pyLife Documentation

Release 1.1.4

pyLife Developer Team

Nov 10, 2022

About

CHAPTER 1

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.

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 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

1.3 Disclaimer

pyLife is in continuous development. We hope to keep the interfaces more or less stable. However depending on the practical use of pyLife in the future interface changes might occur. If that happens, we probably won't be able to put too much effort into backwards compatibility. So be prepared to react to deprecations.

1.4 License

CHAPTER 2

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 pip
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
```

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. The reason for that is that there is one package that pyLife depends on (mystic), that does not have a conda package. So we cannot provide a conda packages that fetches all the required dependencies to use all of pyLife's functionality.

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

```
./install_pylife_linux.sh
```

on Linux and

```
install_pylife_linux.bat
```

on Windows

which will take a couple of minutes.

Then activate it:

```
conda activate ./_venv
```

2.3.3 Setup for development

In order to make the package usable in your environment do

```
python ./setup.py develop
```

2.3.4 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 `DeprecationWarning`s. Ignore them for now.

2.3.5 Building the documentation (optional)

The docs are available at readthedocs. You can also build them on your own, in order to check your docstrings.

```
./batch_scripts/build_docs.sh
```

on Linux or

```
./batch_scripts/build_docs.bat
```

The docs then are in `doc/build/index.html`.

CHAPTER 3

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.

pyLife being a library for numerical data analysis and numerical data processing makes 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.

For some specific functionality other libraries are used like for example scipy.

CHAPTER 4

The pyLife Signal API

The signal api is the higher level API of pyLife. It is the API that you probably should be using. Most of the domain specific functions are also available as pure numpy functions, if your application for some reason makes the use of pandas harder. However, in such a case we highly recommend you to take a closer look at pandas and consider to adapt your application to the pandas way of doing things.

4.1 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
- cyclic load information over a component's geometry
- ...

Signals are usually kept in a `pandas.DataFrame`. In cases, the signal is only a small record of parameters, it is kept in a `pandas.Series`.

To implement signal processors we implement accessor classes using `pandas.api.extensions.register_dataframe_accessor()` resp. `pandas.api.extensions.register_series_accessor()`.

For predefined accessors that can be subclassed from see

- `meshsignal`
- `stresssignal`

4.1.1 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 `meshsignal`) needs to be imported explicitly.

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

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.PlainMeshAccessor object at 0x7f66da8d4d10>
```

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
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmu3si/Devel/pylife/_venv/lib/python3.7/site-packages/pandas/core/
↪generic.py", line 5175, in __getattr__
    return object.__getattribute__(self, name)
  File "/home/jmu3si/Devel/pylife/_venv/lib/python3.7/site-packages/pandas/core/
↪accessor.py", line 175, in __get__
    accessor_obj = self._accessor(obj)
  File "/home/jmu3si/Devel/pylife/pylife/mesh/meshsignal.py", line 79, in __init__
    self._validate(pandas_obj)
  File "/home/jmu3si/Devel/pylife/pylife/mesh/meshsignal.py", line 84, in _validate
    signal.fail_if_key_missing(obj, self._coord_keys)
  File "/home/jmu3si/Devel/pylife/pylife/core/signal.py", line 88, in fail_if_key_
↪missing
    raise AttributeError(msg % (', '.join(keys_to_check), ', '.join(missing_keys)))
AttributeError: PlainMeshAccessor must have the items x, y. Missing y.
```

Example for accessing a property

Get the coordinates of a 2D plain mesh

```
>>> import pandas as pd
>>> import pylife.mesh.meshsignal
>>> df = pd.DataFrame({'x': [1.0], 'y': [1.0], 'foo': [42.0], 'bar': [23.0]})
>>> df.plain_mesh.coordinates
     x    y
0  1.0  1.0
```

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
     x    y    z
0  1.0  1.0  1.0
```

4.2 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 *MeshAccessor*. This class you register as a pandas DataFrame accessor using a decorator

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

@pd.api.extensions.register_dataframe_accessor('my_mesh_processor')
class MyMeshAccessor(meshsignal.MeshAccessor):
    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 *MyMeshAccessor* is derived from *MeshAccessor* the validation of *MeshAccessor* 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.

```
>>> df = pd.read_hdf('demos/plate_with_hole.h5', '/node_data')
>>> result = df.my_mesh_processor.do_something()
```

4.2.1 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.meshsignal
from pylife import signal

@pd.api.extensions.register_dataframe_accessor('my_only_for_3D_mesh_processor')
class MyOnlyFor3DMeshAccessor(meshsignal.PlainMeshAccessor):
    def _validate(self, obj):
        super(MyOnlyFor3DMeshAccessor, obj) # call PlainMeshAccessor._validate()
        signal.fail_if_key_missing(['z'])
```

4.3 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 signal

@pd.api.extensions.register_dataframe_accessor('my_signal')
class MySignalAccessor(signal.PylifeSignal):
    def _validate(self, obj):
        signal.fail_if_key_missing(obj, ['alpha', 'beta', 'gamma'])
        for k in ['alpha', 'beta', 'gamma']:
            if (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):
        # some code
```

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

Validation fails because of missing *gamma* column.

```
>>> import my_signal_mod
>>> df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, -1.0]})
>>> df.my_signal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmu3si/Devel/pylife/_venv/lib/python3.7/site-packages/pandas/core/
↪generic.py", line 5175, in __getattr__
    return object.__getattribute__(self, name)
  File "/home/jmu3si/Devel/pylife/_venv/lib/python3.7/site-packages/pandas/core/
↪accessor.py", line 175, in __get__
    accessor_obj = self._accessor(obj)
  File "/home/jmu3si/Devel/pylife/signal_test.py", line 7, in __init__
    self._validate(pandas_obj)
  File "/home/jmu3si/Devel/pylife/signal_test.py", line 11, in _validate
    signal.fail_if_key_missing(obj, ['alpha', 'beta', 'gamma'])
  File "/home/jmu3si/Devel/pylife/pylife/core/signal.py", line 88, in fail_if_key_
↪missing
    raise AttributeError(msg % (', '.join(keys_to_check), ', '.join(missing_keys)))
AttributeError: MySignalAccessor 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
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmu3si/Devel/pylife/_venv/lib/python3.7/site-packages/pandas/core/
↪accessor.py", line 175, in __get__
    accessor_obj = self._accessor(obj)
  File "/home/jmu3si/Devel/pylife/signal_test.py", line 7, in __init__
    self._validate(pandas_obj)
```

(continues on next page)

(continued from previous page)

```
  File "/home/jmu3si/Devel/pylife/signal_test.py", line 15, in _validate
    "At least one is less than 0" % k)
ValueError: All values of beta need to be positive. At least one is less than 0
```

Validation success.

```
>>> df = pd.DataFrame({'alpha': [1.0, 2.0], 'beta': [1.0, 0.0], 'gamma': [1.0, 2.0]})
>>> df.my_signal
<signal_test.MySignalAccessor object at 0x7fb3268c4f50>
```

Call *some_method()*

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

4.3.1 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 signal

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

    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)
```

4.4 Registering a method to an existing accessor class

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 *StressTensorEquistressAccessor*, 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.StressTensorEquistressAccessor, 'my_equistress')
def my_equistress_method(df)
```

(continues on next page)

(continued from previous page)

```
    # 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.StressTensorEquistressAccessor, '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)
```

CHAPTER 5

pyLife Cookbook

5.1 Life time Calculation

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

5.1.1 Stress derivation

First we read in different time signals (coming from a test bench or a vehicle measurement e.g.).

1. Import the time series into a pandas Data Frame
2. Resample the time series if necessary
3. Filter the time series with a bandpass filter if necessary
4. Edititing time series using Running Statistics methods
5. Rainflow Calculation
6. Mean stress correction
7. 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

from pylife.stress.histogram import *
import pylife.stress.timesignal as ts
from pylife.stress.rainflow import *
import pylife.stress.equistress

import pylife.strength.meanstress
from pylife.strength import miner
from pylife.strength import sn_curve
from pylife.strength.miner import MinerElementar, MinerHaibach
from pylife.strength import failure_probability as fp
import pylife.vmap

import pyvista as pv

from pylife.materialdata.woehler.diagrams.woehler_curve_diagrams import␣
↪WoehlerCurveDiagrams

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

from scipy.stats import norm

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import io
from IPython.display import HTML
import base64

# mpl.style.use('seaborn')
# mpl.style.use('seaborn-notebook')
mpl.style.use('bmh')
%matplotlib inline

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

```
---------------------------------------------------------------------------
NoSectionError                            Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:238, in TheanoConfigParser.fetch_val_for_key(self,␣
↪key, delete_key)
    237 try:
--> 238     return self._theano_cfg.get(section, option)
    239 except InterpolationError:
```

(continues on next page)

(continued from previous page)

```
File ~/.pyenv/versions/3.8.6/lib/python3.8/configparser.py:781, in RawConfigParser.
↪get(self, section, option, raw, vars, fallback)
    780 try:
--> 781     d = self._unify_values(section, vars)
    782 except NoSectionError:

File ~/.pyenv/versions/3.8.6/lib/python3.8/configparser.py:1149, in RawConfigParser._
↪unify_values(self, section, vars)
   1148     if section != self.default_section:
-> 1149         raise NoSectionError(section) from None
   1150 # Update with the entry specific variables

NoSectionError: No section: 'blas'

During handling of the above exception, another exception occurred:

KeyError                                  Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:354, in ConfigParam.__get__(self, cls, type_,␣
↪delete_key)
    353 try:
--> 354     val_str = cls.fetch_val_for_key(self.name, delete_key=delete_key)
    355     self.is_default = False

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:242, in TheanoConfigParser.fetch_val_for_key(self,␣
↪key, delete_key)
    241 except (NoOptionError, NoSectionError):
--> 242     raise KeyError(key)

KeyError: 'blas__ldflags'

During handling of the above exception, another exception occurred:

AttributeError                            Traceback (most recent call last)
Cell In [1], line 18
     14 import pylife.vmap
     16 import pyvista as pv
---> 18 from pylife.materialdata.woehler.diagrams.woehler_curve_diagrams import␣
↪WoehlerCurveDiagrams
     20 import pylife.mesh.meshplot
     21 import matplotlib.pyplot as plt

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pylife/materialdata/woehler/__init__.py:26
     24 from .analyzers.probit import Probit
     25 from .analyzers.maxlike import MaxLikeInf, MaxLikeFull
---> 26 from .analyzers.bayesian import Bayesian

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pylife/materialdata/woehler/analyzers/bayesian.py:22
     20 import numpy as np
     21 import pandas as pd
---> 22 import theano.tensor as tt
     23 import pymc3 as pm
     25 from .elementary import Elementary
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/__init__.py:83
     75 # This is the api version for ops that generate C code.  External ops
     76 # might need manual changes if this number goes up.  An undefined
     77 # __api_version__ can be understood to mean api version 0.
     78 #
     79 # This number is not tied to the release version and should change
     80 # very rarely.
     81 __api_version__ = 1
---> 83 from theano import scalar, tensor
     84 from theano.compile import (
     85     In,
     86     Mode,
   (...)
     93     shared,
     94 )
     95 from theano.compile.function import function, function_dump

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/__init__.py:20
      9 from theano.compile import SpecifyShape, specify_shape
     10 from theano.gradient import (
     11     Lop,
     12     Rop,
   (...)
     18     verify_grad,
     19 )
---> 20 from theano.tensor import nnet  # used for softmax, sigmoid, etc.
     21 from theano.tensor import sharedvar  # adds shared-variable constructors
     22 from theano.tensor import (
     23     blas,
     24     blas_c,
   (...)
     29     xlogx,
     30 )

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/nnet/__init__.py:3
      1 import warnings
----> 3 from . import opt
      4 from .abstract_conv import conv2d as abstract_conv2d
      5 from .abstract_conv import conv2d_grad_wrt_inputs, conv3d, separable_conv2d

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/nnet/opt.py:32
     24 from theano.tensor.nnet.blocksparse import (
     25     SparseBlockGemv,
     26     SparseBlockOuter,
     27     sparse_block_gemv_inplace,
     28     sparse_block_outer_inplace,
     29 )
     31 # Cpu implementation
---> 32 from theano.tensor.nnet.conv import ConvOp, conv2d
     33 from theano.tensor.nnet.corr import CorrMM, CorrMM_gradInputs, CorrMM_
↪gradWeights
     34 from theano.tensor.nnet.corr3d import Corr3dMM, Corr3dMMGradInputs,␣
↪Corr3dMMGradWeights
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/nnet/conv.py:20
     18 from theano.graph.basic import Apply
     19 from theano.graph.op import OpenMPOp
---> 20 from theano.tensor import blas
     21 from theano.tensor.basic import (
     22     NotScalarConstantError,
     23     as_tensor_variable,
     24     get_scalar_constant_value,
     25     patternbroadcast,
     26 )
     27 from theano.tensor.nnet.abstract_conv import get_conv_output_shape, get_conv_
↪shape_1axis

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/blas.py:163
    161 from theano.scalar import bool as bool_t
    162 from theano.tensor import basic as tt
--> 163 from theano.tensor.blas_headers import blas_header_text, blas_header_version
    164 from theano.tensor.opt import in2out, local_dimshuffle_lift
    165 from theano.tensor.type import values_eq_approx_remove_inf_nan

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/blas_headers.py:1016
    997             header += textwrap.dedent(
    998                 """\
    999                     static float sdot_(int* Nx, float* x, int* Sx, float* y,␣
↪int* Sy)
   (...)
   1010                     """
   1011             )
   1013     return header + blas_code
-> 1016 if not config.blas__ldflags:
   1017     _logger.warning("Using NumPy C-API based implementation for BLAS␣
↪functions.")
   1020 def mkl_threads_text():

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:358, in ConfigParam.__get__(self, cls, type_,␣
↪delete_key)
    356 except KeyError:
    357     if callable(self.default):
--> 358         val_str = self.default()
    359     else:
    360         val_str = self.default

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/link/c/cmodule.py:2621, in default_blas_ldflags()
   2617 try:
   2618     if hasattr(numpy.distutils, "__config__") and numpy.distutils.__config__:
   2619         # If the old private interface is available use it as it
   2620         # don't print information to the user.
-> 2621         blas_info = numpy.distutils.__config__.blas_opt_info
   2622     else:
   2623         # We do this import only here, as in some setup, if we
   2624         # just import theano and exit, with the import at global
```

