0.25
              (1.5, 2.0]      0.00
(-1.5, -1.0]  (-1.0, -0.5]   -1.00
              (-0.5, 0.0]    -0.75
              (0.0, 0.5]     -0.50
              (0.5, 1.0]     -0.25
              (1.0, 1.5]      0.00
              (1.5, 2.0]      0.25
(-1.0, -0.5]  (-1.0, -0.5]   -0.75
              (-0.5, 0.0]    -0.50
              (0.0, 0.5]     -0.25
```

(continues on next page)

(continued from previous page)

```
              (0.5, 1.0]      0.00
              (1.0, 1.5]      0.25
              (1.5, 2.0]      0.50
(-0.5, 0.0]   (-1.0, -0.5]   -0.50
              (-0.5, 0.0]    -0.25
              (0.0, 0.5]      0.00
              (0.5, 1.0]      0.25
              (1.0, 1.5]      0.50
              (1.5, 2.0]      0.75
(0.0, 0.5]    (-1.0, -0.5]   -0.25
              (-0.5, 0.0]     0.00
              (0.0, 0.5]      0.25
              (0.5, 1.0]      0.50
              (1.0, 1.5]      0.75
              (1.5, 2.0]      1.00
(0.5, 1.0]    (-1.0, -0.5]    0.00
              (-0.5, 0.0]     0.25
              (0.0, 0.5]      0.50
              (0.5, 1.0]      0.75
              (1.0, 1.5]      1.00
              (1.5, 2.0]      1.25
Name: meanstress, dtype: float64,
from          to
(-2.0, -1.5]  (-1.0, -0.5]    2.333333
              (-0.5, 0.0]     7.000000
              (0.0, 0.5]     -7.000000
              (0.5, 1.0]     -2.333333
              (1.0, 1.5]     -1.400000
              (1.5, 2.0]     -1.000000
(-1.5, -1.0]  (-1.0, -0.5]    1.666667
              (-0.5, 0.0]     5.000000
              (0.0, 0.5]     -5.000000
              (0.5, 1.0]     -1.666667
              (1.0, 1.5]     -1.000000
              (1.5, 2.0]     -0.714286
(-1.0, -0.5]  (-1.0, -0.5]    1.000000
              (-0.5, 0.0]     3.000000
              (0.0, 0.5]     -3.000000
              (0.5, 1.0]     -1.000000
              (1.0, 1.5]     -0.600000
              (1.5, 2.0]     -0.428571
(-0.5, 0.0]   (-1.0, -0.5]    3.000000
              (-0.5, 0.0]     1.000000
              (0.0, 0.5]     -1.000000
              (0.5, 1.0]     -0.333333
              (1.0, 1.5]     -0.200000
              (1.5, 2.0]     -0.142857
(0.0, 0.5]    (-1.0, -0.5]   -3.000000
              (-0.5, 0.0]    -1.000000
              (0.0, 0.5]      1.000000
              (0.5, 1.0]      0.333333
              (1.0, 1.5]      0.200000
```

(continues on next page)

(continued from previous page)

```
              (1.5, 2.0]      0.142857
(0.5, 1.0]    (-1.0, -0.5]   -1.000000
              (-0.5, 0.0]    -0.333333
              (0.0, 0.5]      0.333333
              (0.5, 1.0]      1.000000
              (1.0, 1.5]      0.600000
              (1.5, 2.0]      0.428571
Name: R, dtype: float64)
```

This might look a bit confusing as this only shows the amplitudes, meanstresses and R-values correspond to the bins of the histogram. Remember, that they were all except two empty. So let's restrict the histogram to bins that are not empty:

```
[10]: not_empty = histogram > 0.0
hi.amplitude[not_empty], hi.cycles[not_empty]
```

```
[10]: (from          to
 (-2.0, -1.5]  (1.5, 2.0]      1.75
 (0.5, 1.0]    (-1.0, -0.5]    0.75
 Name: amplitude, dtype: float64,
 from          to
 (-2.0, -1.5]  (1.5, 2.0]      2.0
 (0.5, 1.0]    (-1.0, -0.5]    3.0
 Name: cycles, dtype: float64)
```

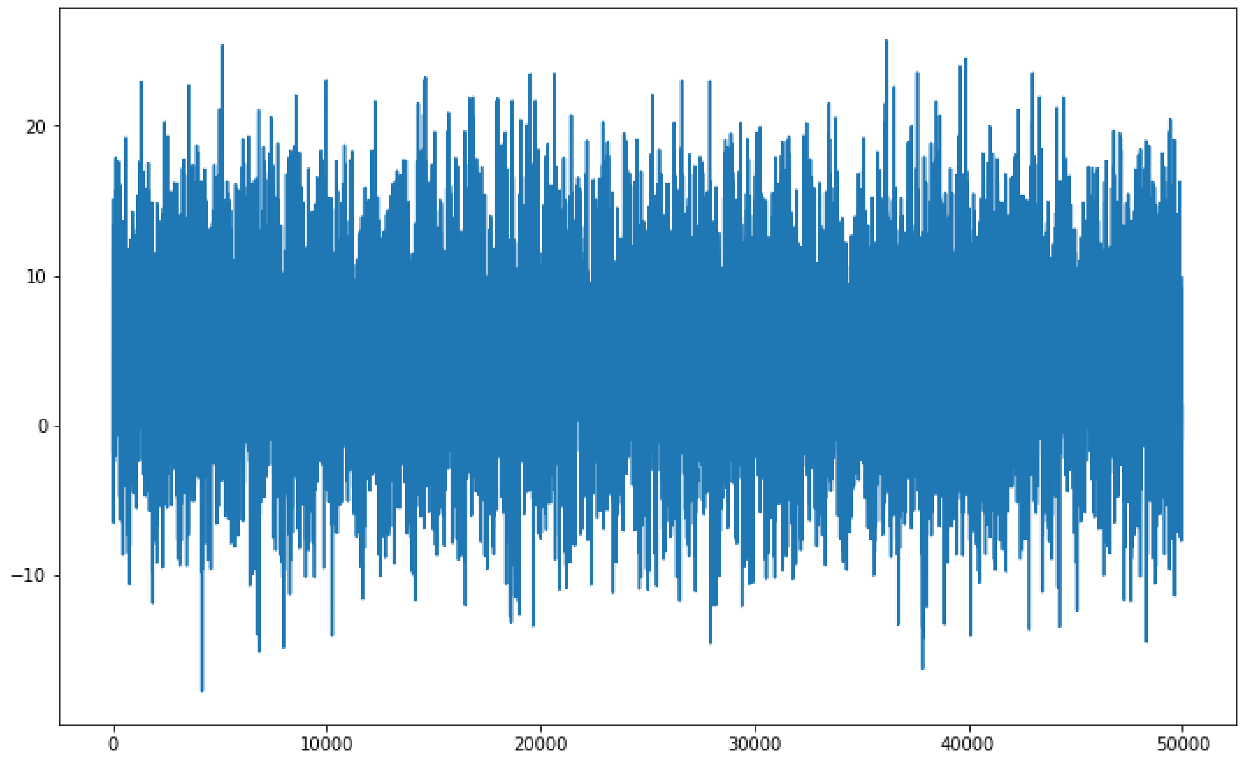
The amplitude values 1.75 and 0.75 correspond to 2.0 and 1.0. They are in the middle of the histogram bins.

3.2.3 A more complex example

Now let's take a look at a more complex load collective. We use the `TimeSignalGenerator` to generate a load signal.

```
[11]: load_signal = TS.TimeSignalGenerator(
    10,
    {
        'number': 50,
        'amplitude_median': 1.0, 'amplitude_std_dev': 0.5,
        'frequency_median': 4, 'frequency_std_dev': 3,
        'offset_median': 0, 'offset_std_dev': 0.4
    }, None, None
).query(50000)
plt.plot(load_signal)
```

```
[11]: [<matplotlib.lines.Line2D at 0x7f1f194eff70>]
```



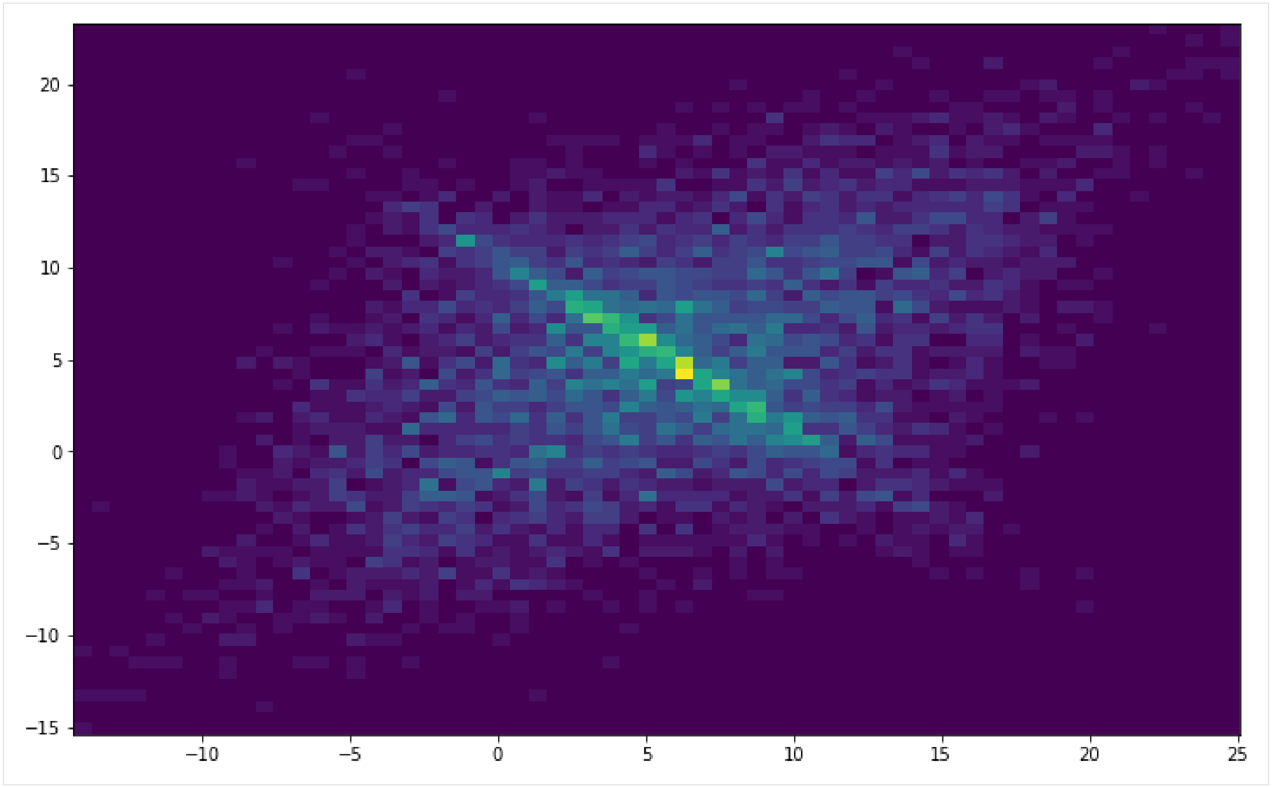
Again we perform a rainflow analysis to obtain the load histogram.

```
[12]: detector = RF.FourPointDetector(recorder=RF.LoopValueRecorder())
      detector.process(load_signal)

      histogram = detector.recorder.histogram(64)
```

We can plot the histogram with a bit of processing.

```
[13]: fr, to = histogram.index.levels[0], histogram.index.levels[1]
      numpy_hist = np.flipud(histogram.values.reshape(len(fr),len(to)))
      X, Y = np.meshgrid(fr.left, to.left)
      plt.pcolormesh(X, Y, numpy_hist)
```

```
[13]: <matplotlib.collections.QuadMesh at 0x7f1f1889d130>
```



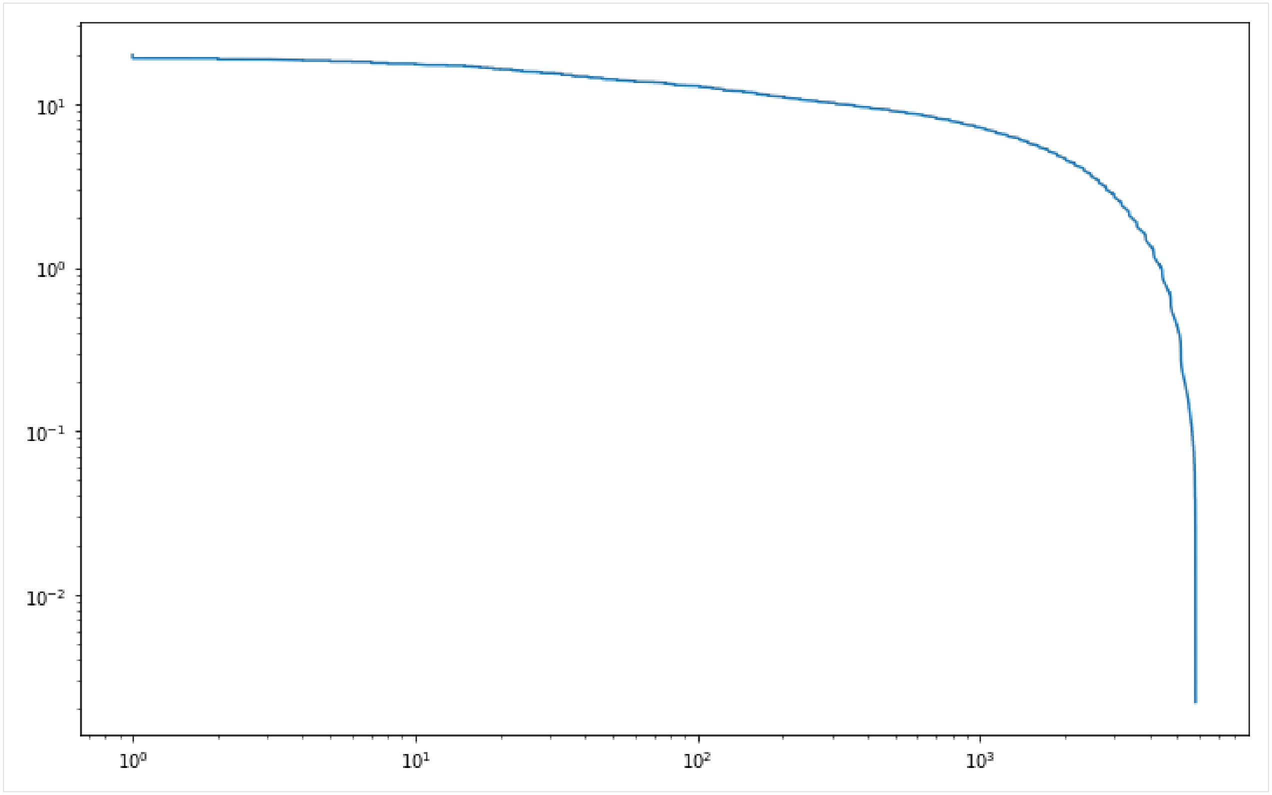
We can also plot the cumulated version of the histogram. Therefor we put the amplitude and the cycles into a dataframe.

```
[14]: df = pd.DataFrame({
          'cycles': histogram.load_collective.cycles,
          'amplitude': histogram.load_collective.amplitude,
      }).sort_values('amplitude', ascending=False)
```

Now we can plot the amplitude against the cumulated sum of the cycles:

```
[15]: plt.plot(np.cumsum(df.cycles), df.amplitude)
      plt.loglog()
```

```
[15]: []
```



3.3 The concept of stress and strength

The fundamental principle of component lifetime and reliability design is to calculate the superposition of *stress* and *strength*. Sometimes you would also say *load* and strength. The basic assumption is that as soon as the stress exceeds the strength the component fails. Usually stress and strength are statistically distributed. In this tutorial we learn how to work with material data and material laws to model the strength and then calculate the damage using a given load.

3.3.1 Material laws

The material load that is used to model the strength for component fatigue is the `pylife.materiallaws.WoehlerCurve` class.

First we need to import`pandas` and the `WoehlerCurve` class.

[1]:
```
import pandas as pd
import numpy as np
from pylife.materiallaws import WoehlerCurve
import matplotlib.pyplot as plt
```

The material data for a Wöhler curve is usually stored in a `pandas.Series`. In the simplest form like this:

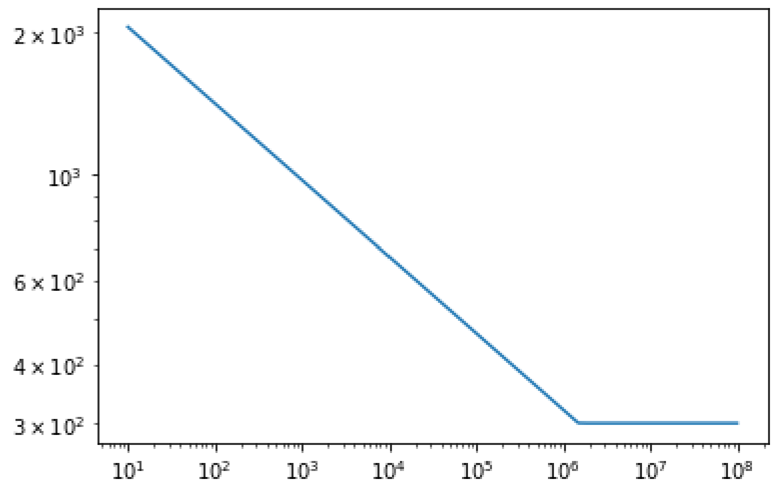
[2]:
```
woehler_curve_data = pd.Series({
    'SD': 300.0,
    'ND': 1.5e6,
    'k_1': 6.2
})
```

Using the `WoehlerCurve` class can do operations on the data. To instantiate the class we use the accessor attribute `woehler`. Then we can calculate the cycle number for a given load.

[3]:
```
woehler_curve_data.woehler.cycles(350.0)
```

[3]:
```
array(576794.57014027)
```

[4]:
```
cycles = np.logspace(1., 8., 70)
load = woehler_curve_data.woehler.load(cycles)
plt.plot(cycles, load)
plt.loglog()
plt.show()
```



This basically means that a material of the given Wöhler curve will fail after about 577k cycles when charged with a load of 350. Note that we don't use any units here. They just have to be consistent.

3.3.2 Damage sums

Usually we don't have a single load amplitude but a collective. We can describe a collective using a python object that has an `amplitude` and a `cycle` attribute. We can do that for example with a simple `pandas.DataFrame`:

[5]:
```
load_collective = pd.DataFrame({
    'cycles': [2e5, 3e4, 5e3, 2e2, 7e1],
    'amplitude': [374.0, 355.0, 340.0, 320.0, 290.0]
})
```

Using the `pylife.strength.fatigue.damage` function we can calculate the damage of each block of the load collective. Therefore we use the `fatigue` accessor to operate on the Wöhler data.

[6]:
```
from pylife.strength import fatigue
woehler_curve_data.fatigue.damage(load_collective)
```

[6]:
```
0    0.523107
1    0.056793
2    0.007243
3    0.000199
4    0.000000
Name: damage, dtype: float64
```

Now we know the damage contribution of each block of the load collective. Of course we can also easily calculate the damage sum by just summing up:

[7]:
```
woehler_curve_data.fatigue.damage(load_collective).sum()
```

[7]:
```
0.5873418943984274
```

3.3.3 Broadcasting to a FEM mesh

Oftentimes we want to map a load collective to a whole FEM mesh to map a load collective to every FEM node. For those kinds of mappings pyLife provides the`pylife.Broadcaster` facility.

In order to operate properly the `Broadcaster` needs to know the meanings of the rows of a `pandas.Series` or a `pandas.DataFrame`. For that it uses the index names. Therefore we have to set the index names appropriately.

[8]:
```
load_collective.index.name = 'load_block'
```

Then we setup simple node stress distribution and broadcast the load collective to it.

[9]:
```
node_stress = pd.Series(
    [1.0, 0.8, 1.3],
    index=pd.Index([1, 2, 3], name='node_id')
)

from pylife import Broadcaster

load_collective, node_stress = Broadcaster(node_stress).broadcast(load_collective)
```

As you can see, the `Broadcaster` returns two objects. The first is the object that has been broadcasted, in our case the load collective:

[10]:
```
load_collective
```

[10]:
```
                      cycles  amplitude
node_id load_block
1       0           200000.0      374.0
        1            30000.0      355.0
        2             5000.0      340.0
        3              200.0      320.0
        4               70.0      290.0
2       0           200000.0      374.0
        1            30000.0      355.0
        2             5000.0      340.0
        3              200.0      320.0
        4               70.0      290.0
3       0           200000.0      374.0
        1            30000.0      355.0
        2             5000.0      340.0
        3              200.0      320.0
        4               70.0      290.0
```

The second is the object that has been broadcasted to, in our case the node stress distribution.

[11]:
```
node_stress
```

[11]:
```
node_id  load_block
1        0             1.0
         1             1.0
         2             1.0
         3             1.0
         4             1.0
2        0             0.8
         1             0.8
         2             0.8
         3             0.8
         4             0.8
3        0             1.3
         1             1.3
         2             1.3
         3             1.3
         4             1.3
dtype: float64
```

As you can see, both have the same index, which is a cross product of the indices of the two initial objects. Now we can easily scale the load collective to the node stress distribution.

[12]:
```
load_collective['amplitude'] *= node_stress
load_collective
```

[12]:
```
                      cycles  amplitude
node_id load_block
1       0           200000.0      374.0
        1            30000.0      355.0
        2             5000.0      340.0
        3              200.0      320.0
        4               70.0      290.0
2       0           200000.0      299.2
        1            30000.0      284.0
        2             5000.0      272.0
        3              200.0      256.0
        4               70.0      232.0
3       0           200000.0      486.2
        1            30000.0      461.5
        2             5000.0      442.0
        3              200.0      416.0
        4               70.0      377.0
```

Now we have for each `load_block` for each `node_id` the corresponding amplitudes and cycle numbers. Again we can use the `damage` function to calculate the damage contribution of each load block on each node.

[13]:
```
damage_contributions = woehler_curve_data.fatigue.damage(load_collective)
damage_contributions
```

[13]:
```
node_id  load_block
1        0             0.523107
         1             0.056793
         2             0.007243
         3             0.000199
         4             0.000000
```

(continues on next page)

(continued from previous page)

```
2        0              0.000000
         1              0.000000
         2              0.000000
         3              0.000000
         4              0.000000
3        0              2.660968
         1              0.288897
         2              0.036842
         3              0.001012
         4              0.000192
Name: damage, dtype: float64
```

In order to calculate the damage sum for each node, we have to group the damage contributions by the node and sum them up:

[14]:
```
damage_contributions.groupby('node_id').sum()
```

[14]:
```
node_id
1    0.587342
2    0.000000
3    2.987911
Name: damage, dtype: float64
```

As you can see the damage sum for node 3 is higher than 1, which means that the stress exceeds the strength. So we would expect failure at node 3.

[ ]:
```

```

CHAPTER

FOUR

PYLIFE USER GUIDE

This document aims to briefly describe the overall design of pyLife and how to use it inside your own applications, for example Jupyter Notebooks.

4.1 Overview

pyLife provides facilities to perform different kinds of tasks. They can be roughly grouped as follows

4.1.1 Fitting material data

This is about extracting material parameters from experimental data. As of now this is a versatile set of classes to fit Wöhler curve (aka SN-curve) parameters from experimental fatigue data. Mid term we would like to see there a module to fit tensile test data and things like that.

- `pylife.materialdata.woehler`

4.1.2 Predicting material behavior

These modules use material parameters, e.g. the ones fitted by the corresponding module about data fitting, to predict material behavior. As of now these are

- `pylife.materiallaws.RambergOsgood`
- `pylife.materiallaws.WoehlerCurve`
- Functions to calculate the true stress and true strain, see `pylife.materiallaws.true_stress_strain`

4.1.3 Analyzing load collectives and stresses

These modules perform basic operations on time signals as well as more complpex things as rainflow counting.

- `pylife.stress.collective` – facilities to handle load collectives
- `pylife.stress.rainflow` – a versatile module for rainflow counting
- `pylife.stress.equistress` for equivalent stress calculations from stress tensors
- `pylife.stress.timesignal` for operations on time signals

4.1.4 Lifetime assessment of components

Calculate lifetime, failure probabilities, nominal endurance limits of components based on load collective and material data.

4.1.5 Mesh operations

For operations on FEM meshes

- *`pylife.mesh.meshsignal`* – accessor classes for general mesh operations
- *`pylife.mesh.HotSpot`* for hotspot detection
- *`pylife.mesh.Gradient`* to calculate gradients of scalar values along a mesh
- *`pylife.mesh.Meshmapper`* to map one mesh to another of the same geometry by interpolating

4.1.6 VMAP interface

Import and export mesh data from/to VMAP files.

4.1.7 Utilities

Some mathematical helper functions, that are useful throughout the code base.

4.2 General Concepts

pyLife aims to provide toolbox of calculation tools that can be plugged together in order to perform complex operations. We try to make the use of the existing modules as well as writing custom ones as easy as possible while at the same time performing well on larger amounts of data. Moreover we try keep data that belongs together in self explaining data structures, rather than in individual variables.

In order to achieve all that we make extensive use of pandas and numpy. In this guide we suppose that you have a basic understanding of these libraries and the data structures and concepts they are using.

The page *Data Model* describes the way how data should be stored in pandas objects.

The page *Signal API* describes how mathematical operations are to be performed on those pandas objects.

4.2.1 The Data Model of pyLife

pyLife stores data for calculations most often in *pandas* objects. Modules that want to operate on such pandas objects should use the *pandas* methods wherever possible.

Dimensionality of data

pandas comes with two data classes, `pandas.Series` for one dimensional data and `pandas.DataFrame` for two dimensional data.

This dimensionality has nothing to do with mathematical or geometrical dimensions, it only relates to the dimensionality from a data structure perspective. There are two dimensions, we call them the one in *row direction* and the one in *column direction*.

In row direction, all the values represent the same physical quantity, like a stress, a length, a volume, a time. It means that it is easily thinkable to add infinitely more values in row direction. Furthermore it mostly would make sense to perform statistical operations in row direction, like the maximum stress, the average volume, etc.

An one channel time signal is an example for a one dimensional data structure in row direction. You could add infinitely more samples, every sample represents the same physical quantity, it makes sense to perform statistical operations on the whole series.

```
In [4]: row_direction
Out[4]:
time
0     0.497646
1     0.278503
2     0.649374
3     0.419474
4     0.614923
5     0.961856
... <infinitely more could be added>
Name: Load, dtype: float64
```

In column direction, the values usually represent different physical quantities. You might be able to think of adding a few more values in column direction, but not infinitely. It almost never makes sense to perform statistical operations in column direction.

A Wöhler curve is an example for a one dimensional example in column direction. All the members (columns) represent different physical quantities, it is not obvious to add more values to the data structure and it makes no sense to perform statistical operations to it.

```
In [8]: column_direction
Out[8]:
SD      3.000000e+02
ND      2.300000e+06
k_1     7.000000e+00
TS      1.234363e+00
TN      3.420000e+00
dtype: float64
```

Two dimensional data structures have both dimensions. An example would be a FEM-mesh.

```
In [9]: vmap.make_mesh('1', 'STATE-2').join_coordinates().join_variable('STRESS_CAUCHY').
↪join_variable('DISPLACEMENT').to_frame()
Out[9]:
                            x         y    z        S11       S22  S33        S12  S13 ␣
↪S23        dx        dy   dz
element_id node_id
1          1734     14.897208  5.269875  0.0  27.080811  6.927080  0.0 -13.687358  0.0 ␣
↪0.0  0.005345  0.000015  0.0
```

(continues on next page)

(continued from previous page)

```
           1582     14.555333  5.355806  0.0  28.319006  1.178649  0.0 -10.732705  0.0 ␣
↪0.0  0.005285  0.000003  0.0
           1596     14.630658  4.908741  0.0  47.701195  5.512213  0.0 -17.866833  0.0 ␣
↪0.0  0.005376  0.000019  0.0
           4923     14.726271  5.312840  0.0  27.699907  4.052865  0.0 -12.210032  0.0 ␣
↪0.0  0.005315  0.000009  0.0
           4924     14.592996  5.132274  0.0  38.010101  3.345431  0.0 -14.299768  0.0 ␣
↪0.0  0.005326  0.000013  0.0
...                       ...       ...  ...        ...       ...  ...        ...  ...  .
↪..       ...       ...  ...
4770       3812    -13.189782 -5.691876  0.0  36.527439  2.470588  0.0 -14.706686  0.0 ␣
↪0.0 -0.005300  0.000027  0.0
           12418   -13.560289 -5.278386  0.0  32.868889  3.320898  0.0 -14.260107  0.0 ␣
↪0.0 -0.005444  0.000002  0.0
           14446   -13.673285 -5.569107  0.0  34.291058  3.642457  0.0 -13.836027  0.0 ␣
↪0.0 -0.005404  0.000009  0.0
           14614   -13.389065 -5.709927  0.0  36.063541  2.828889  0.0 -13.774759  0.0 ␣
↪0.0 -0.005330  0.000022  0.0
           14534   -13.276068 -5.419206  0.0  33.804211  2.829817  0.0 -14.580153  0.0 ␣
↪0.0 -0.005371  0.000014  0.0

[37884 rows x 12 columns]
```

Even though a two dimensional rainflow matrix is two dimensional from a mathematical point of view, it is one dimensional from a data structure perspective because every element represents the same physical quantity (occurrence frequency of loops). You could add infinitely more samples by a finer bin and it can make sense to perform statistical operations on the whole matrix, for example in order to normalize it to the maximum frequency value.

In order to represent mathematically multidimensional structures like a two dimensional rainflow matrix we use `pandas.MultiIndex`. Example for a rainflow matrix in a two dimensional `pandas.IntervalIndex`.

```
In [11]: rainflow_matrix
Out[11]:
from                                      to
(-60.14656996518033, -49.911160871492996] (-50.39826857402471, -40.36454994053457]     ␣
↪7.0
                                          (-40.36454994053457, -30.33083130704449]    ␣
↪10.0
                                          (-30.33083130704449, -20.29711267355441]     ␣
↪5.0
                                          (-20.29711267355441, -10.26339404006427]     ␣
↪4.0
                                          (-10.26339404006427, -0.2296754065741311]    ␣
↪6.0
                                                                                       .
↪..
(42.20752097169333, 52.44293006538072]    (9.804043226915951, 19.837761860406033]      ␣
↪9.0
                                          (19.837761860406033, 29.871480493896172]     ␣
↪6.0
                                          (29.871480493896172, 39.90519912738631]      ␣
↪5.0
                                          (39.90519912738631, 49.93891776087639]       ␣
↪11.0
```

(continues on next page)

(continued from previous page)

```
                                              (49.93891776087639, 59.972636394366475]     ␣
↪5.0
Name: frequency, Length: 121, dtype: float64

In [12]: type(rainflow_matrix)
Out[12]: pandas.core.series.Series
```

4.2.2 The pyLife Signal API

The signal api is the higher level API of pyLife. It is the API that you probably should be using. Some of the domain specific functions are also available as pure numpy functions. However, we highly recommend you to take a closer look at pandas and consider to adapt your application to the pandas way of doing things.

Motivation

In pyLife's domain, we often deal with data structures that consist multiple numerical values. For example a Wöhler curve (see *WoehlerCurve*) consists at least of the parameters `k_1`, `ND` and `SD`. Optionally it can have the additional parameters `k_2`, `TN` and `TS`. As experience teaches us, it is advisable to put these kinds of values into one variable to avoid confusion. For example a function with more than five positional arguments often leads to hard to debugable bugs due to wrong positioning.

Moreover some of these data structures can come in different dimensionalities. For example data structures describing material behavior can come as one dataset, describing one material. They can also come as a map to a FEM mesh, for example when you are dealing with case hardened components. Then every element of your FEM mesh can have a different associated Wöhler curve dataset. In pyLife we want to deal with these kinds of mappings easily without much coding overhead for the programmer.

The pyLife signal API provides the class *pylife.PylifeSignal* to facilitate handling these kinds of data structures and broadcasting them to another signal instance.

This page describes the basic concept of the pyLife signal API. The next page describes the broadcasting mechanism of a *pylife.PylifeSignal*.

The basic concept

The basic idea is to have all the data in a signal like data structure, that can be piped through the individual calculation process steps. Each calculation process step results in a new signal, that then can be handed over to the next process step.

Signals can be for example

- stress tensors like from an FEM-solver
- load collectives like time signals or a rainflow matrix
- material data like Wöhler curve parameters
- …

From a programmer's point of view, signals are objects of either `pandas.Series` or `pandas.DataFrame`, depending if they are one or two dimensional (see here about *dimensionality*).

Functions that operate on a signal are usually written as methods of an instance of as class derived from `PylifeSignal`. These classes are usually decorated as Series or DataFrame accessor using `pandas.api.extensions.register_series_accessor()` resp. `pandas.api.extensions.register_dataframe_accessor()`.

Due to the decorators, signal accessor classes can be instantiated also as an attribute of a pandas.Series or pandas.DataFrame. The following two lines are equivalent.

Usual class instantiation:

```
PlainMesh(df).coordinates
```

Or more convenient using the accessor decorator attribute:

```
df.plain_mesh.coordinates
```

There is also the convenience function `from_parameters()` to instantiate the signal class from individual parameters. So a `pylife.materialdata.WoehlerCurve` can be instantiated in three ways.

- directly with the class constructor

```
data = pd.Series({
    'k_1': 7.0,
    'ND': 2e6,
    'SD': 320.
})
wc = WoehlerCurve(data)
```

- using the pandas accessor

```
data = pd.Series({
    'k_1': 7.0,
    'ND': 2e6,
    'SD': 320.
})
wc = data.woehler
```

- from individual parameters

```
wc = WoehlerCurve.from_parameters(k_1=7.0, ND=2e6, SD= 320.)
```

How to use predefined signal accessors

There are too reasons to use a signal accessor:

- let it validate the accessed DataFrame
- use a method or access a property that the accessor defines

Example for validation

In the following example we are validating a DataFrame that if it is a valid plain mesh, i.e. if it has the columns *x* and *y*.

Import the modules. Note that the module with the signal accessors (here `mesh`) needs to be imported explicitly.

```
import pandas as pd
import pylife.mesh
```

Create a DataFrame and have it validated if it is a valid plain mesh, i.e. has the columns *x* and *y*.

```
df = pd.DataFrame({'x': [1.0], 'y': [1.0]})
df.plain_mesh
```

```
<pylife.mesh.meshsignal.PlainMesh at 0x7f0d94b6bc70>
```

Now create a DataFrame which is not a valid plain mesh and try to have it validated:

```
df = pd.DataFrame({'x': [1.0], 'a': [1.0]})
df.plain_mesh
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [3], in <module>
      1 df = pd.DataFrame({'x': [1.0], 'a': [1.0]})
----> 2 df.plain_mesh

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/generic.py:5583, in NDFrame.__getattr__(self, name)
   5576 if (
   5577     name not in self._internal_names_set
   5578     and name not in self._metadata
   5579     and name not in self._accessors
   5580     and self._info_axis._can_hold_identifiers_and_holds_name(name)
   5581 ):
   5582     return self[name]
-> 5583 return object.__getattribute__(self, name)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/accessor.py:182, in CachedAccessor.__get__(self, obj, cls)
    179 if obj is None:
    180     # we're accessing the attribute of the class, i.e., Dataset.geo
    181     return self._accessor
--> 182 accessor_obj = self._accessor(obj)
    183 # Replace the property with the accessor object. Inspired by:
    184 # https://www.pydanny.com/cached-property.html
    185 # We need to use object.__setattr__ because we overwrite __setattr__ on
    186 # NDFrame
    187 object.__setattr__(obj, self._name, accessor_obj)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:62, in PylifeSignal.__init__(self, pandas_obj)
     54 """Instantiate a :class:`signal.PyLifeSignal`.
     55 
     56 Parameters
   (...)
     59 
     60 """
     61 self._obj = pandas_obj
---> 62 self._validate()

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/mesh/meshsignal.py:102, in PlainMesh._validate(self)
```

(continues on next page)

(continued from previous page)

```
    100 def _validate(self):
    101     self._coord_keys = ['x', 'y']
--> 102     self.fail_if_key_missing(self._coord_keys)
    103     if 'z' in self._obj.columns:
    104         self._coord_keys.append('z')

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:184, in PylifeSignal.fail_if_key_missing(self,␣
↪keys_to_check, msg)
    153 def fail_if_key_missing(self, keys_to_check, msg=None):
    154     """Raise an exception if any key is missing in a self._obj object.
    155 
    156     Parameters
   (...)
    182     :class:`stresssignal.StressTensorVoigt`
    183     """
--> 184     DataValidator().fail_if_key_missing(self._obj, keys_to_check)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/data_validator.py:131, in DataValidator.fail_if_key_missing(self,␣
↪signal, keys_to_check, msg)
    129     the_class = stack[2][0].f_locals['self'].__class__
    130     msg = the_class.__name__ + ' must have the items %s. Missing %s.'
--> 131 raise AttributeError(msg % (', '.join(keys_to_check), ', '.join(missing_keys)))

AttributeError: PlainMesh must have the items x, y. Missing y.
```

Example for accessing a property

Get the coordinates of a 2D plain mesh

```
df = pd.DataFrame({'x': [1.0, 2.0, 3.0], 'y': [1.0, 2.0, 3.0]})
df.plain_mesh.coordinates
```

Now a 3D mesh

```
df = pd.DataFrame({'x': [1.0], 'y': [1.0], 'z': [1.0], 'foo': [42.0], 'bar': [23.0]})
df.plain_mesh.coordinates
```

Defining your own signal accessors

If you want to write a processor for signals you need to put the processing functionality in an accessor class that is derived from the signal accessor base class like for example *Mesh*. This class you register as a pandas DataFrame accessor using a decorator

```
import pandas as pd
import pylife.mesh

@pd.api.extensions.register_dataframe_accessor('my_mesh_processor')
class MyMesh(meshsignal.Mesh):
```

(continues on next page)

(continued from previous page)

```
    def do_something(self):
        # ... your code here
        # the DataFrame is accessible by self._obj
        # usually you would calculate a DataFrame df to return it.
        df = ...
        # you might want copy the index of self._obj to the returned
        # DataFrame.
        return df.set_index(self._obj.index)
```

As *MyMesh* is derived from `Mesh` the validation of *Mesh* is performed. So in the method *do_something()* you can rely on that *self._obj* is a valid mesh DataFrame.

You then can use the class in the following way when the module is imported.

Performing additional validation

Sometimes your signal accessor needs to perform an additional validation on the accessed signal. For example you might need a mesh that needs to be 3D. Therefore you can reimplement *_validate()* to perform the additional validation. Make sure to call *_validate()* of the accessor class you are deriving from like in the following example.

```
import pandas as pd
import pylife.mesh

@pd.api.extensions.register_dataframe_accessor('my_only_for_3D_mesh_processor')
class MyOnlyFor3DMesh(pylife.mesh.PlainMesh):
    def _validate(self):
        super()._validate() # call PlainMesh._validate()
        self.fail_if_key_missing(['z'])

df = pd.DataFrame({'x': [1.0], 'y': [1.0]})
df.my_only_for_3D_mesh_processor
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [6], in <module>
      8         self.fail_if_key_missing(['z'])
     10 df = pd.DataFrame({'x': [1.0], 'y': [1.0]})
---> 11 df.my_only_for_3D_mesh_processor

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/generic.py:5583, in NDFrame.__getattr__(self, name)
   5576 if (
   5577     name not in self._internal_names_set
   5578     and name not in self._metadata
   5579     and name not in self._accessors
   5580     and self._info_axis._can_hold_identifiers_and_holds_name(name)
   5581 ):
   5582     return self[name]
-> 5583 return object.__getattribute__(self, name)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/accessor.py:182, in CachedAccessor.__get__(self, obj, cls)
```

(continues on next page)

(continued from previous page)

```
    179 if obj is None:
    180     # we're accessing the attribute of the class, i.e., Dataset.geo
    181     return self._accessor
--> 182 accessor_obj = self._accessor(obj)
    183 # Replace the property with the accessor object. Inspired by:
    184 # https://www.pydanny.com/cached-property.html
    185 # We need to use object.__setattr__ because we overwrite __setattr__ on
    186 # NDFrame
    187 object.__setattr__(obj, self._name, accessor_obj)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:62, in PylifeSignal.__init__(self, pandas_obj)
     54 """Instantiate a :class:`signal.PyLifeSignal`.
     55 
     56 Parameters
   (...)
     59 
     60 """
     61 self._obj = pandas_obj
---> 62 self._validate()

Input In [6], in MyOnlyFor3DMesh._validate(self)
      6 def _validate(self):
      7     super()._validate() # call PlainMesh._validate()
----> 8     self.fail_if_key_missing(['z'])

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:184, in PylifeSignal.fail_if_key_missing(self,␣
↪keys_to_check, msg)
    153 def fail_if_key_missing(self, keys_to_check, msg=None):
    154     """Raise an exception if any key is missing in a self._obj object.
    155 
    156     Parameters
   (...)
    182     :class:`stresssignal.StressTensorVoigt`
    183     """
--> 184     DataValidator().fail_if_key_missing(self._obj, keys_to_check)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/data_validator.py:131, in DataValidator.fail_if_key_missing(self,␣
↪signal, keys_to_check, msg)
    129     the_class = stack[2][0].f_locals['self'].__class__
    130     msg = the_class.__name__ + ' must have the items %s. Missing %s.'
--> 131 raise AttributeError(msg % (', '.join(keys_to_check), ', '.join(missing_keys)))

AttributeError: MyOnlyFor3DMesh must have the items z. Missing z.
```

Defining your own signals

The same way the predefined pyLife signals are defined you can define your own signals. Let's say, for example, that in your signal there needs to be the columns *alpha*, *beta*, *gamma* all of which need to be positive.

You would put the signal class into a module file *my_signal_mod.py*

```
import pandas as pd
from pylife import PylifeSignal

@pd.api.extensions.register_dataframe_accessor('my_signal')
class MySignal(PylifeSignal):
    def _validate(self):
        self.fail_if_key_missing(['alpha', 'beta', 'gamma'])
        for k in ['alpha', 'beta', 'gamma']:
            if (self._obj[k] < 0).any():
                raise ValueError("All values of %s need to be positive. "
                                 "At least one is less than 0" % k)

    def some_method(self):
        return self._obj[['alpha', 'beta', 'gamma']] * -3.0
```

You can then validate signals and/or call `some_method()`.

Validation success.

```
df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, 0.0], 'gamma': [1.0, 2.0]})
df.my_signal.some_method()
```

Validation fails because of missing *gamma* column.

```
df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, -1.0]})
df.my_signal.some_method()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [9], in <module>
      1 df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, -1.0]})
----> 2 df.my_signal.some_method()

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/generic.py:5583, in NDFrame.__getattr__(self, name)
   5576 if (
   5577     name not in self._internal_names_set
   5578     and name not in self._metadata
   5579     and name not in self._accessors
   5580     and self._info_axis._can_hold_identifiers_and_holds_name(name)
   5581 ):
   5582     return self[name]
-> 5583 return object.__getattribute__(self, name)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/accessor.py:182, in CachedAccessor.__get__(self, obj, cls)
    179 if obj is None:
```

(continues on next page)

(continued from previous page)

```
    180     # we're accessing the attribute of the class, i.e., Dataset.geo
    181     return self._accessor
--> 182 accessor_obj = self._accessor(obj)
    183 # Replace the property with the accessor object. Inspired by:
    184 # https://www.pydanny.com/cached-property.html
    185 # We need to use object.__setattr__ because we overwrite __setattr__ on
    186 # NDFrame
    187 object.__setattr__(obj, self._name, accessor_obj)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:62, in PylifeSignal.__init__(self, pandas_obj)
     54 """Instantiate a :class:`signal.PyLifeSignal`.
     55 
     56 Parameters
   (...)
     59 
     60 """
     61 self._obj = pandas_obj
---> 62 self._validate()

Input In [7], in MySignal._validate(self)
      6 def _validate(self):
----> 7     self.fail_if_key_missing(['alpha', 'beta', 'gamma'])
      8     for k in ['alpha', 'beta', 'gamma']:
      9         if (self._obj[k] < 0).any():

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:184, in PylifeSignal.fail_if_key_missing(self,␣
↪keys_to_check, msg)
    153 def fail_if_key_missing(self, keys_to_check, msg=None):
    154     """Raise an exception if any key is missing in a self._obj object.
    155 
    156     Parameters
   (...)
    182     :class:`stresssignal.StressTensorVoigt`
    183     """
--> 184     DataValidator().fail_if_key_missing(self._obj, keys_to_check)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/data_validator.py:131, in DataValidator.fail_if_key_missing(self,␣
↪signal, keys_to_check, msg)
    129     the_class = stack[2][0].f_locals['self'].__class__
    130     msg = the_class.__name__ + ' must have the items %s. Missing %s.'
--> 131 raise AttributeError(msg % (', '.join(keys_to_check), ', '.join(missing_keys)))

AttributeError: MySignal must have the items alpha, beta, gamma. Missing gamma.
```

Validation fail because one *beta* is negative.

```
df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, -1.0], 'gamma': [1.0, 2.0]})
df.my_signal.some_method()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [10], in <module>
      1 df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, -1.0], 'gamma': [1.0, 2.0]}
↪)
----> 2 df.my_signal.some_method()

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pandas/core/accessor.py:182, in CachedAccessor.__get__(self, obj, cls)
    179 if obj is None:
    180     # we're accessing the attribute of the class, i.e., Dataset.geo
    181     return self._accessor
--> 182 accessor_obj = self._accessor(obj)
    183 # Replace the property with the accessor object. Inspired by:
    184 # https://www.pydanny.com/cached-property.html
    185 # We need to use object.__setattr__ because we overwrite __setattr__ on
    186 # NDFrame
    187 object.__setattr__(obj, self._name, accessor_obj)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/2.0.0/lib/python3.8/site-
↪packages/pylife/core/pylifesignal.py:62, in PylifeSignal.__init__(self, pandas_obj)
     54 """Instantiate a :class:`signal.PyLifeSignal`.
     55 
     56 Parameters
   (...)
     59 
     60 """
     61 self._obj = pandas_obj
---> 62 self._validate()

Input In [7], in MySignal._validate(self)
      8 for k in ['alpha', 'beta', 'gamma']:
      9     if (self._obj[k] < 0).any():
---> 10         raise ValueError("All values of %s need to be positive. "
     11                          "At least one is less than 0" % k)

ValueError: All values of beta need to be positive. At least one is less than 0
```

Additional attributes in your own signals

If your accessor class needs to have attributes other than the accessed object itself you can define default values in the *__init__()* of your accessor and set these attributes with setter methods.

```
import pandas as pd
from pylife import PylifeSignal

@pd.api.extensions.register_dataframe_accessor('my_signal')
class MySignal(PylifeSignal):
    def __init__(self, pandas_obj):
        super(MySignal, self).__init__(pandas_obj)
        self._my_attribute = 'the default value'
```

(continues on next page)

(continued from previous page)

```
    def set_my_attribute(self, my_attribute):
        self._my_attribute = my_attribute
        return self

    def do_something(self, some_parameter):
        # ... use some_parameter, self._my_attribute and self._obj
```

```
>>> df.my_signal.set_my_attribute('foo').do_something(2342)
```

Registering a method to an existing accessor class

Note: This functionality might be dropped on the way to *pyLife-2.0* as it turns out that it is not that much used.

One drawback of the accessor class API is that you cannot extend accessors by deriving from them. For example if you need a custom equivalent stress function you cannot add it by deriving from *StressTensorEquistress*, and register it by the same accessor *equistress*.

The solution for that is `register_method()` that lets you monkey patch a new method to any class deriving from *PylifeSignal*.

```
from pylife import equistress

@pl.signal_register_method(equistress.StressTensorEquistress, 'my_equistress')
def my_equistress_method(df)
    # your code here
    return ...
```

Then you can call the method on any *DataFrame* that is accessed by *equistress*:

```
>>> df.equistress.my_equistress()
```

You can also have additional arguments in the registered method:

```
from pylife import equistress

@pl.signal_register_method(equistress.StressTensorEquistress, 'my_equistress_with_arg')
def my_equistress_method_with_arg(df, additional_arg)
    # your code here
    return ...
```

```
>>> df.equistress.my_equistress_with_arg(my_additional_arg)
```

4.2.3 The Signal Broadcaster

Motivation

pyLife tries to provide a flexible API for its functionality with respect to sizes of the datasets involved. No matter if you want to perform some calculation on just a single value or on a whole FEM-mesh. No matter if you want to calculate the damage that a certain load amplitude does on a certain material, or if you have a FEM-mesh with different materials associated with to element and every node has its own rainflow matrix.

Example

Take for example the function *cycles()*. Imagine you have a single Wöhler curve dataset like

```
import pandas as pd
from pylife.materiallaws import WoehlerCurve

woehler_curve_data = pd.Series({
    'k_1': 7.0,
    'ND': 2e5,
    'SD': 320.0,
    'TN': 2.3,
    'TS': 1.25
})

woehler_curve_data
```

```
k_1          7.00
ND      200000.00
SD         320.00
TN           2.30
TS           1.25
dtype: float64
```

Now you can calculate the cycles along the Basquin equation for a single load value:

```
woehler_curve_data.woehler.cycles(load=350.)
```

```
array(106807.93865297)
```

Now let's say, you have different loads for each *element_id* if your FEM-mesh:

```
amplitude = pd.Series([320., 340., 330., 320.], index=pd.Index([1, 2, 3, 4], name=
↪'element_id'))
amplitude
```

```
element_id
1    320.0
2    340.0
```

(continues on next page)

(continued from previous page)

```
3    330.0
4    320.0
dtype: float64
```

`cycles()` now gives you a result for every *element_id*.

```
woehler_curve_data.woehler.cycles(load=amplitude)
```

```
element_id
1    200000.000000
2    130836.050152
3    161243.517913
4    200000.000000
dtype: float64
```

In the next step, even the Wöhler curve data is different for every element, like for example for a hardness gradient in your component:

```
woehler_curve_data = pd.DataFrame({
    'k_1': 7.0,
    'ND': 2e5,
    'SD': [370., 320., 280, 280],
    'TN': 2.3,
    'TS': 1.25
}, index=pd.Index([1, 2, 3, 4], name='element_id'))

woehler_curve_data
```

In this case the broadcaster determines from the identical index name *element_id* that the two structures can be aligned, so every element is associated with its load and with its Wöhler curve:

```
woehler_curve_data.woehler.cycles(load=amplitude)
```

```
element_id
1             inf
2    1.308361e+05
3    6.331967e+04
4    7.853918e+04
dtype: float64
```

In another case we assume that you have a Wöhler curve associated to every element, and the loads are constant throughout the component but different for different load scenarios.

```
amplitude_scenarios = pd.Series([320., 340., 330., 320.], index=pd.Index([1, 2, 3, 4],␣
↪name='scenario'))
amplitude_scenarios
```

```
scenario
1    320.0
2    340.0
3    330.0
```

(continues on next page)

(continued from previous page)

```
4    320.0
dtype: float64
```

In this case the broadcaster makes a cross product of load *scenario* and *element_id*, i.e. for every *element_id* for every load *scenario* the allowable cycles are calculated:

```
woehler_curve_data.woehler.cycles(load=amplitude_scenarios)
```

```
element_id  scenario
1           1                    inf
            2                    inf
            3                    inf
            4                    inf
2           1           2.000000e+05
            2           1.308361e+05
            3           1.612435e+05
            4           2.000000e+05
3           1           7.853918e+04
            2           5.137878e+04
            3           6.331967e+04
            4           7.853918e+04
4           1           7.853918e+04
            2           5.137878e+04
            3           6.331967e+04
            4           7.853918e+04
dtype: float64
```

As is very uncommon that the load is constant all over the component like in the previous example we now consider an even more complex one. Let's say we have a different load scenarios, which give us for every *element_id* multiple load `scenario`s:

```
amplitude_scenarios = pd.Series(
    [320., 340., 330., 320, 220., 240., 230., 220, 420., 440., 430., 420],
    index=pd.MultiIndex.from_tuples([
        (1, 1), (1, 2), (1, 3), (1, 4),
        (2, 1), (2, 2), (2, 3), (2, 4),
        (3, 1), (3, 2), (3, 3), (3, 4)
    ], names=['scenario', 'element_id']))
amplitude_scenarios
```

```
scenario  element_id
1         1             320.0
          2             340.0
          3             330.0
          4             320.0
2         1             220.0
          2             240.0
          3             230.0
          4             220.0
3         1             420.0
          2             440.0
          3             430.0
```

(continues on next page)

(continued from previous page)

```
         4             420.0
dtype: float64
```

Now the broadcaster still aligns the *element_id*:

```
woehler_curve_data.woehler.cycles(load=amplitude_scenarios)
```

```
scenario  element_id
1         1                      inf
          2             1.308361e+05
          3             6.331967e+04
          4             7.853918e+04
2         1                      inf
          2                      inf
          3                      inf
          4                      inf
3         1             8.235634e+04
          2             2.152341e+04
          3             9.927892e+03
          4             1.170553e+04
dtype: float64
```

Note that in the above examples the call was always identical

```
woehler_curve_data.woehler.cycles(load=...)
```

That means that when you write a module for a certain functionality **you don't need to know if your code later on receives a single value parameter or a whole FEM-mesh**. Your code will take both and handle them.

Usage

As you might have seen, we did not call the `pylife.Broadcaster` in the above code snippets directly. And that's the way it's meant to be. When you are on the level that you simply want to use pyLife's functionality to perform calculations, you should not be required to think about how to broadcast your datasets to one another. It should simply happen automatically. In our example the the calls to the `pylife.Broadcaster` are done inside `cycles()`.

You do need to deal with the `pylife.Broadcaster` when you implement new calculation methods. Let's go through an example.

Todo: **Sorry**, this is still to be written.

CHAPTER

FIVE

PYLIFE COOKBOOK

5.1 Life time Calculation

demos/dash-apps/pyLife_logo_20200219_FINAL_RGB.png

This Notebook shows a general calculation stream for a nominal and local stress reliability approach.

5.1.1 Stress derivation

We are starting with the imported rainflow matrices. More information about the time series and loading handlin and the RF generation you can find in the notebook time_series_handling

1. Mean stress correction
2. Multiplication with repeating factor of every manoveur

5.1.2 Damage Calculation

1. Select the damage calculation method (Miner elementary, Miner-Haibach, . . .)
2. Calculate the damage for every load level and the damage sum
3. Calculate the failure probability with or w/o field scatter

5.1.3 Local stress approach

1. Load the FE mesh
2. Apply the load history to the FE mesh
3. Calculate the damage

[1]:
```
import numpy as np
import pandas as pd
import pickle
from pylife.utils.histogram import *
import pylife.stress.timesignal as ts

import pylife.stress.equistress

import pylife.stress
import pylife.strength.meanstress as MS
import pylife.strength.fatigue

import pylife.mesh.meshsignal

from pylife.strength import failure_probability as fp
import pylife.vmap

import pyvista as pv

import matplotlib.pyplot as plt
import matplotlib as mpl

from scipy.stats import norm

from helper_functions import plot_rf
# mpl.style.use('seaborn')
# mpl.style.use('seaborn-notebook')
mpl.style.use('bmh')
%matplotlib inline

# pv.set_plot_theme('document')
# pv.set_jupyter_backend('panel')
```

[2]:
```
# read the rf data
rf_dict = pickle.load(open("rf_dict.p", "rb"))
```

Meanstress transformation

Here we are using the *FKM Goodman* approach to calculate the meanstress transformation

[3]:
```
meanstress_sensitivity = pd.Series({
    'M': 0.3,
    'M2': 0.2
})
```

[4]:
```
transformed_dict = {k: rf_act.meanstress_transform.fkm_goodman(meanstress_sensitivity, R_
↪goal=-1.).to_pandas() for k, rf_act in rf_dict.items()}
```

Repeating factor

If you want to apply a repeating factor to your loads you can do it very easily:

[5]:
```
repeating = {
    'wn': 50.0,
    'sine': 25.0,
    'SoR': 25
}
```

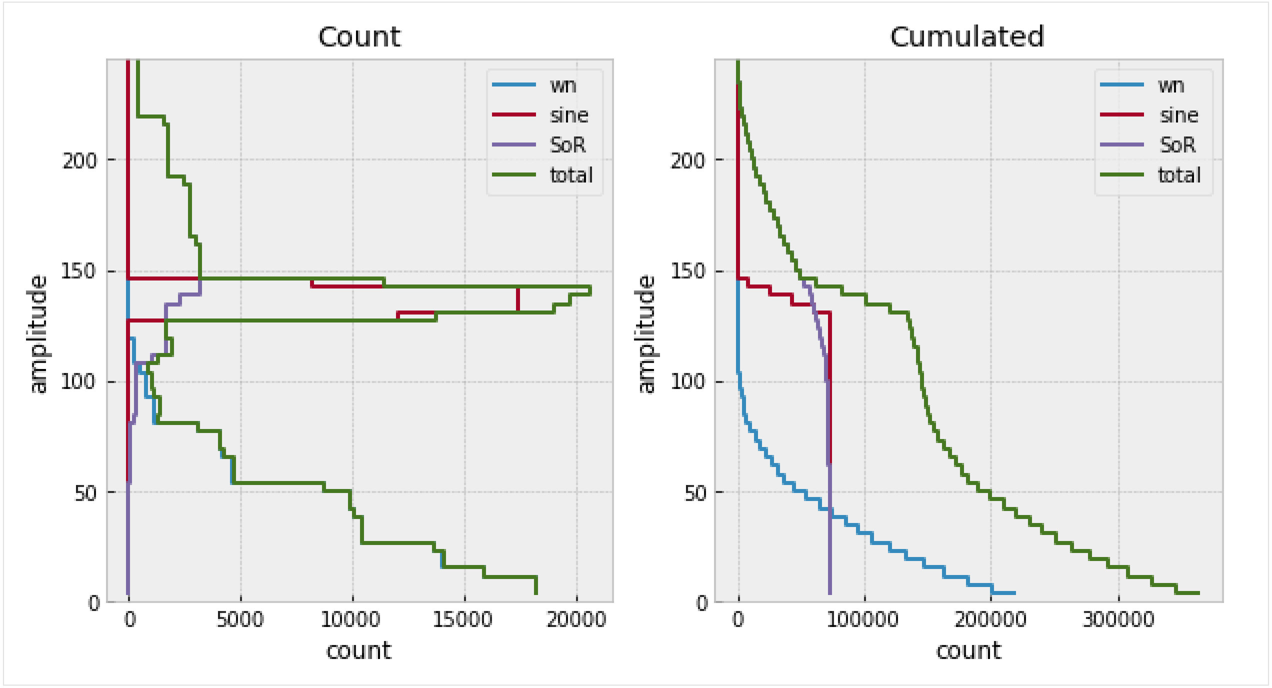
[6]:
```
load_dict = {k: transformed_dict[k] * repeating[k] for k in repeating.keys()}
```

We are calculating a seperat load case, where we summarize the three channels together. Later on we can compare the damage results of this channel with the sum of the other channels.

[7]:
```
load_dict['total'] = pd.concat([load_dict[k] for k in load_dict.keys()])
```

[8]:
```
bins = pd.interval_range(0., load_dict['total'].load_collective.use_class_right().
↪amplitude.max(), 64)
rebinned_dict = {k: rebin_histogram(v.load_collective.amplitude_histogram, bins) for k,␣
↪v in load_dict.items()}
```

[9]:
```
fig, ax = plt.subplots(nrows=1, ncols=2,figsize=(10, 5))

for k, v in rebinned_dict.items():
    amplitude = v.index.right[::-1]
    cycles = v[::-1]
    ax[0].step(cycles, amplitude, label=k)
    ax[1].step(np.cumsum(cycles), amplitude, label=k)

for title, ai in zip(['Count', 'Cumulated'], ax):
    ai.set_title(title)
    ai.xaxis.grid(True)
    ai.legend()
    ai.set_xlabel('count')
    ai.set_ylabel('amplitude')
    ai.set_ylim((0,max(amplitude)))
```



Nominal stress approach

Material parameters

You can create your own material data from Woeler tests using the Notebook woehler_analyzer

```
[10]: k_1 = 8
mat = pd.Series({
    'k_1': k_1,
    'k_2' : 2 * k_1 - 1,
    'ND': 1.0e6,
    'SD': 300.0,
    'TN': 12.,
    'TS': 1.1
})
display(mat)
```

```
k_1           8.0
k_2          15.0
ND      1000000.0
SD          300.0
TN           12.0
TS            1.1
dtype: float64
```

Damage Calculation

Now we can calculate the damage for every loadstep and summarize this damage to get the total damage.

[11]:
```
# damage for every load range
damage_miner_original = {k: mat.fatigue.damage(v.load_collective) for k, v in load_dict.
↪items()}
damage_miner_elementary = {k: mat.fatigue.miner_elementary().damage(v.load_collective)␣
↪for k, v in load_dict.items()}
damage_miner_haibach = {k: mat.fatigue.miner_haibach().damage(v.load_collective) for k,␣
↪v in load_dict.items()}

# and the damage sum
damage_sum_miner_haibach = {k: v.sum() for k, v in damage_miner_haibach.items()}
# ... and so on
print(damage_sum_miner_haibach)

print("total from sum: " + str(damage_sum_miner_haibach["wn"] + damage_sum_miner_haibach[
↪"sine"] + damage_sum_miner_haibach["SoR"]))
```

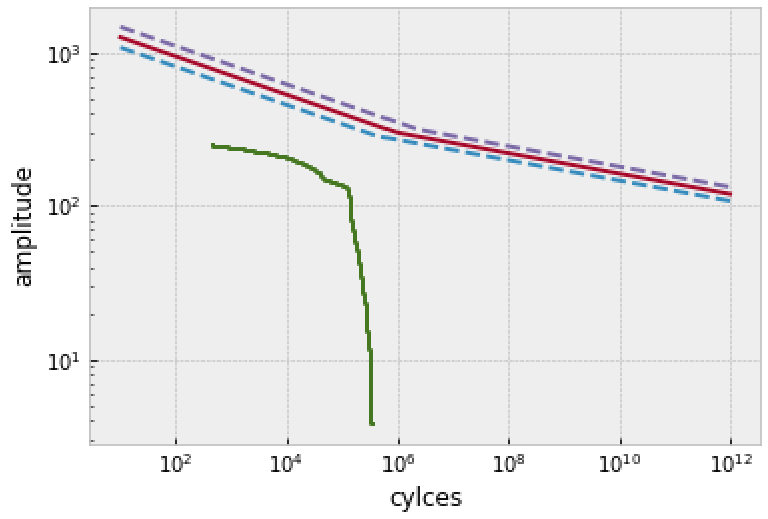
```
{'wn': 5.679531727172357e-10, 'sine': 5.084090614627166e-07, 'SoR': 0.
↪00012047238604791249, 'total': 0.00030263135776530 86}
total from sum: 0.00012098136306254792
```

If we compare the sum of the first three load channels with the 'total' one. The different is based on the fact that we have used 10 bins only. Try to rerun the notebook with a higher bin resolution and you will see the differences.

5.1.4 Plot the damage vs collectives

[12]:
```
wc = mat.woehler
cyc = pd.Series(np.logspace(1, 12, 200))
for pf, style in zip([0.1, 0.5, 0.9], ['--', '-', '--']):
    load = wc.basquin_load(cyc, failure_probability=pf)
    plt.plot(cyc, load, style)

plt.step(np.cumsum(rebinned_dict['total'][::-1]), rebinned_dict['total'].index.right[::-
↪1])
plt.xlabel("cylces"), plt.ylabel("amplitude")
plt.loglog()
```

[12]:
```
[]
```

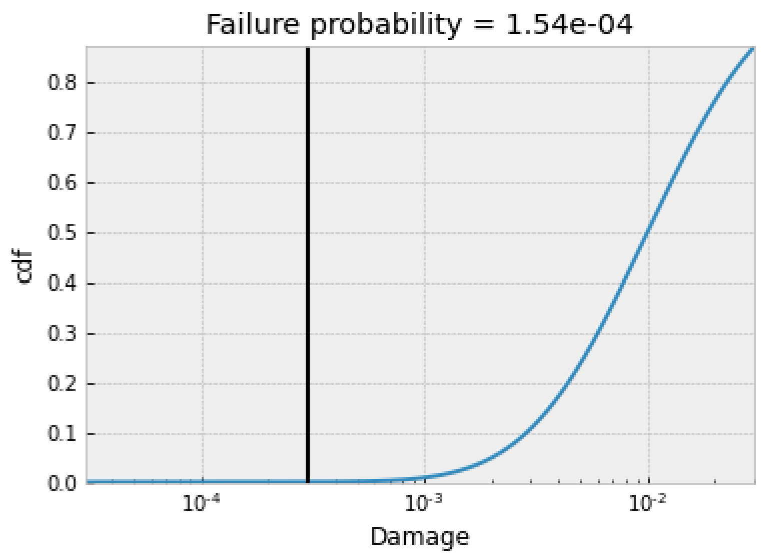


5.1.5 Without field scatter

In the first use case we assume, that we have the material scatter only. With that we can calculate the failure probability using the *FailureProbability* class.