(continues on next page)

(continued from previous page)

```
   (...)
   2630         # This happen with Python 2.7.3 |EPD 7.3-1 and numpy 1.8.1
   2631         # isort: off
   2632         import numpy.distutils.system_info  # noqa

AttributeError: module 'numpy.distutils.__config__' has no attribute 'blas_opt_info'
```

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

[2]:
```
t = np.linspace(0,60,60*2048)
files = ['wn','sine']
wn = pd.DataFrame(index = t, columns = ['sensor_1'], data = 120*np.random.
↪randn(len(t)))
sine = pd.DataFrame(index = t, columns = ['sensor_1'], data = 80*np.sin(2*np.pi*50*t))
input_data = [wn,sine]
```

[3]:
```
# input_data = []
# for upload in files:
#     data_akt = pd.read_csv(data_loc + upload, sep = ",")
#     if len(data_akt.columns) == 1:
#         print ('please use "," as seperator next time')
#         data_akt = pd.read_csv(data_loc + upload, sep = ";")
#         input_data.append(data_akt)
#     print(upload + " imported succesfully")
```

Resampling

[4]:
```
f_resample = widgets.FloatText(value = 1024,min=1,max=100e3,step=1,
    description='Resampling frequency [Hz]',
    disabled=False,readout=True,readout_format='d')
display(f_resample)
# select time column
# timeColumn = widgets.Dropdown(options = data_akt.columns)
# display(timeColumn)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [4], line 1
----> 1 f_resample = widgets.FloatText(value = 1024,min=1,max=100e3,step=1,
      2     description='Resampling frequency [Hz]',
      3     disabled=False,readout=True,readout_format='d')
      4 display(f_resample)
      5 # select time column
      6 # timeColumn = widgets.Dropdown(options = data_akt.columns)
      7 # display(timeColumn)
```

(continues on next page)

(continued from previous page)

```
NameError: name 'widgets' is not defined
```

[5]:
```
meas_resample = []
for file_act in input_data:
#     file_act = file_act.set_index(timeColumn.value)
    meas_resample.append(ts.TimeSignalPrep(file_act).resample_acc(f_resample.value))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [5], line 4
      1 meas_resample = []
      2 for file_act in input_data:
      3 #     file_act = file_act.set_index(timeColumn.value)
----> 4     meas_resample.append(ts.TimeSignalPrep(file_act).resample_acc(f_resample.
↪value))

NameError: name 'f_resample' is not defined
```

[6]:
```
print("select channel to plot")
plotChan = widgets.Dropdown(options = file_act.columns)
display(plotChan)
```

```
select channel to plot
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [6], line 2
      1 print("select channel to plot")
----> 2 plotChan = widgets.Dropdown(options = file_act.columns)
      3 display(plotChan)

NameError: name 'widgets' is not defined
```

[7]:
```
fig, ax = plt.subplots(len(meas_resample))
fig.suptitle('Resampled input data')
ii = 0
for df_act in meas_resample:
    if len(meas_resample) == 1:
        ax.plot(df_act.index, df_act[plotChan.value])
    else:
        ax[ii].plot(df_act.index, df_act[plotChan.value])
    ii += 1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [7], line 1
----> 1 fig, ax = plt.subplots(len(meas_resample))
      2 fig.suptitle('Resampled input data')
      3 ii = 0

NameError: name 'plt' is not defined
```

Filtering

[8]:
```
f_min = widgets.FloatText(value = 5,description='min frequency [Hz]',disabled=False)
f_max = widgets.FloatText(value = 100,description='max frequency [Hz]',disabled=False)
display(f_min)
display(f_max)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [8], line 1
----> 1 f_min = widgets.FloatText(value = 5,description='min frequency [Hz]',
 ↪disabled=False)
      2 f_max = widgets.FloatText(value = 100,description='max frequency [Hz]',
 ↪disabled=False)
      3 display(f_min)

NameError: name 'widgets' is not defined
```

[9]:
```
bandpass = []
for df_act in meas_resample:
    bandpassDF = pd.DataFrame(index = df_act.index)
    for col_act in df_act.columns:
        bandpassDF[col_act] = ts.TimeSignalPrep(df_act[col_act]).butter_bandpass(f_
 ↪min.value,f_max.value,f_resample.value,5)
    bandpass.append(bandpassDF)
display(bandpassDF)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [9], line 7
      5         bandpassDF[col_act] = ts.TimeSignalPrep(df_act[col_act]).butter_
 ↪bandpass(f_min.value,f_max.value,f_resample.value,5)
      6     bandpass.append(bandpassDF)
----> 7 display(bandpassDF)

NameError: name 'bandpassDF' is not defined
```

Running statistics

[10]:
```
print("select channel to for running stats")
runChan = widgets.Dropdown(options = df_act.columns)
display(runChan)
print(" Running statistics method")
method_choice = widgets.Dropdown(options = ['rms','max','min','abs'])
display(method_choice)

paraRunStats = ['window_length', 'buffer_overlap', 'limit']
values = [800,0.1,0.015]
child = [widgets.FloatText(description=name) for name in paraRunStats]
tab = widgets.Tab()
tab.children = child
for i in range(len(child)):
    tab.set_title(i, paraRunStats[i])
    tab.children[i].value = values[i]

tab
```

```
select channel to for running stats
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [10], line 2
      1 print("select channel to for running stats")
----> 2 runChan = widgets.Dropdown(options = df_act.columns)
      3 display(runChan)
      4 print(" Running statistics method")

NameError: name 'widgets' is not defined
```

[11]:
```
""" Running statistics to drop out zero values """
cleaned = []
for df_act in bandpass:
    cleaned_df = ts.TimeSignalPrep(df_act).running_stats_filt(
                            col = runChan.value,
                            window_length = int(tab.children[0].value),
                            buffer_overlap = int(tab.children[1].value),
                            limit = tab.children[2].value,
                            method = method_choice.value)
    cleaned.append(cleaned_df)
# display(cleaned_df)
```

[12]:
```
print("select channel to plot")
plotChan = widgets.Dropdown(options = file_act.columns)
display(plotChan)
```

```
select channel to plot
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [12], line 2
      1 print("select channel to plot")
----> 2 plotChan = widgets.Dropdown(options = file_act.columns)
      3 display(plotChan)

NameError: name 'widgets' is not defined
```

[13]:
```
fig, ax = plt.subplots(len(meas_resample))
fig.suptitle('Cleaned input data')
for i, df_act in enumerate(cleaned):
    if len(meas_resample) == 1:
        ax.plot(df_act.index, df_act[plotChan.value])
    else:
        ax[i].plot(df_act.index, df_act[plotChan.value])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [13], line 1
----> 1 fig, ax = plt.subplots(len(meas_resample))
      2 fig.suptitle('Cleaned input data')
      3 for i, df_act in enumerate(cleaned):

NameError: name 'plt' is not defined
```

Rainflow

[14]:
```
rfcChan = widgets.Dropdown(options = df_act.columns)
display(rfcChan)
binwidget = widgets.IntSlider(value = 64, min=1, max=1024, step=1,description='Bins:')
display(binwidget)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [14], line 1
----> 1 rfcChan = widgets.Dropdown(options = df_act.columns)
      2 display(rfcChan)
      3 binwidget = widgets.IntSlider(value = 64, min=1, max=1024, step=1,description=
↪'Bins:')

NameError: name 'widgets' is not defined
```

[15]:
```
rainflow = []
for df_act in cleaned:
    rfc = RainflowCounterFKM().process(df_act[rfcChan.value].values)
    rfm = rfc.get_rainflow_matrix_frame(binwidget.value)
    rainflow.append(rfm)
```

[16]:
```
colormap = cm.ScalarMappable()
cmap = cm.get_cmap('PuRd')
# fig, ax = plt.subplots(2,len(rainflow))
fig = plt.figure(figsize = (8,11))
fig.suptitle('Rainflow of Channel ' + rfcChan.value)

for i, rf_act in enumerate(rainflow):
    # 2D
    ax = fig.add_subplot(3,2,2*(i+1)-1)
    froms = rf_act.index.get_level_values('from').mid
    tos = rf_act.index.get_level_values('to').mid
    counts = np.flipud((rf_act.values.reshape(rf_act.index.levshape).T))#.ravel()
    ax.set_xlabel('From')
    ax.set_ylabel('To')
    ax.imshow(np.log10(counts), extent=[froms.min(), froms.max(), tos.min(), tos.
↪max()])
    # 3D
    ax = fig.add_subplot(3,2,2*(i+1), projection='3d')
    bottom = np.zeros_like(counts.ravel())
    width = rf_act.index.get_level_values('from').length.min()
    depth = rf_act.index.get_level_values('to').length.min()
    max_height = np.max(counts.ravel())   # get range of colorbars
    min_height = np.min(counts.ravel())
    rgba = [cmap((k-min_height)/max_height) for k in counts.ravel()]
    ax.set_xlabel('From')
    ax.set_ylabel('To')
    ax.set_zlabel('Count')
    ax.bar3d(froms.ravel(), tos.ravel(), bottom, width, depth, counts.ravel(),␣
↪shade=True, color=rgba, zsort='average')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [16], line 1
----> 1 colormap = cm.ScalarMappable()
      2 cmap = cm.get_cmap('PuRd')
```

(continues on next page)

(continued from previous page)

```
      3 # fig, ax = plt.subplots(2,len(rainflow))

NameError: name 'cm' is not defined
```

Meanstress transformation

[17]:
```
meanstress_para = ['M', 'M2', 'R_Goal']
values = [0.3,0.2,-1]
child = [widgets.FloatText(description=name) for name in meanstress_para]
tab_mean = widgets.Tab()
tab_mean.children = child
for i in range(len(child)):
    tab_mean.set_title(i, meanstress_para[i])
    tab_mean.children[i].value = values[i]

tab_mean
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [17], line 3
      1 meanstress_para = ['M', 'M2', 'R_Goal']
      2 values = [0.3,0.2,-1]
----> 3 child = [widgets.FloatText(description=name) for name in meanstress_para]
      4 tab_mean = widgets.Tab()
      5 tab_mean.children = child

Cell In [17], line 3, in <listcomp>(.0)
      1 meanstress_para = ['M', 'M2', 'R_Goal']
      2 values = [0.3,0.2,-1]
----> 3 child = [widgets.FloatText(description=name) for name in meanstress_para]
      4 tab_mean = widgets.Tab()
      5 tab_mean.children = child

NameError: name 'widgets' is not defined
```

[18]:
```
transformed = []
for rf_act in rainflow:
    transformed.append(rf_act.meanstress_hist.FKM_goodman(pd.Series({'M': tab_mean.
↪children[0].value,
                                                                    'M2': tab_mean.
↪children[1].value})
                                                          , R_goal = tab_mean.
↪children[2].value))
```