```
[13]: D50 = 0.01

damage = damage_sum_miner_haibach["total"]

di = np.logspace(np.log10(1e-1*damage), np.log10(1e2*damage), 1000)
std = pylife.utils.functions.scattering_range_to_std(mat.TN)
failprob = fp.FailureProbability(D50, std).pf_simple_load(di)

fig, ax = plt.subplots()
ax.semilogx(di, failprob, label='cdf')
plt.vlines(damage, ymin=0, ymax=1, color="black")
plt.xlabel("Damage")
plt.ylabel("cdf")
plt.title("Failure probability = %.2e" %fp.FailureProbability(D50,std).pf_simple_
↪load(damage))
plt.ylim(0,max(failprob))
plt.xlim(min(di), max(di))
```

```
[13]: (3.026313577653087e-05, 0.03026313577653087)
```

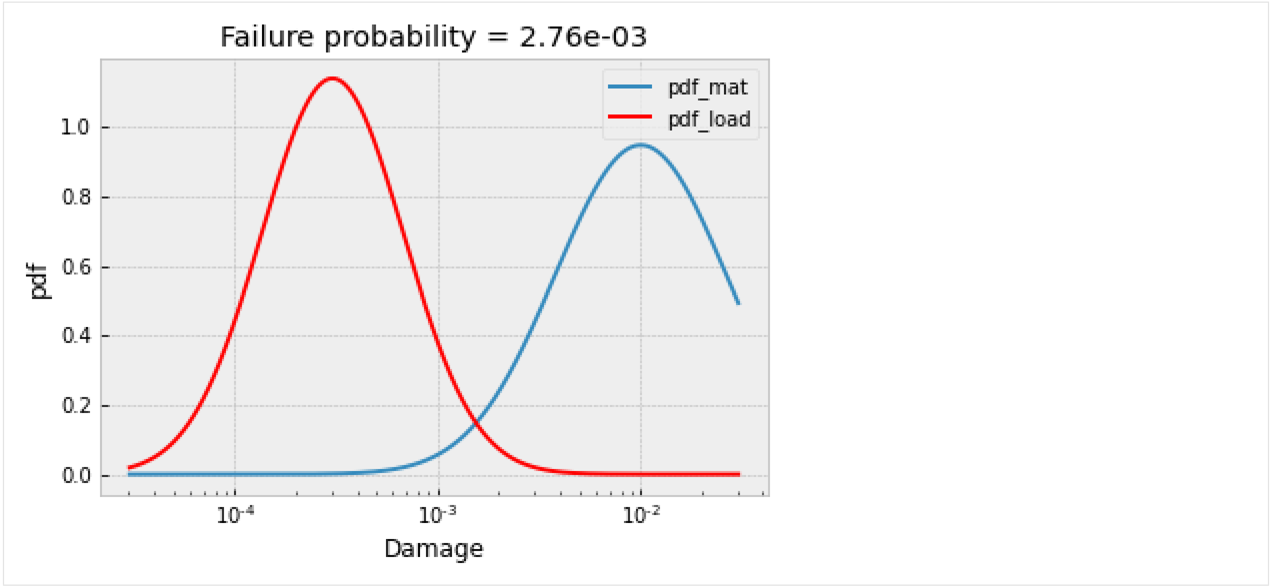


5.1.6 With field scatter

If we have the field scatter we can calculate the failure probability using convoluation of the probility density functions of the load and the strength.

[14]:
```
field_std = 0.35
fig, ax = plt.subplots()
# plot pdf of material
mat_pdf = norm.pdf(np.log10(di), loc=np.log10(D50), scale=std)
ax.semilogx(di, mat_pdf, label='pdf_mat')
# plot pdf of load
field_pdf = norm.pdf(np.log10(di), loc=np.log10(damage), scale=field_std)
ax.semilogx(di, field_pdf, label='pdf_load',color = 'r')
plt.xlabel("Damage")
plt.ylabel("pdf")
plt.title("Failure probability = %.2e" %fp.FailureProbability(D50, std).pf_norm_
↪load(damage, field_std))
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7faab5d80190>
```



5.1.7 FE based failure probability calculation

5.1.8 FE Data

```
[15]: vm_mesh = pylife.vmap.VMAPImport("plate_with_hole.vmap")
      pyLife_mesh = (vm_mesh.make_mesh('1', 'STATE-2')
                     .join_coordinates()
                     .join_variable('STRESS_CAUCHY')
                     .to_frame())
```

```
[16]: mises = pyLife_mesh.groupby('element_id')['S11', 'S22', 'S33', 'S12', 'S13', 'S23'].
      ↪mean().equistress.mises()
      mises /= 150.0  # the nominal load level in the FEM analysis
      #mises
```

5.1.9 Damage Calculation

```
[17]: scaled_collective = load_dict['total'].load_collective.scale(mises)
```

```
[18]: damage = mat.fatigue.damage(scaled_collective)
```

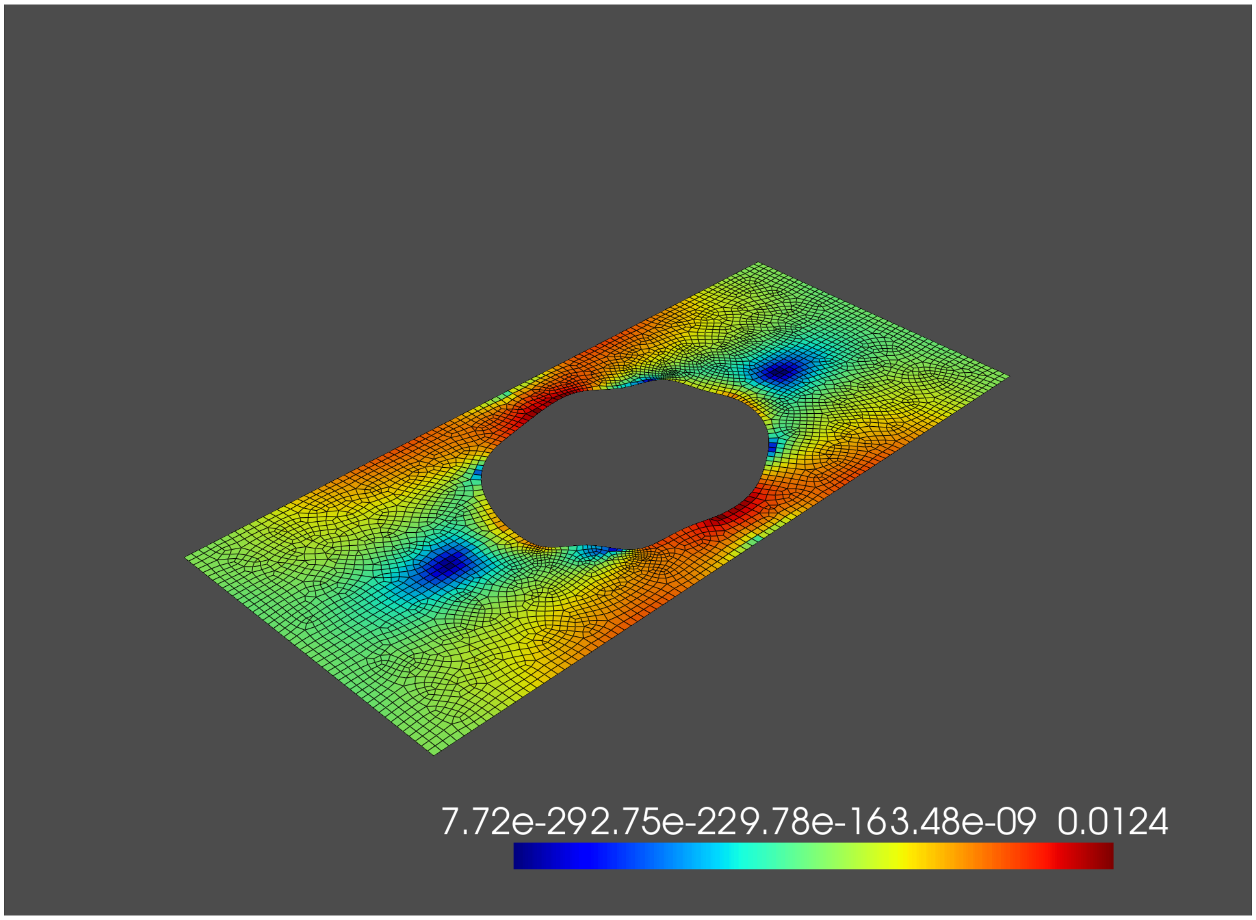
```
[19]: damage = damage.groupby(['element_id']).sum()
      #damage
```

```
[20]: grid = pv.UnstructuredGrid(*pyLife_mesh.mesh.vtk_data())
      plotter = pv.Plotter(window_size=[1920, 1400])
      plotter.add_mesh(grid, scalars=damage.to_numpy(), log_scale=True,
```

(continues on next page)

(continued from previous page)

```
                  show_edges=True, cmap='jet')
plotter.add_scalar_bar()
plotter.show()
```



[21]:
```
print("Maximal damage sum: %f" % damage.max())
```

```
Maximal damage sum: 0.012388
```

Failure probability of the plate

Often we don't get the volume of the FE data from the result file. But with pyVista we can calculate the volume (or area for 2d elements) easily:

[22]:
```
areas =  grid.compute_cell_sizes().cell_arrays["Area"]
```

To get the failure probability we have to proceed the following steps:

- get the failure probality of every element
- get the probality of survival for every element
- get the probality of survival for the whole component normed based on the volume (or area in 2d) of the element
- get the failure probality for the whole component

[23]:
```
fp_per_element = fp.FailureProbability(D50, std).pf_simple_load(damage)
probability_of_survival_per_ele = 1 - fp_per_element
probability_of_survival_component = (probability_of_survival_per_ele ** (areas/areas.
↪sum())).prod()
fp_component = 1 - probability_of_survival_component
```

[24]:
```
print('\033[1m' + "Failure probability of the component is %.2e" %fp_component)
```

```
Failure probability of the component is 4.23e-04
```

5.2 Ramberg Osgood relation

The RambGood module allows you to easily calculate stress strain curves and stress strain hytesresis loops using the Ramberg Osgood relation starting from the Hollomon parameters and Young's modulus of a material.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylife.materiallaws as ML
```

5.2.1 Initialize the RambergOsgood class

[2]:
```
ramberg_osgood = ML.RambergOsgood(E=210e3, K=1800, n=0.2)
```

5.2.2 Calculate the monotone branch

[3]:
```
max_stress = 800
monotone_stress = np.linspace(0, max_stress, 150)
monotone_strain = ramberg_osgood.strain(monotone_stress)
```

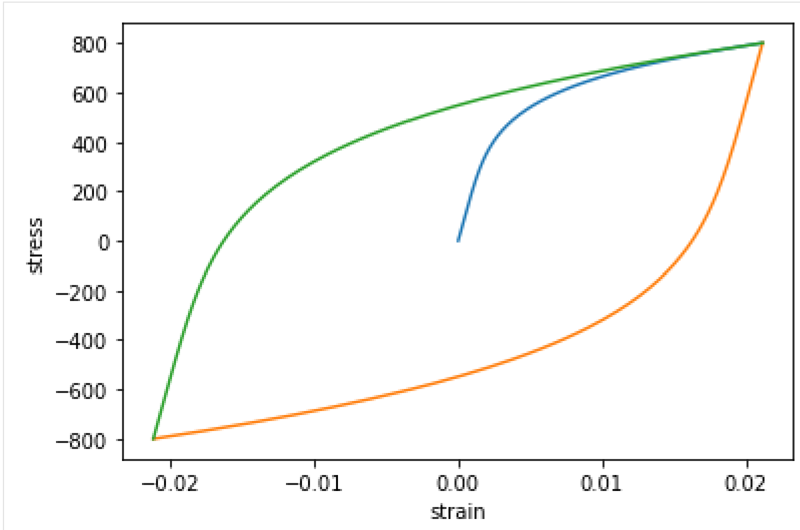
5.2.3 Calculate the cyclic branch

We calculate the lower branch of the hyteresis loop. By flipping it we get the upper branch.

[4]:
```
hyst_stress = np.linspace(-max_stress, max_stress, 150)
hyst_strain = ramberg_osgood.lower_hysteresis(hyst_stress, max_stress)
```

[5]:
```
plt.plot(monotone_strain, monotone_stress)

plt.plot(hyst_strain, hyst_stress)
plt.plot(-hyst_strain, np.flip(hyst_stress))
plt.xlabel('strain')
plt.ylabel('stress')
```

[5]:
```
Text(0, 0.5, 'stress')
```



5.3 Wöhler analyzing functions

Developed by Mustapha Kassem in scope of a master thesis at TU München

[1]:
```
import copy
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go

import pylife.materialdata.woehler as woehler
from pylife.materiallaws import WoehlerCurve
```

```
WARNING (theano.tensor.blas): Using NumPy C-API based implementation for BLAS functions.
```

The Wölhler analysis module takes fatigue data, i. e. values of the form `cycles load fracture` that have been measured by a fatigue testing lab and analyze it to return the parameters of a Wöhler curve.

These are: * the slope `k_1` * the cycle number of the endurance limit `ND` * the load level of the endurance limit `SD` * the scatter line cycle (lifetime) direction `TN` * the scatter in load direction `TS`

We will see several methods to perform the analysis below.

5.3.1 Data import

Data is made up of two columns:

- The first column is made up of the load values
- The scond column is made up of the load-cycle values

[2]:
```
file_name = 'data/woehler/fatigue-data-plain.csv'
```

```
[3]: df = pd.read_csv(file_name, sep='\t')
     df.columns=['load', 'cycles']
     px.scatter(df, x='cycles', y='load', log_x=True, log_y=True)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Guessing the fractures

In this case there is no information if the specimen was a runout or a fracture. We can guess it based on the value for the `load_cycle_limit` which defaults to `1e7`.

```
[4]: load_cycle_limit = None # or for example 1e7
     df = woehler.determine_fractures(df, load_cycle_limit)
     px.scatter(df, x='cycles', y='load', color='fracture', log_x=True, log_y=True)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.3.2 Analysis

Genaral preparations

Before we do the actual analysis, we make some preparations like, guessing the initial `fatigue_limit`, distinguishing between `runouts` and `fractures` and between `infinite_zone` and `finite_zone`.

```
[5]: fatigue_data = df.fatigue_data
     fatigue_data.fatigue_limit
```

```
[5]: 308.909475
```

We can distinguish between the finite and infinite zones.

```
[6]: infinite_zone = fatigue_data.infinite_zone
     finite_zone = fatigue_data.finite_zone

     go.Figure([
         go.Scatter(x=finite_zone.cycles, y=finite_zone.load, mode='markers', name='finite'),
         go.Scatter(x=infinite_zone.cycles, y=infinite_zone.load, mode='markers', name=
     ↪'infinite'),
         go.Scatter(x=[df.cycles.min(), df.cycles.max()], y=[fatigue_data.fatigue_limit]*2,␣
     ↪mode='lines', name='fatigue limit')
     ]).update_xaxes(type='log').update_yaxes(type='log').update_layout(xaxis_title='Cycles',␣
     ↪yaxis_title='Load')
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

We can separate fractures from runouts.

[7]:
```
fractures = fatigue_data.fractures
runouts = fatigue_data.runouts

fig = go.Figure([
    go.Scatter(x=fractures.cycles, y=fractures.load, mode='markers', name='fractures'),
    go.Scatter(x=runouts.cycles, y=runouts.load, mode='markers', name='runouts'),
    go.Scatter(x=[df.cycles.min(), df.cycles.max()], y=[fatigue_data.fatigue_limit]*2,␣
↪mode='lines', name='fatigue limit')
]).update_xaxes(type='log').update_yaxes(type='log').update_layout(xaxis_title='Cycles',␣
↪yaxis_title='Load')
fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Elementary analysis

The `Elementary` analysis is the first step of the analysis. It determines the slope `k_1` in the finite region and the scatter in cycle region `TN` using the pearl chain method.

The endurance limit in load direction `SN` is guessed from the tentative fatigue limit. The scatter in load direction `TS` is transformed from `TN` using the slope `k_1`.

[8]:
```
elementary_result = woehler.Elementary(fatigue_data).analyze()
elementary_result
```

[8]:
```
k_1                         8.626165
ND                     898426.345672
SD                        308.909475
TN                         12.059468
TS                          1.334610
failure_probability         0.500000
dtype: float64
```

[9]:
```
wc = elementary_result.woehler

cycles = np.logspace(np.log10(df.cycles.min()), np.log10(df.cycles.max()), 100)
elementary_fig = copy.deepcopy(fig)

elementary_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles), mode='lines', name=
↪'Elementary 50%')
elementary_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.1),
                           mode='lines', name='Elementary 10%')
elementary_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.9),
                           mode='lines', name='Elementary 90%')

elementary_fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Probit

[10]:
```
probit_result = woehler.Probit(fatigue_data).analyze()
probit_result
```

[10]:
```
k_1                    8.626165e+00
ND                     1.199879e+06
SD                     2.987202e+02
TN                     1.205947e+01
TS                     1.110052e+00
failure_probability    5.000000e-01
dtype: float64
```

[11]:
```
wc = probit_result.woehler

cycles = np.logspace(np.log10(df.cycles.min()), np.log10(df.cycles.max()), 100)
probit_fig = copy.deepcopy(fig)
probit_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles), mode='lines', name='Probit 50
↪%')
probit_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.1),␣
↪mode='lines', name='Probit 10%')
probit_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.9),␣
↪mode='lines', name='Probit 90%')

probit_fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Maximum Likelihood Infinite

The Maximum Likelihood Infinite method takes the parameters for the finite regime fitted by `Elementary` (`k_1`, `TN`) and fits the ones for the infinite regime (`SD`, `ND`, `TS`), hence the name.

[12]:
```
maxlike_inf_result = woehler.MaxLikeInf(fatigue_data).analyze()
maxlike_inf_result
```

[12]:
```
k_1                    8.626165e+00
ND                     1.326971e+06
SD                     2.952540e+02
TN                     1.205947e+01
TS                     1.106812e+00
failure_probability    5.000000e-01
dtype: float64
```

[13]:
```
wc = maxlike_inf_result.woehler

cycles = np.logspace(np.log10(df.cycles.min()), np.log10(df.cycles.max()), 100)
maxlike_inf_fig = copy.deepcopy(fig)
maxlike_inf_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles), mode='lines', name=
↪'MaxLikeInf 50%')
maxlike_inf_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.1),
↪ mode='lines', name='MaxLikeInf 10%')
```

(continues on next page)

(continued from previous page)

```
maxlike_inf_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.9),
↪ mode='lines', name='MaxLikeInf 90%')

maxlike_inf_fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Maximum Likelihood Full

The Maximum Likelihood Full method just takes the elementary result as starting values but fits all the parameters.

[14]:
```
maxlike_full_result = woehler.MaxLikeFull(fatigue_data).analyze()
maxlike_full_result
```

[14]:
```
k_1                    8.626165e+00
ND                     1.326971e+06
SD                     2.952540e+02
TN                     9.862007e+00
TS                     1.106811e+00
failure_probability    5.000000e-01
dtype: float64
```

[15]:
```
wc = maxlike_full_result.woehler

cycles = np.logspace(np.log10(df.cycles.min()), np.log10(df.cycles.max()), 100)
maxlike_full_fig = copy.deepcopy(fig)
maxlike_full_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles), mode='lines', name=
↪'MaxLikeFull 50%')
maxlike_full_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.
↪1), mode='lines', name='MaxLikeFull 10%')
maxlike_full_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.
↪9), mode='lines', name='MaxLikeFull 90%')

maxlike_full_fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Maximum Likelihood Full with fixed parameters

Sometimes it is desirable to pin one or more parameters of the Wöhler curve to a predefined value based on assumptions of the Wöhler curve. You can archive that by handing the fixed parameters to the `MaxLikeFull` object using the `fixed_parameters` argument.

[16]:
```
fixed_parameters = {
    'k_1': 7.,
    'ND': 1e6
}
```

(continues on next page)

(continued from previous page)

```
maxlike_fixed_result = woehler.MaxLikeFull(fatigue_data).analyze(fixed_parameters=fixed_
↪parameters)
maxlike_fixed_result
```

[16]:
```
k_1                          7.000000
ND                     1000000.000000
SD                         296.971820
TN                          10.201474
TS                           1.114568
failure_probability          0.500000
dtype: float64
```

[17]:
```
wc = maxlike_fixed_result.woehler

cycles = np.logspace(np.log10(df.cycles.min()), np.log10(df.cycles.max()), 100)
maxlike_fixed_fig = copy.deepcopy(fig)
maxlike_fixed_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles), mode='lines', name=
↪'MaxLikefixed 50%')
maxlike_fixed_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.
↪1), mode='lines', name='MaxLikeFixed 10%')
maxlike_fixed_fig.add_scatter(x=cycles, y=wc.basquin_load(cycles, failure_probability=0.
↪9), mode='lines', name='MaxLikeFixed 90%')

maxlike_fixed_fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[ ]:

5.4 Hotspot calculation demo

This notebook detects and classifies hotspots of the von Mises stress in a connected FEM mesh. Each element/node entry of the mesh receives a number of the hotspot it is member of. “0” means the element/node is not part of any hotspots. “1” means that the element/node is part of the hotspot with the highes peak, “2” the same for the second highest peak and so forth.

See module documentation further details.

[1]:
```
import numpy as np
import pylife
import pandas as pd
import scipy.stats as stats
import pylife.stress.equistress
import pylife.strength.meanstress
import pylife.mesh.meshsignal
import pylife.mesh.hotspot
import pylife.vmap
```

(continues on next page)

(continued from previous page)

```
import pyvista as pv

pv.set_plot_theme('document')
pv.set_jupyter_backend('panel')
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

```
[2]: vm_mesh = pylife.vmap.VMAPImport("plate_with_hole.vmap")
     mesh = (vm_mesh.make_mesh('1', 'STATE-2')
             .join_coordinates()
             .join_variable('STRESS_CAUCHY')
             .join_variable('DISPLACEMENT')
             .to_frame())
```

5.4.1 Equivalent stress calculation

```
[3]: mesh['mises'] = mesh.equistress.mises()
```

5.4.2 Hot spot Calculation

```
[4]: threshold = .9 # factor of the maximum local value
     mesh['hotspot'] = mesh.hotspot.calc("mises", threshold)
     display(mesh[['x', 'y', 'z', 'mises', 'hotspot']])
```

```
                                x         y    z      mises  hotspot
element_id node_id
1          1734     14.897208  5.269875  0.0  33.996987        0
           1582     14.555333  5.355806  0.0  33.399850        0
           1596     14.630658  4.908741  0.0  54.777007        0
           4923     14.726271  5.312840  0.0  33.446991        0
           4924     14.592996  5.132274  0.0  44.070962        0
...                       ...       ...  ...        ...      ...
4770       3812    -13.189782 -5.691876  0.0  43.577209        0
           12418   -13.560289 -5.278386  0.0  39.903508        0
           14446   -13.673285 -5.569107  0.0  40.478974        0
           14614   -13.389065 -5.709927  0.0  42.140169        0
           14534   -13.276068 -5.419206  0.0  41.143837        0

[37884 rows x 5 columns]
```

```
[5]: print("%d hotspots found over the threshold" % mesh['hotspot'].max())
```

```
3 hotspots found over the threshold
```

```
[6]: grid = pv.UnstructuredGrid(*mesh.mesh.vtk_data())
     plotter = pv.Plotter(window_size=[1920, 1080])
     plotter.add_mesh(grid, scalars=mesh.groupby('element_id')['hotspot'].first().to_numpy(),
                      show_edges=True, cmap='prism_r')
     plotter.add_scalar_bar()
     plotter.show()
```

First hotspot

```
[7]: first_hotspot = mesh[mesh['hotspot'] == 1]
     display(first_hotspot)
```

```
                        x    y    z         S11        S22  S33       S12  S13  \
element_id node_id
456        5          0.0  6.3  0.0  300.899658  30.574533  0.0 -7.081042  0.0

                      S23           dx        dy   dz       mises  hotspot
element_id node_id
456        5          0.0  1.365477e-35 -0.005435  0.0  287.099222        1
```

```
[8]: second_hotspot = mesh[mesh['hotspot'] == 2]
     display(second_hotspot)
```

```
                        x    y    z         S11        S22  S33       S12  S13  \
element_id node_id
2852       9          0.0 -6.3  0.0  284.085327  30.704277  0.0  3.546472  0.0

                      S23           dx        dy   dz       mises  hotspot
element_id node_id
2852       9          0.0  1.400164e-35  0.005436  0.0  270.115389        2
```

```
[ ]:
```

```
[ ]:
```

5.5 Stress gradient calculation

This demo shows the stress gradient calculation module. A gradient is calculated by fitting a plane into a node and its neighbor nodes of an FEM mesh.

See documentation for details on the module.

```
[1]: import numpy as np
     import pandas as pd

     import pylife.stress.equistress
     import pylife.mesh.gradient
     import pylife.vmap
```

(continues on next page)

(continued from previous page)

```
import pyvista as pv

pv.set_plot_theme('document')
pv.set_jupyter_backend('panel')
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
```

```
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
```

Read in demo data and add the stress tensor dimensions for the third dimension.

```
[2]: vm_mesh = pylife.vmap.VMAPImport("plate_with_hole.vmap")
     pyLife_mesh = (vm_mesh.make_mesh('1', 'STATE-2')
                    .join_coordinates()
                    .join_variable('STRESS_CAUCHY')
                    .to_frame())
```

Calculate and add von Mises stress

```
[3]: pyLife_mesh['mises'] = pyLife_mesh.equistress.mises()
```

Calculate stress gradient on von Mises stress

```
[4]: grad = pyLife_mesh.gradient.gradient_of('mises')
```

```
[5]: grad["abs_grad"] = np.linalg.norm(grad, axis = 1)
     pyLife_mesh = pyLife_mesh.join(grad)
     display(pyLife_mesh)
```

```
                            x         y    z        S11       S22  S33  \
element_id node_id
1          1734     14.897208  5.269875  0.0  27.080811  6.927080  0.0
           1582     14.555333  5.355806  0.0  28.319006  1.178649  0.0
           1596     14.630658  4.908741  0.0  47.701195  5.512213  0.0
           4923     14.726271  5.312840  0.0  27.699907  4.052865  0.0
           4924     14.592996  5.132274  0.0  38.010101  3.345431  0.0
...                       ...       ...  ...        ...       ...  ...
4770       3812    -13.189782 -5.691876  0.0  36.527439  2.470588  0.0
           12418   -13.560289 -5.278386  0.0  32.868889  3.320898  0.0
           14446   -13.673285 -5.569107  0.0  34.291058  3.642457  0.0
           14614   -13.389065 -5.709927  0.0  36.063541  2.828889  0.0
           14534   -13.276068 -5.419206  0.0  33.804211  2.829817  0.0

                          S12  S13  S23      mises  dmises_dx  dmises_dy  \
element_id node_id
1          1734    -13.687358  0.0  0.0  33.996987  -0.693487  -1.256807
           1582    -10.732705  0.0  0.0  33.399850  -3.300877  -0.779737
           1596    -17.866833  0.0  0.0  54.777007  -4.654205   1.511868
           4923    -12.210032  0.0  0.0  33.446991  -1.438945  -4.702223
```

(continues on next page)

(continued from previous page)

```
            4924    -14.299768  0.0  0.0  44.070962  35.610803 -51.987913
...                        ...  ...  ...        ...        ...        ...
4770        3812    -14.706686  0.0  0.0  43.577209  -0.154659  -6.003801
            12418   -14.260107  0.0  0.0  39.903508   0.174451  -3.892492
            14446   -13.836027  0.0  0.0  40.478974   2.091836   4.804640
            14614   -13.774759  0.0  0.0  42.140169  -0.540695  -6.376289
            14534   -14.580153  0.0  0.0  41.143837  -0.141019  -4.978514

                     dmises_dz   abs_grad
element_id node_id
1           1734           0.0   1.435440
            1582           0.0   3.391722
            1596           0.0   4.893605
            4923           0.0   4.917465
            4924           0.0  63.014858
...                        ...        ...
4770        3812           0.0   6.005793
            12418          0.0   3.896399
            14446          0.0   5.240262
            14614          0.0   6.399172
            14534          0.0   4.980511

[37884 rows x 14 columns]
```

[6]:
```
grid = pv.UnstructuredGrid(*pyLife_mesh.mesh.vtk_data())
plotter = pv.Plotter(window_size=[1920, 1080])
plotter.add_mesh(grid, scalars=pyLife_mesh.groupby('element_id')["abs_grad"].mean().to_
↪numpy(),
                show_edges=True, cmap='jet')
plotter.add_scalar_bar()
plotter.show()
```

5.6 Local stress approach

5.6.1 FE based failure probability calculation

FE Data

we are using VMAP data format and rst file formats. It is also possible to use odb data,

[1]:
```
import numpy as np
import pandas as pd
import pickle
import pylife.vmap
import pylife.mesh
import pylife.mesh.meshsignal
import pylife.stress.equistress
import pylife.stress
```

(continues on next page)

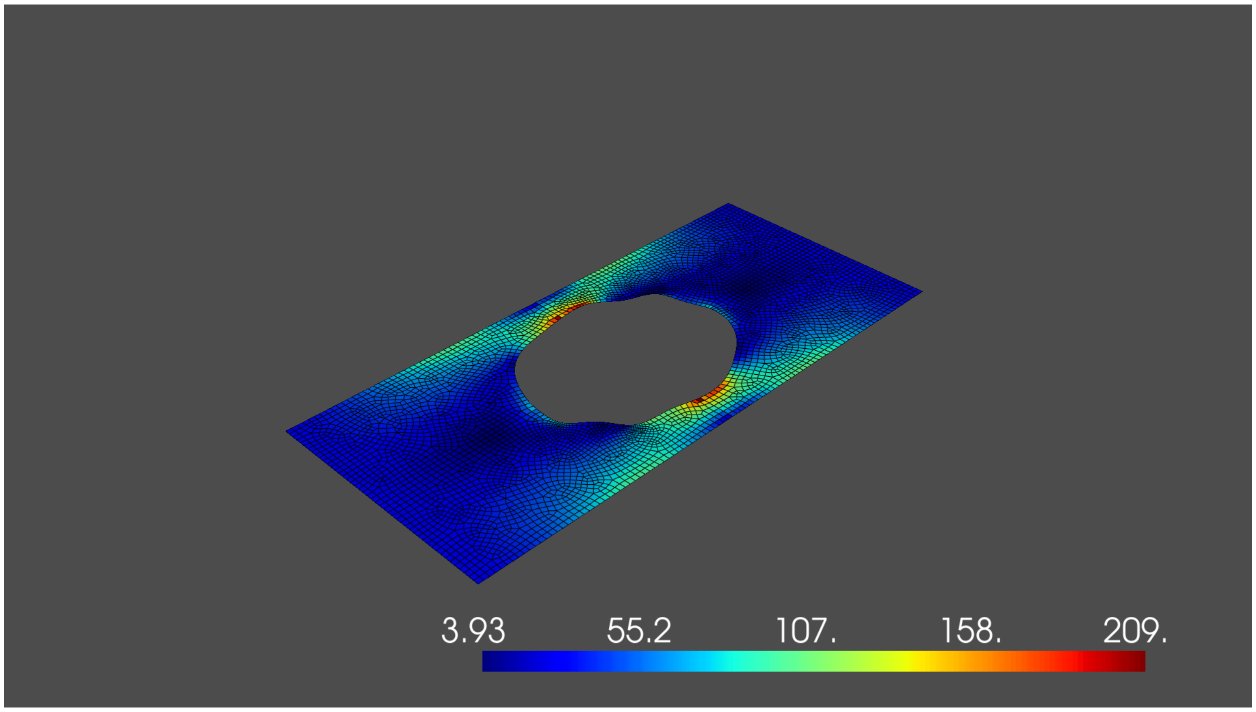
(continued from previous page)

```
import pylife.strength.fatigue
import pylife.utils.histogram as psh
import pyvista as pv

# from ansys.dpf import post
# pv.set_plot_theme('document')
# pv.set_jupyter_backend('ipyvtklink')
# get_ipython().run_line_magic('matplotlib', 'inline')
```

5.6.2 VMAP

For plotting of VMAP data we are using pyVista.

```
[2]: pyLife_mesh = (pylife.vmap.VMAPImport("plate_with_hole.vmap").make_mesh('1', 'STATE-2')
                    .join_coordinates()
                    .join_variable('STRESS_CAUCHY')
                    .to_frame())

     pyLife_mesh['mises'] = pyLife_mesh.equistress.mises()
     grid = pv.UnstructuredGrid(*pyLife_mesh.mesh.vtk_data())
     plotter = pv.Plotter(window_size=[1920, 1080])
     plotter.add_mesh(grid, scalars=pyLife_mesh.groupby('element_id')['mises'].mean().to_
     ↪numpy(),
                     show_edges=True, cmap='jet')
     plotter.add_scalar_bar()
     plotter.show()
```



Now we want to apply the collectives to the mesh

```
[3]: mises = pyLife_mesh.groupby('element_id')['S11', 'S22', 'S33', 'S12', 'S13', 'S23'].
     ↪mean().equistress.mises()
     mises /= mises.max()  # the nominal load level in the FEM analysis is set, that s_max = 1
     collectives = pickle.load(open("collectives.p", "rb"))
     collectives = collectives.unstack().T.fillna(0)
     collectives_sorted = psh.combine_histogram([collectives[col] for col in collectives],
                                             method="sum")

     scaled_collectives = collectives_sorted.load_collective.scale(mises)
     display(scaled_collectives.to_pandas().sample(5))
```

```
range                                      element_id
(13.809263752657428, 20.71389562898614]    2915             16.0
(75.51855116742843, 94.39818895928553]     602             918.0
(67.96887403296256, 76.46498328708287]     2604            360.0
(54.517686917879075, 72.69024922383876]    758           13940.0
(14.54541186824786, 29.09082373649572]     381               0.0
Name: cycles, dtype: float64
```

Define the material parameters

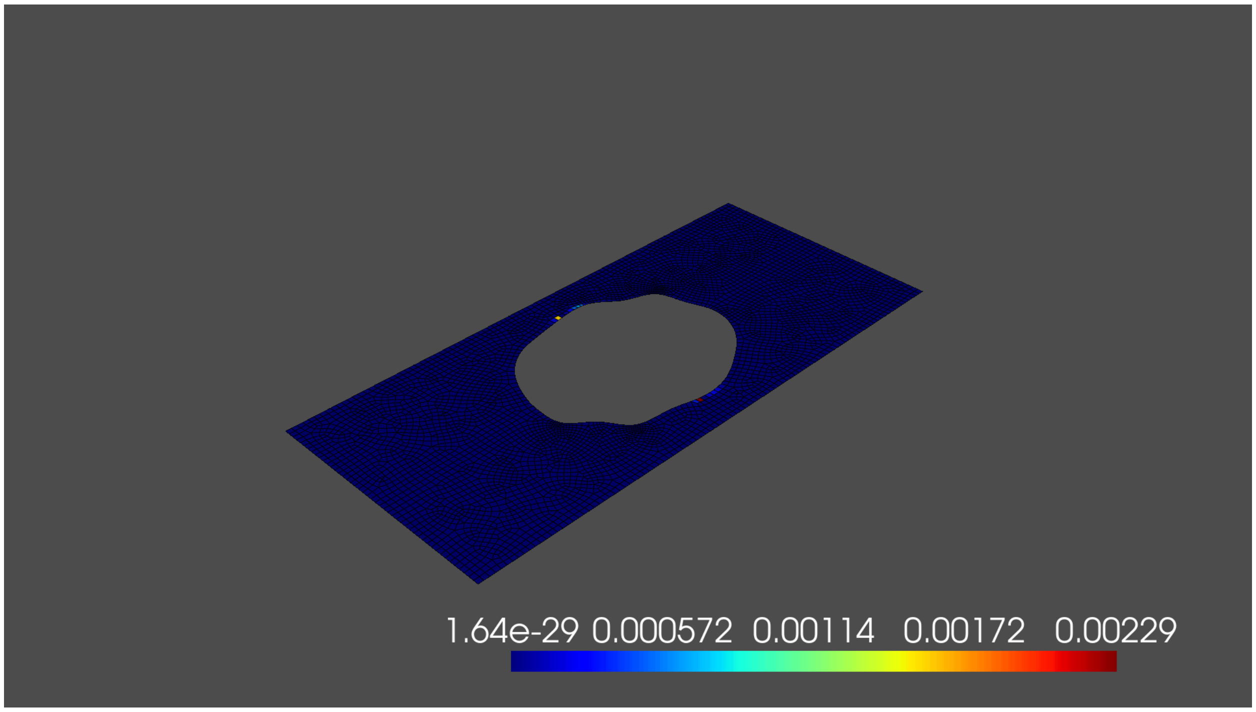
```
[4]: mat = pd.Series({
         'k_1': 8.,
         'ND': 1.0e6,
         'SD': 200.0, # range
         'TN': 1./12.,
         'TS': 1./1.1
     })
```

Damage Calculation

```
[5]: damage = mat.fatigue.miner_haibach().damage(scaled_collectives)
     print("Max damage : %f" % damage.max())
     damage = damage.groupby(['element_id']).sum()
```

```
Max damage : 0.001484
```

```
[6]: grid = pv.UnstructuredGrid(*pyLife_mesh.mesh.vtk_data())
     plotter = pv.Plotter(window_size=[1920, 1080])
     plotter.add_mesh(grid, scalars=damage.to_numpy(),
                     show_edges=True, cmap='jet')
     plotter.add_scalar_bar()
     plotter.show()
```



5.6.3 ANSYS

[7]:
```
#%% Ansys (license is necessary)
# For Ansys  *.rst files we are using pymapdl
# from ansys.mapdl import reader as pymapdl_reader
# # for more information please go to pymapdl
# # rst_input = post.load_solution("beam_3d.rst")
# # # pymapdl has some nice features
# # rst_input.plot_nodal_displacement(0)
# # rst_input.plot_nodal_stress(0,"X")
# ansys_mesh = pymapdl_reader.read_binary('beam_3d.rst')
# grid_ansys = ansys_mesh.grid
# plotter = pv.Plotter(window_size=[1920, 1080])
# _, volume, _  = ansys_mesh.element_solution_data(0,"ENG")
# volume = pd.DataFrame(volume)[1]

# nodes, ansys_mesh_mises = ansys_mesh.nodal_stress(0)
# ansys_mesh_mises = pd.DataFrame(data = ansys_mesh_mises,
#                                  columns=['S11', 'S22', 'S33', 'S12', 'S13', 'S23']).equistress.
↪mises()


# test = pd.DataFrame(ansys_mesh.mesh.elem).iloc[:, 8:]
# #%%
# plotter.add_mesh(grid_ansys, scalars=ansys_mesh_mises,
#                  show_edges=True, cmap='jet')
# plotter.add_scalar_bar()
# plotter.show()
```

5.7 PSD Optimizer

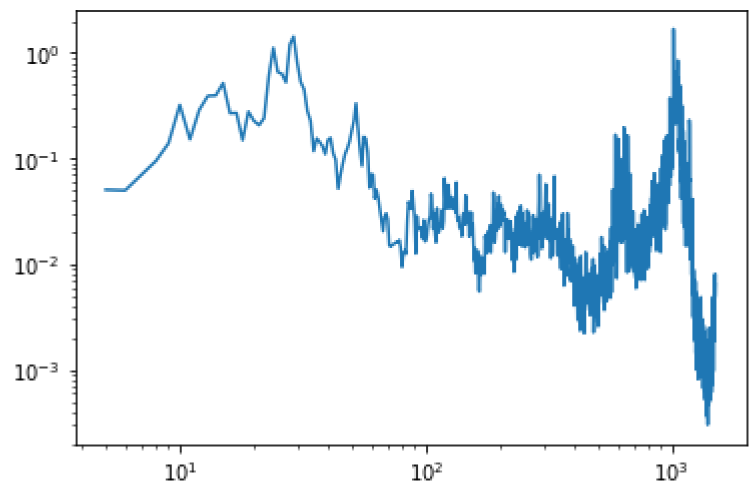
Example for the derivation of an equivalent PSD signal for shaker testing or other purposes. See the docu of the pylife psd_smoother function in the psdSignal class for more details

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from scipy import optimize as op
import sys
from pylife.stress.frequencysignal import psdSignal
import ipywidgets as wd
%matplotlib inline
```

[2]:
```
psd = pd.DataFrame(pd.read_csv("data/PSD_values.csv",index_col = 0).iloc[5:1500,0])
```

[3]:
```
#fsel = np.array([30,50,80,460,605,1000])
fsel = []
fig = plt.figure()
plt.loglog(psd)
txt = wd.Textarea(
    value='',
    placeholder='',
    description='fsel = ',
    disabled=False
)
display(txt)
def onclick(event):
    yval = psd.values[psd.index.get_loc(event.xdata,method ='nearest')]
    plt.vlines(event.xdata,0,yval)
    fsel.append(event.xdata)
    txt.value = "{:.2f}".format(event.xdata)
cid = fig.canvas.mpl_connect('button_press_event', onclick)
```

```
Textarea(value='', description='fsel = ', placeholder='')
```

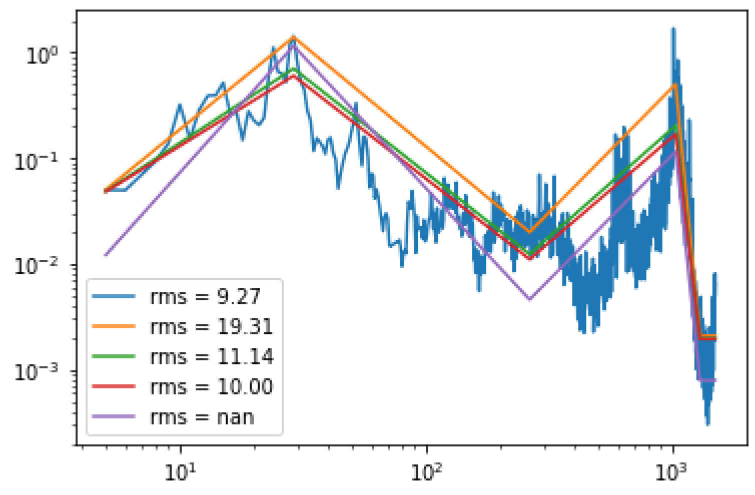


[4]:
```
fsel = np.asarray(fsel)
# please uncomment the following line. It is just for utomatization purpose
fsel = np.asarray([5,29,264,1035,1300,])
print(fsel)
```

```
[   5   29  264 1035 1300]
```

[5]:
```
def rms_psd(psd):
    return (sum(((psd.diff()+psd).dropna().values.flatten()*np.diff(psd.index.
↪values)))**0.5)
plt.loglog(psd, label="rms = {:.2f}".format(rms_psd(psd)))
for i in np.linspace(0,1,4):
    psd_fit = psdSignal.psd_smoother(psd, fsel, i)
    plt.loglog(psd_fit, label="rms = {:.2f}".format(rms_psd(psd_fit)))
plt.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7f1efc615490>
```



[ ]:

5.8 Time series handling

demos/dash-apps/pyLife_logo_20200219_FINAL_RGB.png

This Notebook shows a general calculation stream for time series. You will see how to * read in time series * plot the data in time and frequency domain * filter the time series with a bandpass filter * remove spikes using running statistics * calculate and plot the rainflow matrices of the time series * combine the PSD to an envelope PSD.

If you have any question feel free to contact us.

[1]:
```
import numpy as np
import pandas as pd

import pylife.utils.histogram as psh
import pylife.stress.timesignal as ts
import pylife.stress.rainflow as RF
import pylife.stress.rainflow.recorders as RFR
import pickle

import pyvista as pv

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib as mpl

from scipy import signal as sg

# mpl.style.use('seaborn')
mpl.style.use('bmh')
get_ipython().run_line_magic('matplotlib', 'inline')
```

some functionality to plot the rainflow matrices

```
[2]: from helper_functions import plot_rf
```

5.8.1 Time series signal

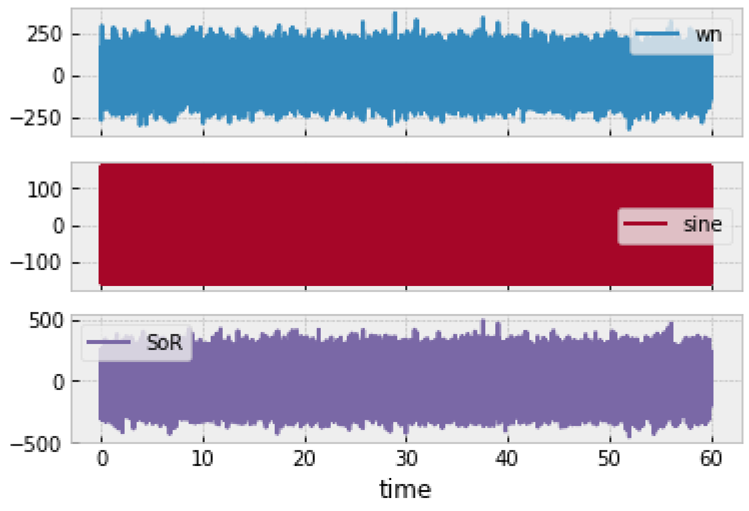
import, filtering and so on. You can import your own signal with

- pd.read_csv()
- pd.read_excel()
- scipy.io.loadmat() for matlab files

and so on. Here we define a white noise, a sine and a sine on random signal.

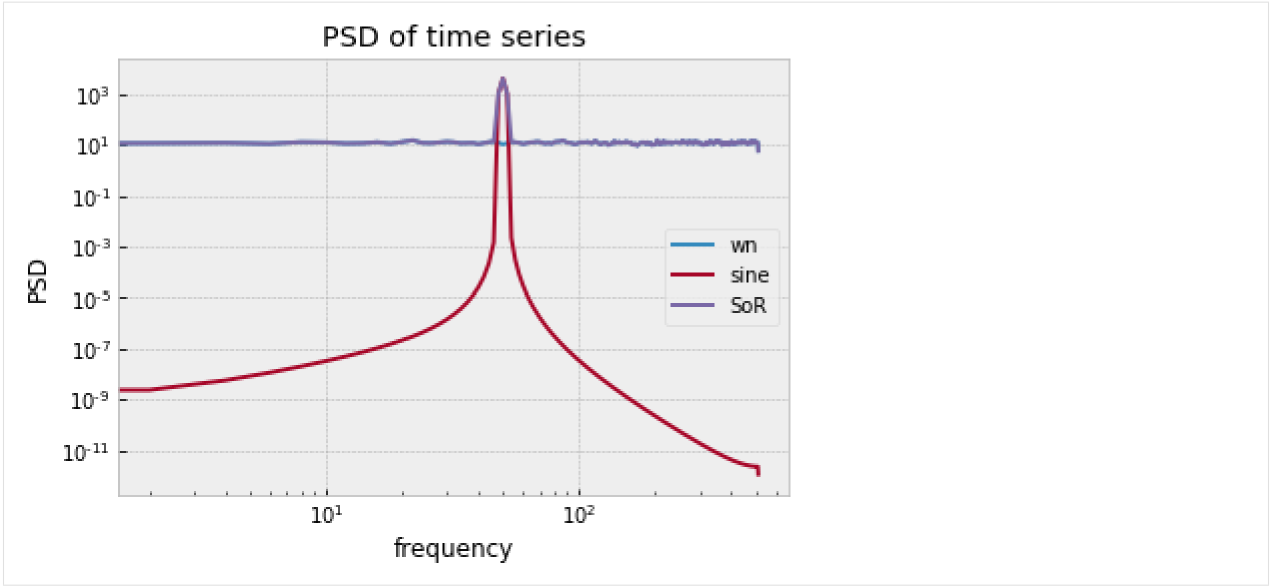
```
[3]: np.random.seed(4711)
     sample_frequency = 1024
     t = np.linspace(0, 60, 60 * sample_frequency)
     signal_df = pd.DataFrame(data = np.array([80 * np.random.randn(len(t)),
                                               160 * np.sin(2 * np.pi * 50 * t)]).T,
                              columns=["wn", "sine"],
                              index=pd.Index(t, name="time"))
     signal_df["SoR"] = signal_df["wn"] + signal_df["sine"]
     signal_df.plot(subplots=True)
```

```
[3]: array([<AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
            <AxesSubplot:xlabel='time'>], dtype=object)
```



```
[4]: ts.psd_df(signal_df, NFFT = 512).plot(loglog=True, ylabel="PSD", title="PSD of time␣
     ↪series")
```

```
[4]: <AxesSubplot:title={'center':'PSD of time series'}, xlabel='frequency', ylabel='PSD'>
```



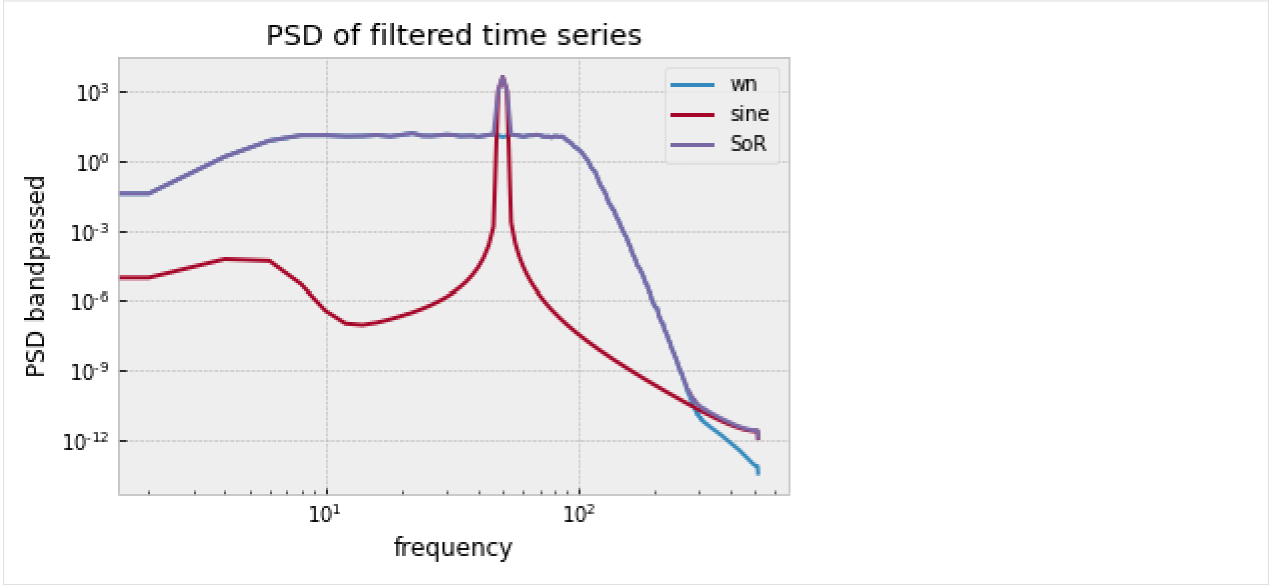
5.8.2 Filtering

We are using a butterworth bandpass filter from scipy.signal to filter the time series.

```
[5]: f_min = 5.0     # Hz
     f_max = 100.0   # Hz
```

```
[6]: bandpass_df = ts.butter_bandpass(signal_df, f_min, f_max)

     df_psd = ts.psd_df(bandpass_df, NFFT = 512)
     df_psd.plot(loglog=True, ylabel="PSD bandpassed", title="PSD of filtered time series")
```

```
[6]: <AxesSubplot:title={'center':'PSD of filtered time series'}, xlabel='frequency', ylabel=
     ↪'PSD bandpassed'>
```

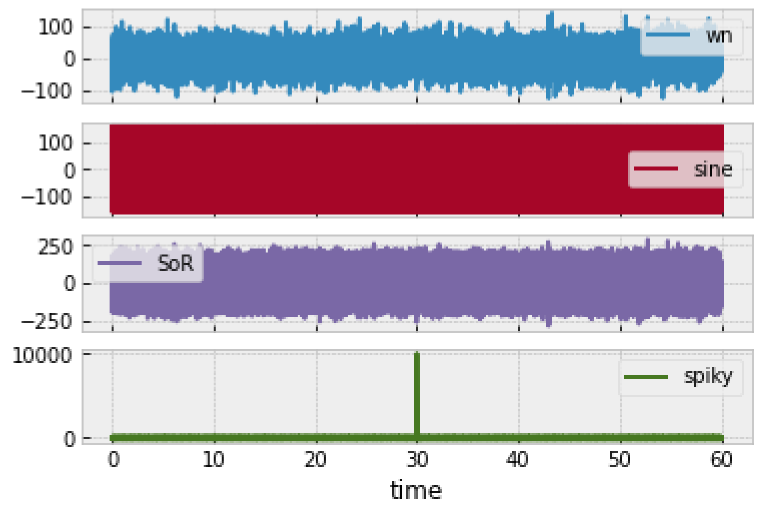


5.8.3 Running statistics

First we create a spike in our existing data set

```
[7]: bandpass_df["spiky"] = bandpass_df["sine"] + 1e4 * sg.unit_impulse(signal_df.shape[0],␣
     ↪idx="mid")
     bandpass_df.plot(subplots=True)
```

```
[7]: array([<AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
            <AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>],
           dtype=object)
```

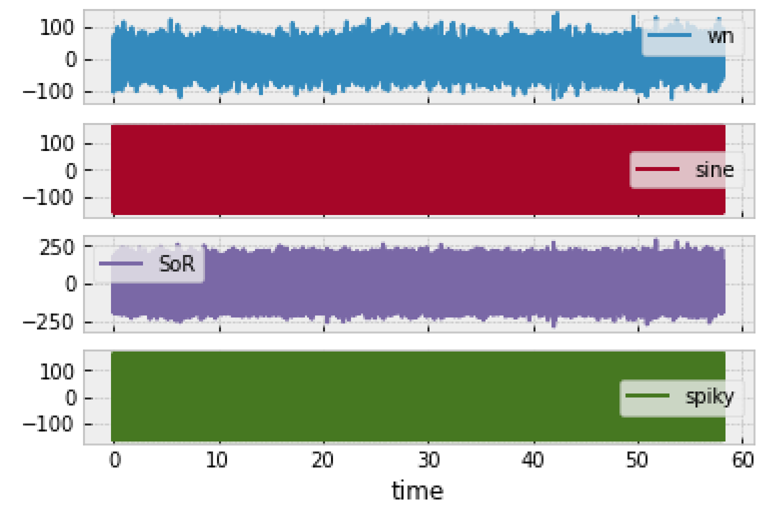


Now we want to clean this spike automatically

```
[8]: cleaned_df = ts.clean_timeseries(bandpass_df, "spiky", window_size=1024, overlap=32,
                                      feature="maximum", method="remove", n_gridpoints=3,
                                      percentage_max=0.05, order=3).drop(["time"], axis=1)

     cleaned_df.plot(subplots=True)
```

```
Feature Extraction: 100%|| 244/244 [00:00<00:00, 6203.20it/s]
```

```
[8]: array([<AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
            <AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>],
           dtype=object)
```



5.8.4 Rainflow

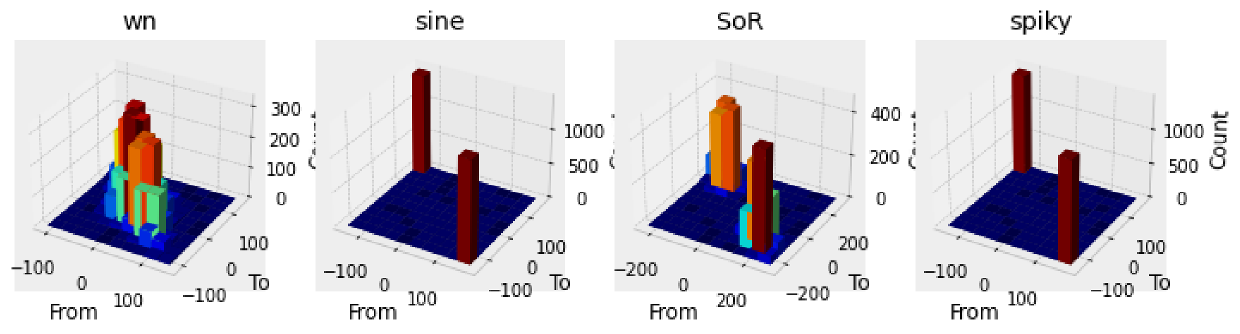
The rainflow module in pyLife can be used with different counting methods:

- FKM
- Three point
- Four point enhanced

[9]:
```
rainflow_bins = 10
```

[10]:
```
#%% Rainflow for a multiple time series
recorder_dict = {key: RFR.FullRecorder() for key in cleaned_df}
detector_dict = {key: RF.FKMDetector(recorder=recorder_dict[key]).process(cleaned_
↪df[key]) for key in cleaned_df}
rf_series_dict = {key: detector_dict[key].recorder.histogram(rainflow_bins) for  key in␣
↪detector_dict.keys()}
```

[11]:
```
f = plot_rf(rf_series_dict)
```

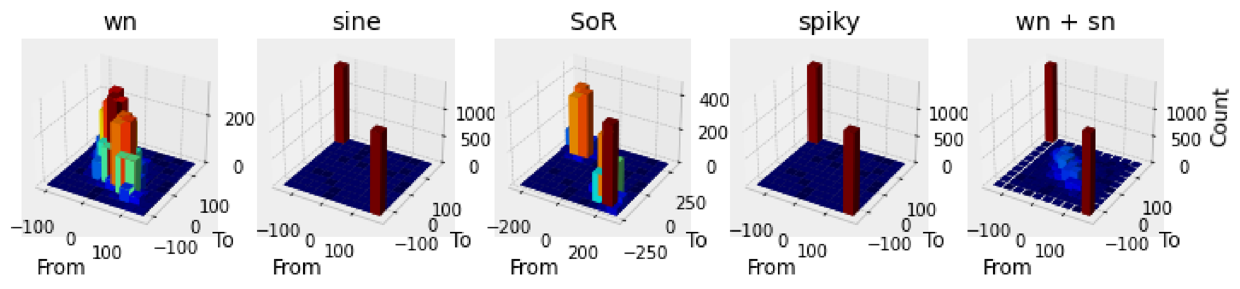


[12]:
```
#%% Now Combining different RFs to one
rf_series_dict["wn + sn"] = psh.combine_histogram([rf_series_dict["wn"],rf_series_dict[
↪"sine"]],
```

(continues on next page)

(continued from previous page)

```
                                                  method="sum")
f = plot_rf(rf_series_dict)
```



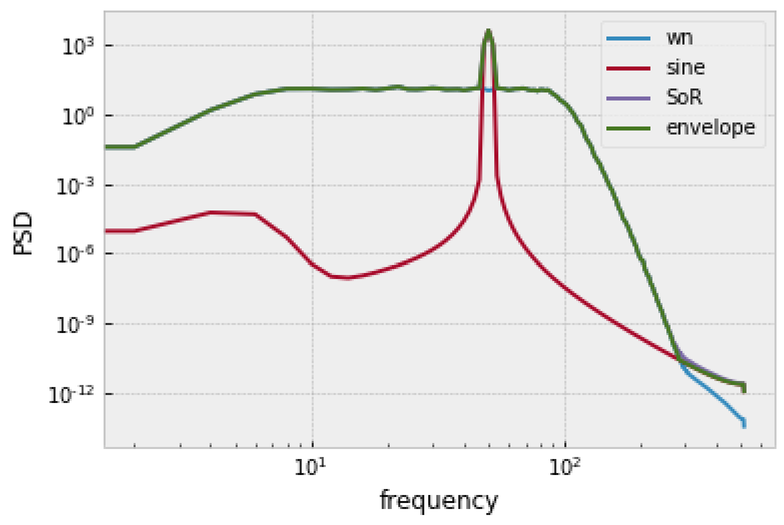
You can see the difference of the rainflow matrices of *SoR* and *wn+sn*.

5.8.5 PSD combinig

It is also possible to combine spectra

```
[13]: df_psd["envelope"] =  df_psd[["sine", "wn"]].max(axis = 1)
      df_psd.plot(loglog=True, ylabel="PSD")
```

```
[13]: <AxesSubplot:xlabel='frequency', ylabel='PSD'>
```



5.8.6 Saving

Now we saving the rainflow data into a pickle file. If you want to have an introduction to damage and failure probability calculation, please have a look on the notebook lifetime_calc

```
[14]: rf_out = {k: rf_series_dict[k] for k in ["wn", "sine", "SoR"] if k in rf_series_dict}
      pickle.dump(rf_out, open("rf_dict.p", "wb"))
```

CHAPTER

SIX

PYLIFE REFERENCE

6.1 General

6.1.1 pyLife core

class pylife.**PylifeSignal**(*pandas_obj*)

Base class for signal accessor classes.

Notes

Derived classes need to implement the method *_validate(self, obj)* that gets *pandas_obj* as *obj* parameter. This *validate()* method must raise an Exception (e.g. AttributeError or ValueError) in case *obj* is not a valid DataFrame for the kind of signal.

For these validation `fail_if_key_missing()` and `get_missing_keys()` might be helpful.

For a derived class you can register methods without modifying the class' code itself. This can be useful if you want to make signal accessor classes extendable.

See also:

`fail_if_key_missing()` `get_missing_keys()` `register_method()`

fail_if_key_missing(*keys_to_check*, *msg=None*)

Raise an exception if any key is missing in a self._obj object.

Parameters

- **self._obj** (*pandas.DataFrame or pandas.Series*) – The object to be checked
- **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Raises

- **AttributeError** – if *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*
- **AttributeError** – if any of the keys is not found in the self._obj's keys.

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* meed to be found in the *self._obj.columns*.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* meed to be found in the *self._obj.index*.

See also:

get_missing_keys(), `stresssignal.StressTensorVoigt`

classmethod from_parameters(***kwargs*)

Make a signal instance from a parameter set.

This is a convenience function to instantiate a signal from individual parameters rather than pandas objects.

A signal class like

```
@pd.api.extensions.register_dataframe_accessor('foo_signal')
class FooSignal(PylifeSignal):
    pass
```

The following two blocks are equivalent:

```
pd.Series({'foo': 1.0, 'bar': 2.0}).foo_signal
```

```
FooSignal.from_parameters(foo=1.0, bar=1.0)
```

get_missing_keys(*keys_to_check*)

Get a list of missing keys that are needed for a self._obj object.

Parameters **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Returns **missing_keys** – a list of missing keys

Return type list

Raises `AttributeError` – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* not found in the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* not found in the *self._obj.index* are returned.

keys()

Get a list of missing keys that are needed for a signal object.

Returns **keys** – a pandas index of keys

Return type pd.Index

Raises `AttributeError` – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, the *self._obj.index* are returned.

to_pandas()

Expose the pandas object of the signal.

Returns **pandas_object** – The pandas object representing the signal

Return type pd.DataFrame or pd.Series

Notes

The default implementation just returns the object given when instantiating the signal class. Derived classes may return a modified object or augmented, if they store some extra information.

By default the object is **not** copied. So make a copy yourself, if you intent to modify it.

class pylife.**Broadcaster**(*pandas_obj*)

The Broadcaster to align pyLife signals to operands.

Parameters

- **pandas_obj** (`pandas.Series` or `pandas.DataFrame`) – the object of the `Broadcaster`
- **The** (*In most cases the Broadcaster class is not used directly.*) –
- **class** (*functionality is in most cases used by the derived*) –

:param *pylife.PylifeSignal*.: :param The purpose of the `Broadcaster` is to take two numerical objects and: :param return two objects of the same numerical data with an aligned index. That: :param means that mathematical operations using the two objects as operands can be: :param implemented using numpy's broadcasting functionality.: :param See method **:method:`pylife.Broadcaster.broadcast`** documentation for details.: :param The broadcasting is done in the following ways:

```
object                  parameter               returned object         returned␣
↪parameter

Series                  Scalar                  Series                  Scalar
|------|-----|                                  |------|-----|
| idx  |     |                                  | idx  |     |
|------|-----|          5.0                 ->  |------|-----|          5.0
| foo  | 1.0 |                                  | foo  | 1.0 |
| bar  | 2.0 |                                  | bar  | 2.0 |
|------|-----|                                  |------|-----|


DataFrame               Scalar                  DataFrame               Series
|------|-----|-----|                            |------|-----|-----|    |------|-----|
| idx  | foo | bar |                            | idx  | foo | bar |    | idx  |     |
|------|-----|-----|                            |------|-----|-----|    |------|-----|
| 0    | 1.0 | 2.0 |    5.0                 ->  | 0    | 1.0 | 2.0 |    | 0    | 5.0 |
| 1    | 1.0 | 2.0 |                            | 1    | 1.0 | 2.0 |    | 1    | 5.0 |
| ...  | ... | ... |                            | ...  | ... | ... |    | ...  | ... |
|------|-----|-----|                            |------|-----|-----|    |------|-----|
```

(continues on next page)

(continued from previous page)

```
Series                    Series/DataFrame          DataFrame                 Series/
↪DataFrame
|------|-----|            |------|-----|            |------|-----|-----|      |------|-----|
| None |     |            | idx  |     |            | idx  | foo | bar |      | idx  |     |
|------|-----|            |------|-----|    ->      |------|-----|-----|      |------|-----|
| foo  | 1.0 |            | 0    | 5.0 |            | 0    | 1.0 | 2.0 |      | 0    | 5.0 |
| bar  | 2.0 |            | 1    | 6.0 |            | 1    | 1.0 | 2.0 |      | 1    | 6.0 |
|------|-----|            | ...  | ... |            | ...  | ... | ... |      | ...  | ... |
                          |------|-----|            |------|-----|-----|      |------|-----|


Series/DataFrame          Series/DataFrame          Series/DataFrame          Series/
↪DataFrame
|------|-----|            |------|-----|            |------|-----|            |------|-----|
| xidx |     |            | xidx |     |            | xidx |     |            | xidx |     |
|------|-----|            |------|-----|    ->      |------|-----|            |------|-----|
| foo  | 1.0 |            | tau  | 5.0 |            | foo  | 1.0 |            | foo  | nan |
| bar  | 2.0 |            | bar  | 6.0 |            | bar  | 2.0 |            | bar  | 6.0 |
|------|-----|            |------|-----|            | tau  | nan |            | tau  | 5.0 |
                                                    |------|-----|            |------|-----|


Series/DataFrame          Series/DataFrame          Series/DataFrame          Series/
↪DataFrame
|------|-----|            |------|-----|            |------|------|-----|     |------|------
↪|-----|
| xidx |     |            | yidx |     |            | xidx | yidx |     |     | xidx | yidx␣
↪|     |
|------|-----|            |------|-----|   ->       |------|------|-----|     |------|------
↪|-----|
| foo  | 1.0 |            | tau  | 5.0 |            | foo  | tau  | 1.0 |     | foo  | tau ␣
↪| 5.0 |
| bar  | 2.0 |            | chi  | 6.0 |            |      | chi  | 1.0 |     |      | chi ␣
↪| 6.0 |
|------|-----|            |------|-----|            | bar  | tau  | 2.0 |     | bar  | tau ␣
↪| 5.0 |
                                                    |      | chi  | 2.0 |     |      | chi ␣
↪| 6.0 |
                                                    |------|------|-----|     |------|------
↪|-----|
```

broadcast(*parameter*, *droplevel=[]*)

Broadcast the parameter to the object of `self`.

Parameters **parameters** (*scalar, numpy array or pandas object*) – The parameter to broadcast to

Returns **parameter, object**

Return type index aligned numerical objects

The

Examples

The behavior of the Broadcaster is best illustrated by examples:

- Broadcasting `pandas.Series` to a scalar results in a scalar and a `pandas.Series`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['foo', 'bar'], name='idx'))
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
array(5.)
```

```
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

- Broadcasting `pandas.DataFrame` to a scalar results in a `pandas.DataFrame` and a `pandas.Series`.

```
obj = pd.DataFrame({
    'foo': [1.0, 2.0],
    'bar': [3.0, 4.0]
}, index=pd.Index([1, 2], name='idx'))
obj
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
idx
1    5.0
2    5.0
dtype: float64
```

```
obj
```

- Broadcasting `pandas.DataFrame` to a a `pandas.Series` results in a `pandas.DataFrame` and a `pandas.Series`, **if and only if** the index name of the object is `None`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['tau', 'chi']))
obj
```

```
tau    1.0
chi    2.0
dtype: float64
```

```
parameter = pd.Series([3.0, 4.0], index=pd.Index(['foo', 'bar'], name='idx
↪'))
parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(parameter)

parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
obj
```

class pylife.**DataValidator**

fail_if_key_missing(*signal*, *keys_to_check*, *msg=None*)

Raise an exception if any key is missing in a signal object.

Parameters

- **signal** (*pandas.DataFrame or pandas.Series*) – The object to be checked
- **keys_to_check** (*list*) – A list of keys that need to be available in *signal*

Raises

- **AttributeError** – if *signal* is neither a *pandas.DataFrame* nor a *pandas.Series*
- **AttributeError** – if any of the keys is not found in the signal's keys.

Notes

If *signal* is a *pandas.DataFrame*, all keys of *keys_to_check* meed to be found in the *signal.columns*.

If *signal* is a *pandas.Series*, all keys of *keys_to_check* meed to be found in the *signal.index*.

See also:

`signal.get_missing_keys()`, `stresssignal.StressTensorVoigt`

get_missing_keys(*signal*, *keys_to_check*)

Get a list of missing keys that are needed for a signal object.

Parameters

- **signal** (*pandas.DataFrame or pandas.Series*) – The object to be checked

- **keys_to_check** (*list*) – A list of keys that need to be available in *signal*

Returns **missing_keys** – a list of missing keys

Return type list

Raises `AttributeError` – If *signal* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *signal* is a *pandas.DataFrame*, all keys of *keys_to_check* not found in the *signal.columns* are returned.

If *signal* is a *pandas.Series*, all keys of *keys_to_check* not found in the *signal.index* are returned.

keys(*signal*)

Get a list of missing keys that are needed for a signal object.

Parameters **signal** (*pandas.DataFrame or pandas.Series*) – The object to be checked

Returns **keys** – a pandas index of keys

Return type pd.Index

Raises `AttributeError` – If *signal* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *signal* is a *pandas.DataFrame*, the *signal.columns* are returned.

If *signal* is a *pandas.Series*, the *signal.index* are returned.

6.2 Stress

6.2.1 The pyLife stress subpackage

The `stress` subpackage contains all the pyLife modules that deal with stress resp. load analysis.

There are the two central classes `LoadCollective` and `LoadHistogram` to describe load collectives.

6.2.2 The `equistress` module

Equivalent Stresses

Library to calculate the equivalent stress values of a FEM stress tensor.

By now the following calculation methods are implemented:

- Principal stresses
- Maximum principal stress
- Minimum principal stress
- Absolute maximum principal stress
- Von Mises
- Signed von Mises, sign from trace

- Signed von Mises, sign from absolute maximum principal stress
- Tresca
- Signed Tresca, sign from trace
- Signed Tresca, sign from absolute maximum principal stress

`class pylife.stress.equistress.`**`StressTensorEquistress`**(*pandas_obj*)

  `abs_max_principal``()`

  `max_principal``()`

  `min_principal``()`

  `mises``()`

  `principals``()`

  `signed_mises_abs_max_principal``()`

  `signed_mises_trace``()`

  `signed_tresca_abs_max_principal``()`

  `signed_tresca_trace``()`

  `tresca``()`

`pylife.stress.equistress.`**`abs_max_principal`**(*s11*, *s22*, *s33*, *s12*, *s13*, *s23*)

Calculate absolute maximum principal stress (maximum of absolute eigenvalues with corresponding sign).

Parameters

- **s11** (*array_like*) – Component 11 of 3D tensor.
- **s22** (*array_like*) – Component 22 of 3D tensor.
- **s33** (*array_like*) – Component 33 of 3D tensor.
- **s12** (*array_like*) – Component 12 of 3D tensor.
- **s13** (*array_like*) – Component 13 of 3D tensor.
- **s23** (*array_like*) – Component 23 of 3D tensor.

Returns Absolute maximum principal stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`eigenval`**(*s11*, *s22*, *s33*, *s12*, *s13*, *s23*)

Calculate eigenvalues of a symmetric 3D tensor.

Parameters

- **s11** (*array_like*) – Component 11 of 3D tensor.
- **s22** (*array_like*) – Component 22 of 3D tensor.
- **s33** (*array_like*) – Component 33 of 3D tensor.
- **s12** (*array_like*) – Component 12 of 3D tensor.
- **s13** (*array_like*) – Component 13 of 3D tensor.
- **s23** (*array_like*) – Component 23 of 3D tensor.

Returns Array containing eigenvalues sorted in ascending order. Shape is (length of components, 3) or simply 3 if components are single values.

Return type numpy.ndarray

`pylife.stress.equistress.`**`max_principal`**`(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Calculate maximum principal stress (maximum of eigenvalues).

Parameters

- **`s11`** (*`array_like`*) – Component 11 of 3D tensor.
- **`s22`** (*`array_like`*) – Component 22 of 3D tensor.
- **`s33`** (*`array_like`*) – Component 33 of 3D tensor.
- **`s12`** (*`array_like`*) – Component 12 of 3D tensor.
- **`s13`** (*`array_like`*) – Component 13 of 3D tensor.
- **`s23`** (*`array_like`*) – Component 23 of 3D tensor.

Returns Maximum principal stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`min_principal`**`(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Calculate minimum principal stress (minimum of eigenvalues).

Parameters

- **`s11`** (*`array_like`*) – Component 11 of 3D tensor.
- **`s22`** (*`array_like`*) – Component 22 of 3D tensor.
- **`s33`** (*`array_like`*) – Component 33 of 3D tensor.
- **`s12`** (*`array_like`*) – Component 12 of 3D tensor.
- **`s13`** (*`array_like`*) – Component 13 of 3D tensor.
- **`s23`** (*`array_like`*) – Component 23 of 3D tensor.

Returns Minimum principal stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`mises`**`(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Calculate equivalent stress according to von Mises.

Parameters

- **`s11`** (*`array_like`*) – Component 11 of 3D tensor.
- **`s22`** (*`array_like`*) – Component 22 of 3D tensor.
- **`s33`** (*`array_like`*) – Component 33 of 3D tensor.
- **`s12`** (*`array_like`*) – Component 12 of 3D tensor.
- **`s13`** (*`array_like`*) – Component 13 of 3D tensor.
- **`s23`** (*`array_like`*) – Component 23 of 3D tensor.

Returns Von Mises equivalent stress. Shape is the same as the components.

Return type numpy.ndarray

Raises **`AssertionError:`** – Components' shape is not consistent.

`pylife.stress.equistress.`**`principals`**`(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Calculate all principal stress components (eigenvalues).

Parameters

- `s11` (*array_like*) – Component 11 of 3D tensor.
- `s22` (*array_like*) – Component 22 of 3D tensor.
- `s33` (*array_like*) – Component 33 of 3D tensor.
- `s12` (*array_like*) – Component 12 of 3D tensor.
- `s13` (*array_like*) – Component 13 of 3D tensor.
- `s23` (*array_like*) – Component 23 of 3D tensor.

Returns All principal stresses. Shape *(…, 3)*.

Return type numpy.ndarray

`pylife.stress.equistress.`**`signed_mises_abs_max_principal`**(*s11*, *s22*, *s33*, *s12*, *s13*, *s23*)

Calculate equivalent stress according to von Mises, signed with the sign of the absolute maximum principal stress.

Parameters

- `s11` (*array_like*) – Component 11 of 3D tensor.
- `s22` (*array_like*) – Component 22 of 3D tensor.
- `s33` (*array_like*) – Component 33 of 3D tensor.
- `s12` (*array_like*) – Component 12 of 3D tensor.
- `s13` (*array_like*) – Component 13 of 3D tensor.
- `s23` (*array_like*) – Component 23 of 3D tensor.

Returns Signed von Mises equivalent stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`signed_mises_trace`**(*s11*, *s22*, *s33*, *s12*, *s13*, *s23*)

Calculate equivalent stress according to von Mises, signed with the sign of the trace (i.e s11 + s22 + s33).

Parameters

- `s11` (*array_like*) – Component 11 of 3D tensor.
- `s22` (*array_like*) – Component 22 of 3D tensor.
- `s33` (*array_like*) – Component 33 of 3D tensor.
- `s12` (*array_like*) – Component 12 of 3D tensor.
- `s13` (*array_like*) – Component 13 of 3D tensor.
- `s23` (*array_like*) – Component 23 of 3D tensor.

Returns Signed von Mises equivalent stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`signed_tresca_abs_max_principal`**(*s11*, *s22*, *s33*, *s12*, *s13*, *s23*)

Calculate equivalent stress according to Tresca, signed with the sign of the absolute maximum principal stress.

Parameters

- `s11` (*array_like*) – Component 11 of 3D tensor.
- `s22` (*array_like*) – Component 22 of 3D tensor.

- **`s33`** (*`array_like`*) – Component 33 of 3D tensor.
- **`s12`** (*`array_like`*) – Component 12 of 3D tensor.
- **`s13`** (*`array_like`*) – Component 13 of 3D tensor.
- **`s23`** (*`array_like`*) – Component 23 of 3D tensor.

Returns Signed Tresca equivalent stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`signed_tresca_trace`**`(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Calculate equivalent stress according to Tresca, signed with the sign of the trace (i.e s11 + s22 + s33).

Parameters

- **`s11`** (*`array_like`*) – Component 11 of 3D tensor.
- **`s22`** (*`array_like`*) – Component 22 of 3D tensor.
- **`s33`** (*`array_like`*) – Component 33 of 3D tensor.
- **`s12`** (*`array_like`*) – Component 12 of 3D tensor.
- **`s13`** (*`array_like`*) – Component 13 of 3D tensor.
- **`s23`** (*`array_like`*) – Component 23 of 3D tensor.

Returns Signed Tresca equivalent stress. Shape is the same as the components.

Return type numpy.ndarray

`pylife.stress.equistress.`**`tresca`**`(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Calculate equivalent stress according to Tresca.

Parameters

- **`s11`** (*`array_like`*) – Component 11 of 3D tensor.
- **`s22`** (*`array_like`*) – Component 22 of 3D tensor.
- **`s33`** (*`array_like`*) – Component 33 of 3D tensor.
- **`s12`** (*`array_like`*) – Component 12 of 3D tensor.
- **`s13`** (*`array_like`*) – Component 13 of 3D tensor.
- **`s23`** (*`array_like`*) – Component 23 of 3D tensor.

Returns Equivalent Tresca stress. Shape is the same as the components.

Return type numpy.ndarray

6.2.3 The `rainflow` module

A module performing rainflow counting

Overview over pyLife's rainflow counting module

From pyLife-2.0.0 on rainflow counting has been split into two different subtasks:

- hysteresis loop detection, done by a subclass of `AbstractDetector`.
- hysteresis loop recording, done by a subclass of *`AbstractRecorder`*.

That means you can combine detectors and recorders freely. You can choose recorders and detectors that come with pyLife but also write your own custom detectors and custom recorders.

Detectors

Detectors process a one dimensional time signal and detect hysteresis loops in them. A hysteresis loop consists of the sample point where the hysteresis starts, and the sample of the turning point where the hysteresis loop starts to turn back towards the load level of the starting point.

Once the detector has detected such a sample pair that makes a closed hysteresis loop it reports it to the recorder. All detectors report the load levels, some detectors also the index to the samples defining the loop limits.

pyLife's detectors are implemented in a way that the samples are chunkable. That means that you don't need to feed them the complete signal at once, but you can resume the rainflow analysis later when you have the next sample chunk.

As of now, pyLife comes with the following detectors:

- *`ThreePointDetector`*, classic three point algorithm, reports sample index
- *`FourPointDetector`*, recent four point algorithm, reports sample index
- *`FKMDetector`*, algorithm described by Clormann & Seeger, recommended by FKM, does not report sample index.

Recorders

Recorders are notified by detectors about loops and will process the loop information as they wish.

As of now, pyLife comes with the following recorders:

- *`LoopValueRecorder`*, only records the *from* and *to* values of all the closed hysteresis loops.`
- *`FullRecorder`*, records additionally to the *from* and *to* values also the indices of the loop turning points in the original time series, so that additional data like temperature during the loop or dwell times can be looked up in the original time series data.

API Documentation

Detectors

The `ThreePointDetector` class

class pylife.stress.rainflow.**ThreePointDetector**(*recorder*)

Classic three point rainflow counting algorithm.

```python
from pylife.stress.timesignal import TimeSignalGenerator
import pylife.stress.rainflow as RF

ts = TimeSignalGenerator(10, {
    'number': 50,
    'amplitude_median': 1.0, 'amplitude_std_dev': 0.5,
    'frequency_median': 4, 'frequency_std_dev': 3,
    'offset_median': 0, 'offset_std_dev': 0.4}, None, None).query(10000)

rfc = RF.ThreePointDetector(recorder=RF.LoopValueRecorder())
rfc.process(ts)

rfc.recorder.collective
```

Alternatively you can ask the recorder for a histogram matrix:

```python
rfc.recorder.histogram(bins=16)
```

```
from                                          to
(-11.364740270468808, -9.216993554273254]  (-10.233349189089544, -8.
↪126164018303303]     0.0
                                           (-8.126164018303303, -6.018978847517063]␣
↪     0.0
                                           (-6.018978847517063, -3.911793676730823]␣
↪     0.0
                                           (-3.911793676730823, -1.
↪8046085059445822]     0.0
                                           (-1.8046085059445822, 0.
↪30257666484165924]    0.0
                                                                                   ␣
↪    ...
(20.851460472464503, 22.999207188660062]   (12.9456876895591, 15.052872860345342]  ␣
↪     0.0
                                           (15.052872860345342, 17.160058031131584]␣
↪     0.0
                                           (17.160058031131584, 19.26724320191782] ␣
↪     0.0
                                           (19.26724320191782, 21.374428372704063] ␣
↪     0.0
                                           (21.374428372704063, 23.4816135434903]  ␣
↪     0.0
Length: 256, dtype: float64
```

We take three turning points into account to detect closed hysteresis loops.

- start: the point where the loop is starting from
- front: the turning point after the start
- back: the turning point after the front

A loop is considered closed if following conditions are met:

- the load difference between front and back is bigger than or equal the one between start and front. In other words: if the back goes beyond the starting point. For example (A-B-C) and (B-C-D) not closed, whereas (C-D-E) is.

- the loop init has not been a loop front in a prior closed loop. For example F would close the loops (D-E-F) but D is already front of the closed loop (C-D-E).
- the load level of the front has already been covered by a prior turning point. Otherwise it is considered part of the front residuum.

When a loop is closed it is possible that the loop back also closes unclosed loops of the past by acting as loop back for an unclosed start/front pair. For example E closes the loop (C-D-E) and then also (A-B-E).

```
Load -----------------------------
|        x B               F x
--------/-\-----------------/-----
|      /   \   x D         /
------/-----\-/-\---------/-------
|    /     C x   \       /
--\-/-------------\-----/---------
|  x A             \   /
--------------------\-/-----------
|                    x E
----------------------------------
|              Time
```

`__init__`(*recorder*)
Instantiate a ThreePointDetector.

Parameters **`recorder`** (subclass of *`AbstractRecorder`*) – The recorder that the detector will report to.

`process`(*samples*)
Process a sample chunk.

Parameters **`samples`** (*`array_like, shape (N, )`*) – The samples to be processed

Returns **self** – The `self` object so that processing can be chained

Return type *ThreePointDetector*

The `FourPointDetector` class

class `pylife.stress.rainflow.`**`FourPointDetector`**(*recorder*)
Implements four point rainflow counting algorithm.

```python
from pylife.stress.timesignal import TimeSignalGenerator
import pylife.stress.rainflow as RF

ts = TimeSignalGenerator(10, {
    'number': 50,
    'amplitude_median': 1.0, 'amplitude_std_dev': 0.5,
    'frequency_median': 4, 'frequency_std_dev': 3,
    'offset_median': 0, 'offset_std_dev': 0.4}, None, None).query(10000)

rfc = RF.FourPointDetector(recorder=RF.LoopValueRecorder())
rfc.process(ts)

rfc.recorder.collective
```

Alternatively you can ask the recorder for a histogram matrix:

```
rfc.recorder.histogram(bins=16)
```

```
from                                      to
(-18.84179414559365, -16.70855774672485]  (-21.756306907873412, -19.
↪438175359364863]    0.0
                                          (-19.438175359364863, -17.
↪120043810856316]    0.0
                                          (-17.120043810856316, -14.
↪801912262347768]    0.0
                                          (-14.801912262347768, -12.48378071383922]␣
↪     0.0
                                          (-12.48378071383922, -10.165649165330672]␣
↪     0.0
                                                                                   ␣
↪    ...
(13.156751837438321, 15.289988236307122]  (3.7431401257206183, 6.061271674229165]  ␣
↪     0.0
                                          (6.061271674229165, 8.37940322273771]    ␣
↪     0.0
                                          (8.37940322273771, 10.697534771246264]   ␣
↪     0.0
                                          (10.697534771246264, 13.01566631975481]  ␣
↪     0.0
                                          (13.01566631975481, 15.333797868263353]  ␣
↪     0.0
Length: 256, dtype: float64
```

We take four turning points into account to detect closed hysteresis loops.

Consider four consecutive peak/valley points say, A, B, C, and D If B and C are contained within A and B, then a cycle is counted from B to C; otherwise no cycle is counted.

i.e, If X Y AND Z Y then a cycle exist FROM=B and TO=C where, ranges X=**|D–C|**, Y=**|C–B|**, and Z=**|B–A|**

```
Load -----------------------------
|        x B               F x
--------/-\-----------------/-----
|      /   \   x D         /
------/-----\-/-\---------/-------
|    /     C x   \       /
--\-/-------------\-----/---------
|  x A             \   /
--------------------\-/-----------
|                    x E
----------------------------------
|              Time
```

So, if a cycle exsist from B to C then delete these peaks from the turns array and perform next iteration by joining A&D else if no cylce exsists, then B would be the next strarting point.

`__init__`(*recorder*)

Instantiate a FourPointDetector.

Parameters **`recorder`** (subclass of *`AbstractRecorder`*) – The recorder that the detector will report to.

process(*samples*)
Process a sample chunk.

Parameters samples (*array_like, shape (N,)*) – The samples to be processed

Returns self – The self object so that processing can be chained

Return type *FourPointDetector*

The FKMDetector class

class pylife.stress.rainflow.**FKMDetector**(*recorder*)
Rainflow detector as described in FKM non linear.

The algorithm has been published by Clormann & Seeger 1985 and has been cited heavily since.

```
from pylife.stress.timesignal import TimeSignalGenerator
import pylife.stress.rainflow as RF

ts = TimeSignalGenerator(10, {
    'number': 50,
    'amplitude_median': 1.0, 'amplitude_std_dev': 0.5,
    'frequency_median': 4, 'frequency_std_dev': 3,
    'offset_median': 0, 'offset_std_dev': 0.4}, None, None).query(10000)

rfc = RF.FKMDetector(recorder=RF.LoopValueRecorder())
rfc.process(ts)

rfc.recorder.collective
```

Alternatively you can ask the recorder for a histogram matrix:

```
rfc.recorder.histogram(bins=16)
```

```
from                                         to
(-16.622041286664835, -14.507581641845505]  (-16.56195196949719, -14.
↪528172800076922]     0.0
                                             (-14.528172800076922, -12.
↪494393630656653]    0.0
                                             (-12.494393630656653, -10.
↪460614461236384]    0.0
                                             (-10.460614461236384, -8.
↪426835291816115]     0.0
                                             (-8.426835291816115, -6.
↪393056122395846]      0.0
                                                                                    ␣
↪     ...
(15.094853385625107, 17.209313030444434]    (5.809618894125766, 7.8433980635460365]␣
↪      0.0
                                             (7.8433980635460365, 9.877177232966307]␣
↪      0.0
                                             (9.877177232966307, 11.910956402386574]␣
↪      0.0
                                             (11.910956402386574, 13.
↪944735571806842]      0.0
```

(continues on next page)

(continued from previous page)

```
                                              (13.944735571806842, 15.
↪978514741227114]      0.0
Length: 256, dtype: float64
```

Note: This detector **does not** report the loop index.

`__init__`(*recorder*)

Instantiate a FKMDetector.

Parameters **`recorder`** (subclass of *`AbstractRecorder`*) – The recorder that the detector will report to.

`process`(*samples*)

Process a sample chunk.

Parameters **`samples`** (*`array_like, shape (N, )`*) – The samples to be processed

Returns **self** – The `self` object so that processing can be chained

Return type *FKMDetector*

Recorders

The `LoopValueRecorder` class

`class pylife.stress.rainflow.`**`LoopValueRecorder`**

Rainflow recorder that collects the loop values.

`__init__`()

Instantiate a LoopRecorder.

`property collective`

The overall collective recorded as `pandas.DataFrame`.

The columns are named `from`, `to`.

`histogram`(*bins=10*)

Calculate a histogram of the recorded values into a `pandas.Series`.

An interval index is used to index the bins.

Parameters **`bins`** (*`int or array_like or [int, int] or [array, array], optional`*) – The bin specification (see numpy.histogram2d)

Returns A pandas.Series using a multi interval index in order to index data point for a given from/to value pair.

Return type pandas.Series

`histogram_numpy`(*bins=10*)

Calculate a histogram of the recorded values into a plain numpy.histogram2d.

Parameters **`bins`** (*`int or array_like or [int, int] or [array, array], optional`*) – The bin specification (see numpy.histogram2d)

Returns

- **H** (*ndarray, shape(nx, ny)*) – The bi-dimensional histogram of samples (see numpy.histogram2d)
- **xedges** (*ndarray, shape(nx+1,)*) – The bin edges along the first dimension.
- **yedges** (*ndarray, shape(ny+1,)*) – The bin edges along the second dimension.

`record_values`(*values_from*, *values_to*)
Record the loop values.

`property values_from`
1-D float array containing the values from which the loops start.

`property values_to`
1-D float array containing the values the loops go to before turning back.

The `FullRecorder` class

`class pylife.stress.rainflow.`**`FullRecorder`**
Rainflow recorder that collects the loop values and the loop index.

Same functionality like *`LoopValueRecorder`* but additionally collects the loop index.

`__init__`()
Instantiate a FullRecorder.

`property collective`
The overall collective recorded as `pandas.DataFrame`.

The columns are named `from`, `to`, `index_from`, `index_to`.

`property index_from`
1-D int array containing the index to the samples from which the loops start.

`property index_to`
1-D int array containing the index to the samples the loops go to before turning back.

`record_index`(*index_from*, *index_to*)
Record the index.

The `AbstractRecorder` class

`class pylife.stress.rainflow.`**`AbstractRecorder`**
A common base class for rainflow recorders.

Subclasses implementing a rainflow recorder are supposed to implement the following methods:

- `record_values()`
- `record_index()`

`__init__`()
Instantiate an AbstractRecorder.

`chunk_local_index`(*global_index*)
Transform the global index to an index valid in a certain chunk.

Parameters **`global_index`** (*`array-like int`*) – The global index to be transformed.

Returns

- **chunk_number** (*array of ints*) – The number of the chunk the indexed sample is in.

- **chunk_local_index** (*array of ints*) – The index of the sample in its chunk.

`property chunks`

The limits index of the chunks processed so far.

Note: The first chunk limit is the length of the first chunk, so identical to the index to the first sample of the second chunk, if a second chunk exists.

`record_index`(*indeces_from*, *indeces_to*)

Record hysteresis loop index to the recorder.

Parameters

- **indeces_from** (*list of ints*) – The sample indeces where the hysteresis loop starts from.
- **indeces_to** (*list of ints*) – The sample indeces where the hysteresis loop goes to and turns back from.

Note: Default implementation does nothing. Can be implemented by recorders interested in the hysteresis loop values.

`record_values`(*values_from*, *values_to*)

Report hysteresis loop values to the recorder.

Parameters

- **values_from** (*list of floats*) – The sample values where the hysteresis loop starts from.
- **values_to** (*list of floats*) – The sample values where the hysteresis loop goes to and turns back from.

Note: Default implementation does nothing. Can be implemented by recorders interested in the hysteresis loop values.

`report_chunk`(*chunk_size*)

Report a chunk.

Parameters **chunk_size** (*int*) – The length of the chunk previously processed by the detector.

Note: Should be called by the detector after the end of `process()`.

Utility functions

`pylife.stress.rainflow.`**`find_turns`**`(`*samples*`)`

Find the turning points in a sample chunk.

Parameters **`samples`** (*`1D numpy.ndarray`*) – the sample chunk

Returns

- **index** (*1D numpy.ndarray*) – the indeces where sample has a turning point
- **turns** (*1D numpy.ndarray*) – the values of the turning points

Notes

In case of plateaus i.e. multiple directly neighbored samples with exactly the same values, building a turning point together, the first sample of the plateau is indexed.

Compatibility

The old pylife-1.x rainflow counting API

In order to not to break existing code, the old pylife-1.x API is still in place as wrappers around the new API. Using it is strongly discouraged. It will be deprecated and eventually removed.

6.2.4 The `LoadCollective` class

A Load collective.

The usual use of this signal is to process hysteresis loop data from a rainflow recording. Usually the keys `from` and `to` are used to describe the hysteresis loops. Alternatively also the keys `range` and `mean` can be given. In that case the frame is internally converted to `from` and `to` where the `from` values are the lower ones.

6.2.5 The `LoadHistogram` class

Base class for signal accessor classes.

Notes

Derived classes need to implement the method *_validate(self, obj)* that gets *pandas_obj* as *obj* parameter. This *validate()* method must raise an Exception (e.g. AttributeError or ValueError) in case *obj* is not a valid DataFrame for the kind of signal.

For these validation `fail_if_key_missing()` and `get_missing_keys()` might be helpful.