Repeating factor

[19]:
```
child = [widgets.FloatText(description=name) for name in files]
tab_repeat = widgets.Tab()
tab_repeat.children = child
for i in range(len(child)):
    tab_repeat.set_title(i, files[i])
    tab_repeat.children[i].value = int(50/(i+1))
tab_repeat
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [19], line 1
----> 1 child = [widgets.FloatText(description=name) for name in files]
      2 tab_repeat = widgets.Tab()
      3 tab_repeat.children = child

Cell In [19], line 1, in <listcomp>(.0)
----> 1 child = [widgets.FloatText(description=name) for name in files]
      2 tab_repeat = widgets.Tab()
      3 tab_repeat.children = child

NameError: name 'widgets' is not defined
```

[20]:
```
for ii in range(len(files)):
    transformed[ii] = transformed[ii]*tab_repeat.children[ii].value
range_only_total = combine_hist(transformed,method = "sum")
display(range_only_total)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In [20], line 2
      1 for ii in range(len(files)):
----> 2     transformed[ii] = transformed[ii]*tab_repeat.children[ii].value
      3 range_only_total = combine_hist(transformed,method = "sum")
      4 display(range_only_total)

IndexError: list index out of range
```

[21]:
```
fig, ax = plt.subplots(nrows=1, ncols=2,figsize=(10, 5))
# plot total
amplitude = range_only_total.index.get_level_values('range').left.values[::-1]/2
cycles = range_only_total.values[::-1].ravel()
ax[0].step(cycles,amplitude,c = "black",linewidth = 3, label = "total")
ax[1].step(np.cumsum(cycles),amplitude,c = "black",linewidth = 3, label = "total")
ii = 0
for range_only in transformed:
    amplitude = range_only.index.get_level_values('range').mid.values[::-1]/2
    cycles = range_only.values[::-1].ravel()
    ax[0].step(cycles,amplitude,label = files [ii])
    ax[1].step(np.cumsum(cycles),amplitude,label = files [ii])
    ii += 1
ax[0].set_title('Count')
ax[1].set_title('Cumulated sum count')
ax[1].legend()
for ai in ax:
    ai.xaxis.grid(True)
    ai.set_xlabel('count')
    ai.set_ylabel('amplitude of ' + rfcChan.value)
    ai.set_ylim((0,max(amplitude)))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [21], line 1
----> 1 fig, ax = plt.subplots(nrows=1, ncols=2,figsize=(10, 5))
      2 # plot total
      3 amplitude = range_only_total.index.get_level_values('range').left.values[::-
↪1]/2
```

(continues on next page)

(continued from previous page)

```
NameError: name 'plt' is not defined
```

Nominal stress approach

Material parameters

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

[22]:
```
mat = pd.Series(index = ['k_1', 'ND_50', 'SD_50', '1/TN', '1/TS'],
                data = [8, 1.5e+06, 1.5e+02, 12, 1.1])
display(mat)
```

```
k_1            8.0
ND_50    1500000.0
SD_50        150.0
1/TN          12.0
1/TS           1.1
dtype: float64
```

Damage Calculation

[23]:
```
SNmethod = widgets.Dropdown(options = ['Miner Elementar','Miner Haibach','Miner␣
↪original'])
display(SNmethod)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [23], line 1
----> 1 SNmethod = widgets.Dropdown(options = ['Miner Elementar','Miner Haibach',
↪'Miner original'])
      2 display(SNmethod)

NameError: name 'widgets' is not defined
```

[24]:
```
damage_calc = sn_curve.FiniteLifeCurve(**mat.drop(['1/TN','1/TS']))
damage = damage_calc.calc_damage(range_only_total,method = 'original')
# display(damage)
print("\033[5m  Total Damage of channel %s: %.2e  \033[0m" % (rfcChan.value,damage.
↪sum()))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [24], line 2
      1 damage_calc = sn_curve.FiniteLifeCurve(**mat.drop(['1/TN','1/TS']))
----> 2 damage = damage_calc.calc_damage(range_only_total,method = 'original')
      3 # display(damage)
      4 print("\033[5m  Total Damage of channel %s: %.2e  \033[0m" % (rfcChan.value,
↪damage.sum()))

NameError: name 'range_only_total' is not defined
```

[25]:
```
SRI = mat['SD_50']*(mat['ND_50']**(1/mat['k_1']))
# Plotting
diagdata = WoehlerCurveDiagrams(mat, fatigue_data = None, analyzer = None,
                                y_min=2, y_max=SRI, x_min=1e1, x_max=1e12, ax = None)
diagdata.plot_fitted_curve( k_2=15)
plt.step(np.cumsum(cycles),2*amplitude)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [25], line 3
      1 SRI = mat['SD_50']*(mat['ND_50']**(1/mat['k_1']))
      2 # Plotting
----> 3 diagdata = WoehlerCurveDiagrams(mat, fatigue_data = None, analyzer = None,
      4                                 y_min=2, y_max=SRI, x_min=1e1, x_max=1e12, ax␣
↪= None)
      5 diagdata.plot_fitted_curve( k_2=15)
      6 plt.step(np.cumsum(cycles),2*amplitude)

NameError: name 'WoehlerCurveDiagrams' is not defined
```

Failure Probaility

5.1.4 Without field scatter

[26]:
```
D50 = 0.05
d = damage.sum()
di = np.logspace(np.log10(1e-1*d),np.log10(1e3*d),1000).flatten()
std = np.log10(mat['1/TN'])/2.5631031311
failprob = fp.FailureProbability(D50,std).pf_simple_load(di)
#print(failprob)
fig, ax = plt.subplots()
ax.semilogx(di, failprob, label='cdf')
ax.vlines(d, max(failprob), fp.FailureProbability(D50,std).pf_simple_load(d))
#
plt.xlabel("Damage")
plt.ylabel("cdf")
plt.title("Failure probability = %.2e" %fp.FailureProbability(D50,std).pf_simple_
↪load(d))
plt.ylim(0,max(failprob))
plt.xlim(min(di),max(di))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [26], line 2
      1 D50 = 0.05
----> 2 d = damage.sum()
      3 di = np.logspace(np.log10(1e-1*d),np.log10(1e3*d),1000).flatten()
      4 std = np.log10(mat['1/TN'])/2.5631031311

NameError: name 'damage' is not defined
```

5.1.5 With field scatter

[27]:
```
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(d), 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(d,
↪field_std))
plt.legend()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [27], line 2
      1 field_std = 0.35
----> 2 fig, ax = plt.subplots()
      3 # plot pdf of material
      4 mat_pdf = norm.pdf(np.log10(di), loc=np.log10(D50), scale=std)

NameError: name 'plt' is not defined
```

5.1.6 FE based failure probability calculation

5.1.7 FE Data

[28]:
```
vm_mesh = pylife.vmap.VMAPImport("plate_with_hole.vmap")
pyLife_mesh = (vm_mesh.make_mesh('1', 'STATE-2')
               .join_coordinates()
               .join_variable('STRESS_CAUCHY')
               .join_variable('DISPLACEMENT')
               .to_frame())

pyLife_mesh.sample(10)
```

[28]:

		x	y	z	S11	S22	S33	\
element_id	node_id							
4340	12701	-19.003136	-6.563058	0.0	30.617290	3.366126	0.0	
4664	14465	-15.953177	-1.456723	0.0	22.672516	10.300152	0.0	
3280	11389	13.956087	-0.760905	0.0	14.329796	7.645057	0.0	
3350	11852	9.406816	-5.121030	0.0	32.910194	-7.088352	0.0	
1378	2508	-9.787263	5.310039	0.0	35.433792	-5.238566	0.0	
2810	3668	15.679893	-6.507764	0.0	36.186066	2.594980	0.0	
459	5715	8.115341	0.373289	0.0	-0.687680	-54.724014	0.0	
4094	13736	-5.833673	-5.547660	0.0	34.483360	-5.376860	0.0	
732	6907	8.137152	4.620015	0.0	24.873665	-13.072495	0.0	
3675	12700	-18.797379	-6.738377	0.0	30.907495	3.077386	0.0	

		S12	S13	S23	dx	dy	dz
element_id	node_id						
4340	12701	-2.674491	0.0	0.0	-0.006085	-0.000099	0.0
4664	14465	-4.626424	0.0	0.0	-0.006229	-0.000028	0.0

(continues on next page)

(continued from previous page)

```
3280       11389     3.507645  0.0  0.0  0.005861 -0.000013  0.0
3350       11852    12.856589  0.0  0.0  0.004541  0.000459  0.0
1378        2508    13.803806  0.0  0.0 -0.004777 -0.000393  0.0
2810        3668     9.571038  0.0  0.0  0.005290  0.000014  0.0
459         5715    -1.767934  0.0  0.0  0.005724 -0.000094  0.0
4094       13736    -3.259853  0.0  0.0 -0.004014  0.002027  0.0
732         6907    -6.053735  0.0  0.0  0.004549 -0.000888  0.0
3675       12700    -2.945733  0.0  0.0 -0.006039 -0.000081  0.0
```

[29]:
```
# Equivalent stress range
pyLife_mesh['mises'] = 2*pyLife_mesh.equistress.mises()
# Scaling with amplitude
pyLife_mesh['mises'] = 2*pyLife_mesh['mises']/pyLife_mesh['mises'].max()
ampl_fe = pd.DataFrame(data = amplitude, columns = ["ampl"] ,index =cycles)
s_vm_scaled = pd.DataFrame(data = ampl_fe.values*pyLife_mesh['mises'].transpose().
↪values,index = ampl_fe.index,columns = pyLife_mesh['mises'].index)
display(s_vm_scaled)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [29], line 5
      3 # Scaling with amplitude
      4 pyLife_mesh['mises'] = 2*pyLife_mesh['mises']/pyLife_mesh['mises'].max()
----> 5 ampl_fe = pd.DataFrame(data = amplitude, columns = ["ampl"] ,index =cycles)
      6 s_vm_scaled = pd.DataFrame(data = ampl_fe.values*pyLife_mesh['mises'].
↪transpose().values,index = ampl_fe.index,columns = pyLife_mesh['mises'].index)
      7 display(s_vm_scaled)

NameError: name 'amplitude' is not defined
```

5.1.8 Damage Calculation

[30]:
```
N = damage_calc.calc_N(s_vm_scaled,ignore_limits = True)
d_mesh_cycle =  1/(N.div(N.index.values, axis = 'index'))
#np.sum(data_act[range_mid > sn_curve_parameters["sigma_ak"]].values/N)

d_mesh = d_mesh_cycle.sum()
display(d_mesh)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [30], line 1
----> 1 N = damage_calc.calc_N(s_vm_scaled,ignore_limits = True)
      2 d_mesh_cycle =  1/(N.div(N.index.values, axis = 'index'))
      3 #np.sum(data_act[range_mid > sn_curve_parameters["sigma_ak"]].values/N)

NameError: name 's_vm_scaled' is not defined
```

[31]:
```
pyLife_mesh = pyLife_mesh.join(pd.DataFrame(data = d_mesh,columns = ['d']))
display(pyLife_mesh)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [31], line 1
----> 1 pyLife_mesh = pyLife_mesh.join(pd.DataFrame(data = d_mesh,columns = ['d']))
```

(continues on next page)

(continued from previous page)

```
      2 display(pyLife_mesh)

NameError: name 'd_mesh' is not defined
```

[32]:
```
# plotting using pyvista
pyLife_nodes = pyLife_mesh.groupby('node_id').mean()
mesh = pv.PolyData(pyLife_nodes[['x', 'y', 'z']].values)
mesh.point_arrays["d"] = pyLife_nodes["d"].values
mesh.plot(scalars="d",log_scale = True)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pandas/core/indexes/base.py:3803, in Index.get_loc(self, key, method,␣
↪tolerance)
   3802 try:
-> 3803     return self._engine.get_loc(casted_key)
   3804 except KeyError as err:

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pandas/_libs/index.pyx:138, in pandas._libs.index.IndexEngine.get_loc()

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pandas/_libs/index.pyx:165, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:5745, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:5753, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

KeyError: 'd'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In [32], line 4
      2 pyLife_nodes = pyLife_mesh.groupby('node_id').mean()
      3 mesh = pv.PolyData(pyLife_nodes[['x', 'y', 'z']].values)
----> 4 mesh.point_arrays["d"] = pyLife_nodes["d"].values
      5 mesh.plot(scalars="d",log_scale = True)

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pandas/core/frame.py:3804, in DataFrame.__getitem__(self, key)
   3802 if self.columns.nlevels > 1:
   3803     return self._getitem_multilevel(key)
-> 3804 indexer = self.columns.get_loc(key)
   3805 if is_integer(indexer):
   3806     indexer = [indexer]

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pandas/core/indexes/base.py:3805, in Index.get_loc(self, key, method,␣
↪tolerance)
   3803     return self._engine.get_loc(casted_key)
   3804 except KeyError as err:
-> 3805     raise KeyError(key) from err
   3806 except TypeError:
```