For a derived class you can register methods without modifying the class' code itself. This can be useful if you want to make signal accessor classes extendable.

See also:

`fail_if_key_missing()` `get_missing_keys()` `register_method()`

6.2.6 The `stresssignal` module

class `pylife.stress.stresssignal.`**`StressTensorVoigt`**(*pandas_obj*)

DataFrame accessor class for Voigt noted stress tensors

Raises `AttributeError` – if at least one of the needed columns is missing.

Notes

Base class to access `pandas.DataFrame` objects containing Voigt noted stress tensors. The stress tensor components are assumed to be in the columns *S11*, *S22*, *S33*, *S12*, *S13*, *S23*.

See also:

`pandas.api.extensions.register_dataframe_accessor()`

Examples

For an example see `equistress.StressTensorEquistress`.

6.2.7 The `timesignal` module

A module for time signal handling

Warning: This module is not considered finalized even though it is part of pylife-2.0. Breaking changes might occur in upcoming minor releases.

class `pylife.stress.timesignal.`**`TimeSignalGenerator`**(*sample_rate*, *sine_set*, *gauss_set*, *log_gauss_set*)

Generates mixed time signals

The generated time signal is a mixture of random sets of sinus signals

For each set the user supplys a dict describing the set:

```
sinus_set = {
    'number': number of signals
    'amplitude_median':
    'amplitude_std_dev':
    'frequency_median':
    'frequency_std_dev':
    'offset_median':
    'offset_std_dev':
}
```

The amplitudes (A), fequencies (ω) and offsets (c) are then norm distributed. Each sinus signal looks like

$$s = A \sin(\omega t + \phi) + c$$

where phi is a random value between 0 and 2π.

So the whole sinus S set is given by the following expression:

$$S = \sum_i^n A_i \sin(\omega_i t + \phi_i) + c_i.$$

`query`(*sample_num*)

Gets a sample chunk of the time signal

Parameters `sample_num` (*int*) – number of the samples requested

Returns **samples** – the requested samples

Return type 1D numpy.ndarray

You can query multiple times, the newly delivered samples will smoothly attach to the previously queried ones.

`reset`()

Resets the generator

A resetted generator behaves like a new generator.

`pylife.stress.timesignal.butter_bandpass`(*df*, *lowcut*, *highcut*, *order=5*)

Use the functonality of scipy

Parameters

- `df` (*DataFrame*) –
- `lowcut` (*float*) – low frequency
- `highcut` (*float*) – high freqency.
- `order` (*int, optional*) – Butterworth filter order. The default is 5.

Returns **TSout**

Return type DataFrame

`pylife.stress.timesignal.clean_timeseries`(*df*, *comparison_column*, *window_size=1000*, *overlap=800*, *feature='abs_energy'*, *method='keep'*, *n_gridpoints=3*, *percentage_max=0.05*, *order=3*)

Removes segments of the data in which the extracted feature value is lower as percentage_max and fills the gaps with polynomial regression

Parameters

- `df` (*input pandas DataFrame that shall be cleaned*) –
- `comparison_column` (*str, column that is used for the feature*) – comparison with percentage max
- `window_size` (*int, optional*) – window size of the rolled segments - The default is 1000.
- `overlap` (*int, optional*) – overlap between 2 adjecent windows -The default is 200.
- `feature` (*string, optional*) – extracted feature - only supports one at a time - and only features form tsfresh that dont need extra parameters. The default is “maximum”.
- `method` (*string, optional*) –
  - ‘keep’: keeps the windows which are extracted,
  - ’remove’: removes the windows which are extracted
- `n_gridpoints` (*TYPE, optional*) – number of gridpoints. The default is 3.
- `percentage_max` (*float, optional*) – min percentage of the maximum to keep the window. The default is 0.05.
- `order` (*int, optional*) – order of polynom The default is 3.

Returns **df_poly** – cleaned DataFrame

Return type pandas DataFrame

`pylife.stress.timesignal.`**`fs_calc`**`(`*df*`)`

Calculates the sample frequency of a DataFrame time series

Parameters **`df`** (*`DataFrame`*) – time series.

Returns **fs** – sample freqency

Return type int, float

`pylife.stress.timesignal.`**`psd_df`**`(`*df_ts*, *NFFT=512*`)`

calculates the psd using Welch algorithm from matplotlib functionality

Parameters

- **`df_ts`** (*`DataFram`*) – time series dataframe
- **`NFFT`** (*`int, optional`*) – BufferSize. The default is 512.

Returns **df_psd** – PSD.

Return type DataFrame

`pylife.stress.timesignal.`**`resample_acc`**`(`*df*, *fs=1*`)`

Resamples a pandas time series DataFrame

Parameters

- **`df`** (*`DataFrame`*) –
- **`time_col`** (*`str`*) – column name of the time column
- **`fs`** (*`float`*) – sample rate of the resampled time series

Return type DataFrame

6.2.8 The `frequencysignal` module

A module for frequency signal handling

Warning: This module is not considered finalized even though it is part of pylife-2.0. Breaking changes might occur in upcoming minor releases.

`class pylife.stress.frequencysignal.`**`psdSignal`**`(`*df*`)`

Handles different routines for self signals

Remark: We are using the pandas data frame schema. The index contains the discrete frequency step. Every single column one self.

Some functions of these class:

- psd_optimizer
- …

`psd_smoother``(`*fsel*, *factor_rms_nodes=0.5*`)`

Smoothen a PSD using nodes and a penalty factor weighting the errors for the RMS and for the node PSD values

Parameters

- **`self`** (*`DataFrame`*) – unsmoothed PSD

- **fsel** (*list or np.array*) – nodes
- **factor_rms_nodes** (*float (0 <= factor_rms_nods <= 1)*) – penalty error weighting the errors:
  - 0: only error of node PSD values is considered
  - 1: only error of the RMS is considered

Return type DataFrame

rms_psd()

6.3 Strength

6.3.1 The Fatigue class

class pylife.strength.**Fatigue**(*pandas_obj*)

Extension for *WoehlerCurve* accessor class for fatigue calculations.

Note: This class is accessible by the *fatigue* accessor attribute.

damage(*load_collective*)

Calculate the damage to the material caused by a given load collective.

Parameters load_collective (*pandas object or object behaving like a load collective*) – The given load collective

Returns damage – The calculated damage values. The index is the broadcast between *load_collective* and *self*.

Return type pd.Series

security_cycles(*load_distribution*, *allowed_failure_probability*)

Calculate the security factor in cycles direction for given load distribution.

Parameters load_distribution (*pandas object or object behaving like a load collective*) – The given load distribution

Returns security_factor – The calculated security_factors. The index is the broadcast between *load_distribution* and *self*.

Return type pd.Series

security_load(*load_distribution*, *allowed_failure_probability*)

Calculate the security factor in load direction for given load distribution.

Parameters load_distribution (*pandas object or object behaving like a load collective*) – The given load distribution

Returns security_factor – The calculated security_factors. The index is the broadcast between *load_distribution* and *self*.

Return type pd.Series

6.3.2 The `meanstress` module

Meanstress routines

Mean stress transformation methods

- FKM Goodman
- Five Segment Correction

`class pylife.strength.meanstress.`**`HaighDiagram`**`(`*pandas_obj*`)`

Model for a Haigh diagram in order to perform meanstress transformations.

A Haigh diagram a set of meanstress sensitivity slopes M that is changing with the R-values. The values of the `` `pd.Series `` represents that slopes M and the *pd.IntervalIndex* represents the R-ranges.

`classmethod five_segment``(`*five_segment_haigh_diagram*`)`

Create a five segment slope Haigh diagram.

Parameters **`five_segment_haigh_diagram`** (`pandas.Series` or `pandas.DataFrame`) – The five segment meanstress slope data.

Notes

`five_segment_hagih_diagram` has to provide the following keys:

- `M0`: the mean stress sensitivity between `R==-inf` and `R==0`
- `M1`: the mean stress sensitivity between `R==0` and `R==R12`
- `M2`: the mean stress sensitivity betwenn `R==R12` and `R==R23`
- `M3`: the mean stress sensitivity between `R==R23` and `R==1`
- `M4`: the mean stress sensitivity beyond `R==1`
- `R12`: R-value between `M1` and `M2`
- `R23`: R-value between `M2` and `M3`

`classmethod fkm_goodman``(`*haigh_fkm_goodman*`)`

Create a Haigh diagram according to FKM Goodman.

Parameters

- **`haigh_fkm_goodman`** (*`pd.Series or pd.DataFrame`*) – a series containing one or a dataframe containing multiple values for *M* and optionally *M2*.
- **`M`** (*`The Haigh diagram according to FKM Goodman comes with the slope`*) –
- **`slope`** (*`which is valid between R==-inf and R==0. Beyond R==0 the`*) –
- **`not.`** (*`` is M2` if ``M2 is given or M/3 if ``*) –

`classmethod from_dict``(`*segments_dict*`)`

Create a Haigh diagram from a dict.

Parameters **`segments_dict`** (*`dict`*) – dict resolving the R-value intervals to the meanstress slope

Example

```
>>> hd = MST.HaighDiagram.from_dict({
>>>     (1.0, np.inf): 0.0,
>>>     (-np.inf, 0.0): 0.5,
>>>     (0.0, 1.0): 0.167
>>> })
```

sets up a FKM Goodman like Haigh diagram.

transform(*cycles*, *R_goal*)

Transform a load collective to defined R-value.

Parameters cycles (pd.Series accepted by class:LoadCollective` or class: `LoadHistogram) – The load collective

Returns transformed_cycles – The transformed cycles

Return type pd.Series

class pylife.strength.meanstress.**MeanstressTransformCollective**(*pandas_obj*)

five_segment(*haigh*, *R_goal*)

fkm_goodman(*ms_sens*, *R_goal*)

class pylife.strength.meanstress.**MeanstressTransformMatrix**(*pandas_obj*)

fkm_goodman(*haigh*, *R_goal*)

pylife.strength.meanstress.**experimental_mean_stress_sensitivity**(*sn_curve_R0*, *sn_curve_Rn1*, *N_c=inf*)

Estimate the mean stress sensitivity from two *FiniteLifeCurve* objects for the same amount of cycles *N_c*.

The formula for calculation is taken from: "Betriebsfestigkeit", Haibach, 3. Auflage 2006

Formula (2.1-24):

$$M_{\sigma} = {S_a}^{R=-1}(N_c) / {S_a}^{R=0}(N_c) - 1$$

Alternatively the mean stress sensitivity is calculated based on both SD values (if N_c is not given).

Parameters

- **sn_curve_R0** (*pylife.strength.sn_curve.FiniteLifeCurve*) – Instance of FiniteLifeCurve for R == 0
- **sn_curve_Rn1** (*pylife.strength.sn_curve.FiniteLifeCurve*) – Instance of FiniteLifeCurve for R == -1
- **N_c** (*float, (default=np.inf)*) – Amount of cycles where the amplitudes should be compared. If N_c is higher than a fatigue transition point (ND) for the SN-Curves, SD is taken. If N_c is None, SD values are taken as stress amplitudes instead.

Returns Mean stress sensitivity M_sigma

Return type float

Raises ValueError – If the resulting M_sigma doesn't lie in the range from 0 to 1 a ValueError is raised, as this value would suggest higher strength with additional loads.

pylife.strength.meanstress.**five_segment_correction**(*amplitude*, *meanstress*, *M0*, *M1*, *M2*, *M3*, *M4*, *R12*, *R23*, *R_goal*)

Performs a mean stress transformation to R_goal according to the Five Segment Mean Stress Correction

Parameters

- `Sa` – the stress amplitude
- `Sm` – the mean stress
- `Rgoal` – the R-value to transform to
- `M` – the mean stress sensitivity between R=-inf and R=0
- `M1` – the mean stress sensitivity between R=0 and R=R12
- `M2` – the mean stress sensitivity betwenn R=R12 and R=R23
- `M3` – the mean stress sensitivity between R=R23 and R=1
- `M4` – the mean stress sensitivity beyond R=1
- `R12` – R-value between M1 and M2
- `R23` – R-value between M2 and M3

Returns the transformed stress range

`pylife.strength.meanstress.`**`fkm_goodman`**(*amplitude*, *meanstress*, *M*, *M2*, *R_goal*)

6.3.3 The `FailureProbability` class

Strength representation to calculate failure probabilities

The strength is represented as a log normal distribution of strength_median and strength_std.

Failure probabilities can be calculated for a given load or load distribution.

param strength_median The median value of the strength

type strength_median array_like, shape (N,)

param strength_std The standard deviation of the strength

type strength_std array_like, shape (N,)

Note: We assume that the load and the strength are statistically distributed values. In case the load is higher than the strength we get failure. So if we consider a quantile of our load distribution of a probability p_load, the probability of failure due to a load of this quantile is p_load times the probability that the strength lies within this quantile or below.

So in order to calculate the total failure probability, we need to integrate the load's pdf times the strength' cdf from -inf to +inf.

6.3.4 The `miner` module

Implementation of the miner rule for fatigue analysis

Currently, the following implementations are part of this module:

- Miner Elementary
- Miner Haibach

References

- M. Wächter, C. Müller and A. Esderts, "Angewandter Festigkeitsnachweis nach {FKM}-Richtlinie" Springer Fachmedien Wiesbaden 2017, https://doi.org/10.1007/978-3-658-17459-0
- E. Haibach, "Betriebsfestigkeit", Springer-Verlag 2006, https://doi.org/10.1007/3-540-29364-7

`class pylife.strength.miner.`**`MinerBase`**`(`*pandas_obj*`)`

Basic functions related to miner-rule (original).

Uses the constructor of *`WoehlerCurve`*.

`effective_damage_sum``(`*collective*`)`

Compute *effective damage sum* D_m.

Refers to the formula given in Waechter2017, p. 99

Parameters **`collective`** (*`a load collective`*) – the multiple of the lifetime

Returns **effective_damage_sum** – The effective damage sums for the collective

Return type float or `pandas.Series`

`finite_life_factor``(`*N*`)`

Calculate *finite life factor* according to Waechter2017 (p. 96).

Parameters **`N`** (*`float`*) – Collective range (sum of cycle numbers) of load collective

`gassner_cycles``(`*collective*`)`

Compute the cycles of the Gassner line for a certain load collective.

Parameters **`collective`** (`LoadCollective` or similar) – The load collective

Returns The cycles for the collective

Return type cycles

Note: The absolute load levels of the collective are important.

`abstract` **`lifetime_multiple`**`(`*collective*`)`

Compute the lifetime multiple according to the corresponding Miner rule.

Needs to be implemented in the class implementing the Miner rule.

Parameters **`collective`** (`LoadCollective` or similar) – The load collective

Returns **lifetime_multiple** – lifetime multiple

Return type float > 0

`class pylife.strength.miner.`**`MinerElementary`**`(`*pandas_obj*`)`

Implementation of Miner Elementary according to Waechter2017.

`gassner``(`*collective*`)`

Calculate the Gaßner shift according to Miner Elementary.

Parameters **`collective`** (`LoadCollective` or similar) – The load collective

Returns **gassner** – The Gaßner shifted fatigue strength object.

Return type `Fatigue`

`lifetime_multiple``(`*collective*`)`

Compute the lifetime multiple according to Miner Elementary.

Described in Waechter2017 as "Lebensdauervielfaches, A_ele".

Parameters **`collective`** (`LoadCollective` or similar) – The load collective

Returns **lifetime_multiple** – lifetime multiple

Return type float > 0

`class pylife.strength.miner.`**`MinerHaibach`**`(`*pandas_obj*`)`

Miner-modified according to Haibach (2006).

> **Warning:** Contrary to Miner Elementary, the lifetime multiple is not constant but dependent on the evaluated load level! That is why there is no method for the Gaßner shift.

`lifetime_multiple``(`*collective*`)`

Compute the lifetime multiple for Miner-modified according to Haibach.

Refer to Haibach (2006), p. 291 (3.21-61). The lifetime multiple can be expressed in respect to the maximum amplitude so that N_lifetime = N_Smax * A

Parameters **`collective`** (`LoadCollective` or similar) – The load collective

Returns **lifetime_multiple** – lifetime multiple return value is 'inf' if maximum collective amplitude < SD

Return type float > 0

`pylife.strength.miner.`**`effective_damage_sum`**`(`*lifetime_multiple*`)`

Compute *effective damage sum*.

Refers to the formula given in Waechter2017, p. 99

Parameters

- **`A`** (*`float or np.ndarray (with 1 element)`*) – the multiple of the lifetime
- **`Returns`** –
- **`d_m`** (*`float`*) – the effective damage sum

6.4 Materiallaws

6.4.1 The `hookeslaw` module

`class pylife.materiallaws.hookeslaw.`**`HookesLaw1d`**`(`*E*`)`

Implementation of the one dimensional Hooke's Law

Parameters **`E`** (*`float`*) – Young's modulus

`property E`

Get Young's modulus

`strain``(`*stress*`)`

Get the elastic strain for a given stress

Parameters **`stress`** (*`array-like float`*) – The stress

Returns **strain** – The resulting elastic strain

Return type array-like float

`stress`(*strain*)
Get the stress for a given elastic strain

Parameters **strain** (*array-like float*) – The elastic strain

Returns **strain** – The resulting stress

Return type array-like float

`class pylife.materiallaws.hookeslaw.HookesLaw2dPlaneStrain`(*E*, *nu*)
Implementation of the Hooke's Law under plane strain conditions.

Parameters

- **E** (*float*) – Young's modulus
- **nu** (*float*) – Poisson's ratio. Must be between -1 and 1./2.

Notes

A cartesian coordinate system is assumed. The strain components in 3 direction are assumed to be zero, e33 = g13 = g23 = 0.

`property E`
Get Young's modulus

`property G`
Get the sheer modulus

`property K`
Get the bulk modulus

`property nu`
Get Poisson's ratio

`strain`(*s11*, *s22*, *s12*)
Get the elastic strain components for given stress components

Parameters

- **s11** (*array-like float*) – The normal stress component with basis 1-1
- **s22** (*array-like float*) – The normal stress component with basis 2-2
- **s12** (*array-like float*) – The shear stress component with basis 1-2

Returns

- **e11** (*array-like float*) – The resulting elastic normal strain component with basis 1-1
- **e22** (*array-like float*) – The resulting elastic normal strain component with basis 2-2
- **g12** (*array-like float*) – The resulting elastic engineering shear strain component with basis 1-2, (1. / 2 * g12 is the tensor component)

`stress`(*e11*, *e22*, *g12*)
Get the stress components for given elastic strain components

Parameters

- **e11** (*array-like float*) – The elastic normal strain component with basis 1-1
- **e22** (*array-like float*) – The elastic normal strain component with basis 2-2
- **g12** (*array-like float*) – The elastic engineering shear strain component with basis 1-2, (1. / 2 * g12 is the tensor component)

Returns

- **s11** (*array-like float*) – The resulting normal stress component with basis 1-1
- **s22** (*array-like float*) – The resulting normal stress component with basis 2-2
- **s33** (*array-like float*) – The resulting normal stress component with basis 3-3
- **s12** (*array-like float*) – The resulting shear stress component with basis 1-2

`class pylife.materiallaws.hookeslaw.HookesLaw2dPlaneStress(E, nu)`

Implementation of the Hooke's Law under plane stress conditions.

Parameters

- **E** (*float*) – Young's modulus
- **nu** (*float*) – Poisson's ratio. Must be between -1 and 1./2.

Notes

A cartesian coordinate system is assumed. The stress components in 3 direction are assumed to be zero, s33 = s13 = s23 = 0.

`property E`

Get Young's modulus

`property G`

Get the sheer modulus

`property K`

Get the bulk modulus

`property nu`

Get Poisson's ratio

`strain(s11, s22, s12)`

Get the elastic strain components for given stress components

Parameters

- **s11** (`array-like float`) – The normal stress component with basis 1-1
- **s22** (`array-like float`) – The normal stress component with basis 2-2
- **s12** (`array-like float`) – The shear stress component with basis 1-2

Returns

- **e11** (*array-like float*) – The resulting elastic normal strain component with basis 1-1
- **e22** (*array-like float*) – The resulting elastic normal strain component with basis 2-2
- **e33** (*array-like float*) – The resulting elastic normal strain component with basis 3-3
- **g12** (*array-like float*) – The resulting elastic engineering shear strain component with basis 1-2, (1. / 2 * g12 is the tensor component)

`stress(e11, e22, g12)`

Get the stress components for given elastic strain components

Parameters

- **e11** (`array-like float`) – The elastic normal strain component with basis 1-1
- **e22** (`array-like float`) – The elastic normal strain component with basis 2-2

- **g12** (`array-like float`) – The elastic engineering shear strain component with basis 1-2, (1. / 2 * g12 is the tensor component)

Returns

- **s11** (*array-like float*) – The resulting normal stress component with basis 1-1
- **s22** (*array-like float*) – The resulting normal stress component with basis 2-2
- **s12** (*array-like float*) – The resulting shear stress component with basis 1-2

`class pylife.materiallaws.hookeslaw.HookesLaw3d(`*E*, *nu*`)`

Implementation of the Hooke's Law in three dimensions.

Parameters

- **E** (`float`) – Young's modulus
- **nu** (`float`) – Poisson's ratio. Must be between -1 and 1./2

Notes

A cartesian coordinate system is assumed.

`property E`

Get Young's modulus

`property G`

Get the sheer modulus

`property K`

Get the bulk modulus

`property nu`

Get Poisson's ratio

`strain(`*s11*, *s22*, *s33*, *s12*, *s13*, *s23*`)`

Get the elastic strain components for given stress components

Parameters

- **s11** (`array-like float`) – The resulting normal stress component with basis 1-1
- **s22** (`array-like float`) – The resulting normal stress component with basis 2-2
- **s33** (`array-like float`) – The resulting normal stress component with basis 3-3
- **s12** (`array-like float`) – The resulting shear stress component with basis 1-2
- **s13** (`array-like float`) – The resulting shear stress component with basis 1-3
- **s23** (`array-like float`) – The resulting shear stress component with basis 2-3

Returns

- **e11** (*array-like float*) – The resulting elastic normal strain component with basis 1-1
- **e22** (*array-like float*) – The resulting elastic normal strain component with basis 2-2
- **e33** (*array-like float*) – The resulting elastic normal strain component with basis 3-3
- **g12** (*array-like float*) – The resulting elastic engineering shear strain component with basis 1-2, (1. / 2 * g12 is the tensor component)
- **g13** (*array-like float*) – The resulting elastic engineering shear strain component with basis 1-3, (1. / 2 * g13 is the tensor component)

- **g23** (*array-like float*) – The resulting elastic engineering shear strain component with basis 2-3, (1. / 2 * g23 is the tensor component)

stress(*e11*, *e22*, *e33*, *g12*, *g13*, *g23*)

Get the stress components for given elastic strain components

Parameters

- **e11** (*array-like float*) – The elastic normal strain component with basis 1-1
- **e22** (*array-like float*) – The elastic normal strain component with basis 2-2
- **e33** (*array-like float*) – The elastic normal strain component with basis 3-3
- **g12** (*array-like float*) – The elastic engineering shear strain component with basis 1-2, (1. / 2 * g12 is the tensor component)
- **g13** (*array-like float*) – The elastic engineering shear strain component with basis 1-3, (1. / 2 * g13 is the tensor component)
- **g23** (*array-like float*) – The elastic engineering shear strain component with basis 2-3, (1. / 2 * g23 is the tensor component)

Returns

- **s11** (*array-like float*) – The resulting normal stress component with basis 1-1
- **s22** (*array-like float*) – The resulting normal stress component with basis 2-2
- **s33** (*array-like float*) – The resulting normal stress component with basis 3-3
- **s12** (*array-like float*) – The resulting shear stress component with basis 1-2
- **s13** (*array-like float*) – The resulting shear stress component with basis 1-3
- **s23** (*array-like float*) – The resulting shear stress component with basis 2-3

6.4.2 The RambergOsgood class

class pylife.materiallaws.**RambergOsgood**(*E*, *K*, *n*)

Simple implementation of the Ramberg-Osgood relation

Parameters

- **E** (*float*) – Young's Modulus
- **K** (*float*) – The strength coefficient
- **n** (*float*) – The strain hardening coefficient

Notes

The equation implemented is the one that Wikipedia refers to as "Alternative Formulation". The parameters *n* and *k* in this are formulation are the Hollomon parameters.

property E

Get Young's Modulus

property K

Get the strength coefficient

delta_strain(*delta_stress*)

Calculate the cyclic Masing strain span for a given stress span

Parameters **delta_stress** (*array-like float*) – The stress span

Returns **delta_strain** – The corresponding strain span

Return type array-like float

Notes

A Masing like behavior is assumed for the material as described in Kerbgrundkonzept.

delta_stress(*delta_strain*)
Calculate the cyclic Masing stress span for a given strain span

Parameters **delta_strain** (*array-like float*) – The strain span

Returns **delta_stress** – The corresponding stress span

Return type array-like float

Notes

A Masing like behavior is assumed for the material as described in Kerbgrundkonzept.

elastic_strain(*stress*)
Calculate the elastic strain for a given stress

Parameters **stress** (*array-like float*) – The stress

Returns **strain** – The resulting elastic strain

Return type array-like float

lower_hysteresis(*stress*, *max_stress*)
Calculate the lower (relaxation to compression) hysteresis starting from a given maximum stress

Parameters

- **stress** (*array-like float*) – The stress (must be below the maximum stress)
- **max_stress** (*float*) – The maximum stress of the hysteresis look

Returns **lower_hysteresis** – The lower hysteresis branch from *max_stress* all the way to *stress*

Return type array-like float

Raises **ValueError if stress > max_stress** –

property n
Get the strain hardening coefficient

plastic_strain(*stress*)
Calculate the plastic strain for a given stress

Parameters **stress** (*array-like float*) – The stress

Returns **strain** – The resulting plastic strain

Return type array-like float

strain(*stress*)
Calculate the elastic plastic strain for a given stress

Parameters **stress** (*array-like float*) – The stress

Returns **strain** – The resulting strain

Return type array-like float

`stress`(*strain*, *rtol=1e-05*, *tol=1e-06*)
Calculate the stress for a given strain

Parameters **strain** (`array-like float`) – The strain

Returns **stress** – The resulting stress

Return type array-like float

`tangential_compliance`(*stress*)
Calculate the derivative of the strain with respect to the stress for a given stress

Parameters **stress** (`array-like float`) – The stress

Returns **dstrain** – The resulting derivative

Return type array-like float

`tangential_modulus`(*stress*)
Calculate the derivative of the stress with respect to the strain for a given stress

Parameters **stress** (`array-like float`) – The stress

Returns **dstress** – The resulting derivative

Return type array-like float

6.4.3 The `WoehlerCurve` class

`class pylife.materiallaws.WoehlerCurve`(*pandas_obj*)
A PylifeSignal accessor for Wöhler Curve data.

Wöhler Curve (aka SN-curve) determines after how many load cycles at a certain load amplitude the component is expected to fail.

The signal has the following mandatory keys:

- `k_1` : The slope of the Wöhler Curve
- `ND` : The cycle number of the endurance limit
- `SD` : The load level of the endurance limit

The `_50` suffixes imply that the values are valid for a 50% probability of failure.

There are the following optional keys:

- **k_2** [The slope of the Wöhler Curve below the endurance limit] If the key is missing it is assumed to be infinity, i.e. perfect endurance
- **TN** [The scatter in cycle direction, (N_90/N_10)] If the key is missing it is assumed to be 1.0 or calculated from `TS` if given.
- **TS** [The scatter in cycle direction, (S_90/S_10)] If the key is missing it is assumed to be 1.0 or calculated from `TN` if given.

`property ND`

`property SD`

`property TN`
The load direction scatter value TN.

`property TS`

The load direction scatter value TS.

`basquin_cycles`(*load*, *failure_probability=0.5*)

Calculate the cycles numbers from loads according to the Basquin equation.

Parameters

- **load** (*array_like*) – The load levels for which the corresponding cycle numbers are to be calculated.
- **failure_probability** (*float, optional*) – The failure probability with which the component should fail when charged with *load* for the calculated cycle numbers. Default 0.5

Returns **cycles** – The cycle numbers at which the component fails for the given *load* values

Return type numpy.ndarray

`basquin_load`(*cycles*, *failure_probability=0.5*)

Calculate the load values from loads according to the Basquin equation.

Parameters

- **cycles** (*array_like*) – The cycle numbers for which the corresponding load levels are to be calculated.
- **failure_probability** (*float, optional*) – The failure probability with which the component should fail when charged with *load* for the calculated cycle numbers. Default 0.5

Returns **cycles** – The cycle numbers at which the component fails for the given *load* values

Return type numpy.ndarray

`broadcast`(*parameter*, *droplevel=[]*)

Broadcast the parameter to the object of `self`.

Parameters **parameters** (*scalar, numpy array or pandas object*) – The parameter to broadcast to

Returns **parameter, object**

Return type index aligned numerical objects

The

Examples

The behavior of the Broadcaster is best illustrated by examples:

- Broadcasting `pandas.Series` to a scalar results in a scalar and a `pandas.Series`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['foo', 'bar'], name='idx'))
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
array(5.)
```

```
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

- Broadcasting `pandas.DataFrame` to a scalar results in a `pandas.DataFrame` and a `pandas.Series`.

```
obj = pd.DataFrame({
    'foo': [1.0, 2.0],
    'bar': [3.0, 4.0]
}, index=pd.Index([1, 2], name='idx'))
obj
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
idx
1    5.0
2    5.0
dtype: float64
```

```
obj
```

- Broadcasting `pandas.DataFrame` to a a `pandas.Series` results in a `pandas.DataFrame` and a `pandas.Series`, **if and only if** the index name of the object is `None`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['tau', 'chi']))
obj
```

```
tau    1.0
chi    2.0
dtype: float64
```

```
parameter = pd.Series([3.0, 4.0], index=pd.Index(['foo', 'bar'], name='idx
↪'))
parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(parameter)

parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
obj
```

`cycles`(*load*, *failure_probability=0.5*)
Calculate the cycles numbers from loads.

Parameters

- **load** (*array_like*) – The load levels for which the corresponding cycle numbers are to be calculated.
- **failure_probability** (*float, optional*) – The failure probability with which the component should fail when charged with *load* for the calculated cycle numbers. Default 0.5

Returns **cycles** – The cycle numbers at which the component fails for the given *load* values

Return type numpy.ndarray

Notes

By default the calculation is performed according to the Basquin equation using *basquin_cycles()*. Derived classes can choose to override this in order to implement a different fatigue law.

`fail_if_key_missing`(*keys_to_check*, *msg=None*)
Raise an exception if any key is missing in a self._obj object.

Parameters

- **self._obj** (*pandas.DataFrame or pandas.Series*) – The object to be checked
- **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Raises

- **AttributeError** – if *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*
- **AttributeError** – if any of the keys is not found in the self._obj's keys.

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* meed to be found in the *self._obj.columns*.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* meed to be found in the *self._obj.index*.

See also:

get_missing_keys(), `stresssignal.StressTensorVoigt`

property failure_probability

classmethod from_parameters(***kwargs*)

Make a signal instance from a parameter set.

This is a convenience function to instantiate a signal from individual parameters rather than pandas objects.

A signal class like

```
@pd.api.extensions.register_dataframe_accessor('foo_signal')
class FooSignal(PylifeSignal):
    pass
```

The following two blocks are equivalent:

```
pd.Series({'foo': 1.0, 'bar': 2.0}).foo_signal
```

```
FooSignal.from_parameters(foo=1.0, bar=1.0)
```

get_missing_keys(*keys_to_check*)

Get a list of missing keys that are needed for a self._obj object.

Parameters **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Returns **missing_keys** – a list of missing keys

Return type list

Raises **AttributeError** – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* not found in the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* not found in the *self._obj.index* are returned.

property k_1

The second Wöhler slope.

property k_2

The second Wöhler slope.

keys()

Get a list of missing keys that are needed for a signal object.

Returns **keys** – a pandas index of keys

Return type pd.Index

Raises **AttributeError** – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, the *self._obj.index* are returned.

`load`(*cycles*, *failure_probability=0.5*)

Calculate the load values from loads.

Parameters

- **cycles** (*array_like*) – The cycle numbers for which the corresponding load levels are to be calculated.
- **failure_probability** (*float, optional*) – The failure probability with which the component should fail when charged with *load* for the calculated cycle numbers. Default 0.5

Returns **cycles** – The cycle numbers at which the component fails for the given *load* values

Return type numpy.ndarray

Notes

By default the calculation is performed according to the Basquin equation using `basquin_cycles()`. Derived classes can choose to override this in order to implement a different fatigue law.

`miner_elementary()`

Set k_2 to k_1 according Miner Elementary method (k_2 = k_1).

Return type self

`miner_haibach()`

Set k_2 to value according Miner Haibach method (k_2 = 2 * k_1 - 1).

Return type self

`to_pandas()`

Expose the pandas object of the signal.

Returns **pandas_object** – The pandas object representing the signal

Return type pd.DataFrame or pd.Series

Notes

The default implementation just returns the object given when instantiating the signal class. Derived classes may return a modified object or augmented, if they store some extra information.

By default the object is **not** copied. So make a copy yourself, if you intent to modify it.

`transform_to_failure_probability`(*failure_probability*)

6.4.4 The `true_stress_strain` module

Simple conversion functions from technical stress/strain to true stress/strain including fracture stress/strain.

`pylife.materiallaws.true_stress_strain.`**`true_fracture_strain`**`(`*reduction_area_fracture*`)`

Calculation of the true frature strain (in the FKM Non Linear (static assessment)) :param reduction_area_fracture: directly measured on the fractures sample :type reduction_area_fracture: float

Returns **true_fracture_strain** – descrivbes the calculated true fracture strain.

Return type float

`pylife.materiallaws.true_stress_strain.`**`true_fracture_stress`**`(`*fracture_force*, *initial_cross_section*, *reduction_area_fracture*`)`

Calculation of the true fracture stress (euqation FKM Non-linear (static assessment))

Parameters

- **`fracture_force`** (*`float`*) – from experimental results
- **`initial_cross_section`** (*`float`*) – cross section of initial tensile sample.
- **`reduction_area_fracture`** (*`float`*) – directly measured on the fractures sample.

Returns **true_fracture_stress** – calculated true fracture stress of the sample.

Return type float

`pylife.materiallaws.true_stress_strain.`**`true_strain`**`(`*tech_strain*`)`

Calculation of true strain data (from experimental data generated by tensile experiments)

Parameters **`tech_strain`** (*`array-like float`*) –

Returns **true_strain**

Return type array-like float

`pylife.materiallaws.true_stress_strain.`**`true_stress`**`(`*tech_stress*, *tech_strain*`)`

Calculate the true stress data from technical data

Parameters

- **`tech_stress`** (*`array-like float`*) – stress data from tensile experiments
- **`tech_strain`** (*`list of float`*) – strain data from tensile experiments

Returns **true_stress**

Return type array-like float

6.5 Materialdata

6.5.1 The `woehler` module

Module description

The `woehler` module overview

A module for Wöhler curve fatigue data analysis

Overview

FatigueData is a signal accessor class to handle fatigue data from a Wöhler test. They can be analyzed by several analyzers according to your choice

- *Elementary* only treats the finite zone of the fatigue data and calculates the slope and the scatter in lifetime direction. It is the base class for all other analyzers
- *Probit* calculates parameters not calculated by *Elementary* using the Probit method.
- *MaxLikeInf* calculates parameters not calculated by *Elementary* using the maximum likelihood method.
- *MaxLikeFull* calculates all parameters using the maximum likelihood method. The result from *Elementary* is used as start values.

Fatigue data handling

The `FatigueData` class

class `pylife.materialdata.woehler.`**`FatigueData`**(*pandas_obj*)

class for fatigue data

Mandatory keys are

- `load` : float, the load level
- `cycles` : float, the cycles of failure or runout
- `fracture`: bool, `True` iff the test is a runout

`conservative_fatigue_limit()`

Sets a lower fatigue limit that what is expected from the algorithm given by Mustafa Kassem. For calculating the fatigue limit, all amplitudes where runouts and fractures are present are collected. To this group, the maximum amplitude with only runouts present is added. Then, the fatigue limit is the mean of all these amplitudes.

Return type self

See also:

`Kassem, Mustafa`

property cycles

the cycle numbers

property fatigue_limit

The start value of the load endurance limit.

It is determined by searching for the lowest load level before the appearance of a runout data point, and the first load level where a runout appears. Then the median of the two load levels is the start value.

property finite_zone

All the tests with load levels above `fatigue_limit`, i.e. the finite zone

property fractured_loads

property fractures

Only the fracture tests

property infinite_zone

All the tests with load levels below `fatigue_limit`, i.e. the infinite zone

property load
The load levels

property max_runout_load

property mixed_loads

property non_fractured_loads

property num_fractures
The number of fractures

property num_runouts
The number of runouts

property num_tests
The number of tests

property runout_loads

property runouts
Only the runout tests

Analyzers

The Elementary class

class `pylife.materialdata.woehler.`**Elementary**(*fatigue_data*)
Base class to analyze SN-data.

The common base class for all SN-data analyzers calculates the first estimation of a Wöhler curve in the finite zone of the SN-data. It calculates the slope *k*, the fatigue limit *SD*, the transition cycle number *ND* and the scatter in load direction *1/TN*.

The result is just meant to be a first guess. Derived classes are supposed to use those first guesses as starting points for their specific analysis. For that they should implement the method *_specific_analysis()*.

analyze(***kwargs*)
Analyze the SN-data.

Parameters **kwargs : kwargs arguments

Arguments to be passed to the derived class

bayesian_information_criterion()
The Bayesian Information Criterion

Bayesian Information Criterion is a criterion for model selection among a finite set of models; the model with the lowest BIC is preferred. https://www.statisticshowto.datasciencecentral.com/bayesian-information-criterion/

Basically the lower the better the fit.

pearl_chain_estimator()

The `Probit` class

class `pylife.materialdata.woehler.`**`Probit`**(*fatigue_data*)

`fitter`()

The `MaxLikeInf` class

class `pylife.materialdata.woehler.`**`MaxLikeInf`**(*fatigue_data*)

The `MaxLikeFull` class

class `pylife.materialdata.woehler.`**`MaxLikeFull`**(*fatigue_data*)

The `Bayesian` class

class `pylife.materialdata.woehler.`**`Bayesian`**(*fatigue_data*)

A Wöhler analyzer using Bayesian optimization

Warning: We are considering switching from pymc3 to GPyOpt as calculation engine in the future. Maybe this will lead to breaking changes without new major release.

Helpers

The `pearl_chain` module

class `pylife.materialdata.woehler.pearl_chain.`**`PearlChainProbability`**(*fractures*, *slope*)

property normed_cycles

property normed_load

The `likelihood` module

class `pylife.materialdata.woehler.likelihood.`**`Likelihood`**(*fatigue_data*)

Calculate the likelihood a fatigue dataset matches with Wöhler curve parameters.

`likelihood_finite`(*SD*, *k_1*, *ND*, *TN*)

`likelihood_infinite`(*SD*, *TS*)

Produces the likelihood functions that are needed to compute the endurance limit and the scatter in load direction. The likelihood functions are represented by a cummalative distribution function. The likelihood function of a runout is 1-Li(fracture).

Parameters

- **SD** – Endurnace limit start value to be optimzed, unless the user fixed it.
- **TS** – The scatter in load direction 1/TS to be optimzed, unless the user fixed it.

Returns Sum of the log likelihoods. The negative value is taken since optimizers in statistical packages usually work by minimizing the result of a function. Performing the maximum likelihood estimate of a function is the same as minimizing the negative log likelihood of the function.

Return type neg_sum_lolli

likelihood_total(*SD*, *TS*, *k_1*, *ND*, *TN*)

Produces the likelihood functions that are needed to compute the parameters of the woehler curve. The likelihood functions are represented by probability and cummalative distribution functions. The likelihood function of a runout is 1-Li(fracture). The functions are added together, and the negative value is returned to the optimizer.

Parameters

- **SD** – Endurnace limit start value to be optimzed, unless the user fixed it.
- **TS** – The scatter in load direction 1/TS to be optimzed, unless the user fixed it.
- **k_1** – The slope k_1 to be optimzed, unless the user fixed it.
- **ND** – Load-cycle endurance start value to be optimzed, unless the user fixed it.
- **TN** – The scatter in load-cycle direction 1/TN to be optimzed, unless the user fixed it.

Returns Sum of the log likelihoods. The negative value is taken since optimizers in statistical packages usually work by minimizing the result of a function. Performing the maximum likelihood estimate of a function is the same as minimizing the negative log likelihood of the function.

Return type neg_sum_lolli

6.6 Mesh utilities

6.6.1 The `mesh` module

Overview

Helper to process mesh based data

Data that is distributed over a geometrical body, e.g. a stress tensor distribution on a component, is usually transported via a mesh. The meshes are a list of items (e.g. nodes or elements of a FEM mesh), each being described by the geometrical coordinates and the local data values, like for example the local stress tensor data.

In a plain mesh (see *PlainMesh*) there is no further relation between the items is known, whereas a complete FEM mesh (see *Mesh*) there is also information on the connectivity of the nodes and elements.

Examples

Read in a mesh from a vmap file:

```
>>> df = (vm = pylife.vmap.VMAPImport('demos/plate_with_hole.vmap')
            .make_mesh('1', 'STATE-2')
            .join_variable('STRESS_CAUCHY')
            .join_variable('DISPLACEMENT')
            .to_frame())
```

(continues on next page)

(continued from previous page)

```
>>> df.head()
                              x          y    z        S11        S22  S33        S12  S13 ␣
↪S23        dx        dy   dz
element_id node_id
1          1734     14.897208  5.269875  0.0  27.080811  6.927080  0.0 -13.687358  0.0 ␣
↪0.0  0.005345  0.000015  0.0
           1582     14.555333  5.355806  0.0  28.319006  1.178649  0.0 -10.732705  0.0 ␣
↪0.0  0.005285  0.000003  0.0
           1596     14.630658  4.908741  0.0  47.701195  5.512213  0.0 -17.866833  0.0 ␣
↪0.0  0.005376  0.000019  0.0
           4923     14.726271  5.312840  0.0  27.699907  4.052865  0.0 -12.210032  0.0 ␣
↪0.0  0.005315  0.000009  0.0
           4924     14.592996  5.132274  0.0  38.010101  3.345431  0.0 -14.299768  0.0 ␣
↪0.0  0.005326  0.000013  0.0
```

Get the coordinates of the mesh.

```
>>> df.plain_mesh.coordinates.head()
                              x          y    z
element_id node_id
1          1734     14.897208  5.269875  0.0
           1582     14.555333  5.355806  0.0
           1596     14.630658  4.908741  0.0
           4923     14.726271  5.312840  0.0
           4924     14.592996  5.132274  0.0
```

Now the same with a 2D mesh:

```
>>> df.drop(columns=['z']).plain_mesh.coordinates.head()
                              x          y
element_id node_id
1          1734     14.897208  5.269875
           1582     14.555333  5.355806
           1596     14.630658  4.908741
           4923     14.726271  5.312840
           4924     14.592996  5.132274
```

The signal classes

The `PlainMesh` class

class pylife.mesh.**PlainMesh**(*pandas_obj*)

DataFrame accessor to access plain 2D and 3D mesh data, i.e. without connectivity

Raises `AttributeError` – if at least one of the columns *x*, *y* is missing

Notes

The PlainMesh describes meshes whose only geometrical information is the coordinates of the nodes or elements. Unlike *Mesh* they don't know about connectivity, not even about elements and nodes.

See also:

Mesh: accesses meshes with connectivity information `pandas.api.extensions.register_dataframe_accessor()`: concept of DataFrame accessors

broadcast(*parameter*, *droplevel=[]*)

Broadcast the parameter to the object of `self`.

Parameters parameters (*scalar, numpy array or pandas object*) – The parameter to broadcast to

Returns parameter, object

Return type index aligned numerical objects

The

Examples

The behavior of the Broadcaster is best illustrated by examples:

- Broadcasting `pandas.Series` to a scalar results in a scalar and a `pandas.Series`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['foo', 'bar'], name='idx'))
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
array(5.)
```

```
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

- Broadcasting `pandas.DataFrame` to a scalar results in a `pandas.DataFrame` and a `pandas.Series`.

```
obj = pd.DataFrame({
    'foo': [1.0, 2.0],
    'bar': [3.0, 4.0]
```

(continues on next page)

(continued from previous page)

```
}, index=pd.Index([1, 2], name='idx'))
obj
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
idx
1    5.0
2    5.0
dtype: float64
```

```
obj
```

- Broadcasting `pandas.DataFrame` to a `pandas.Series` results in a `pandas.DataFrame` and a `pandas.Series`, **if and only if** the index name of the object is `None`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['tau', 'chi']))
obj
```

```
tau    1.0
chi    2.0
dtype: float64
```

```
parameter = pd.Series([3.0, 4.0], index=pd.Index(['foo', 'bar'], name='idx
↪'))
parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(parameter)

parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
obj
```

property coordinates

Returns the coordinate colums of the accessed DataFrame

Returns coordinates – The coordinates x, y and if 3D z of the accessed mesh

Return type pandas.DataFrame

property dimensions

The dimensions of the mesh (2 for 2D and 3 for 3D)

Note: If all the coordinates in z-direction are equal the mesh is considered 2D.

fail_if_key_missing(*keys_to_check*, *msg=None*)

Raise an exception if any key is missing in a self._obj object.

Parameters

- **self._obj** (*pandas.DataFrame or pandas.Series*) – The object to be checked
- **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Raises

- AttributeError – if *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*
- AttributeError – if any of the keys is not found in the self._obj's keys.

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* meed to be found in the *self._obj.columns*.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* meed to be found in the *self._obj.index*.

See also:

get_missing_keys(), stresssignal.StressTensorVoigt

classmethod from_parameters(***kwargs*)

Make a signal instance from a parameter set.

This is a convenience function to instantiate a signal from individual parameters rather than pandas objects.

A signal class like

```
@pd.api.extensions.register_dataframe_accessor('foo_signal')
class FooSignal(PylifeSignal):
    pass
```

The following two blocks are equivalent:

```
pd.Series({'foo': 1.0, 'bar': 2.0}).foo_signal
```

```
FooSignal.from_parameters(foo=1.0, bar=1.0)
```

get_missing_keys(*keys_to_check*)

Get a list of missing keys that are needed for a self._obj object.

Parameters **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Returns **missing_keys** – a list of missing keys

Return type list

Raises AttributeError – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* not found in the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* not found in the *self._obj.index* are returned.

`keys()`
Get a list of missing keys that are needed for a signal object.

Returns **keys** – a pandas index of keys

Return type pd.Index

Raises `AttributeError` – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, the *self._obj.index* are returned.

`to_pandas()`
Expose the pandas object of the signal.

Returns **pandas_object** – The pandas object representing the signal

Return type pd.DataFrame or pd.Series

Notes

The default implementation just returns the object given when instantiating the signal class. Derived classes may return a modified object or augmented, if they store some extra information.

By default the object is **not** copied. So make a copy yourself, if you intent to modify it.

The `Mesh` class

`class pylife.mesh.`**`Mesh`**`(`*pandas_obj*`)`
DataFrame accessor to access FEM mesh data (2D and 3D)

Raises

- `AttributeError` – if at least one of the columns *x*, *y* is missing
- `AttributeError` – if the index of the DataFrame is not a two level MultiIndex with the names *node_id* and *element_id*

Notes

The Mesh describes how we expect FEM data to look like. It consists of nodes identified by *node_id* and elements identified by *element_id*. A node playing a role in several elements and an element consists of several nodes. So in the DataFrame a *node_id* can appear multiple times (for each element, the node is playing a role in). Likewise each *element_id* appears multiple times (for each node the element consists of).

The combination *node_id*:*element_id* however, is unique. So the table is indexed by a `pandas.MultiIndex` with the level names *node_id*, *element_id*.

See also:

`PlainMesh`: accesses meshes without connectivity information `pandas.api.extensions.register_dataframe_accessor()`: concept of DataFrame accessors

Examples

For an example see `hotspot`.

`broadcast`(*parameter*, *droplevel=[]*)

Broadcast the parameter to the object of `self`.

Parameters **`parameters`** (*`scalar, numpy array or pandas object`*) – The parameter to broadcast to

Returns **parameter, object**

Return type index aligned numerical objects

The

Examples

The behavior of the Broadcaster is best illustrated by examples:

- Broadcasting `pandas.Series` to a scalar results in a scalar and a `pandas.Series`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['foo', 'bar'], name='idx'))
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
array(5.)
```

```
obj
```

```
idx
foo    1.0
bar    2.0
dtype: float64
```

- Broadcasting `pandas.DataFrame` to a scalar results in a `pandas.DataFrame` and a `pandas.Series`.

```
obj = pd.DataFrame({
    'foo': [1.0, 2.0],
    'bar': [3.0, 4.0]
}, index=pd.Index([1, 2], name='idx'))
obj
```

```
parameter, obj = Broadcaster(obj).broadcast(5.0)

parameter
```

```
idx
1    5.0
2    5.0
dtype: float64
```

```
obj
```

- Broadcasting `pandas.DataFrame` to a a `pandas.Series` results in a `pandas.DataFrame` and a `pandas.Series`, **if and only if** the index name of the object is `None`.

```
obj = pd.Series([1.0, 2.0], index=pd.Index(['tau', 'chi']))
obj
```

```
tau    1.0
chi    2.0
dtype: float64
```

```
parameter = pd.Series([3.0, 4.0], index=pd.Index(['foo', 'bar'], name='idx
↪'))
parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
parameter, obj = Broadcaster(obj).broadcast(parameter)

parameter
```

```
idx
foo    3.0
bar    4.0
dtype: float64
```

```
obj
```

property connectivity
The connectivity of the mesh.

property coordinates
Returns the coordinate colums of the accessed DataFrame

Returns **coordinates** – The coordinates *x*, *y* and if 3D *z* of the accessed mesh

Return type pandas.DataFrame

property dimensions
The dimensions of the mesh (2 for 2D and 3 for 3D)

Note: If all the coordinates in z-direction are equal the mesh is considered 2D.

fail_if_key_missing(*keys_to_check*, *msg=None*)
Raise an exception if any key is missing in a self._obj object.

Parameters

- **self._obj** (*pandas.DataFrame or pandas.Series*) – The object to be checked
- **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Raises

- `AttributeError` – if *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*
- `AttributeError` – if any of the keys is not found in the self._obj's keys.

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* meed to be found in the *self._obj.columns*.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* meed to be found in the *self._obj.index*.

See also:

get_missing_keys(), `stresssignal.StressTensorVoigt`

classmethod from_parameters(***kwargs*)
Make a signal instance from a parameter set.

This is a convenience function to instantiate a signal from individual parameters rather than pandas objects.

A signal class like

```
@pd.api.extensions.register_dataframe_accessor('foo_signal')
class FooSignal(PylifeSignal):
    pass
```

The following two blocks are equivalent:

```
pd.Series({'foo': 1.0, 'bar': 2.0}).foo_signal
```

```
FooSignal.from_parameters(foo=1.0, bar=1.0)
```

`get_missing_keys(keys_to_check)`

Get a list of missing keys that are needed for a self._obj object.

Parameters **keys_to_check** (*list*) – A list of keys that need to be available in *self._obj*

Returns **missing_keys** – a list of missing keys

Return type list

Raises `AttributeError` – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, all keys of *keys_to_check* not found in the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, all keys of *keys_to_check* not found in the *self._obj.index* are returned.

`keys()`

Get a list of missing keys that are needed for a signal object.

Returns **keys** – a pandas index of keys

Return type pd.Index

Raises `AttributeError` – If *self._obj* is neither a *pandas.DataFrame* nor a *pandas.Series*

Notes

If *self._obj* is a *pandas.DataFrame*, the *self._obj.columns* are returned.

If *self._obj* is a *pandas.Series*, the *self._obj.index* are returned.

`to_pandas()`

Expose the pandas object of the signal.

Returns **pandas_object** – The pandas object representing the signal

Return type pd.DataFrame or pd.Series

Notes

The default implementation just returns the object given when instantiating the signal class. Derived classes may return a modified object or augmented, if they store some extra information.

By default the object is **not** copied. So make a copy yourself, if you intent to modify it.

`vtk_data()`

Make VTK data structure easily plot the mesh with pyVista.

Returns

- **offsets** (*ndarray*) – An empty numpy array as `pyVista.UnstructuredGrid()` still demands the argument for the offsets, even though VTK>9 does not accept it.
- **cells** (*ndarray*) – The location of the cells describing the points in a way `pyVista.UnstructuredGrid()` needs it
- **cell_types** (*ndarray*) – The VTK code for the cell types (see https://github.com/Kitware/VTK/blob/master/Common/DataModel/vtkCellType.h)
- **points** (*ndarray*) – The coordinates of the cell points

Notes

This is a convenience function to easily plot a 3D mesh with pyVista. It prepares a data structure which can be passed to `pyVista.UnstructuredGrid()`

Example

```
>>> import pyvista as pv
>>> grid = pv.UnstructuredGrid(*our_mesh.mesh.vtk_data())
>>> plotter = pv.Plotter(window_size=[1920, 1080])
>>> plotter.add_mesh(grid, scalars=our_mesh.groupby('element_id')['val'].mean().
↪to_numpy())
>>> plotter.show()
```

Note the * that needs to be added when calling `pv.UnstructuredGrid()`.

6.6.2 The `HotSpot` class

class `pylife.mesh.`**`HotSpot`**(*pandas_obj*)

`calc`(*value_key*, *limit_frac=0.9*, *artefact_threshold=None*)

Calculates hotspots on a FE mesh

Parameters

- **`value_key`** (*`string`*) – Column name of the field variable, on which the Hot Spot calculation is done.
- **`limit_frac`** (*`float`*, *`optional`*) – Fraction of the max field variable. Example: If you set limit_frac = 0.9, the function finds all nodes and regions which are >= 90% of the maximum value of the field variable. default: 0.9
- **`artefact_threshold`** (*`float`*, *`optional`*) – If set all the values above the *artefact_threshold* limit are not taken into account for the calculation of the maximum value. This is meant to be used for numerical artefacts which would take the threshold value for hotspot determined by *limit_frac* to such a high level, that all the relevant hotspots would “hide” underneath it.

Returns **hotspots** – A Series of integers with the same index of the accessed mesh object indicating which mesh point belongs to which hotspot. A value 0 means below the *limit_frac*.

Return type pandas.Series

Notes

A loop is defined in the following way:

- Select the node with the maximum stress value
- Find all elements > *limit_frac* belonging to this node
- Select all nodes > *limit_frac* belonging to these elements
- Start loop again until all nodes > *limit_frac* are assigned to a hotspot

Attention: All stress values are node based, not integration point based

6.6.3 The `Gradient` class

class `pylife.mesh.`**`Gradient`**(*pandas_obj*)

Computes the gradient of a value in a 3D mesh

Accesses a *mesh* registered in `meshsignal`

Raises

- `AttributeError` – if at least one of the columns *x*, *y* is missing
- `AttributeError` – if the index of the DataFrame is not a two level MultiIndex with the names *node_id* and *element_id*

Notes

The gradient is calculated by fitting a plane into the nodes of each coordinate and the neighbor nodes using least square fitting.

The method is described in a thread on stackoverflow.

`gradient_of`(*value_key*)

returns the gradient

Parameters **`value_key`** (*`str`*) – The key of the value that forms the gradient. Needs to be found in `df`

Returns **gradient** – A table describing the gradient indexed by `node_id`. The keys for the components of the gradients are `['d{value_key}_dx', 'd{value_key}_dy', 'd{value_key}_dz']`.

Return type pd.DataFrame

6.6.4 The `Meshmapper` class

class `pylife.mesh.`**`Meshmapper`**(*pandas_obj*)

Mapper to map points of one mesh to another

Notes

The accessed DataFrame needs to be accessible by a *`PlainMesh`*.

`process`(*from_df*, *value_key*, *method='linear'*)

Performs the mapping

Parameters **`from_df`** (pandas.DataFrame accessible by a *`PlainMesh`*.) – The DataFrame that is to be mapped to the accessed one. Needs to have the same dimensions (2D or 3D) as the accessed one

6.7 VMAP Interface

6.7.1 VMAP interface for pyLife

VMAP *is a vendor-neutral standard for CAE data storage to enhance interoperability in virtual engineering workflows.*

As for now, the VMAP support in pyLife is in an experimental stage. That is mainly because there are not many other software packages available that do support it. Nevertheless we would like to encourage the usage of VMAP, so we decided to put the experimental code into the release.

pyLife supports a growing subset of the VMAP standard. That means that only features relevant for pyLife's addressed real life use cases are or will be implemented. Probably there are features missing, that are important for some valid use cases. In that case please file a feature request at https://github.com/boschresearch/pylife/issues

6.7.2 Reading a VMAP file

The most common use case is to get the element nodal stress tensor for a certain geometry 1 and a certain load state STATE-2 out of the vmap file. The vmap interface provides you the nodal geometry (node coordinates), the mesh connectivity index and the field variables.

You can retrieve a DataFrame of the mesh with the desired variables in just one statement.

```
>>> (pylife.vmap.VMAPImport('demos/plate_with_hole.vmap')
     .make_mesh('1', 'STATE-2')
     .join_coordinates()
     .join_variable('STRESS_CAUCHY')
     .join_variable('E')
     .to_frame())
                              x         y    z        S11       S22  S33        S12  S13 ␣
↪S23       E11       E22  E33       E12  E13  E23
element_id node_id
1          1734     14.897208  5.269875  0.0  27.080811  6.927080  0.0 -13.687358  0.0 ␣
↪0.0  0.000119 -0.000006  0.0 -0.000169  0.0  0.0
           1582     14.555333  5.355806  0.0  28.319006  1.178649  0.0 -10.732705  0.0 ␣
↪0.0  0.000133 -0.000035  0.0 -0.000133  0.0  0.0
           1596     14.630658  4.908741  0.0  47.701195  5.512213  0.0 -17.866833  0.0 ␣
↪0.0  0.000219 -0.000042  0.0 -0.000221  0.0  0.0
           4923     14.726271  5.312840  0.0  27.699907  4.052865  0.0 -12.210032  0.0 ␣
↪0.0  0.000126 -0.000020  0.0 -0.000151  0.0  0.0
           4924     14.592996  5.132274  0.0  38.010101  3.345431  0.0 -14.299768  0.0 ␣
↪0.0  0.000176 -0.000038  0.0 -0.000177  0.0  0.0
...                       ...       ...  ...        ...       ...  ...        ...  ...  .
↪..        ...       ...  ...       ...  ...  ...
4770       3812    -13.189782 -5.691876  0.0  36.527439  2.470588  0.0 -14.706686  0.0 ␣
↪0.0  0.000170 -0.000040  0.0 -0.000182  0.0  0.0
           12418   -13.560289 -5.278386  0.0  32.868889  3.320898  0.0 -14.260107  0.0 ␣
↪0.0  0.000152 -0.000031  0.0 -0.000177  0.0  0.0
           14446   -13.673285 -5.569107  0.0  34.291058  3.642457  0.0 -13.836027  0.0 ␣
↪0.0  0.000158 -0.000032  0.0 -0.000171  0.0  0.0
           14614   -13.389065 -5.709927  0.0  36.063541  2.828889  0.0 -13.774759  0.0 ␣
↪0.0  0.000168 -0.000038  0.0 -0.000171  0.0  0.0
           14534   -13.276068 -5.419206  0.0  33.804211  2.829817  0.0 -14.580153  0.0 ␣
↪0.0  0.000157 -0.000035  0.0 -0.000181  0.0  0.0
```

[37884 rows x 15 columns]

Supported features

So far the following data can be read from a vmap file

Geometry

- node positions
- node element index

Field variables

Any field variables can be read and joined to the node element index from the following locations:

- element
- node
- element nodal

In particular, field variables at integration point location *cannot* cannot be read, as that would require extrapolating them to the node positions. This functionality is not available in pyLife.

The VMAPImport Class

`class pylife.vmap.`**`VMAPImport`**`(`*filename*`)`

The interface class to import a vmap file

Parameters **`filename`** (*`string`*) – The path to the vmap file to be read

Raises `Exception` – If the file cannot be read an exception is raised. So far any exception from the `h5py` module is passed through.

`element_sets``(`*geometry*`)`

Returns a list of the element_sets present in the vmap file

`filter_element_set``(`*element_set*`)`

Filters a node set out of the current mesh

Parameters **`element_set`** (*`string, optional`*) – The element set defined in the vmap file as geometry set

Return type self

Raises `APIUseError` – If the mesh has not been initialized using `make_mesh()`

`filter_node_set``(`*node_set*`)`

Filters a node set out of the current mesh

Parameters **`node_set`** (*`string`*) – The node set defined in the vmap file as geometry set

Return type self

Raises `APIUseError` – If the mesh has not been initialized using `make_mesh()`

`geometries``()`

Returns a list of geometry strings of geometries present in the vmap data

join_coordinates()

Join the coordinates of the predefined geometry in the mesh

Return type self

Raises **APIUseError** – If the mesh has not been initialized using `make_mesh()`

Examples

Receive the mesh with the node coordinates

```
>>> pylife.vmap.VMAPImport('demos/plate_with_hole.vmap').make_mesh('1').join_
↪coordinates().to_frame()
                            x         y    z
element_id node_id
1          1734      14.897208  5.269875  0.0
           1582      14.555333  5.355806  0.0
           1596      14.630658  4.908741  0.0
           4923      14.726271  5.312840  0.0
           4924      14.592996  5.132274  0.0
...                        ...       ...  ...
4770       3812     -13.189782 -5.691876  0.0
           12418    -13.560289 -5.278386  0.0
           14446    -13.673285 -5.569107  0.0
           14614    -13.389065 -5.709927  0.0
           14534    -13.276068 -5.419206  0.0
```

[37884 rows x 3 columns]

join_variable(*var_name*, *state=None*, *column_names=None*)

Joins a field output variable to the mesh

Parameters

- **var_name** (*string*) – The name of the field variables
- **state** (*string, opional*) – The load state of which the field variable is to be read If not given, the last defined state, either defined in `make_mesh()` or defeined in `join_variable()` is used.
- **column_names** (*list of string, optional*) – The names of the columns names to be used in the DataFrame If not provided, it will be chosen according to the list shown below. The length of the list must match the dimension of the variable.

Return type self

Raises

- **APIUseError** – if the mesh has not been initialized using `make_mesh()`
- **KeyError** – if the geometry, state or varname is not found of if the vmap file is corrupted
- **KeyError** – if there are no column names given and known for the variable.
- **ValueError** – if the length of the column_names does not match the dimension of the variable

Notes

The mesh must be initialized with `make_mesh()`. The final DataFrame can be retrieved with `to_frame()`.

If the `column_names` argument is not provided the following column names are chosen

- 'DISPLACEMENT': `['dx', 'dy', 'dz']`
- 'STRESS_CAUCHY': `['S11', 'S22', 'S33', 'S12', 'S13', 'S23']`
- 'E': `['E11', 'E22', 'E33', 'E12', 'E13', 'E23']`

If that fails a `KeyError` exception is risen.

Examples

Receiving the 'DISPLACEMENT' of 'STATE-1' , the stress and strain tensors of 'STATE-2'