(continues on next page)

(continued from previous page)

```
   3807     # If we have a listlike key, _check_indexing_error will raise
   3808     #  InvalidIndexError. Otherwise we fall through and re-raise
   3809     #  the TypeError.
   3810     self._check_indexing_error(key)

KeyError: 'd'
```

[33]:
```
print("Maximal damage sum: %f" % d_mesh.max())
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [33], line 1
----> 1 print("Maximal damage sum: %f" % d_mesh.max())

NameError: name 'd_mesh' is not defined
```

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

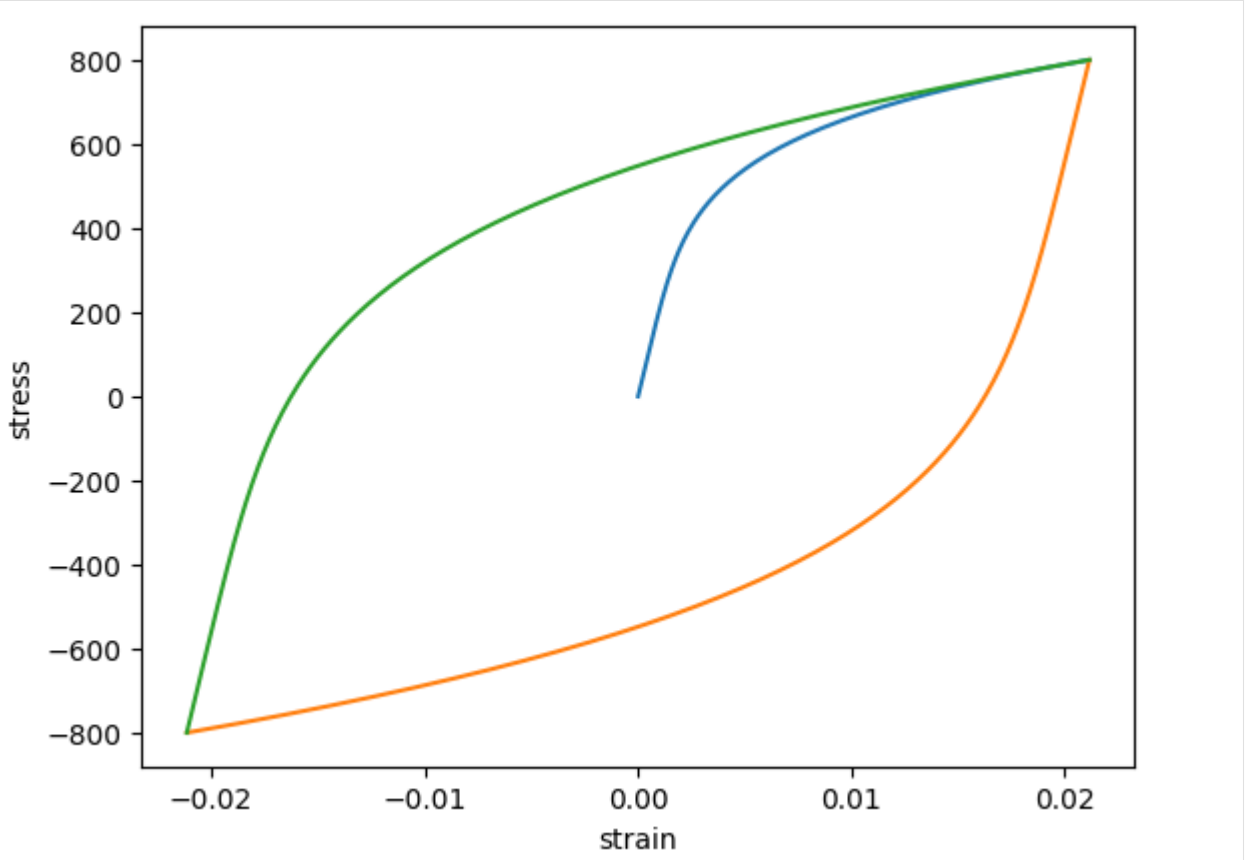
We calculate the lower branch of the hyteresis loop. By flipping it we get the upper branch.

[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 tool

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

5.3.1 Pylife Woehler-curve evaluation script

Initialization

[1]:
```
import numpy as np
import pandas as pd
from os import path
import sys, os
import json

import pylife.materialdata.woehler as woehler
import pylife.utils.diagrams.probability_data as probdiagram
from pylife.materialdata.woehler.controls.data_file_display import DataFileDisplay
from pylife.materialdata.woehler.controls.woehler_curve_analyzer_options import␣
↪WoehlerCurveAnalyzerOptions
from pylife.materialdata.woehler.controls.woehler_curve_data_plotter import␣
↪WoehlerCurveDataPlotter
```

(continues on next page)

(continued from previous page)

```
from pylife.materialdata.woehler.controls.whole_woehler_curve_plotter import␣
↪WholeWoehlerCurvePlotter
from pylife.materialdata.woehler.diagrams.woehler_curve_diagrams import␣
↪WoehlerCurveDiagrams
```

```
---------------------------------------------------------------------------
NoSectionError                            Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:238, in TheanoConfigParser.fetch_val_for_key(self,␣
↪key, delete_key)
    237 try:
--> 238     return self._theano_cfg.get(section, option)
    239 except InterpolationError:

File ~/.pyenv/versions/3.8.6/lib/python3.8/configparser.py:781, in RawConfigParser.
↪get(self, section, option, raw, vars, fallback)
    780 try:
--> 781     d = self._unify_values(section, vars)
    782 except NoSectionError:

File ~/.pyenv/versions/3.8.6/lib/python3.8/configparser.py:1149, in RawConfigParser._
↪unify_values(self, section, vars)
   1148     if section != self.default_section:
-> 1149         raise NoSectionError(section) from None
   1150 # Update with the entry specific variables

NoSectionError: No section: 'blas'

During handling of the above exception, another exception occurred:

KeyError                                  Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:354, in ConfigParam.__get__(self, cls, type_,␣
↪delete_key)
    353 try:
--> 354     val_str = cls.fetch_val_for_key(self.name, delete_key=delete_key)
    355     self.is_default = False

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:242, in TheanoConfigParser.fetch_val_for_key(self,␣
↪key, delete_key)
    241 except (NoOptionError, NoSectionError):
--> 242     raise KeyError(key)

KeyError: 'blas__ldflags'

During handling of the above exception, another exception occurred:

AttributeError                            Traceback (most recent call last)
Cell In [1], line 7
      4 import sys, os
      5 import json
----> 7 import pylife.materialdata.woehler as woehler
      8 import pylife.utils.diagrams.probability_data as probdiagram
      9 from pylife.materialdata.woehler.controls.data_file_display import␣
↪DataFileDisplay
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pylife/materialdata/woehler/__init__.py:26
     24 from .analyzers.probit import Probit
     25 from .analyzers.maxlike import MaxLikeInf, MaxLikeFull
---> 26 from .analyzers.bayesian import Bayesian

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/pylife/materialdata/woehler/analyzers/bayesian.py:22
     20 import numpy as np
     21 import pandas as pd
---> 22 import theano.tensor as tt
     23 import pymc3 as pm
     25 from .elementary import Elementary

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/__init__.py:83
     75 # This is the api version for ops that generate C code.  External ops
     76 # might need manual changes if this number goes up.  An undefined
     77 # __api_version__ can be understood to mean api version 0.
     78 #
     79 # This number is not tied to the release version and should change
     80 # very rarely.
     81 __api_version__ = 1
---> 83 from theano import scalar, tensor
     84 from theano.compile import (
     85     In,
     86     Mode,
   (...)
     93     shared,
     94 )
     95 from theano.compile.function import function, function_dump

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/__init__.py:20
      9 from theano.compile import SpecifyShape, specify_shape
     10 from theano.gradient import (
     11     Lop,
     12     Rop,
   (...)
     18     verify_grad,
     19 )
---> 20 from theano.tensor import nnet  # used for softmax, sigmoid, etc.
     21 from theano.tensor import sharedvar  # adds shared-variable constructors
     22 from theano.tensor import (
     23     blas,
     24     blas_c,
   (...)
     29     xlogx,
     30 )

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/nnet/__init__.py:3
      1 import warnings
----> 3 from . import opt
      4 from .abstract_conv import conv2d as abstract_conv2d
      5 from .abstract_conv import conv2d_grad_wrt_inputs, conv3d, separable_conv2d
```

(continues on next page)

(continued from previous page)

```
File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/nnet/opt.py:32
     24 from theano.tensor.nnet.blocksparse import (
     25     SparseBlockGemv,
     26     SparseBlockOuter,
     27     sparse_block_gemv_inplace,
     28     sparse_block_outer_inplace,
     29 )
     31 # Cpu implementation
---> 32 from theano.tensor.nnet.conv import ConvOp, conv2d
     33 from theano.tensor.nnet.corr import CorrMM, CorrMM_gradInputs, CorrMM_
↪gradWeights
     34 from theano.tensor.nnet.corr3d import Corr3dMM, Corr3dMMGradInputs,␣
↪Corr3dMMGradWeights

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/nnet/conv.py:20
     18 from theano.graph.basic import Apply
     19 from theano.graph.op import OpenMPOp
---> 20 from theano.tensor import blas
     21 from theano.tensor.basic import (
     22     NotScalarConstantError,
     23     as_tensor_variable,
     24     get_scalar_constant_value,
     25     patternbroadcast,
     26 )
     27 from theano.tensor.nnet.abstract_conv import get_conv_output_shape, get_conv_
↪shape_1axis

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/blas.py:163
    161 from theano.scalar import bool as bool_t
    162 from theano.tensor import basic as tt
--> 163 from theano.tensor.blas_headers import blas_header_text, blas_header_version
    164 from theano.tensor.opt import in2out, local_dimshuffle_lift
    165 from theano.tensor.type import values_eq_approx_remove_inf_nan

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/tensor/blas_headers.py:1016
    997             header += textwrap.dedent(
    998                 """\
    999                     static float sdot_(int* Nx, float* x, int* Sx, float* y,␣
↪int* Sy)
   (...)
   1010                     """
   1011             )
   1013     return header + blas_code
-> 1016 if not config.blas__ldflags:
   1017     _logger.warning("Using NumPy C-API based implementation for BLAS␣
↪functions.")
   1020 def mkl_threads_text():

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/configparser.py:358, in ConfigParam.__get__(self, cls, type_,␣
↪delete_key)
    356 except KeyError:
```

(continues on next page)

(continued from previous page)

```
    357     if callable(self.default):
--> 358         val_str = self.default()
    359     else:
    360         val_str = self.default

File ~/checkouts/readthedocs.org/user_builds/pylife/envs/1.1.4/lib/python3.8/site-
↪packages/theano/link/c/cmodule.py:2621, in default_blas_ldflags()
   2617 try:
   2618     if hasattr(numpy.distutils, "__config__") and numpy.distutils.__config__:
   2619         # If the old private interface is available use it as it
   2620         # don't print information to the user.
-> 2621         blas_info = numpy.distutils.__config__.blas_opt_info
   2622     else:
   2623         # We do this import only here, as in some setup, if we
   2624         # just import theano and exit, with the import at global
   (...)
   2630         # This happen with Python 2.7.3 |EPD 7.3-1 and numpy 1.8.1
   2631         # isort: off
   2632         import numpy.distutils.system_info  # noqa

AttributeError: module 'numpy.distutils.__config__' has no attribute 'blas_opt_info'
```

[2]:
```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from IPython.display import display
import warnings
import pdb
```

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

[3]:
```
file_name = 'data/woehler/fatigue-data-plain.csv'
```

Transforming data in csv to python arrays

[4]:
```
data = pd.read_csv(file_name, sep='\t')
data.columns=['load', 'cycles']
rb = DataFileDisplay(data)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [4], line 3
      1 data = pd.read_csv(file_name, sep='\t')
      2 data.columns=['load', 'cycles']
----> 3 rb = DataFileDisplay(data)
```

(continues on next page)

(continued from previous page)

```
NameError: name 'DataFileDisplay' is not defined
```

2. Enter the load cycle limit (The load-cycle value that seperates Fractures from Run-outs):

Note: in case the load cycle limit is the highest number found in the column set it to *max(data[:,1])*

[5]:
```
ld_cyc_lim = None
data = woehler.determine_fractures(data, ld_cyc_lim)
data
fatigue_data = data.fatigue_data
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [5], line 2
      1 ld_cyc_lim = None
----> 2 data = woehler.determine_fractures(data, ld_cyc_lim)
      3 data
      4 fatigue_data = data.fatigue_data

NameError: name 'woehler' is not defined
```

5.3.2 Parameters

[6]:
```
woehler_curve_analyzer_options = WoehlerCurveAnalyzerOptions(fatigue_data)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [6], line 1
----> 1 woehler_curve_analyzer_options = WoehlerCurveAnalyzerOptions(fatigue_data)

NameError: name 'WoehlerCurveAnalyzerOptions' is not defined
```

5.3.3 Visualization of Results

4. Choose the plot type to be visualized in the following cell

[7]:
```
woehler_curve = woehler_curve_analyzer_options.woehler_curve
analyzer = woehler_curve_analyzer_options.analyzer()
woehler_curve_data_plotter = WoehlerCurveDataPlotter(woehler_curve, fatigue_data,␣
↪analyzer)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [7], line 1
----> 1 woehler_curve = woehler_curve_analyzer_options.woehler_curve
      2 analyzer = woehler_curve_analyzer_options.analyzer()
      3 woehler_curve_data_plotter = WoehlerCurveDataPlotter(woehler_curve, fatigue_
↪data, analyzer)
```

(continues on next page)

(continued from previous page)

```
NameError: name 'woehler_curve_analyzer_options' is not defined
```

5. Choose the probability curve type to be visualized in the following cell

[8]:
```
probdiag_finite = probdiagram.ProbabilityDataDiagram(analyzer.pearl_chain_estimator(),
                                                     occurrences_name='Load',
                                                     title='Failure probability finite
↪')
probdiag_finite.plot()
if isinstance(analyzer, woehler.analyzers.probit.Probit):
    probdiag_infinite = probdiagram.ProbabilityDataDiagram(analyzer.pearl_chain_
↪estimator(),
                                                           occurrences_name='Load',
                                                           title='Failure probability␣
↪inffinite')
    probdiag_infinite.plot()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [8], line 1
----> 1 probdiag_finite = probdiagram.ProbabilityDataDiagram(analyzer.pearl_chain_
↪estimator(),
      2                                                     occurrences_name='Load',
      3                                                     title='Failure␣
↪probability finite')
      4 probdiag_finite.plot()
      5 if isinstance(analyzer, woehler.analyzers.probit.Probit):

NameError: name 'probdiagram' is not defined
```

Final Woehler-curve plot

6. Plot of the complete Woehler curve.

Choose the value of k_2 to plot the figure.

[9]:
```
whole_woehler_curve_plotter = WoehlerCurveDiagrams(woehler_curve, fatigue_data,␣
↪analyzer)
whole_woehler_curve_plotter.plot_fatigue_data()
whole_woehler_curve_plotter.plot_fitted_curve(k_2=12)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [9], line 1
----> 1 whole_woehler_curve_plotter = WoehlerCurveDiagrams(woehler_curve, fatigue_
↪data, analyzer)
      2 whole_woehler_curve_plotter.plot_fatigue_data()
      3 whole_woehler_curve_plotter.plot_fitted_curve(k_2=12)

NameError: name 'WoehlerCurveDiagrams' is not defined
```

[10]:
```
print(woehler_curve)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [10], line 1
----> 1 print(woehler_curve)

NameError: name 'woehler_curve' is not defined
```

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.hotspot
import pylife.vmap

import pyvista as pv

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

[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')
               .join_variable('DISPLACEMENT')
               .to_frame())
```

5.4.1 Equivalent stress calculation

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

5.4.2 Hot spot Calculation

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

```
                           x         y    z      mises  hotspot
element_id node_id
```

(continues on next page)

(continued from previous page)

```
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" % pyLife_mesh['hotspot'].max())
```

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

[6]:
```
# plotting using pyvista
pyLife_nodes = pyLife_mesh[['x', 'y', 'z', 'hotspot']].groupby('node_id').mean()
mesh = pv.PolyData(pyLife_nodes[['x', 'y', 'z']].values)
mesh.point_arrays["hotspot"] = pyLife_nodes["hotspot"].values
mesh.plot(scalars="hotspot",log_scale = False, off_screen=True)
```

```
AppLayout(children=(VBox(children=(HTML(value='<h3>hotspot</h3>'),␣
↪Dropdown(description='Colormap:', options=(...
```

First hotspot

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

```
                       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
```

[ ]:
```
```

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
```

(continues on next page)

(continued from previous page)

```
import pylife.mesh.gradient
import pylife.vmap

import pyvista as pv
```

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')
               .join_variable('DISPLACEMENT')
               .to_frame())
```

Calculate and add von Mises stress

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

Calculate stress gradient on von Mises stress

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

[5]:
```
pyLife_nodes = pd.concat([pyLife_mesh.groupby('node_id').mean(), grad], axis=1)
pyLife_nodes["abs_grad"] = np.linalg.norm(grad,axis = 1)
display(pyLife_nodes)
```

```
                 x         y    z         S11        S22  S33        S12  S13  \
node_id
1         7.900000  0.000000  0.0   -0.802995 -68.944250  0.0  -1.869146  0.0
2        20.000000  0.000000  0.0   30.042272  24.801415  0.0  -0.092955  0.0
3        20.000000  9.000000  0.0   30.024656   0.024460  0.0   0.005555  0.0
4         0.000000  9.000000  0.0   18.584425  -0.299882  0.0   5.858656  0.0
5         0.000000  6.300000  0.0  216.279488   5.181528  0.0  -6.305715  0.0
...            ...       ...  ...         ...        ...  ...        ...  ...
14610    -6.424311 -6.312416  0.0   52.471514  -6.999632  0.0  -3.310140  0.0
14611   -11.206949 -2.616344  0.0   10.610512  -1.571398  0.0 -10.754698  0.0
14612   -13.101126 -7.120056  0.0   46.328791   1.005923  0.0 -11.133379  0.0
14613   -10.574732 -4.750605  0.0   27.268641  -2.193988  0.0 -15.060814  0.0
14614   -13.389065 -5.709927  0.0   36.637676   3.454111  0.0 -14.996684  0.0

         S23            dx            dy   dz       mises         dx  \
node_id
1        0.0  5.717073e-03  3.114637e-36  0.0   68.842539  -5.616596
2        0.0  6.448063e-03  6.655135e-38  0.0   27.796941  -0.915970
3        0.0  5.691581e-03  1.163137e-04  0.0   30.012435  -0.000519
4        0.0 -3.303485e-36 -5.751588e-03  0.0   24.948312   0.122343
5        0.0  1.365477e-35 -5.435489e-03  0.0  215.130796  16.186186
...      ...           ...           ...  ...         ...        ...
14610    0.0 -3.627928e-03  1.724823e-03  0.0   56.589972   1.880459
14611    0.0 -5.765039e-03  4.447560e-05  0.0   21.879599  -0.800229
14612    0.0 -4.997560e-03  1.022665e-04  0.0   49.725466   0.651738
14613    0.0 -5.150582e-03  1.811491e-04  0.0   38.583754   0.381298
14614    0.0 -5.329776e-03  2.166348e-05  0.0   43.646014  -0.540695

               dy   dz    abs_grad
```

(continues on next page)

(continued from previous page)

```
node_id
1         5.359001  0.0   7.763056
2        -0.225004  0.0   0.943201
3        -0.000082  0.0   0.000526
4        -0.613853  0.0   0.625926
5         1.395853  0.0  16.246262
...            ...  ...        ...
14610    -4.367367  0.0   4.755000
14611    -6.246270  0.0   6.297321
14612    -5.209006  0.0   5.249620
14613    -6.330618  0.0   6.342090
14614    -6.376289  0.0   6.399172

[14614 rows x 17 columns]
```

[6]:
```
mesh = pv.PolyData(pyLife_nodes[['x', 'y', 'z']].values)
mesh.point_arrays["abs_grad"] = pyLife_nodes["abs_grad"].values
mesh.plot(scalars="abs_grad",log_scale = False)
```

```
ViewInteractiveWidget(height=768, layout=Layout(height='auto', width='100%'),␣
↪width=1024)
```

5.6 VMAP IO Demo

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.vmap
import pylife.stress.equistress
import pyvista as pv

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

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

[2]:
```
vm_mesh = pylife.vmap.VMAPImport("plate_with_hole.vmap")
```

[ ]:
```

```

[3]:
```
pyLife_mesh = (vm_mesh.make_mesh('1', 'STATE-2')
               .join_coordinates()
               .join_variable('STRESS_CAUCHY')
               .join_variable('DISPLACEMENT')
               .to_frame())
pyLife_mesh.sample(10)
```

[3]:
```
                             x         y    z         S11         S22  S33  \
element_id node_id
73         5168      9.977762  2.814110  0.0   6.202906 -12.397347  0.0
```

(continues on next page)

(continued from previous page)

```
879         5205     10.832771  1.218087  0.0   0.034201  -8.013414  0.0
735         944       4.304682  7.614532  0.0  75.976501   0.552377  0.0
3804        3760    -18.383881 -3.052231  0.0  33.285740  13.077278  0.0
455         1034      1.212672  8.591251  0.0  44.854668  -0.034998  0.0
1420        7955    -12.600000  9.000000  0.0  64.384834  -0.005875  0.0
1395        7865    -14.570934  1.273713  0.0  17.545462   8.563627  0.0
4577        4629     -8.282072 -7.820222  0.0  76.414841  -1.502231  0.0
571         777       8.121799  3.010482  0.0   0.445920 -27.766199  0.0
1621        8651     -8.470181  2.969894  0.0   2.845855 -22.873436  0.0

                          S12  S13  S23        dx        dy   dz
element_id node_id
73          5168    -10.235822  0.0  0.0  0.005410 -0.000214  0.0
879         5205     -4.701056  0.0  0.0  0.005735 -0.000048  0.0
735         944       7.389172  0.0  0.0  0.001712 -0.003518  0.0
3804        3760     -5.681770  0.0  0.0 -0.006374 -0.000084  0.0
455         1034      6.703181  0.0  0.0  0.000226 -0.005442  0.0
1420        7955      0.026578  0.0  0.0 -0.004523 -0.000300  0.0
1395        7865      5.216332  0.0  0.0 -0.006120  0.000023  0.0
4577        4629     -5.641978  0.0  0.0 -0.003399  0.001000  0.0
571         777       2.567467  0.0  0.0  0.005259 -0.000719  0.0
1621        8651      1.800592  0.0  0.0 -0.005524 -0.000535  0.0
```

[4]:
```
pyLife_mesh['mises'] = pyLife_mesh.equistress.mises()
```

5.6.1 We will use Pyvista for plotting FEM in the future

still work in progress, sorry dudes

[5]:
```
pyLife_nodes = pyLife_mesh[['x', 'y', 'z', 'mises']].groupby('node_id').mean()
mesh = pv.PolyData(pyLife_nodes[['x', 'y', 'z']].values)
mesh.point_arrays["mises"] = pyLife_nodes["mises"].values
mesh.plot(scalars="mises")
```

```
AppLayout(children=(VBox(children=(HTML(value='<h3>mises</h3>'), Dropdown(description=
↪'Colormap:', options=(('...
```

if you are using jupyter-lab you can use some lines, too:

for more information see here: https://pyvista-doc.readthedocs.io/en/latest/plotting/itk_plotting.html

[6]:
```
pl = pv.PlotterITK()
pl.add_mesh(mesh, scalars="mises", smooth_shading=True)
pl.show(True)
```

```
Viewer(geometries=[{'vtkClass': 'vtkPolyData', 'points': {'vtkClass': 'vtkPoints',
↪'name': '_points', 'numberO...
```

```
Viewer(geometries=[{'vtkClass': 'vtkPolyData', 'points': {'vtkClass': 'vtkPoints',
↪'name': '_points', 'numberO...
```

[ ]:
```

```

5.7 PSD Optimizer

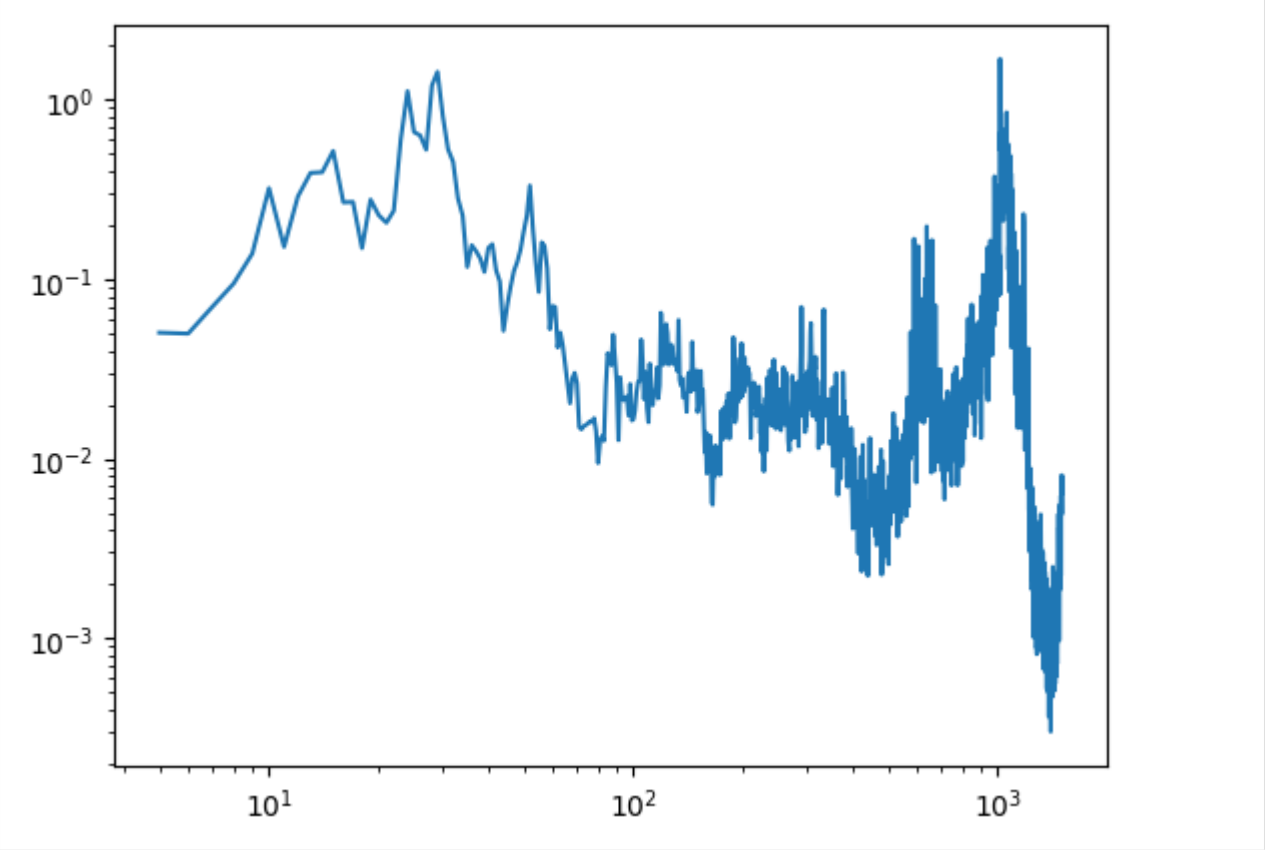
Example for the derivation of an equivalent PSD signal for shaker testing or other purposes. See the docu of the pylife psd_smoother function in the psdSignal class for more details

[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.index.values,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='')
```

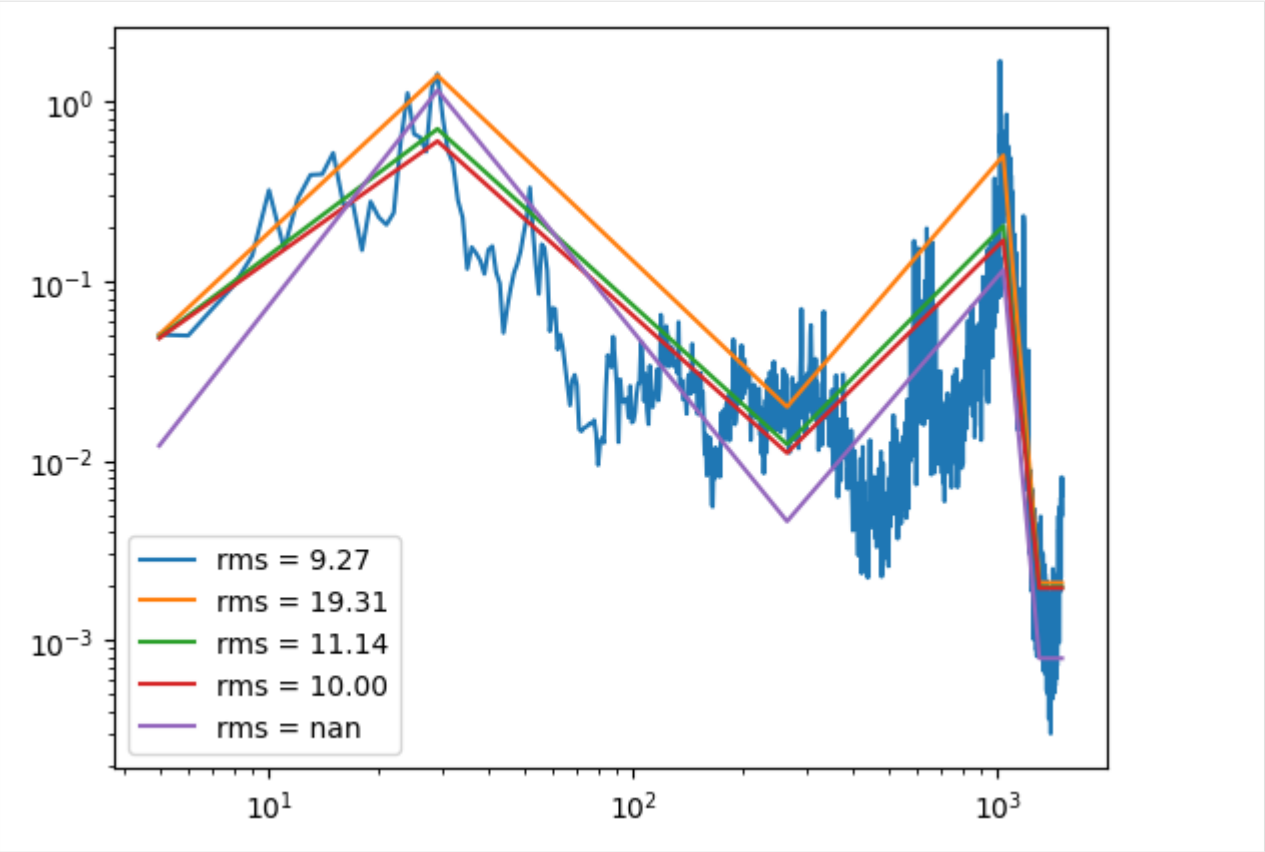


[4]:
```
fsel = np.asarray(fsel)
# please uncomment the following line. It is just for utomatization purpose
fsel = np.asarray([5,29,264,1035,1300,])
print(fsel)
```

```
[   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.index.values,psd,label = "rms = {:.2f}".format(rms_psd(psd)))
for ii in np.linspace(0,1,4):
    psd_fit = psdSignal.psd_smoother(psd,fsel,ii)
    plt.loglog(psd_fit.index.values,psd_fit,label = "rms = {:.2f}".format(rms_psd(psd_
↪fit)))
plt.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7f3ca24c25e0>
```


10^0
10^{-1}
10^{-2}
10^{-3}
10^1
10^2
10^3
rms = 9.27
rms = 19.31
rms = 11.14
rms = 10.00
rms = nan

[ ]:

CHAPTER 6

pyLife Reference

6.1 General

6.1.1 pyLife core

class pylife.core.signal.**PylifeSignal**(*pandas_obj*)
Base class for signal accessor classes

Parameters **pandas_obj** (*pandas.DataFrame or pandas.Series*) –

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()*

pylife.core.signal.**register_method**(*cls*, *method_name*)
Registers a method to a class derived from `PyifeSignal`

Parameters

- **cls** (*class*) – The class the method is registered to.
- **method_name** (*str*) – The name of the method

Raises

- `ValueError` – if *method_name* is already registered for the class

- `ValueError` – if *method_name* the class has already an attribute *method_name*

Notes

The function is meant to be used as a decorator for a function that is to be installed as a method for a class. The class is assumed to contain a pandas object in *self._obj*.

Examples

```
import pandas as pd
import pylife as pl

@pd.api.extensions.register_dataframe_accessor('foo')
class FooAccessor(pl.signal.PyifeSignal):
    def __init__(self, obj):
        # self._validate(obj) could come here
        self._obj = obj


@pl.signal.register_method(FooAccessor, 'bar')
def bar(df):
    return pd.DataFrame({'baz': df['foo'] + df['bar']})
```

```
>>> df = pd.DataFrame({'foo': [1.0, 2.0], 'bar': [-1.0, -2.0]})
>>> df.foo.bar()
   baz
0  0.0
1  0.0
```

class pylife.core.data_validator.**DataValidator**

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

Raises 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.StressTensorVoigtAccessor`

static fill_member(*key*, *values*)

get_missing_keys(*signal*, *keys_to_check*)
Gets 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.

6.2 Stress

6.2.1 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:

- 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.**StressTensorEquistressAccessor**(*pandas_obj*)

abs_max_principal()

max_principal()

min_principal()

mises()

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.**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

6.2.2 The `rainflow` module

Rainflow counting

A module performing rainflow counting

class pylife.stress.rainflow.**AbstractRainflowCounter**

The common base class for rainflow counters

Subclasses implementing a specific rainflow counting algorithm are supposed to implement a method `process()` that takes the signal samples as a parameter, append all the hysteresis loop limits to `self._loops_from` and `self.loops_to` and return `self`. The `process()` method is supposed to be implemented in a way, that the result is independent of the sample chunksize, so `rfc.process(signal)` should be equivalent to `rfc.process(signal[:n]).process(signal[n:])` for any 0 < n < signal length.

Todo:

- write a 4 point rainflow counter
- accept the histogram binning upfront so that loop information has not to be stored explicitly. This is important to ensure that the memory consumption remains O(1) rather than O(n).

---

get_rainflow_matrix

Calculates a histogram of the recorded loops

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.

get_rainflow_matrix_frame

Calculates a histogram of the recorded loops into a pandas.DataFrame.

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.DataFrame using a multi interval index in order to index data point for a given from/to value pair.

Return type pandas.DataFrame

residuals
Returns the residual turning points of the time signal so far

The residuals are the loops not (yet) closed.

class pylife.stress.rainflow.**RainflowCounterFKM**
Implements a rainflow counter as described in FKM non linear

See the here in the demo for an example.

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

process
Processes 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 *RainflowCounterFKM*

Example

```
>>> rfc = RainflowCounterFKM().process(samples)
>>> rfc.get_rainflow_matrix_frame(128)
```

class pylife.stress.rainflow.**RainflowCounterThreePoint**
Implements 3 point rainflow counting algorithm

See the here in the demo for an example.

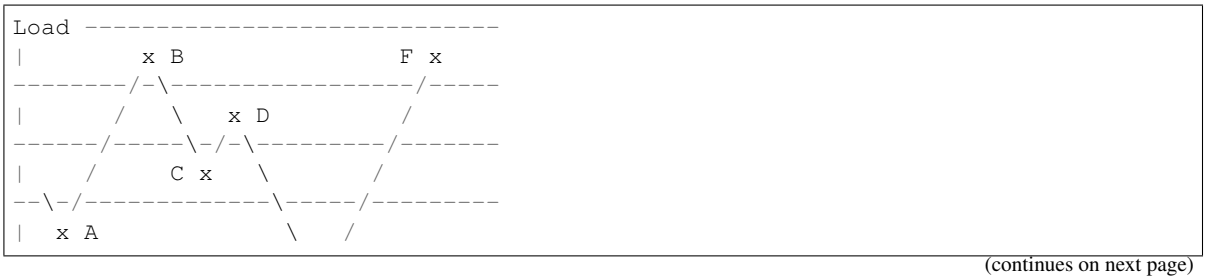
We take three turning points into account to detect closed hysteresis loops.

- 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).



(continues on next page)

(continued from previous page)

```
--------------------\-/-----------
|                    x E
-----------------------------------
|               Time
```

process
Processes 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 *RainflowCounterThreePoint*

Example

```
>>> rfc = RainflowCounterThreePoint().process(samples)
>>> rfc.get_rainflow_matrix_frame(128)
```

pylife.stress.rainflow.**get_turns**
Finds the turning points in a sample chunk

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

Returns

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

6.2.3 The `stresssignal` module

class pylife.stress.stresssignal.**CyclicStressAccessor**(*pandas_obj*)
DataFrame accessor class for cyclic stress data

Raises `AttributeError` – if there is no stress amplitide *sigma_a* key in the data

Notes

Base class to access `pandas.DataFrame` objects containing cyclic stress data consisting of at least a stress amplitude using the key *sigma_a*. Optionally an value for the stress ratio *R* or for meanstress *sigma_m* can be supplied. If neither *R* nor *sigma_a* are supplied a mean stress of zero i.e. *R=-1* is assumed.

Todo: Handle also input data with lower and upper stress.

constant_R(*R*)
Sets *sigma_m* in a way that *R* is a constant value.

Parameters **R** (*float*) – The value for *R*.

Returns **the accessed DataFrame**

Return type `pandas.DataFrame`

class pylife.stress.stresssignal.**StressTensorVoigtAccessor**(*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.StressTensorEquistressAccessor`.

6.2.4 The `timesignal` module

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
- gauss signals (not yet)
- log gauss signals (not yet)

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.

class pylife.stress.timesignal.**TimeSignalPrep**(*df*)

butter_bandpass(*lowcut*, *highcut*, *fs*, *order=5*)
Use the functonality of scipy

resample_acc(*sample_rate_new=1*)
Resampling the time series

Parameters

- **self** (*DataFrame*) –
- **time_col** (*str*) – column name of the time column
- **sample_rate_new** (*float*) – sample rate of the resampled time series

Returns

Return type DataFrame

running_stats_filt(*col*, *window_length=2048*, *buffer_overlap=0.1*, *limit=0.05*, *method='rms'*)
Calculates the running statistics of one DataFrame column and drops the rejected data points from the whole DataFrame.

Attention: Reset_index is used

Parameters

- **self** (*DataFrame*) –
- **col** (*str*) – column name of the signal for the runnings stats calculation
- **window_length** (*int*) – window length of the single time snippet, default is 2048
- **buffer_overlap** (*float*) – overlap parameter, 0.1 is equal to 10 % overlap of every buffer, default is 0.1
- **limit** (*float*) – limit value of skipping values, 0.05 for example skips all values which buffer method parameter is lower than 5% of the total max value, default is 0.05
- **method** (*str*) – method: 'rms', 'min', 'max', 'abs', default is 'rms'

Returns

Return type DataFrame

6.2.5 The frequencysignal module

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

Returns

Return type DataFrame

rms_psd()

6.2.6 The histogram module

pylife.stress.histogram.**combine_hist** (*hist_list*, *method='sum'*, *nbins=64*)
Performs the combination of multiple Histograms.

Parameters

- **hist_list** (*list*) – list of histograms with all histograms (saved as DataFrames in pyLife format)
- **method** (*str*) – method: 'sum', 'min', 'max', 'mean', 'std' default is 'sum'
- **nbins** (*int*) – number of bins of the combined histogram

Returns

- *DataFrame* – Combined histogram
- *list* – list with the reindexed input histograms

6.3 Strength

6.3.1 The infinite strength module

class pylife.strength.infinite.**InfiniteSecurityAccessor** (*pandas_obj*)
Compute Security factor for infinite cyclic stress.

factors (*woehler_data*, *allowed_failure_probability*)
compute security factor between strength and existing stress.

Parameters

- **woehler_data** (*dict*) – strength_inf: double - value for the allowed infinite cyclic strength strength_scatter: double - scattering of strength
- **allowed_failure_probability** (*double*) – Failure Probability we allow

Returns quotient of strength and existing stress

Return type double

6.3.2 The `meanstress` module

Meanstress routines

Mean stress transformation methods

- FKM Goodman
- Five Segment Correction

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

pylife.strength.meanstress.**FKM_goodman**(*Sa*, *Sm*, *M*, *M2*, *R_goal*)

Performs a mean stress transformation to R_goal according to the FKM-Goodman model

Parameters

- **Sa** – the stress amplitude
- **Sm** – the mean stress
- **M** – the mean stress sensitivity between R=-inf and R=0
- **M2** – the mean stress sensitivity beyond R=0
- **R_goal** – the R-value to transform to

Returns the transformed stress range

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

class pylife.strength.meanstress.**MeanstressHist**(*df*)

FKM_goodman(*haigh*, *R_goal*)

five_segment(*haigh*, *R_goal*)

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

FKM_goodman(*haigh*, *R_goal*)

five_segment(*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_50 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_50) for the SN-Curves, SD_50 is taken. If N_c is None, SD_50 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**(*Sa*, *Sm*, *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

6.3.3 The failure_probability module

class pylife.strength.failure_probability.**FailureProbability**(*strength_median*, *strength_std*)

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.

Parameters

- **strength_median** (*array_like, shape (N,)*) – The median value of the strength
- **strength_std** (*array_like, shape (N,)*) – The standard deviation of the strength

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.

pf_arbitrary_load(*load_values*, *load_pdf*)
Calculates the failure probability for an arbitrary load

Parameters

- **load_values** (*array_like, shape (N,)*) – The load values of the load distribution
- **load_pdf** (*array_like, shape (N,)*) – The probability density values for the `load_value` values to occur

Returns failure probability

Return type numpy.ndarray or float

pf_norm_load(*load_median*, *load_std*, *lower_limit=None*, *upper_limit=None*)
Failure probability for a log normal distributed load

Parameters

- **load_median** (*array_like, shape (N,) consistent with class parameters*) – The median of the load distribution for which the failure probability is calculated.
- **load_std** (*array_like, shape (N,) consistent with class parameters*) – The standard deviation of the load distribution
- **lower_limit** (*float, optional*) – The lower limit of the integration, default None
- **upper_limit** (*float, optional*) – The upper limit of the integration, default None

Returns failure probability

Return type numpy.ndarray or float

Notes

The log normal distribution of the load is determined by the load parameters. Only load distribution between `lower_limit` and `upper_limit` is considered.

For small values for `load_std` this function gives the same result as `pf_simple_load`.

Note: The load and strength distributions are transformed in a way, that the median of the load distribution is zero. This guarantees that in any case we can provide a set of relevant points to take into account for the integration.

pf_simple_load(*load*)
Failure probability for a simple load value

Parameters **load** (*array_like, shape (N,) consistent with class parameters*) – The load of for which the failure probability is calculated.

Returns failure probability

Return type numpy.ndarray or float

Notes

This is the case of a non statistical load. So failure occurs if the strength is below the given load, i.e. the strength' cdf at the load.

6.3.4 The `miner` module

Implementation of the miner rule for fatigue analysis

Currently, the following implementations are part of this module:

- Miner-elementar
- Miner-haibach

The source will be given in the function/class

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**(*ND_50*, *k_1*, *SD_50*)

Basic functions related to miner-rule (original)

Definitions will be based on the given references. Therefore, the original names are used so that they can be looked up easily.

Parameters

- **ND_50** (*float*) – number of cycles of the fatigue strength of the S/N curve [number of cycles]
- **k_1** (*float*) – slope of the S/N curve [unitless]
- **SD_50** (*float*) – fatigue strength of the S/N curve [MPa]

N_predict(*load_level*, *A=None*)

The predicted lifetime according to damage sum of the collective

Parameters

- **load_level** (*float*) – the maximum (stress) amplitude of the collective
- **A** (*float*) – the lifetime multiple A BEWARE: this relation is only valid in a specific representation of the predicted (Miner) lifetime where the sn-curve is expressed via the point of the maximum amplitude of the collective: N_predicted = N(S = S_max) * A

calc_A(*collective*)

Compute multiple of the lifetime

calc_zeitfestigkeitsfaktor(*N*, *total_lifetime=True*)
Calculate "Zeitfestigkeitsfaktor" according to Waechter2017 (p. 96)

collective = None

effective_damage_sum(*A*)
Compute 'effective damage sum' D_m

Refers to the formula given in Waechter2017, p. 99

Parameters **A** (*float or np.ndarray (with 1 element)*) – the multiple of the life-time

setup(*collective*)
Calculations independent from the instantation

Use the setup for functions that might require information that was not yet available at runtime during instantation.

Parameters **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting * column: class values in ascending order * column: accumulated number of cycles first entry is the total number of cycles then in a descending manner till the number of cycles of the highest stress class

class pylife.strength.miner.**MinerElementar**(*ND_50*, *k_1*, *SD_50*)
Implementation of Miner-elementar according to Waechter2017

V_FKM = None

V_haibach = None

calc_A(*collective=None*)
Compute the lifetime multiple according to miner-elementar

Described in Waechter2017 as "Lebensdauervielfaches, A_ele".

Parameters **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting * column: class values in ascending order * column: accumulated number of cycles first entry is the total number of cycles then in a descending manner till the number of cycles of the highest stress class

setup(*collective*)
Calculations independent from the instantation

Use the setup for functions that might require information that was not yet available at runtime during instantation.

Parameters **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting * column: class values in ascending order * column: accumulated number of cycles first entry is the total number of cycles then in a descending manner till the number of cycles of the highest stress class

class pylife.strength.miner.**MinerHaibach**(*ND_50*, *k_1*, *SD_50*)
Miner-modified according to Haibach (2006)

WARNING: Contrary to Miner-elementar, the lifetime multiple A is not constant but dependent on the evaluated load level!

Parameters **MinerBase** (*see*) –

A
the multiple of the life time initiated as dict Since A is different for each load level, the load level is taken as dict key (values are rounded to 0 decimals)

Type dict

N_predict(*load_level*, *A=None*, *ignore_inf_rule=False*)
The predicted lifetime according to damage sum of the collective

Parameters

- **load_level** (*float*) – the maximum (stress) amplitude of the collective
- **A** (*float*) – the lifetime multiple A BEWARE: this relation is only valid in a specific representation of the predicted (Miner) lifetime where the sn-curve is expressed via the point of the maximum amplitude of the collective: N_predicted = N(S = S_max) * A

calc_A(*load_level*, *collective=None*, *ignore_inf_rule=False*)
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

- **load_level** (*float > 0*) – load level in [MPa]
- **collective** (*np.ndarray (optional)*) – the collective can optionally be input to this function if it is not specified, then the attribute is used. If no collective exists as attribute (is set during setup) then an error is thrown
- **ignore_inf_rule** (*boolean*) – By default, the lifetime is returned as inf when the given load level is smaller than the lifetime (see Haibach eq. 3.2-62). This rule can be ignored if an estimate for the lifetime in the region below the fatigue strength is required for investigation.

Returns **A** – lifetime multiple return value is 'inf' if load_level < SD_50

Return type float > 0

evaluated_load_levels = None

setup(*collective*)
Calculations independent from the instantation

Use the setup for functions that might require information that was not yet available at runtime during instantation.

Parameters **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting * column: class values in ascending order * column: accumulated number of cycles first entry is the total number of cycles then in a descending manner till the number of cycles of the highest stress class

pylife.strength.miner.**get_accumulated_from_relative_collective**(*collective*)
Get collective with accumulated frequencies

This function can be used to transform a collective with relative frequencies.

Parameters **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting * column: class values in ascending order * column: relative number of cycles for each load class

6.3.5 The sn_curve module

class pylife.strength.sn_curve.**FiniteLifeBase**(*k_1*, *SD_50*, *ND_50*)
Base class for SN curve calculations - either in logarithmic or regular scale

k_1

class pylife.strength.sn_curve.**FiniteLifeCurve**(*k_1*, *SD_50*, *ND_50*)

Sample points on the finite life curve - either N or S (NOT logarithmic scale)

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

Consider: load collective and life curve have to be consistent:

- range vs range
- amplitude vs amplitude

Parameters

- **k** (*float*) – slope of the SN-curve
- **SD_50** (*float*) – lower stress limit in the finite life region
- **ND_50** (*float*) – number of cycles at stress SD_50

calc_N(*S*, *ignore_limits=False*)

Calculate number of cycles N for a given stress S

Parameters

- **S** (*array like*) – Stress (point(s) on the SN-curve)
- **ignore_limits** (*boolean*) – ignores the upper limit of the number of cycles generally it should be smaller than ND_50 (=the limit of the finite life region) but some special evaluation methods (e.g. according to marquardt2004) require extrapolation to estimate an equivalent stress

Returns **N** – number of cycles corresponding to the given stress value (point on the SN-curve)

Return type array like

calc_S(*N*, *ignore_limits=True*)

Calculate stress S for a given number of cycles N

Parameters

- **N** (*float*) – number of cycles
- **ignore_limits** (*boolean*) – ignores the upper limit of the number of cycles generally it should be smaller than ND_50 (=the limit of the finite life region) but some special evaluation methods (e.g. according to marquardt2004) require extrapolation to estimate an equivalent stress

Returns **S** – stress corresponding to the given number of cycles (point on the SN-curve)

Return type float

calc_damage(*loads*, *method='elementar'*, *index_name='range'*)

Calculate the damage based on the methods

- Miner elementar (k_2 = k)
- Miner Haibach (k_2 = 2k-1)
- Miner original (k_2 = -inf)

Consider: load collective and life curve have to be consistent:

- range vs range

- amplitude vs amplitude

Parameters

- **loads** (*pandas series histogram*) – loads (index is the load, column the cycles)
- **method** (*str*) –
  - 'elementar': Miner elementar (k_2 = k)
  - 'MinerHaibach': Miner Haibach (k_2 = 2k-1)
  - 'original': Miner original (k_2 = -inf)

Returns **damage** – damage for every load horizont based on the load collective and the method

Return type pd.DataFrame

class pylife.strength.sn_curve.**FiniteLifeLine**(*k*, *SD_50*, *ND_50*)

Sample points on the finite life line - either N or S (LOGARITHMIC SCALE)

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

Notes

In contrast to the case

Parameters

- **k** (*float*) – slope of the SN-curve
- **SD_50** (*float*) – lower stress limit in the finite life region
- **ND_50** (*float*) – number of cycles at stress SD_50

calc_N_log(*S_log*, *ignore_limits=False*)

Calculate number of cycles N_log for a given stress S_log

Parameters

- **S_log** (*float*) – logarithmic stress (point on the SN-curve)
- **ignore_limits** (*boolean*) – ignores the upper limit of the number of cycles generally it should be smaller than ND_50_log (=the limit of the finite life region) but some special evaluation methods (e.g. according to marquardt2004) require extrapolation to estimate an equivalent stress

Returns **N_log** – logarithmic number of cycles corresponding to the given stress value (point on the SN-curve)

Return type float

calc_S_log(*N_log*, *ignore_limits=False*)

Calculate stress logarithmic S_log for a given number of cycles N_log

Parameters

- **N_log** (*float*) – logarithmic number of cycles
- **ignore_limits** (*boolean*) – ignores the upper limit of the number of cycles generally it should be smaller than ND_50 (=the limit of the finite life region) but some special evaluation methods (e.g. according to marquardt2004) require extrapolation to estimate an equivalent stress

Returns **S_log** – logarithmic stress corresponding to the given number of cycles (point on the SN-curve)

Return type float

6.3.6 The `helpers` module

Small helper functions for fatigue analysis

class pylife.strength.helpers.**StressRelations**

Namespace for simple relations of stress / amplitude / R-ratio

Refer to: Haibach (2006), p. 21

static get_max_stress_from_amplitude(*amplitude*, *R*)

static get_mean_stress_from_amplitude(*amplitude*, *R*)

pylife.strength.helpers.**irregularity_factor**(*rainflow_matrix*, *residuals=array([], dtype=float64)*, *decision_bin=None*)

Calculate the irregularity factor of a turning point sequence based on a rainflow matrix and its residuals.

Two sided irregularity factor:

..math:: I = N_{mean crossings} / N_{turning points}

Parameters

- **rainflow_matrix** (*np.ndarray[int, int]*) – 2D-rainflow matrix (must be square shaped)
- **residuals** (*np.ndarray[int], Optional*) – 1D array of residuals to consider for accurate calculation. Consecutive duplicates are removed beforehand. Residuals must be provided as bin numbers. Hint: Transformation from physical to binned values possible via np.digitize.
- **decision_bin** (*int, Optional*) – Bin number that equals the mean (two-sided). If not provided the decision_bin is inferred by the matrix entries as the mean value based on the turning points and will be broadcasted to int-type.

Todo: Future version may provide the one-sided irregularity factor as a second option. Formula would be:

One sided irregularity factor:

I = N_{zero bin upwards crossing} / N_{peaks}

N_{zero bin upwards crossings} equals positive_mean_bin_crossing if *decision_bin* is set to the bin of physical 0. Inferring exact amount of peaks from rainflow-matrix and residuals is left to be done.

---

pylife.strength.helpers.**solidity_fkm**(*collective*, *k*)

Compute solidity according to the FKM guideline (2012)

Refer to: FKM-Richtlinie - 6. Auflage (2012) - S.58 - Gl. (2.4.55) + Gl. (2.4.55)

Parameters **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting

1. column: class values in ascending order

2. column: accumulated number of cycles first entry is the total number of cycles then in a descending manner till the number of cycles of the highest stress class k : float slope of the S/N curve

Returns **V** – Völligkeitswert (solidity)

Return type np.ndarray

pylife.strength.helpers.**solidity_haibach**(*collective*, *k*)

Compute solidity according to Haibach

Refer to: Haibach - Betriebsfestigkeit - 3. Auflage (2005) - S.271

Parameters

- **collective** (*np.ndarray*) – numpy array of shape (:, 2) where ":" depends on the number of classes defined for the rainflow counting
  1. column: class values in ascending order
  2. column: accumulated number of cycles first entry is the total number of cycles then in a descending manner till the number of cycles of the highest stress class
- **k** (*float*) – slope of the S/N curve

Returns **V** – Völligkeitswert (solidity)

Return type np.ndarray (1,)

6.4 Materiallaws

6.4.1 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.

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 stress span

Return type array-like float

Raises ValueError if delta_stress is negative

Notes

A Masing like behavior is assumed for the material as described in Kerbgrundkonzept.

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

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

Raises ValueError if stress is negative

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

Raises ValueError if stress is negative

6.4.2 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.accessor module

6.5.2 The bayesian module

6.5.3 The elementary module

class pylife.materialdata.woehler.analyzers.elementary.**Elementary**(*fatigue_data*)

analyze(***kw*)

bayesian_information_criterion()

pearl_chain_estimator()

6.5.4 The likelihood module

class pylife.materialdata.woehler.analyzers.likelihood.**Likelihood**(*fatigue_data*)

likelihood_finite(*SD*, *k*, *N_E*, *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

- **variables** – The start values to be optimized. (Endurance limit SD, Scatter in load direction 1/TS)

- **zone_inf** – The data that our log-likelihood function takes in. This data is found in the infinite zone.
- **load_cycle_limit** – The dependent variable that our model requires, in order to seperate the fractures from the runouts.

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*, *N_E*, *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** – The slope k_1 to be optimzed, unless the user fixed it.
- **N_E** – 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.
- **fractures** – The data that our log-likelihood function takes in. This data represents the fractured data.
- **zone_inf** – The data that our log-likelihood function takes in. This data is found in the infinite zone.
- **load_cycle_limit** – The dependent variable that our model requires, in order to seperate the fractures from the runouts.

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

6.5.5 The maxlike module

class pylife.materialdata.woehler.analyzers.maxlike.**MaxLikeFull**(*fatigue_data*)

class pylife.materialdata.woehler.analyzers.maxlike.**MaxLikeInf**(*fatigue_data*)

6.5.6 The pearl_chain module

class pylife.materialdata.woehler.analyzers.pearl_chain.**PearlChainProbability**(*fractures*, *slope*)

normed_cycles

normed_load

6.5.7 The `probit` module

class pylife.materialdata.woehler.analyzers.probit.**Probit**(*fatigue_data*)

fitter()

6.6 Mesh utilities

6.6.1 The *meshsignal* module

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 *PlainMeshAccessor*) there is no further relation between the items is known, whereas a complete FEM mesh (see *MeshAccessor*) 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())
>>> 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
```

class pylife.mesh.meshsignal.**MeshAccessor**(*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 MeshAccessor 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:

PlainMeshAccessor: accesses meshes without connectivity information pandas.api.extensions.register_dataframe_accessor(): concept of DataFrame accessors

Examples

For an example see meshplot.

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

class pylife.mesh.meshsignal.**PlainMeshAccessor**(*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 PlainMeshAccessor describes meshes whose only geometrical information is the coordinates of the nodes or elements. Unlike *MeshAccessor* they don't know about connectivity, not even about elements and nodes.

See also:

MeshAccessor: accesses meshes with connectivity information pandas.api.extensions.register_dataframe_accessor(): concept of DataFrame accessors

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

6.6.2 The meshplot module

class pylife.mesh.meshplot.**PlotmeshAccessor**(*pandas_obj*)
Plot a value on a 2d mesh

The accessed DataFrame must be accessible by meshsignal.MeshAccessor.

plot(*axis*, *value_key*, ***kwargs*)
Plot the accessed dataframe

Parameters

- **ax** (*matplotlib.pyplot.axis*) – The axis to plot on
- **value_key** (*str*) – The column name in the accessed DataFrame
- ****kwargs** – Arguments passed to the matplotlib.collections.PatchCollection, like for example cmap

See also:

pandas.api.extensions.register_dataframe_accessor()

6.6.3 The meshmapping module

Mesh Mapping

Map values of one FEM mesh into another

class pylife.mesh.meshmapping.**MeshmapperAccessor**(*pandas_obj*)
Mapper to map points of one mesh to another

Notes

The accessed DataFrame needs to be accessible by a PlainMeshAccessor.

process(*from_df*, *value_key*, *method='linear'*)
Performs the mapping

Parameters **from_df** (pandas.DataFrame accessible by a PlainMeshAccessor.) – The DataFrame that is to be mapped to the accessed one. Needs to have the same dimensions (2D or 3D) as the accessed one

6.6.4 The hotspot module

class pylife.mesh.hotspot.**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 intergers 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

6.6.5 The gradient module

class pylife.mesh.gradient.**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 neigbor nodes usine 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 `['dx', 'dy', 'dz']`.

Return type pd.DataFrame

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
```

(continues on next page)

(continued from previous page)

```
           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

Returns

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

Returns

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

Returns

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.

Returns

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
```

(continues on next page)

(continued from previous page)

[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()`.

Returns

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
...                       ...       ...  ...        ...       ...  ...
↪...  ...  ...
```

(continues on next page)

(continued from previous page)

```
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*)

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*) –

Returns

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

Returns

Return type self

add_integration_types(*content*)
Creates system dataset IntegrationTypes with the given content

Parameters **content** (*the content of the dataset*) –

Returns

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*) –

Returns

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

Returns

Return type self

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

Returns

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

Returns

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

Returns

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.**scatteringRange2std**(*T_inv*)

Converts a inverted scattering range (1/T) into standard deviation

Parameters **T_inv** (*float*) – inverted scattering range

Returns **std** – standard deviation corresponding to 1/T assuming a normal distribution

Return type float

Notes

Actually *1/(2*norm.ppf(0.9))*np.log10(T_inv)*

Inverse of *std2scatteringRange()*

pylife.utils.functions.**std2scatteringRange**(*std*)

Converts a standard deviation into inverted scattering range (1/T)

Parameters **std** (*float*) – standard deviation

Returns **T_inv** – inverted scattering range corresponding to *std* assuming a normal distribution

Return type float

Notes

Actually *10**(2*norm.ppf(0.9)*std*

Inverse of *scatteringRange2std()*

6.8.2 The `utils.probability_data` moduble

class pylife.utils.probability_data.**ProbabilityFit**(*probs*, *occurrences*)

intercept

occurrences

percentiles

slope

CHAPTER 7

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.

Your contribution must be licensed under the Apache-2.0 license, the license used by this project.

7.1 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 a 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.

7.2 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.

7.2.1 Docstrings

Document your public API classes, methods, functions and attributes using numpy style docstings unless the naming is *really* self-explanatory.

7.2.2 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.

7.3 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 refactured without changing the feature set

7.4 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.

7.5 Sign your work

This project tracks patch provenance and licensing using a modified Developer Certificate of Origin (DCO; from OSDL) and Signed-off-by tags initially developed by the Linux kernel project.

```
pyLife Developer's Certificate of Origin.  Version 1.0

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 "Apache License, Version 2.0"
    ("Apache-2.0"); or

(b) The contribution is based upon previous work that is covered by
    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 Apache-2.0 license;
    or

(c) The contribution was provided directly to me by some other
    person who certified (a) or (b) 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
    metadata and personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
    consistent with this project and the requirements of the Apache-2.0
```

(continues on next page)

(continued from previous page)

```
    license or any open source license(s) involved, where they are
    relevant.

(e) I am granting the contribution to this project under the terms of
    Apache-2.0.

    http://www.apache.org/licenses/LICENSE-2.0
```

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 pyLife Developer's Certificate of Origin 1.0 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.

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.

7.5.1 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.

7.6 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 8

pyLife coding style guidelines

8.1 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. 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.

8.1.1 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] (https://en.wikipedia.org/wiki/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. You might want to take the time and watch [these two] (https://www.youtube.com/watch?v=7EmboKQH8lM), they are fun to watch.

8.2 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.

8.3 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.

8.4 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.

8.4.1 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.

8.4.2 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:
    ...
```

8.4.3 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.

8.4.4 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.

8.4.5 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.

8.5 Structuring of the code

8.5.1 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'

    def public_method(self):
    ...

    def _private_method(self):
```

8.5.2 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.

8.5.3 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.

8.5.4 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
    ... # 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
```

(continues on next page)

(continued from previous page)

```
    ... # 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()

    # 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 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V