```
>>> (pylife.vmap.VMAPImport('demos/plate_with_hole.vmap')
     .make_mesh('1')
     .join_variable('DISPLACEMENT', 'STATE-1')
     .join_variable('STRESS_CAUCHY', 'STATE-2')
     .join_variable('E').to_frame())
                       dx   dy   dz        S11       S22  S33        S12  S13 ␣
↪S23       E11       E22  E33       E12  E13  E23
element_id node_id
1          1734      0.0  0.0  0.0  27.080811  6.927080  0.0 -13.687358  0.0  0.
↪0  0.000119 -0.000006  0.0 -0.000169  0.0  0.0
           1582      0.0  0.0  0.0  28.319006  1.178649  0.0 -10.732705  0.0  0.
↪0  0.000133 -0.000035  0.0 -0.000133  0.0  0.0
           1596      0.0  0.0  0.0  47.701195  5.512213  0.0 -17.866833  0.0  0.
↪0  0.000219 -0.000042  0.0 -0.000221  0.0  0.0
           4923      0.0  0.0  0.0  27.699907  4.052865  0.0 -12.210032  0.0  0.
↪0  0.000126 -0.000020  0.0 -0.000151  0.0  0.0
           4924      0.0  0.0  0.0  38.010101  3.345431  0.0 -14.299768  0.0  0.
↪0  0.000176 -0.000038  0.0 -0.000177  0.0  0.0
...                  ...  ...  ...        ...       ...  ...        ...  ...  ...
↪       ...       ...  ...       ...  ...  ...
4770       3812      0.0  0.0  0.0  36.527439  2.470588  0.0 -14.706686  0.0  0.
↪0  0.000170 -0.000040  0.0 -0.000182  0.0  0.0
           12418     0.0  0.0  0.0  32.868889  3.320898  0.0 -14.260107  0.0  0.
↪0  0.000152 -0.000031  0.0 -0.000177  0.0  0.0
           14446     0.0  0.0  0.0  34.291058  3.642457  0.0 -13.836027  0.0  0.
↪0  0.000158 -0.000032  0.0 -0.000171  0.0  0.0
           14614     0.0  0.0  0.0  36.063541  2.828889  0.0 -13.774759  0.0  0.
↪0  0.000168 -0.000038  0.0 -0.000171  0.0  0.0
           14534     0.0  0.0  0.0  33.804211  2.829817  0.0 -14.580153  0.0  0.
↪0  0.000157 -0.000035  0.0 -0.000181  0.0  0.0
```

[37884 rows x 15 columns]

Todo: Write a more central document about pyLife's column names.

make_mesh(*geometry*, *state=None*)
Makes the initial mesh

Parameters

- **geometry** (*string*) – The geometry defined in the vmap file
- **state** (*string, optional*) – The load state of which the field variable is to be read. If not given, the state must be defined in `join_variable()`.

Return type self

Raises

- `KeyError` – if the `geometry` is not found of if the vmap file is corrupted
- `KeyError` – if the `node_set` or `element_set` is not found in the geometry.
- **APIUseError** – if both, a `node_set` and an `element_set` are given

Notes

This methods defines the initial mesh to which coordinate data can be joined by `join_coordinates()` and field variables can be joined by `join_variable()`

Examples

Get the mesh data with the coordinates of geometry '1' and the stress tensor of 'STATE-2'

```
>>> (pylife.vmap.VMAPImport('demos/plate_with_hole.vmap')
     .make_mesh('1', 'STATE-2')
     .join_coordinates()
     .join_variable('STRESS_CAUCHY')
     .to_frame()
                                x          y    z        S11        S22  S33        ␣
↪S12  S13  S23
element_id node_id
1          1734      14.897208  5.269875  0.0  27.080811  6.927080  0.0 -13.
↪687358  0.0  0.0
           1582      14.555333  5.355806  0.0  28.319006  1.178649  0.0 -10.
↪732705  0.0  0.0
           1596      14.630658  4.908741  0.0  47.701195  5.512213  0.0 -17.
↪866833  0.0  0.0
           4923      14.726271  5.312840  0.0  27.699907  4.052865  0.0 -12.
↪210032  0.0  0.0
           4924      14.592996  5.132274  0.0  38.010101  3.345431  0.0 -14.
↪299768  0.0  0.0
...                        ...       ...  ...        ...       ...  ...        ..
↪.  ...  ...
4770       3812     -13.189782 -5.691876  0.0  36.527439  2.470588  0.0 -14.
↪706686  0.0  0.0
           12418    -13.560289 -5.278386  0.0  32.868889  3.320898  0.0 -14.
↪260107  0.0  0.0
           14446    -13.673285 -5.569107  0.0  34.291058  3.642457  0.0 -13.
↪836027  0.0  0.0
           14614    -13.389065 -5.709927  0.0  36.063541  2.828889  0.0 -13.
↪774759  0.0  0.0
           14534    -13.276068 -5.419206  0.0  33.804211  2.829817  0.0 -14.
↪580153  0.0  0.0
```

`node_sets`(*geometry*)
Returns a list of the node_sets present in the vmap file

`nodes`(*geometry*)
Retrieves the node positions

Parameters **geometry** (*string*) – The geometry defined in the vmap file

Returns **node_positions** – a DataFrame with the node numbers as index and the columns 'x', 'y' and 'z' for the node coordinates.

Return type DataFrame

Raises `KeyError` – if the geometry is not found of if the vmap file is corrupted

`states`()
Returns a list of state strings of states present in the vmap data

`to_frame`()
Returns the mesh and resets the mesh

Returns **mesh** – The mesh data joined so far

Return type DataFrame

Raises **APIUseError** – if there is no mesh present, i.e. make_mesh() has not been called yet or the mesh has been reset in the meantime.

Notes

This method resets the mesh, i.e. `make_mesh()` must be called again in order to fetch more mesh data in another mesh.

`try_get_geometry_set`(*geometry_name*, *geometry_set_name*)

`try_get_vmap_object`(*group_full_path*)

`variables`(*geometry*, *state*)
Ask for available variables for a certain geometry and state.

Parameters

- **geometry** (*string*) – Name of the geometry
- **state** (*string*) – Name of the state

Returns **variables** – List of available variable names for the geometry state combination

Return type list

Raises `KeyError` – if the geometry state combination is not available.

6.7.3 Writing a VMAP file

The VMAPExport Class

`class pylife.vmap.VMAPExport`(*file_name*)
The interface class to export a vmap file

Parameters **file_name** (*string*) – The path to the vmap file to be read

Raises `Exception` – If the file cannot be read an exception is raised. So far any exception from the h5py module is passed through.

`add_element_set`(*geometry_name*, *indices*, *mesh*, *name=None*)
Exports element-type geometry set into given geometry

Parameters

- **geometry_name** (*string*) – The geometry to where we want to export the geometry set
- **indices** (*Pandas Index*) – List of node indices that we want to export
- **mesh** (*Pandas DataFrame*) – The Data Frame that holds the data of the mesh to export
- **name** (*value of attribute MYSETNAME*) –

Return type self

`add_geometry`(*geometry_name*, *mesh*)
Exports geometry with given name and mesh data

Parameters

- **geometry_name** (*string*) – Name of the geometry to add
- **mesh** (*Pandas DataFrame*) – The Data Frame that holds the data of the mesh to export

Return type self

`add_integration_types`(*content*)
Creates system dataset IntegrationTypes with the given content

Parameters **content** (*the content of the dataset*) –

Return type self

`add_node_set`(*geometry_name*, *indices*, *mesh*, *name=None*)
Exports node-type geometry set into given geometry

Parameters

- **geometry_name** (*string`*) – The geometry to where we want to export the geometry set
- **indices** (*Pandas Index*) – List of node indices that we want to export
- **mesh** (*Pandas DataFrame*) – The Data Frame that holds the data of the mesh to export
- **name** (*value of attribute MYSETNAME*) –

Return type self

`add_variable`(*state_name*, *geometry_name*, *variable_name*, *mesh*, *column_names=None*, *location=None*)
Exports variable into given state and geometry

Parameters

- **state_name** (*string*) – State where we want to export the parameter
- **geometry_name** (*string*) – Geometry where we want to export the parameter
- **variable_name** (*string*) – The name of the variable to export
- **mesh** (*Pandas DataFrame*) – The Data Frame that holds the data of the mesh to export
- **column_names** (*List, optional*) – The columns that the parameter consists of
- **location** (*Enum, optional*) – The location of the parameter * 2 - node * 3 - element - not supported yet * 6 - element nodal

Return type self

property file_name
Gets the name of the VMAP file that we are exporting

set_group_attribute(*object_path*, *key*, *value*)
Sets the 'MYNAME' attribute of the VMAP objects

Parameters

- **object_path** (*string*) – The full path to the object that we want to rename
- **key** (*string*) – The key of the attribute that we want to set
- **value** (*np.dtype*) – The value that we want to set to the attribute

Return type

-

variable_column_names(*parameter_name*)
Gets the column names that the given parameter consists of

Parameters **parameter_name** (*string*) – The name of the parameter

Return type The column names of the given parameter in the mesh

variable_location(*parameter_name*)
Gets the location of the given parameter

Parameters **parameter_name** (*string*) – The name of the parameter

Return type The location of the given parameter

6.8 Utils

6.8.1 The `utils.functions` module

Utility Functions

A collection of functions frequently used in lifetime estimation business.

pylife.utils.functions.**rossow_cumfreqs**(*N*)
Cumulative frequency estimator according to Rossow.

Parameters **N** (*int*) – The sample size of the statistical population

Returns **cumfreqs** – The estimated cumulated frequencies of the N samples

Return type numpy.ndarray

Notes

The returned value is the probability that the next taken sample is below the value of the i-th sample of n sorted samples.

Examples

```
>>> rossow_cumfreqs(1)
array([0.5])
```

If we have one sample, the probability that the next sample will be below it is 0.5.

```
>>> rossow_cumfreqs(3)
array([0.2, 0.5, 0.8])
```

If we have three sorted samples, the probability that the next sample will be * below the first is 0.2 * below the second is 0.5 * below the third is 0.8

References

'Statistics of Metal Fatigue in Engineering' page 16

https://books.google.de/books?isbn=3752857722

`pylife.utils.functions.`**`scattering_range_to_std`**(*T*)

Convert a scattering range (*TS* or *TN* in DIN 50100:2016-12) into standard deviation.

Parameters **T** (*float*) – inverted scattering range

Returns **std** – standard deviation corresponding to TS or TN assuming a normal distribution

Return type float

Notes

Actually *1/(2*norm.ppf(0.9))*np.log10(T)*

Inverse of *std_to_scattering_range()*

`pylife.utils.functions.`**`std_to_scattering_range`**(*std*)

Convert a standard deviation into scattering range (*TS* or *TN* in DIN 50100:2016-12).

Parameters **std** (*float*) – standard deviation

Returns **T** – inverted scattering range corresponding to *std* assuming a normal distribution

Return type float

Notes

Actually *10**(2*norm.ppf(0.9)*std*

Inverse of *scattering_range_to_std()*

6.8.2 The `histogram` module

`pylife.utils.histogram.`**`combine_histogram`**(*hist_list*, *method='sum'*)

Combine a list of histograms to one.

Parameters

- **`hist_list`** (*`list`*) – list of histograms with all histograms as interval indexed `pandas.Series`
- **`method`** (*`str or aggregating function`*) – method used for the aggregation, e.g. 'sum', 'min', 'max', 'mean', 'std' default is 'sum'

Returns **histogram** – The resulting histogram

Return type pd.Series

`pylife.utils.histogram.`**`rebin_histogram`**(*histogram*, *binning*, *nan_default=False*)

Rebin a histogram to a given binning.

Parameters

- **`histogram`** (`pandas.Series` with `pandas.IntervalIndex`) – The histogram data to be rebinned
- **`binning`** (`pandas.IntervalIndex` or int) – The given binning or number of bins
- **`nan_default`** (*`bool`*) – If True non occupied bins will be occupied with `np.nan`, else 0.0 Default False

Returns **rebinned** – The rebinned histogram

Return type `pandas.Series` with `pandas.IntervalIndex`

:raises TypeError : if the `histogram` or the `binning` do not have an `IntervalIndex`.: :raises ValueError : if the binning is not monotonic increasing or has gaps.:

6.8.3 The `utils.probability_data` modubule

`class pylife.utils.probability_data.`**`ProbabilityFit`**(*probs*, *occurrences*)

`property intercept`

`property occurrences`

`property percentiles`

`property slope`

CHAPTER

SEVEN

WHAT IS NEW AND WHAT HAS CHANGED IN PYLIFE-2.0

The pyLife-2.0 release is planned for end of 2021. No promises, though. This document lists changes and new features of pyLife-2.0.

7.1 General changes

In pyLife-1.x the individual modules often where not playing together really well, sometimes we had even the same concept multiple times. For the pyLife-2.0 release we aim to improve that. The concept of the accessor classes will be used more extensively.

7.2 New features

7.2.1 Rainflow counting

The *rainflow counting module* has been vastly redesigned in order to get more flexibility. New possibilities are:

- *Four point rainflow counting*
- Recording of the hysteresis loop information is in a separate class to allow the recording in a customized way.

See docs of the *rainflow counting module* for details.

7.3 Restructuring the code

We are now using PyScaffold to handle the packaging files. That's why we have restructured the code base. Basically the only notable things that have changed is that all the code has been moved from `pylife` to `src/pylife` and the documentation has been moved from `doc/source` to `docs`. Both are the common locations for Python 3.x packages.

7.4 Changes that affect your code

- Strength scattering is now stored as `TS` and `TN`, no longer by `1/TS` and `1/TN`. This only concerns the naming, the underlying values are still the same. With this we are following the newer conventions in DIN 50100:2016-12.
- `self._validate()` is no longer called with arguments. The arguments `obj` and `validator` are no longer needed. `obj` is now accessible by `self._obj`. The methods of `DataValidator` are now accessible as methods of `PylifeSignal` directly.
- Signal accessor class names are no longer suffixed with `Accessor`
- The `PyLifeSignal` is promoted to the toplevel of the `pylife` package. That means that you have to change

```
from pylife import signal

...

class Foo(signal.PylifeSignal):
    ...
```

  to

```
from pylife import PylifeSignal

...

class Foo(PylifeSignal):
    ...
```

- The name of a rainflow matrix series is no longer `frequency` but `cycles`.
- The names of the functions `scatteringRange2std` and `std2scatteringRange` have been adjusted to the naming conventions and are now `scattering_range_to_std` and `std_to_scattering_range`.
- The accessor class `CyclicStress` with the accessor `cyclic_stress` is gone. Use `pylife.LoadCollective` instead.

7.5 Variable names

Currently we are brainstorming on guidelines about variable names. See the article *in the docs* about it. It will be continuously updated.

CHAPTER

EIGHT

CONTRIBUTING

Want to contribute? Great! You can do so through the standard GitHub pull request model. For large contributions we do encourage you to file a ticket in the GitHub issues tracking system prior to any code development to coordinate with the pyLife development team early in the process. Coordinating up front helps to avoid frustration later on.

8.1 Test driven development

The functionality of your contribution (functions, class methods) need to be tested by pytest testing routines.

In order to achieve maintainable code we ask contributors to use test driven development, i. e. follow the Three Rules of Test Driven Development:

1. Do not change production code without writing a failing unit test first. Cleanups and refactorings are not changes in that sense.
2. Write only enough test code as is sufficient to fail.
3. Only write or change minimal production code as is sufficient to make the failing test pass.

We are measuring the testing coverage. Your pull request should not decrease the test coverage.

8.2 Coding style

Please do consult the *CODINGSTYLE* file for codingstyle guide lines. In order to have your contribution merged to main line following guide lines should be met.

8.2.1 Docstrings

Document your public API classes, methods, functions and attributes using numpy style docstings unless the naming is *really* self-explanatory.

8.2.2 Comments

Use as little comments as possible. The code along with docstrings should be expressive enough. Remove any commented code lines before issuing your pull request.

8.3 Making commits

8.3.1 Configure your git client

Please configure your identity in your git client appropriately. From the git command line you can do that using

```
git config user.name <Your Name>
git config user.email <your-email@...>
```

8.3.2 Writing good commit messages

Please consider following the commit guidelines when writing your commit message. We will not enforce this, but we would appreciate if you do. Here is a good read why this makes sense.

8.4 Branching and pull requests

Pull requests must be filed against the `develop` branch, except for urgent bugfixes requiring a special bugfix release. Those can be filed against `master`.

Branches should have meaningful names and whenever it makes sense use one of the following prefixes.

- `bugfix/` for bugfixes, that do not change the API
- `feature/` if a new feature is added
- `doc/` if documentation is added or improved
- `cleanup/` if code is cleaned or refactored without changing the behavior

If your branch does not fit any of those, you can also come up with another appropriate prefix.

8.5 License

Your contribution must be licensed under the Apache-2.0 license, the license used by this project.

8.6 Add / retain copyright notices

Include a copyright notice and license in each new file to be contributed, consistent with the style used by this project. If your contribution contains code under the copyright of a third party, document its origin, license, and copyright holders.

8.7 Sign your work

This project tracks patch provenance and licensing using the Developer Certificate of Origin 1.1 (DCO) from developercertificate.org and Signed-off-by tags initially developed by the Linux kernel project.

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

With the sign-off in a commit message you certify that you authored the patch or otherwise have the right to submit it under an open source license. The procedure is simple: To certify above Developer's Certificate of Origin 1.1 for your contribution just append a line

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

to every commit message using your real name or your pseudonym and a valid email address.

If you have set your `user.name` and `user.email` git configs you can automatically sign the commit by running the git-commit command with the `-s` option. There may be multiple sign-offs if more than one developer was involved in authoring the contribution.

Another option to automatically add the `Signed-off-by:` is to once use the command

```
git config core.hooksPath .githooks
```

in your pyLife working directory. This will then add the `Signed-off-by:` line automatically.

For a more detailed description of this procedure, please see SubmittingPatches which was extracted from the Linux kernel project, and which is stored in an external repository.

8.7.1 Individual vs. Corporate Contributors

Often employers or academic institution have ownership over code that is written in certain circumstances, so please do due diligence to ensure that you have the right to submit the code.

If you are a developer who is authorized to contribute to pyLife on behalf of your employer, then please use your corporate email address in the Signed-off-by tag. Otherwise please use a personal email address.

8.8 Maintain Copyright holder / Contributor list

Each contributor is responsible for identifying themselves in the *NOTICE* file, the project's list of copyright holders and authors. Please add the respective information corresponding to the Signed-off-by tag as part of your first pull request.

If you are a developer who is authorized to contribute to pyLife on behalf of your employer, then add your company / organization to the list of copyright holders in the *NOTICE* file. As author of a corporate contribution you can also add your name and corporate email address as in the Signed-off-by tag.

If your contribution is covered by this project's DCO's clause "(c) The contribution was provided directly to me by some other person who certified (a) or (b) and I have not modified it", please add the appropriate copyright holder(s) to the *NOTICE* file as part of your contribution.

CHAPTER

NINE

PYLIFE CODING STYLE GUIDELINES

9.1 Introduction

One crucial quality criteria of program code is maintainability. In order to maintain code, the code has to be written clearly so that it is easily readable to someone who has not written it. Therefore it is helpful to have a consistent coding style with consistent naming conventions. However, the coding style rules are not supposed to be strict rules. They can be disobeyed if there are good reasons to do so.

As we are programming in Python we vastly stick to the [PEP8 coding style guide][1]. That document is generally recommendable to python programmers. This document therefore covers only things that go beyond the PEP8. So please read PEP8 for the general recommendations on python programming.

9.1.1 Clean code

The notion of code quality that keeps software maintainable, makes it easier to find and fix bugs and so on is nowadays referred to by the expression *Clean Code*.

The iconic figure behind that notion is Robert C. Martin aka Uncle Bob. For the full story about clean code you can read his books *Clean Code* and *Clean Coders*. Some of his lectures about Clean Code are available on Youtube.

9.2 Use a linter and let your editor help you

A linter is a tool that scans your code and shows you where you are not following the coding style guidelines. The anaconda environment of `environment.yml` comes with flake8 and pep8-naming, which warns about a lot of things. Best is to configure your editor in a way that it shows you the linter warnings as you type.

Many editors have some other useful helpers. For example whitespace cleanup, i.e. delete any trailing whitespace as soon as you save the file.

9.3 Line lengths

Lines should not often exceed the 90 characters. Exceeding it sometimes by a bit is ok, though. Please do *never* exceed 125 characters because that's the width of the GitHub code viewer.

9.4 Naming conventions

By naming conventions the programmer can give some indications to the reader of the program, what an identifier is supposed to be or what it is referring to. Therefore some consistency guidelines.

9.4.1 Mandatory names throughout the pyLife code base

For variables representing physical quantities, we have a dedicated document in the documentation. Please follow the points discussed there.

9.4.2 Module names

For module names, try to find one word names like `rainflow`, `gradient`. If you by all means need word separation in a module name, use `snake_case`. *Never* use dashes (`-`) and capital letters in module names. They lead to all kinds of problems.

9.4.3 Class names

Class names are usually short and a single or compound noun. For these short names we use the so called `CamelCase` style:

```
class DataObjectReader:
    ...
```

9.4.4 Function names

Function and variable names can be longer than class names. Especially function names tend to be actual sentences like:

```
def calc_all_data_from_scratch():
    ...
```

These are way more readable in the so called `lowercase_with_underscores` style.

9.4.5 Variable names

Variable names can be shorter as long as they are local. For example when you store the result of a function in a variable that the function is finally to return, don't call it `result_to_be_returned` but only `res`. A rule of thumb is that the name of a variable needs to be descriptive, if the code part in which the variable is used, exceeds the area that you can capture with one eye glimpse.

9.4.6 Class method names

There are a couple of conventions that make it easier to understand an API of a class.

To access the data items of a class we used to use getter and setter functions. A better and more modern way is python's `@property` decorator.

```
class ExampleClass:
        def __init__(self):
                self._foo = 23
                self._bar = 42
                self._sum = None

    @property
    def foo(self):
                ''' getter functions have the name of the accessed data item
                '''
                return self._foo

    @foo.setter
        def foo(self, v):
                ''' setter functions have the name of the accessed data item prefixed
                        with `set_`
                '''
                if v < 0: # sanity check
                        raise Exception("Value for foo must be >= 0")
                self._foo = v

        def calc_sum_of_foo_and_bar(self):
                '''         class methods whose name does not imply that they return data
                        should not return anything.
                '''
                self._sum = self._foo + self._bar
```

The old style getter and setter function like `set_foo(self, new_foo)` are still tolerable but should be avoided in new code. Before major releases we might dig to the code and replace them with `@property` where feasible.

9.5 Structuring of the code

9.5.1 Data encapsulation

One big advantage for object oriented programming is the so called data encapsulation. That means that items of a class that is intended only for internal use can be made inaccessible from outside of the class. Python does not strictly enforce that concept, but in order to make it clear to the reader of the code, we mark every class method and every class member variable that is not meant to be accessed from outside the class with a leading underscore _ like:

```
class Foo:

        def __init__(self):
                self.public_variable = 'bar'
                self._private_variable = 'baz'
```

(continues on next page)

(continued from previous page)

```
        def public_method(self):
        ...

        def _private_method(self):
```

9.5.2 Object orientation

Usually it makes sense to compound data structures and the functions using these data structures into classes. The data structures then become class members and the functions become class methods. This object oriented way of doing things is recommendable but not always necessary. Sets of simple utility routines can also be autonomous functions.

As a rule of thumb: If the user of some functionality needs to keep around a data structure for a longer time and make several different function calls that deal with the same data structure, it is probably a good idea to put everything into a class.

Do not just put functions into a class because they belong semantically together. That is what python modules are there for.

9.5.3 Functions and methods

Functions are not only there for sharing code but also to divide code into easily manageable pieces. Therefore functions should be short and sweet and do just one thing. If a function does not fit into your editor window, you should consider to split it into smaller pieces. Even more so, if you need to scroll in order to find out, where a loop or an if statement begins and ends. Ideally a function should be as short, that it is no longer *possible* to extract a piece of it.

9.5.4 Commenting

Programmers are taught in the basic programming lessons that comments are important. However, a more modern point of view is, that comments are only the last resort, if the code is so obscure that the reader needs the comment to understand it. Generally it would be better to write the code in a way that it speaks for itself. That's why keeping functions short is so important. Extracting a code block of a function into another function makes the code more readable, because the new function has a name.

Bad example:

```
def some_function(data, parameters):
        ... # a bunch of code
        ... # over several lines
        ... # hard to figure out
        ... # what it is doing
        if parameters['use_method_1']:
        ... # a bunch of code
                ... # over several lines
                ... # hard to figure out
                ... # what it is doing
        else:
                ... # a bunch of code
                ... # over several lines
                ... # hard to figure out
                ... # what it is doing
```

(continues on next page)

(continued from previous page)

```
        ... # a bunch of code
        ... # over several lines
        ... # hard to figure out
        ... # what it is doing
```

Good example

```
def prepare(data, parameters):
        ... # a bunch of code
        ... # over several lines
        ... # easily understandable
        ... # by the function's name

def cleanup(data, parameters):
        ... # a bunch of code
        ... # over several lines
        ... # easily understandable
        ... # by the function's name

def method_1(data):
        ... # a bunch of code
        ... # over several lines
        ... # easily understandable
        ... # by the function's name

def other_method(data):
        ... # a bunch of code
        ... # over several lines
        ... # easily understandable
        ... # by the function's name

def some_function(data, parameters):
        prepare(data, parameters)
        if parameters['use_method_1']:
                method_1(data)
        else:
                other_method(data)
        cleanup(data, parameters)
```

Ideally the only comments that you need are docstrings that document the public interface of your functions and classes.

Compare the following functions:

Bad example:

```
def hypot(triangle):

    # reading in a
    a = triangle.get_a()

    # reading in b
    b = triangle.get_b()

```

(continues on next page)

(continued from previous page)

```
    # reading in gamma
    gamma = triangle.get_gamma()

    # calculate c
    c = np.sqrt(a*a + b*b - 2*a*b*np.cos(gamma))

    # return result
    return c
```

Everyone sees that you read in some parameter `a`. Everyone sees that you read in some parameter `b` and `gamma`. Everyone sees that you calculate and return some value `c`. But what is it that you are doing?

Now the *good* example:

```
def hypot(triangle):
    ''' Calculates the hypotenuse of a triangle using the law of cosines

    https://en.wikipedia.org/wiki/Law_of_cosines
    '''
    a = triangle.a
    b = triangle.b
    gamma = triangle.gamma

    return np.sqrt(a*a + b*b - 2*a*b*np.cos(gamma))
```

CHAPTER

TEN

PYLIFE'S VARIABLE NAME CONVENTIONS

10.1 Preamble

In order for source code to be readable and maintainable, variable names should be expressive, i.e. they should imply what the variable represents. By doing that, documenting variable names becomes virtually unnecessary.

However, in scientific programming we often need to deal with fairly complex mathematical equations. Then it is tempting to use the same or at least similar symbols as we find in the equation in the text book. While these symbols are obvious to people with domain knowledge, for programmers focusing on software optimization and industrialization these symbols are often hard to read.

Out of these considerations we decided that in pyLife for physical quantities the variable names as described in this document are *mandatory*. For physical quantities not described in this document, you can either use an expressive variable name or you can document a symbol in your module documentation.

10.2 General rules

10.2.1 Letters

Roman letters can be used as is, capital or small. Greek letters could actually also be written as unicode letters. Yes, `x = x_0 * np.exp(-*t) * np.cos(*t + )` is perfectly valid Python code. However, not all of us are using decent systems which allow you to type them easily. That's why for Greek letters we would spell them out like `alpha`. This does not work for `lambda`, though as it is a keyword in python.

10.2.2 Indices

Indices should be separated with an underscore (`_`) from the symbol. However, in some cases the underscore is not uses (see below.)

10.3 Variable names

10.3.1 Stress values

- Stress tensor variables: `S11`, `S22`, `S33`, `S12`, `S13`, `S23`
- Cyclic stress variables: * `amplitude`: stress or load amplitude, * `meanstress`: meanstress, * `R`: R-value

10.3.2 Coordinate values

- Cartesian coordinates: `x`, `y`, `z`

10.3.3 Displacement values

- Displacements in a Cartesian coordinate system: `dx`, `dy`, `dz`

10.3.4 Strength variables

- `SD` endurance limit in load direction, `SD_xx` for `xx` percent failure probability
- `ND` endurance limit in cycle direction, `ND_xx` for `xx` percent failure probability
- `TS` scatter in load direction (= `SD_10/SD_90`)
- `TN` scatter in load direction (= `ND_10/ND_90`)
- `k` slope of the Wöhler curve, `k_1` above the endurance limit, `k_2` below the endurance limit

CHAPTER ELEVEN

AUTHORS

```
# This is the official list of pyLife copyright holders and authors.
#
# Often employers or academic institutions have ownership over code that is
# written in certain circumstances, so please do due diligence to ensure that
# you have the right to submit the code.
#
# When adding J Random Contributor's name to this file, either J's name on its
# own or J's name associated with J's organization's name should be added,
# depending on whether J's employer (or academic institution) has ownership
# over code that is written for this project.
#
# How to add names to this file:
#     Individual's name <submission email address>.
#
# If Individual's organization is copyright holder of her contributions add the
# organization's name, optionally also the contributor's name:
#
#     Organization's name
#         Individual's name <submission corporate email address>
#
# Please keep the list sorted.

Robert Bosch GmbH
       Vivien Le Baube <vivien.lebaube@de.bosch.com>
       Lisa Katharina Hill <LisaKatharina.Hill@de.bosch.com>
       Mustapha Kassem <fixed-term.Mustapha.Kassem@de.bosch.com>
       Gyöngyvér Kiss <gyongyver.kiss@hu.bosch.com>
       Daniel Christopher Kreuter <DanielChristopher.Kreuter@de.bosch.com>
       Johannes Mueller <johannes.mueller4@de.bosch.com>
       Erik Natkowski <erik.natkowski@de.bosch.com>
       Vishnu Pradeep <fixed-term.Vishnu.Pradeep@de.bosch.com>
       Lena Rapp <fixed-term.Lena.Rapp@de.bosch.com>
       Jakob Riebe <fixed-term.Jakob.Riebe@de.bosch.com>
       Simone Schreijäg <simone.schreijaeg@de.bosch.com>
       Cedric Philip Wagner <fixed-term.CedricPhilip.Wagner@de.bosch.com>
       Matthias Wieler <matthias.wieler@de.bosch.com>
       Andreas Wilmes <andreas.wilmes@de.bosch.com>

TU Darmstadt IFSW
       Alexander Maier <maier@wm.tu-darmstadt.de>
```

CHAPTER TWELVE

LICENSE

(continues on next page)

(continued from previous page)

```
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      copyright license to reproduce, prepare Derivative Works of,
      publicly display, publicly perform, sublicense, and distribute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have made,
      use, offer to sell, sell, import, and otherwise transfer the Work,
      where such license applies only to those patent claims licensable
      by such Contributor that are necessarily infringed by their
      Contribution(s) alone or by combination of their Contribution(s)
      with the Work to which such Contribution(s) was submitted. If You
      institute patent litigation against any entity (including a
      cross-claim or counterclaim in a lawsuit) alleging that the Work
      or a Contribution incorporated within the Work constitutes direct
      or contributory patent infringement, then any patent licenses
      granted to You under this License for that Work shall terminate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
      PARTICULAR PURPOSE. You are solely responsible for determining the
      appropriateness of using or redistributing the Work and assume any
      risks associated with Your exercise of permissions under this License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and grossly
      negligent acts) or agreed to in writing, shall any Contributor be
      liable to You for damages, including any direct, indirect, special,
      incidental, or consequential damages of any character arising as a
      result of this License or out of the use or inability to use the
      Work (including but not limited to damages for loss of goodwill,
      work stoppage, computer failure or malfunction, or any and all
      other commercial damages or losses), even if such Contributor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the following
      boilerplate notice, with the fields enclosed by brackets "[]"
      replaced with your own identifying information. (Don't include
      the brackets!)  The text should be enclosed in the appropriate
      comment syntax for the file format. We also recommend that a
      file or class name and description of purpose be included on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright 2019-2021 Robert Bosch GmbH

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0
```

(continues on next page)

(continued from previous page)

CHAPTER THIRTEEN

3RD PARTY LICENSES

(continues on next page)

(continued from previous page)

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX