FFPACK

Release 0.3.0

Dongping Zhu

May 20, 2023

ABOUT:

FFPACK (Fatigue and Fracture PACKage) is a open source Python library for fatigue and fracture evaluation.

Check out the *Usage* section for further information, including how to create a *Environment* for the project.

Note: Please select the correct version of FFPACK documentation. The version can be found at the upper left under the *FFPACK* title. Three versions of documentation are provided:

- *stable*: the documentation for the most recent released version.
- *latest*: the documentation for *main* branch.
- *docs*: the documentation for *docs* branch.

The *docs* branch and associated *docs* documentation represents the newest documentation under development. Once the documentation is finished, the changes will be merged into *main* branch and the *latest* version will be updated.

To switch the version, Click *Read the Docs* at the bottom left.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 FFPACK - Fatigue and Fracture PACKage

1.1.1 Purpose

FFPACK (Fatigue and Fracture PACKage) is an open-source Python library for fatigue and fracture analysis. It supports ASTM cycle counting, load sequence generation, fatigue damage evaluation, etc. A lot of features are under active development. FFPACK is designed to help engineers analyze fatigue and fracture behavior in engineering practice.

1.1.2 Installation

FFPACK can be installed via PyPI:

```
pip install ffpack
```

1.1.3 Usage

The following example shows the usage of ASTM rainflow counting,

```
# Import the ASTM rainflow counting function
from ffpack.lcc import astmRainflowCounting

# Prepare the data
data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]

# Get counting results
results = astmRainflowCounting( data )
```

See the package document for more details and examples.

1.1.4 Status

FFPACK is currently under active development.

1.1.5 Contents

        - Spectral representation
- Load spectra and matrices
    - Cycle counting matrix
        - ASTM simple range counting matrix
        - ASTM range pair counting matrix
        - ASTM rainflow counting matrix
        - ASTM rainflow counting matrix for repeating history
        - Johannesson min max counting matrix
        - Rychlik rainflow counting matrix
        - Four point rainflow counting matrix
    - Wave spectra
        - Jonswap spectrum
        - Pierson Moskowitz spectrum
        - ISSC spectrum
        - Gaussian Swell spectrum
        - Ochi-Hubble spectrum
    - Wind spectra
        - Davenport spectrum with drag coefficient
        - Davenport spectrum with roughness length
        - EC1 spectrum
        - IEC spectrum
        - API spectrum
    - Sequence spectra
        - Periodogram spectrum
        - Welch spectrum
- Random and probabilistic model
    - Metropolis-Hastings algorithm
        - Metropolis-Hastings sampler
        - Au modified Metropolis-Hastings sampler
    - Nataf algorithm
        - Nataf transformation
- Risk and reliability model
    - First order second moment
        - Mean value FOSM
    - First order reliability method
        - Hasofer-Lind-Rackwitz-Fiessler FORM

        * Constrained optimization FORM
    - Second order reliability method
        * Breitung SORM
        * Tvedt SORM
        * Hohenbichler and Rackwitz SORM
    - Simulation based reliability method
        * Subset simulation
- Utility
    - Aggregation
        * Cycle counting aggregation
    - Counting matrix
        * Counting results to counting matrix
    - Derivatives
        * Derivative
        * Central derivative weights
        * Gradient
        * Hessian matrix
    - Digitization
        * Sequence digitization
    - Fitter
        * SN curve fitter
    - Sequence filter
        * Sequence peakValley filter
        * Sequence hysteresis filter

1.1.6 Document

You can find the latest documentation for setting up `FFPACK` at the Read the Docs site.

1.1.7 Credits

This project was made possible by the help from DM2L lab.

1.1.8 License

GPLv3

1.2 Usage

1.2.1 Environment

We encourage to use conda for environment management but it is not necessary.

To create an environment with a specific Python version,

```
$ conda create -n ffpackEnv python=3.9
$ conda activate ffpackEnv
```

1.2.2 Installation

To use FFPack, install it using pip,

```
$ pip install ffpack
```

1.2.3 Develop version

To try the develop version, clone the source code from the GitHub,

```
$ git clone https://github.com/dpzhuX/ffpack.git
```

Then install it with *pip*,

```
$ pip install -e .
```

We recommend to run the develop version in a seperate environment.

1.3 Fatigue damage model (fdm)

1.3.1 Palmgren-miner damage model

Palmgren-miner damage model also known as the linear damage model is one of the famous damage models used in the engineering field. Based on the Palmgren-miner's model, the cumulative damage can be expressed by the following equation,

$$D = \sum \frac{C_i}{F_i}$$

where C_i and F_i are the counting cycles and the failure cycles at a specific load level.

In essence, the Palmgren-miner damage model treats the fatigue damage on different load levels separately. Therefore, the total damage can be calculated by adding the damage from each load level. Although a discrepancy can be found between the experimental results and the Palmgren-miner damage model, it is still widely used due to its simplicity.

Reference:

- Palmgren, A.G., 1924. Die Lebensdaur von Kugellagern [Life Length of Roller Bearings]. Zeitschrift des Vereines Deutscher Ingenieure (VDI Zeitschrift), 68(14), pp.339-341.
- Miner, M.A., 1945. Cumulative damage in fatigue. Journal of Applied Mechanics 12(3): A159–A164.

Naive Palmgren-miner damage model

Function `minerDamageModelNaive` implements the native Palmgren-miner damage model.

The naive Palmgren-miner damage model refers to the damage calculation directly based on the aforementioned equation. When we know the counting cycles and failure cycles at each level, then the total damage can be calculated by summing the damage from all load levels.

Function help

[1]:
```
from ffpack.fdm import minerDamageModelNaive
help( minerDamageModelNaive )
```

```
Help on function minerDamageModelNaive in module ffpack.fdm.minerModel:

minerDamageModelNaive(fatigueData)
    Naive Palmgren-miner damage model directly calcuates the damage results.

    Parameters
    ----------
    fatigueData: 2d array
        Paired counting and experimental data under each load condition,
        e.g., [ [ C1, F1 ], [ C2, F2 ], ..., [ Ci, Fi ] ]
        where Ci and Fi represent the counting cycles and failure cycles
        under the same load condition.

    Returns
    -------
    rst: scalar
        Fatigue damage calculated based on the Palmgren-miner model

    Raises
    ------
    ValueError
        If fatigueData length is less than 1.
        If counting cycles is less than 0.
        If number of failure cycles is less than or equal 0.
        If number of counting cycles is large than failure cycles.


    Examples
    --------
    >>> from ffpack.fdm import minerDamageModelNaive
    >>> fatigueData = [ [ 10, 100 ], [ 200, 2000 ] ]
    >>> rst = minerDamageModelNaive( fatigueData )
```

Example with default values

[2]:
```
nmdrFatigueData =  [ [ 10, 100 ], [ 200, 2000 ] ]

nmdrResults = minerDamageModelNaive( nmdrFatigueData )
```

[3]:
```
print( nmdrResults )
```

```
0.2
```

Classic Palmgren-miner damage model

Function `minerDamageModelClassic` implements the classic Palmgren-miner damage model.

The classic Palmgren-miner damage model can calculate the total damage based on the experimental SN curve. Since the load level for counting cycles might be unavailable for failure cycles, the experimental SN curve will be fitted first and determine the failure cycles at the same load level.

Notes

The load levels under or equal to the fatigueLimit will be ignored for fatigue damage calculation since these load levels do not contribute to the fatigue damage.

Function help

[4]:
```
from ffpack.fdm import minerDamageModelClassic
help( minerDamageModelClassic )
```

```
Help on function minerDamageModelClassic in module ffpack.fdm.minerModel:

minerDamageModelClassic(lccData, snData, fatigueLimit)
    Classical Palmgren-miner damage model calculates the damage results
    based on the SN curve.

    Parameters
    ----------
    lccData: 2d array
        Load cycle counting results in a 2D matrix,
        e.g., [ [ value, count ], ... ]

    snData: 2d array
        Experimental SN data in 2D matrix,
        e.g., [ [ N1, S1 ], [ N2, S2 ], ..., [ Ni, Si ] ]

    fatigueLimit: scalar
        Fatigue limit indicating the minimum S that can cause fatigue.

    Returns
    -------
    rst: scalar
        Fatigue damage calculated based on the Palmgren-miner model.
```

(continues on next page)

(continued from previous page)

```
    Raises
    ------
    ValueError
        If the lccData dimension is not 2.
        If the lccData length is less than 1.

    Examples
    --------
    >>> from ffpack.fdr import minerDamageModelClassic
    >>> lccData = [ [ 1, 100 ], [ 2, 10 ] ]
    >>> snData = [ [ 10, 3 ], [ 1000, 1 ] ]
    >>> fatigueLimit = 0.5
    >>> rst = minerDamageModelClassic( lccData, snData, fatigueLimit )
```

Example with default values

[5]:
```
cmdrLccData = [ [ 1, 100 ], [ 2, 10 ] ]
cmdrSnData = [ [ 10, 3 ], [ 1000, 1 ] ]
cmdrFatigueLimit = 0.5

cmdrResults = minerDamageModelClassic( cmdrLccData, cmdrSnData, cmdrFatigueLimit )
```

[6]:
```
print( "{:.2f}".format(cmdrResults) )
```

```
0.20
```

1.4 Load counting and correction (lcc)

1.4.1 ASTM counting methods

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

ASTM level crossing counting

Function `astmLevelCrossingCounting` implements the level crossing counting method in ASTM E1049-85 (2017): sec 5.1.1.

By the definition:

> One count is recorded each time the positive sloped portion of the load exceeds a preset level above the reference load, and each time the negative sloped portion of the load exceeds a preset level below the reference load. Reference load crossings are counted on the positive sloped portion of the loading history.

Function help

[2]:
```
from ffpack.lcc import astmLevelCrossingCounting
help( astmLevelCrossingCounting )
```

```
Help on function astmLevelCrossingCounting in module ffpack.lcc.astmCounting:

astmLevelCrossingCounting(data, refLevel=0.0, levels=None, aggregate=True)
    ASTM level crossing counting in E1049-85: sec 5.1.1.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    refLevel: scalar, optional
        Reference level.
    levels: 1d array
        Self-defined levels for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ crossPoint1, corssPoint2, ... ], will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 2 or the data dimension is not 1.

    Examples
    --------
    >>> from ffpack.lcc import astmLevelCrossingCounting
    >>> data = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
    >>>          -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]
    >>> rst = astmLevelCrossingCounting( data )
```

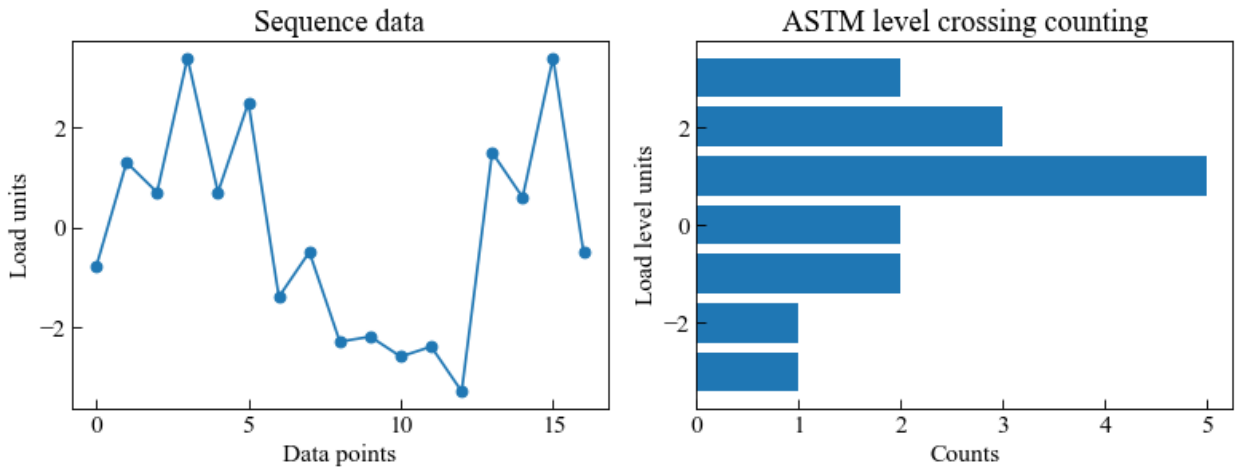
Example with default values

[3]:
```
astmLccSequenceData = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
                        -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]

astmLccCountingResults = astmLevelCrossingCounting( astmLccSequenceData )
```

[4]:
```
print( astmLccCountingResults )
```

```
[[-3.0, 1.0], [-2.0, 1.0], [-1.0, 2.0], [0.0, 2.0], [1.0, 5.0], [2.0, 3.0], [3.0, 2.0]]
```

[5]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( astmLccSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( astmLccCountingResults )[ :, 0 ],
          np.array( astmLccCountingResults )[ :, 1 ] )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Load level units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM level crossing counting" )

plt.tight_layout()
plt.show()
```



ASTM peak counting

Function `astmPeakCounting` implements the peak counting method in ASTM E1049-85 (2017): sec 5.2.1.

By the definition:

> Peaks above the reference load level are counted, and valleys below the reference load level are counted.

Function help

[6]:
```
from ffpack.lcc import astmPeakCounting
help( astmPeakCounting )
```

```
Help on function astmPeakCounting in module ffpack.lcc.astmCounting:

astmPeakCounting(data, refLevel=None, aggregate=True)
    ASTM peak counting in E1049-85: sec 5.2.1.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    refLevel: scalar, optional
        Reference level.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ peak1, peak2, ... ], will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 2 or the data dimension is not 1.

    Examples
    --------
    >>> from ffpack.lcc import astmPeakCounting
    >>> data = [ 0.0, 1.5, 0.5, 3.5, 0.5, 2.5, -1.5, -0.5, -2.5,
    >>>          -2.0, -2.7, -2.5, -3.5, 1.5, 0.5, 3.5, -0.5 ]
    >>> rst = astmPeakCounting( data )
```

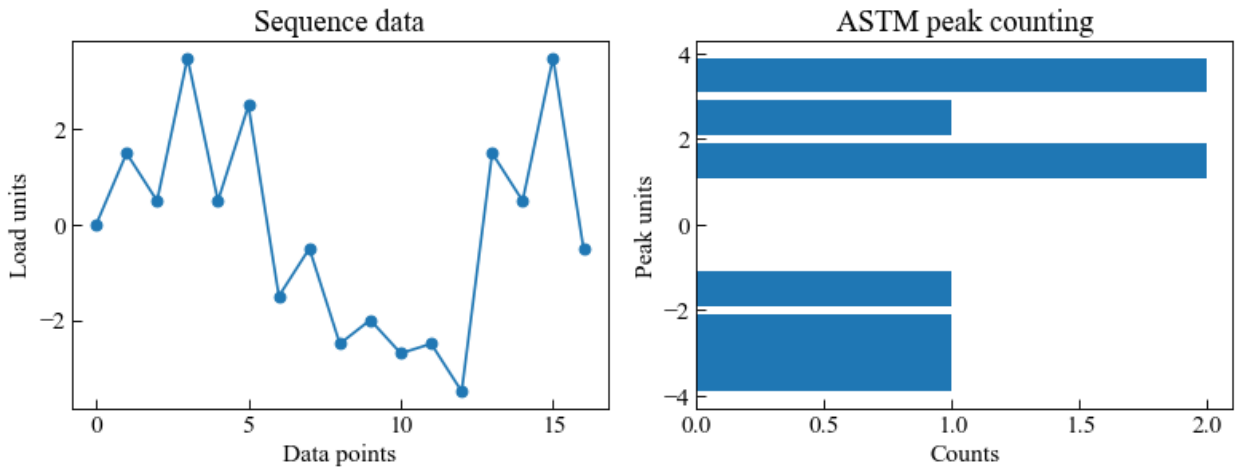
Example with default values

[7]:
```
astmPcSequenceData = [ 0.0, 1.5, 0.5, 3.5, 0.5, 2.5, -1.5, -0.5, -2.5,
                       -2.0, -2.7, -2.5, -3.5, 1.5, 0.5, 3.5, -0.5 ]

astmPcCountingResults = astmPeakCounting( astmPcSequenceData )
```

[8]:
```
print( astmPcCountingResults )
```

```
[[-3.5, 1.0], [-2.7, 1.0], [-2.5, 1.0], [-1.5, 1.0], [1.5, 2.0], [2.5, 1.0], [3.5, 2.0]]
```

[9]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( astmPcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( astmPcCountingResults )[ :, 0 ],
          np.array( astmPcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Peak units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM peak counting" )

plt.tight_layout()
plt.show()
```



ASTM simple range counting

Function `astmSimpleRangeCounting` implements the simple range counting method in ASTM E1049-85 (2017): sec 5.3.1.

By the definition:

> A range is defined as the difference between two successive reversals, the range being positive when a valley is followed by a peak and negative when a peak is followed by a valley. If only positive or only negative ranges are counted, then each is counted as one cycle. If both positive and negative ranges are counted, then each is counted as one-half cycle.

Function help

[10]:
```
from ffpack.lcc import astmSimpleRangeCounting
help( astmSimpleRangeCounting )
```

```
Help on function astmSimpleRangeCounting in module ffpack.lcc.astmCounting:

astmSimpleRangeCounting(data, aggregate=True)
    ASTM simple range counting in E1049-85: sec 5.3.1.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ],
        [ rangeStart2, rangeEnd2, count2 ], ... ], will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 2 or the data dimension is not 1.

    Examples
    --------
    >>> from ffpack.lcc import astmSimpleRangeCounting
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst = astmSimpleRangeCounting( data )
```

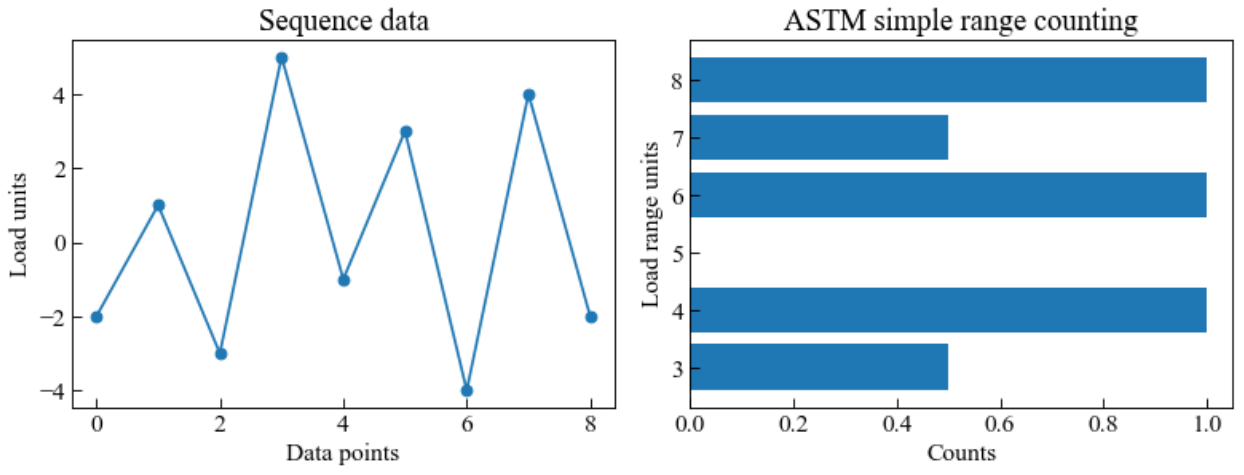
Example with default values

[11]:
```
astmSrcSequenceData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]

astmSrcCountingResults = astmSimpleRangeCounting( astmSrcSequenceData )
```

[12]:
```
print( astmSrcCountingResults )
```

```
[[3.0, 0.5], [4.0, 1.0], [6.0, 1.0], [7.0, 0.5], [8.0, 1.0]]
```

[13]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( astmSrcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( astmSrcCountingResults )[ :, 0 ],
          np.array( astmSrcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM simple range counting" )

plt.tight_layout()
plt.show()
```



ASTM range pair counting

Function `astmRangePairCounting` implements the range pair counting method in ASTM E1049-85 (2017): sec 5.4.3.

By the definition:

> The range-paired method counts a range as a cycle if it can be paired with a subsequent loading in the opposite direction.

Function help

[14]:
```
from ffpack.lcc import astmRangePairCounting
help( astmRangePairCounting )
```

```
Help on function astmRangePairCounting in module ffpack.lcc.astmCounting:

astmRangePairCounting(data, aggregate=True)
    ASTM range pair counting in E1049-85: sec 5.4.3.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ],
        [ rangeStart2, rangeEnd2, count2 ], ... ], will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 2 or the data dimension is not 1.

    Examples
    --------
    >>> from ffpack.lcc import astmRangePairCounting
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst = astmRangePairCounting( data )
```

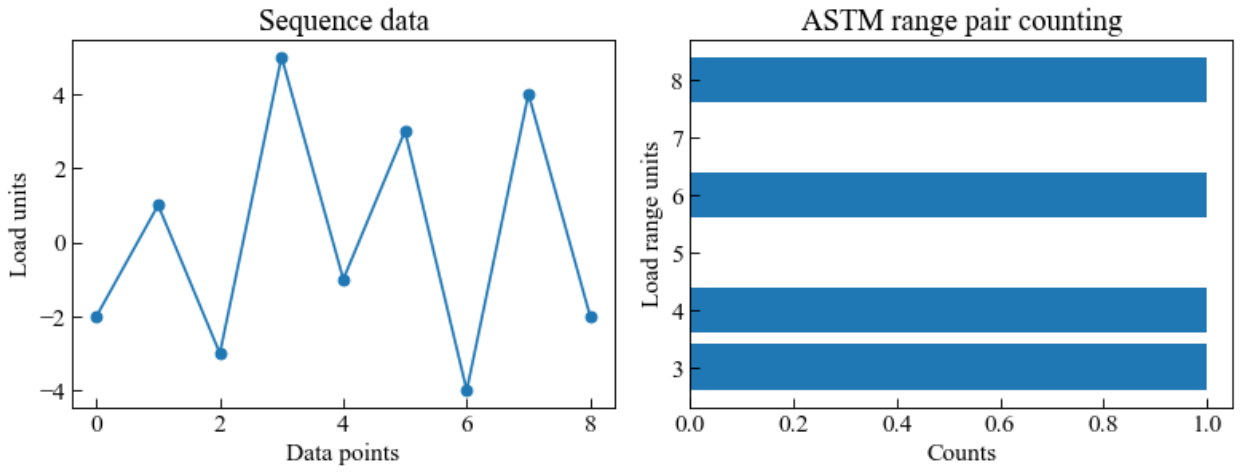
Example with default values

[15]:
```
astmRpcSequenceData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]

astmRpcCountingResults = astmRangePairCounting( astmRpcSequenceData )
```

[16]:
```
print( astmRpcCountingResults )
```

```
[[3.0, 1.0], [4.0, 1.0], [6.0, 1.0], [8.0, 1.0]]
```

[17]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( astmRpcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( astmRpcCountingResults )[ :, 0 ],
          np.array( astmRpcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM range pair counting" )

plt.tight_layout()
plt.show()
```



ASTM rainflow counting

Function `astmRainflowCounting` implements the rainflow counting method in ASTM E1049-85 (2017): sec 5.4.4.

Notes

The original rainflow counting method in ASTM does not provide any preprocessing method for the sequence data or postprocessing method for counting results. To get meaningful results, we provide the preprocessing `sequenceDigitization` function to digitize the input sequence data with a specific resolution, such as 0.5, or the postprocessing `cycleCountingAggregation` function to aggregate the cycle counting results. See the following examples for details.

Function help

[18]:
```
from ffpack.lcc import astmRainflowCounting
help( astmRainflowCounting )
```

```
Help on function astmRainflowCounting in module ffpack.lcc.astmCounting:

astmRainflowCounting(data, aggregate=True)
    ASTM rainflow counting in E1049-85: sec 5.4.4.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ],
        [ rangeStart2, rangeEnd2, count2 ], ... ], will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 2 or the data dimension is not 1.

    Examples
    --------
    >>> from ffpack.lcc import astmRainflowCounting
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst = astmRainflowCounting( data )
```

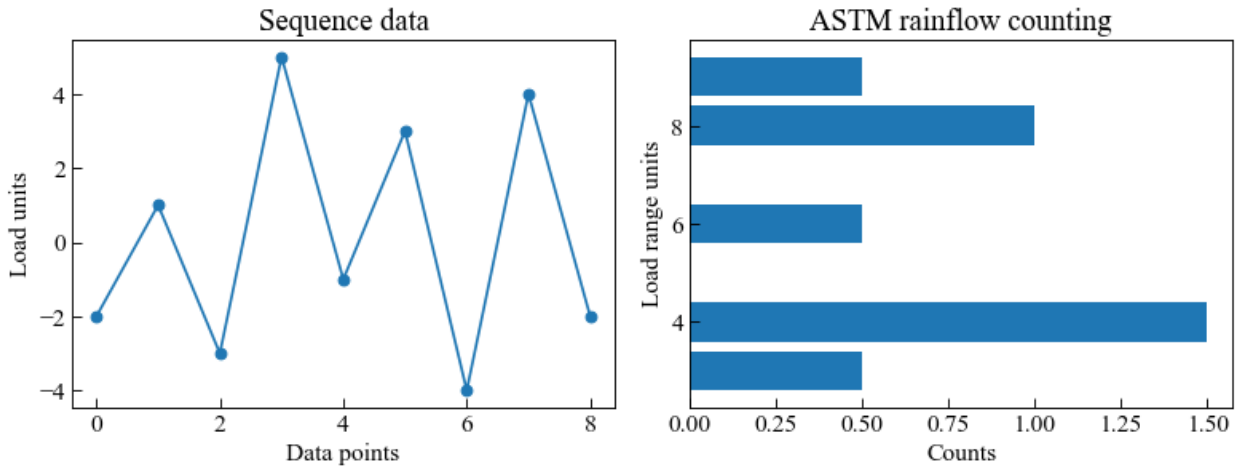
Example with default values

[19]:
```
astmRfcSequenceData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]

astmRfcCountingResults = astmRainflowCounting( astmRfcSequenceData )
```

[20]:
```
print( astmRfcCountingResults )
```

```
[[3.0, 0.5], [4.0, 1.5], [6.0, 0.5], [8.0, 1.0], [9.0, 0.5]]
```

[21]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( astmRfcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( astmRfcCountingResults )[ :, 0 ],
          np.array( astmRfcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM rainflow counting" )

plt.tight_layout()
plt.show()
```



Example with sequence digitization

This example demonstrates the usage of `sequenceDigitization` function **before** the rainflow counting method. The rainflow counting method counts the number of the different load ranges. Therefore, undigitized sequence data can generate noisy output results. With the `sequenceDigitization` function, the output results can be aggregated.

[22]:
```
from ffpack.utils import sequenceDigitization
```

[23]:
```
astmRfcSequenceData = [ -2.4, 1.3, -3.3, 4.6, -1.4, 3.2, -4.4, 4.2, -2.1 ]
digitizedAstmRfcSequenceData = sequenceDigitization( astmRfcSequenceData,
                                                     resolution=1.0 )

originalAstmRfcCountingResults = astmRainflowCounting( astmRfcSequenceData )
digitizedAstmRfcCountingResults = astmRainflowCounting( digitizedAstmRfcSequenceData )
```

[24]:
```
print( "Original sequence data: " )
print( astmRfcSequenceData )
print()
print( "Digitized sequence data with resolution of 1.0: " )
print( digitizedAstmRfcSequenceData )
print()

with np.printoptions( precision=3, suppress=True ):
    print( "Original rainflow counting results: " )
    print( np.array( originalAstmRfcCountingResults ) )
    print()
    print( "Digitized rainflow counting results: " )
    print( np.array( digitizedAstmRfcCountingResults ) )
```

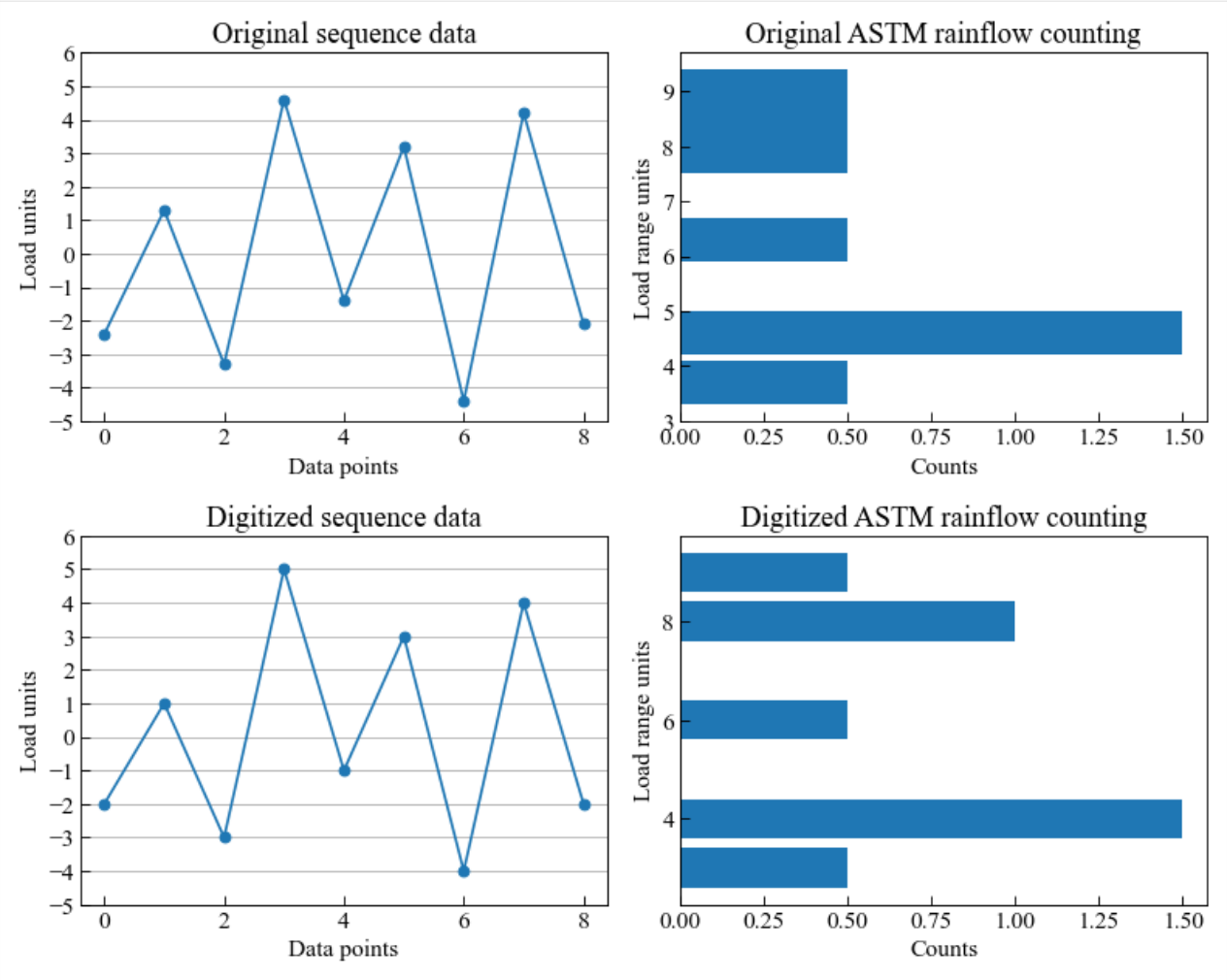
```
Original sequence data:
[-2.4, 1.3, -3.3, 4.6, -1.4, 3.2, -4.4, 4.2, -2.1]

Digitized sequence data with resolution of 1.0:
[-2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0]

Original rainflow counting results:
[[3.7 0.5]
 [4.6 1.5]
 [6.3 0.5]
 [7.9 0.5]
 [8.6 0.5]
 [9.  0.5]]

Digitized rainflow counting results:
[[3.  0.5]
 [4.  1.5]
 [6.  0.5]
 [8.  1. ]
 [9.  0.5]]
```

[25]:
```
fig, ( ax1, ax2 ) = plt.subplots( 2, 2, figsize=( 10, 8 ) )

ax1[0].plot( astmRfcSequenceData, "o-" )
```

(continues on next page)

(continued from previous page)

```
ax1[0].tick_params( axis='x', direction="in", length=5 )
ax1[0].tick_params( axis='y', direction="in", length=5 )
ax1[0].set_ylabel( "Load units" )
ax1[0].set_xlabel( "Data points" )
ax1[0].set_title( "Original sequence data" )
ax1[0].set_yticks( np.arange(-5, 6.5, 1) )
ax1[0].grid( axis='y', color="0.7" )

ax1[1].barh( np.array( originalAstmRfcCountingResults )[ :, 0 ],
             np.array( originalAstmRfcCountingResults )[ :, 1 ] )

ax1[1].tick_params( axis='x', direction="in", length=5 )
ax1[1].tick_params( axis='y', direction="in", length=5 )
ax1[1].set_ylabel( "Load range units" )
ax1[1].set_xlabel( "Counts" )
ax1[1].set_title( "Original ASTM rainflow counting" )

ax2[0].plot( digitizedAstmRfcSequenceData, "o-" )

ax2[0].tick_params( axis='x', direction="in", length=5 )
ax2[0].tick_params( axis='y', direction="in", length=5 )
ax2[0].set_ylabel( "Load units" )
ax2[0].set_xlabel( "Data points" )
ax2[0].set_title( "Digitized sequence data" )
ax2[0].set_yticks( np.arange(-5, 6.5, 1) )
ax2[0].grid( axis='y', color="0.7" )

ax2[1].barh( np.array( digitizedAstmRfcCountingResults )[ :, 0 ],
             np.array( digitizedAstmRfcCountingResults )[ :, 1 ] )

ax2[1].tick_params( axis='x', direction="in", length=5 )
ax2[1].tick_params( axis='y', direction="in", length=5 )
ax2[1].set_ylabel( "Load range units" )
ax2[1].set_xlabel( "Counts" )
ax2[1].set_title( "Digitized ASTM rainflow counting" )

plt.tight_layout()
plt.show()
```



Example with cycle counting aggregation

This example demonstrates the usage of `cycleCountingAggregation` function **after** the rainflow counting method. The rainflow counting method counts the number of the different load ranges. Therefore, undigitized sequence data can generate noisy output results. With the `cycleCountingAggregation` function, the output results can be aggregated.

It should be noted that there exists **discrepancy** of the final counting results with the preprocessing `sequenceDigitization` method and the postprocessing `cycleCountingAggregation` method.

[26]:
```
from ffpack.utils import cycleCountingAggregation
```

[27]:
```
astmRfcSequenceData = [ -2.4, 1.3, -3.3, 4.6, -1.4, 3.2, -4.4, 4.2, -2.1 ]

originalAstmRfcCountingResults = astmRainflowCounting( astmRfcSequenceData )
aggregatedAstmRfcCountingResults = cycleCountingAggregation(
    originalAstmRfcCountingResults )
```

[28]:
```
print( "Original sequence data: " )
print( astmRfcSequenceData )
```

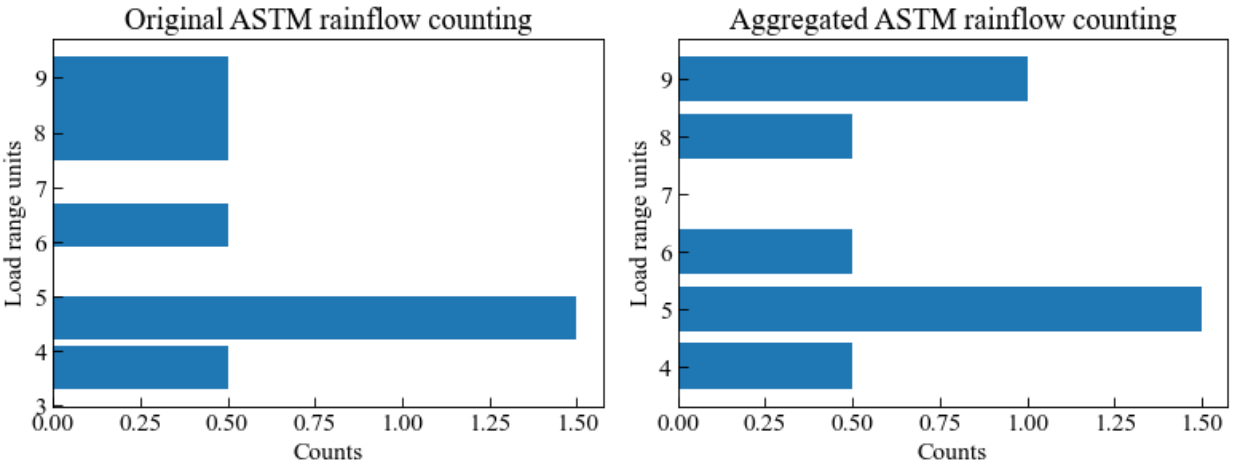
(continues on next page)

(continued from previous page)

```
print()

with np.printoptions( precision=3, suppress=True ):
    print( "Original rainflow counting results: " )
    print( np.array( originalAstmRfcCountingResults ) )
    print()
    print( "Digitized rainflow counting results: " )
    print( np.array( aggregatedAstmRfcCountingResults ) )
```

```
Original sequence data:
[-2.4, 1.3, -3.3, 4.6, -1.4, 3.2, -4.4, 4.2, -2.1]

Original rainflow counting results:
[[3.7 0.5]
 [4.6 1.5]
 [6.3 0.5]
 [7.9 0.5]
 [8.6 0.5]
 [9.  0.5]]

Digitized rainflow counting results:
[[4.  0.5]
 [5.  1.5]
 [6.  0.5]
 [8.  0.5]
 [9.  1. ]]
```

[29]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.barh( np.array( originalAstmRfcCountingResults )[ :, 0 ],
          np.array( originalAstmRfcCountingResults )[ :, 1 ] )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load range units" )
ax1.set_xlabel( "Counts" )
ax1.set_title( "Original ASTM rainflow counting" )

ax2.barh( np.array( aggregatedAstmRfcCountingResults )[ :, 0 ],
          np.array( aggregatedAstmRfcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "Aggregated ASTM rainflow counting" )

plt.tight_layout()
plt.show()
```



ASTM rainflow counting for repeating histories

Function `astmRainflowRepeatHistoryCounting` implements the rainflow counting for repeating histories method in ASTM E1049-85 (2017): sec 5.4.5.

Notes

The original rainflow counting for repeating histories method in ASTM does not provide any preprocessing method for the sequence data or postprocessing method for counting results. To get meaningful results, we provide the preprocessing `sequenceDigitization` function to digitize the input sequence data with a specific resolution, such as 0.5, or the postprocessing `cycleCountingAggregation` function to aggregate the cycle counting results. See the previous "ASTM rainflow counting" section for details.

Function help

[30]:
```
from ffpack.lcc import astmRainflowRepeatHistoryCounting
help( astmRainflowRepeatHistoryCounting )
```

```
Help on function astmRainflowRepeatHistoryCounting in module ffpack.lcc.astmCounting:

astmRainflowRepeatHistoryCounting(data, aggregate=True)
    ASTM simplified rainflow counting for repeating histories in E1049-85: sec 5.4.5.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ],
        [ rangeStart2, rangeEnd2, count2 ], ... ], will be returned.

    Returns
    -------
    rst: 2d array
```

(continues on next page)

(continued from previous page)

```
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 2 or the data dimension is not 1.
        If the data is not repeatable: first data point is different from the
        last data point.

    Examples
    --------
    >>> from ffpack.lcc import astmRainflowRepeatHistoryCounting
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst = astmRainflowRepeatHistoryCounting( data )
```

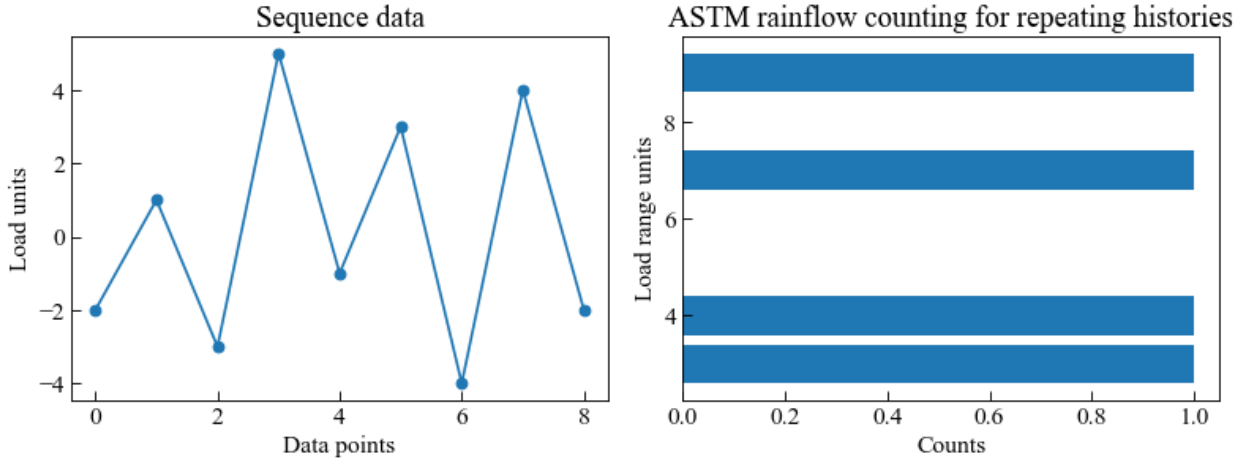
Example with default values

[31]:
```
astmRfcrhSequenceData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]

astmmRfcrhCountingResults = astmRainflowRepeatHistoryCounting( astmRfcrhSequenceData )
```

[32]:
```
print( astmmRfcrhCountingResults )
```

```
[[3.0, 1.0], [4.0, 1.0], [7.0, 1.0], [9.0, 1.0]]
```

[33]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( astmRfcrhSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( astmmRfcrhCountingResults )[ :, 0 ],
          np.array( astmmRfcrhCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM rainflow counting for repeating histories" )

plt.tight_layout()
plt.show()
```



1.4.2 Johannesson counting method

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Johannesson min max counting

Function `johannessonMinMaxCounting` implements the Johannesson min max counting method (Definition 2 in the reference).

Reference:

- Johannesson, P., 1998. Rainflow cycles for switching processes with Markov structure. Probability in the Engineering and Informational Sciences, 12(2), pp.143-175.

Function help

[2]:
```
from ffpack.lcc import johannessonMinMaxCounting
help( johannessonMinMaxCounting )
```

```
Help on function johannessonMinMaxCounting in module ffpack.lcc.johannessonCounting:

johannessonMinMaxCounting(data, aggregate=True)
    Johannesson min-max counting
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        if aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ], [ rangeStart2, rangeEnd2, count2 ],␣
↪... ],
        will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data dimension is not 1
        If the data length is less than 2

    Examples
    --------
    >>> from ffpack.lcc import johannessonMinMaxCounting
    >>> data = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
    >>>          -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]
    >>> rst = johannessonMinMaxCounting( data )
```

Example with default values

[3]:
```
johannessonMmcSequenceData = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
                               -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]

johannessonMmcCountingResults = johannessonMinMaxCounting( johannessonMmcSequenceData )
```

[4]:
```
print( johannessonMmcCountingResults )
```

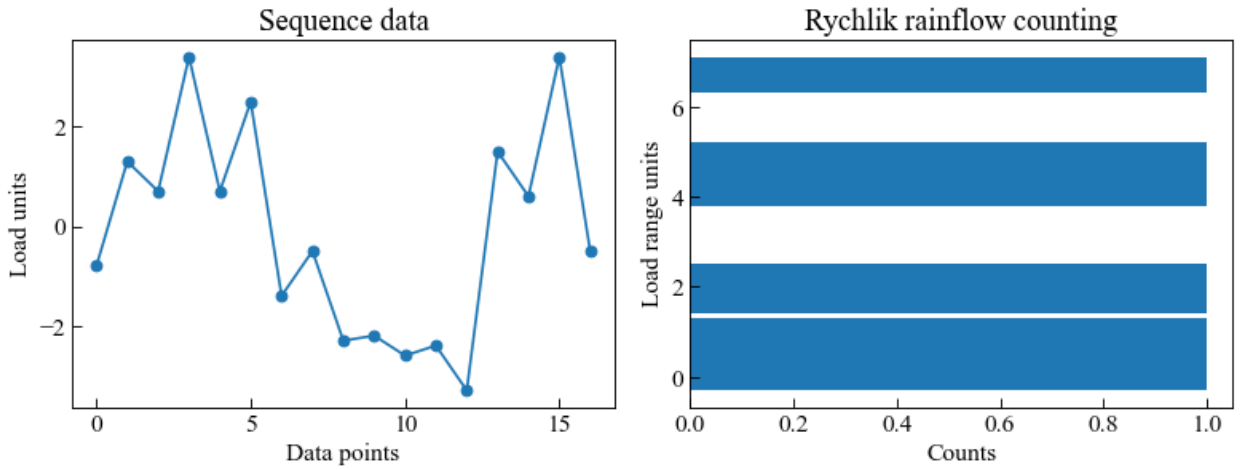
```
[[0.1, 1.0], [0.2, 1.0], [0.9, 1.0], [1.8, 1.0], [2.1, 1.0], [4.2, 1.0], [4.8, 1.0], [6.
↪7, 1.0]]
```

[5]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( johannessonMmcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )
```

(continues on next page)

(continued from previous page)

```
ax2.barh( np.array( johannessonMmcCountingResults )[ :, 0 ],
          np.array( johannessonMmcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "Rychlik rainflow counting" )

plt.tight_layout()
plt.show()
```



1.4.3 Rychlik counting method

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Rychlik rainflow counting

Function `rychlikRainflowCounting` implements the Rychlik rainflow counting method.

Reference:

- Rychlik, I., 1987. A new definition of the rainflow cycle counting method. International journal of fatigue, 9(2), pp.119-121.

Function help

[2]:
```
from ffpack.lcc import rychlikRainflowCounting
help( rychlikRainflowCounting )
```

```
Help on function rychlikRainflowCounting in module ffpack.lcc.rychlikCounting:

rychlikRainflowCounting(data, aggregate=True)
    Rychilk rainflow counting (toplevel-up cycle TUC)

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ], [ rangeStart2, rangeEnd2, count2 ],␣
↪... ],
        will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data dimension is not 1
        If the data length is less than 2

    Examples
    --------
    >>> from ffpack.lcc import rychlikRainflowCycleCounting
    >>> data = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
    >>>          -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]
    >>> rst = rychlikRainflowCycleCounting( data )
```

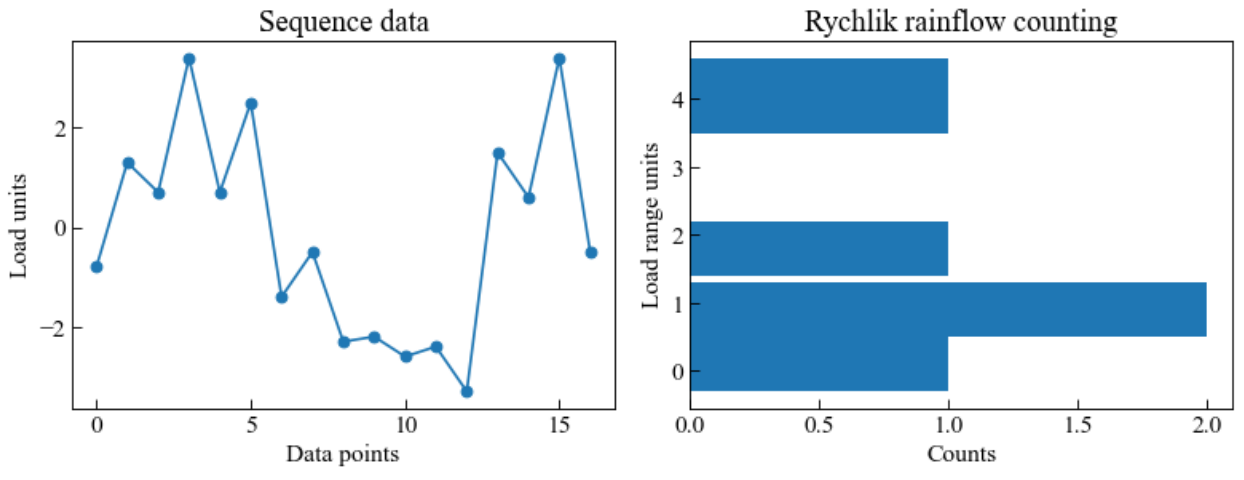
Example with default values

[3]:
```
rychlikRfcSequenceData = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
                    -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]

rychlikRfcCountingResults = rychlikRainflowCounting( rychlikRfcSequenceData )
```

[4]:
```
print( rychlikRfcCountingResults )
```

```
[[0.1, 1.0], [0.2, 1.0], [0.6, 1.0], [0.9, 2.0], [1.8, 1.0], [3.9, 1.0], [4.2, 1.0]]
```

[5]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( rychlikRfcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( rychlikRfcCountingResults )[ :, 0 ],
          np.array( rychlikRfcCountingResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "Rychlik rainflow counting" )

plt.tight_layout()
plt.show()
```



1.4.4 Four pointing counting method

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Four pointing rainflow counting

Function `fourPointRainflowCounting` implements the four pointing counting method in the book by Lee et al.

Reference:

- Lee, Y.L., Barkey, M.E. and Kang, H.T., 2011. Metal fatigue analysis handbook: practical problem-solving techniques for computer-aided engineering. Elsevier.

Function help

[2]:
```
from ffpack.lcc import fourPointRainflowCounting
help( fourPointRainflowCounting )
```

```
Help on function fourPointRainflowCounting in module ffpack.lcc.fourPointCounting:

fourPointRainflowCounting(data, aggregate=True)
    Four point rainflow counting in [Lee2011]_.

    Parameters
    ----------
    data: 1d array
        Load sequence data for counting.
    aggragate: bool, optional
        If aggregate is set to False, the original sequence for internal counting,
        e.g., [ [ rangeStart1, rangeEnd1, count1 ],
        [ rangeStart2, rangeEnd2, count2 ], ... ], will be returned.

    Returns
    -------
    rst: 2d array
        Sorted counting results.

    Raises
    ------
    ValueError
        If the data length is less than 4 or the data dimension is not 1.
```

(continues on next page)

(continued from previous page)

```
    Examples
    --------
    >>> from ffpack.lcc import fourPointRainflowCounting
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst = fourPointRainflowCounting( data )

    References
    ----------
    .. [Lee2011] Lee, Y.L., Barkey, M.E. and Kang, H.T., 2011. Metal fatigue analysis
       handbook: practical problem-solving techniques for computer-aided engineering.
       Elsevier.
```

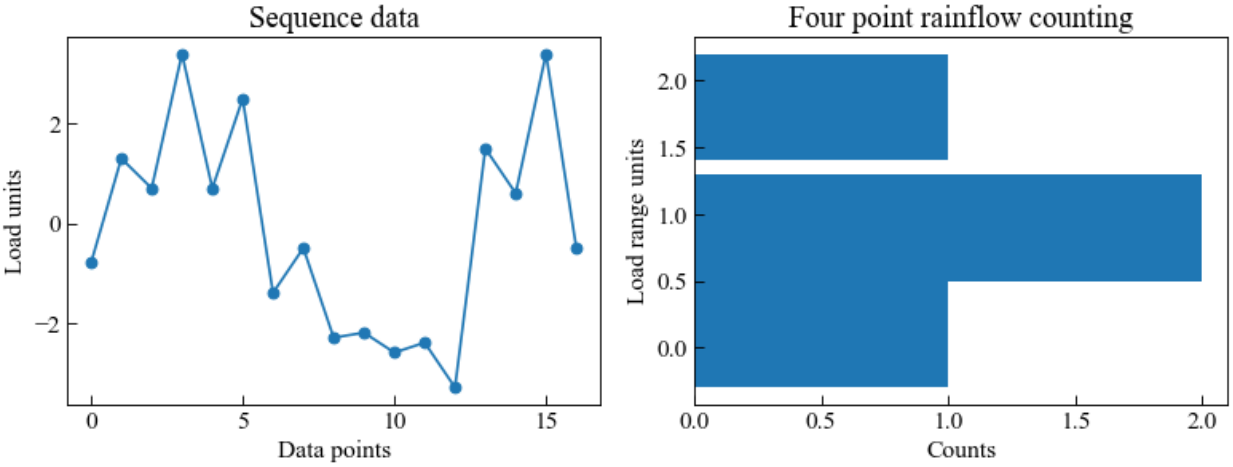
Example with default values

[3]:
```
fpcSequenceData = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
                    -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]

fpcResults = fourPointRainflowCounting( fpcSequenceData )
```

[4]:
```
print( fpcResults )
```

```
[[0.1, 1.0], [0.2, 1.0], [0.6, 1.0], [0.9, 2.0], [1.8, 1.0]]
```

[5]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( fpcSequenceData, "o-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.barh( np.array( fpcResults )[ :, 0 ],
          np.array( fpcResults )[ :, 1 ] )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Load range units" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "Four point rainflow counting" )

plt.tight_layout()
plt.show()
```



1.4.5 Mean stress correction methods

Studies have shown that the mean stress effect plays a critical role in fatigue damage accumulation. The mean value of the fatigue stress response varies under different mean stress levels. In general, fatigue damage increases with the superimposition of static stress to the applied cyclic stress. The mean stress correction is to transform the stress cycle to an equivalent stress cycle with zero mean stress.

For a stress range $[\sigma_{lower}, \sigma_{upper}]$, the mean stress σ_{mean} is calculated with

$$\sigma_m = \frac{\sigma_{lower} + \sigma_{upper}}{2}$$

and the alternating stress, also refered to as stress amplitude, σ_a is calculated with

$$\sigma_a = \frac{\sigma_{upper} - \sigma_{lower}}{2}$$

Reference:

- https://sachinchaturvedi.files.wordpress.com/2012/03/fatigue.pdf.
- https://en.wikipedia.org/wiki/Goodman_relation

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Goodman correction method

Function `goodmanCorrection` implements the Goodman correction method.

The Goodman diagram, originally proposed in 1890, is a graphical representation of this effect. The Goodman correction could be expressed as

$$\frac{\sigma_a}{\sigma_{FL}} + \frac{\sigma_m}{\sigma_u} = 1$$

where σ_u is the ultimate strength; σ_{mean} is the mean stress of the stress range; σ_a is the alternating stress; σ_{FL} is the fatigue limit.

If a safty factor n is considered, the equation becomes

$$\frac{\sigma_a}{\sigma_{FL}} + \frac{\sigma_m}{\sigma_u} = \frac{1}{n}$$

Function help

[2]:
```
from ffpack.lcc import goodmanCorrection

help( goodmanCorrection )
```

```
Help on function goodmanCorrection in module ffpack.lcc.meanStressCorrection:

goodmanCorrection(stressRange, ultimateStrength, n=1.0)
    The Goodman correction in this implementation is applicable to cases with stress
    ratio no less than -1.

    Parameters
    ----------
    stressRange: 1d array
        Stress range, e.g., [ lowerStress, upperStress ].
    ultimateStrength: scalar
        Ultimate tensile strength.
    n: scalar, optional
        Safety factor, default to 1.0.

    Returns
    -------
    rst: scalar
        Fatigue limit.

    Raises
    ------
    ValueError
        If the stressRange dimension is not 1, or stressRange length is not 2.
        If stressRange[ 1 ] <= 0 or stressRange[ 0 ] >= stressRange[ 1 ].
        If ultimateStrength is not a scalar or ultimateStrength <= 0.
        If ultimateStrength is smaller than the mean stress.
        If n < 1.0.
```

(continues on next page)

(continued from previous page)

```
    Examples
    --------
    >>> from ffpack.lcc import goodmanCorrection
    >>> stressRange = [ 1.0, 2.0 ]
    >>> ultimateStrength = 4.0
    >>> rst = goodmanCorrection( stressRange, ultimateStrength )
```

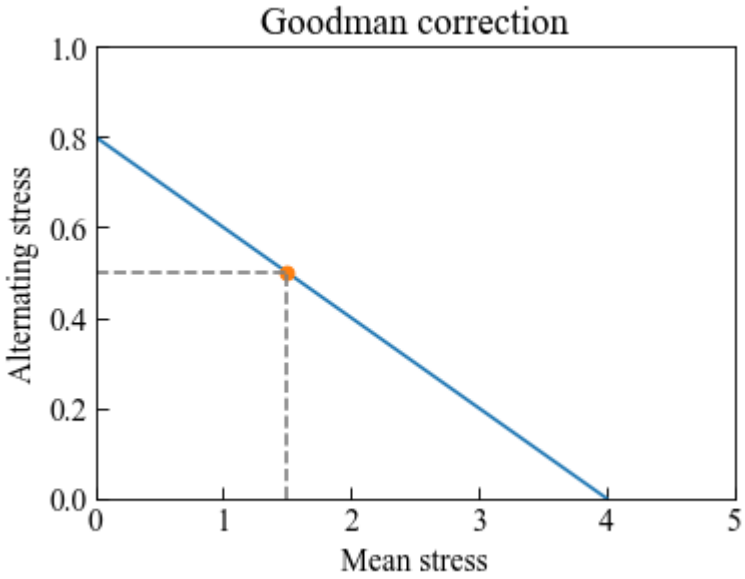
Example with default values

[3]:
```
stressRangeData = [ 1.0, 2.0 ]
ultimateStrength = 4.0

goodmanResult = goodmanCorrection( stressRangeData, ultimateStrength )
```

[4]:
```
print( goodmanResult )
```

```
0.8
```

[5]:
```
fig, ax = plt.subplots()

x = [0, ultimateStrength]
y = [goodmanResult, 0]
ax.plot( x, y, "-" )
ax.tick_params( axis='x', direction="in", length=5 )
ax.tick_params( axis='y', direction="in", length=5 )
plt.xlim( left=0, right=5 )
plt.ylim( bottom=0, top=1 )

sigmaMean = np.mean(stressRangeData)
sigmaAlt = (stressRangeData[1] - stressRangeData[0]) / 2
point = ( sigmaMean, sigmaAlt )
ax.plot( point[0], point[1], "o" )
ax.axvline( point[0], ymin=0, ymax=point[1]/1, linestyle='--', color='gray') # plot␣
↪vertical line
ax.axhline( point[1], xmin=0, xmax=point[0]/5, linestyle='--', color='gray') # plot␣
↪horizontal line

ax.set_xlabel( "Mean stress" )
ax.set_ylabel( "Alternating stress" )
ax.set_title( "Goodman correction" )

plt.tight_layout()
plt.show()
```



Soderberg correction method

Function `soderbergCorrection` implements the Goodman correction method.

The Soderberg diagram, originally proposed in 1890, is a graphical representation of this effect. The Goodman correction could be expressed as

$$\frac{\sigma_a}{\sigma_{FL}} + \frac{\sigma_m}{\sigma_y} = 1$$

where σ_y is the yield strength; σ_{mean} is the mean stress of the stress range; σ_a is the alternating stress; σ_{FL} is the fatigue limit.

If a safty factor n is considered, the equation becomes

$$\frac{\sigma_a}{\sigma_{FL}} + \frac{\sigma_m}{\sigma_y} = \frac{1}{n}$$

Function help

[6]:
```
from ffpack.lcc import soderbergCorrection

help( soderbergCorrection )
```

```
Help on function soderbergCorrection in module ffpack.lcc.meanStressCorrection:

soderbergCorrection(stressRange, yieldStrength, n=1.0)
    The Soderberg correction in this implementation is applicable to cases with stress
    ratio no less than -1.

    Parameters
    ----------
    stressRange: 1d array
```

(continues on next page)

(continued from previous page)

```
        Stress range, e.g., [ lowerStress, upperStress ].
    yieldStrength: scalar
        Yield strength.
    n: scalar, optional
        Safety factor, default to 1.0.

    Returns
    -------
    rst: scalar
        Fatigue limit.

    Raises
    ------
    ValueError
        If the stressRange dimension is not 1, or stressRange length is not 2.
        If stressRange[ 1 ] <= 0 or stressRange[ 0 ] >= stressRange[ 1 ].
        If yieldStrength is not a scalar or yieldStrength <= 0.
        If yieldStrength is smaller than the mean stress.
        If safety factor n < 1.0.

    Examples
    --------
    >>> from ffpack.lcc import soderbergCorrection
    >>> stressRange = [ 1.0, 2.0 ]
    >>> yieldStrength = 3.0
    >>> rst = soderbergCorrection( stressRange, yieldStrength )
```

Example with default values

[7]:
```
stressRangeData = [ 1.0, 2.0 ]
yieldStrength = 3.0

soderbergResult = soderbergCorrection( stressRangeData, yieldStrength )
```

[8]:
```
print( soderbergResult )
```

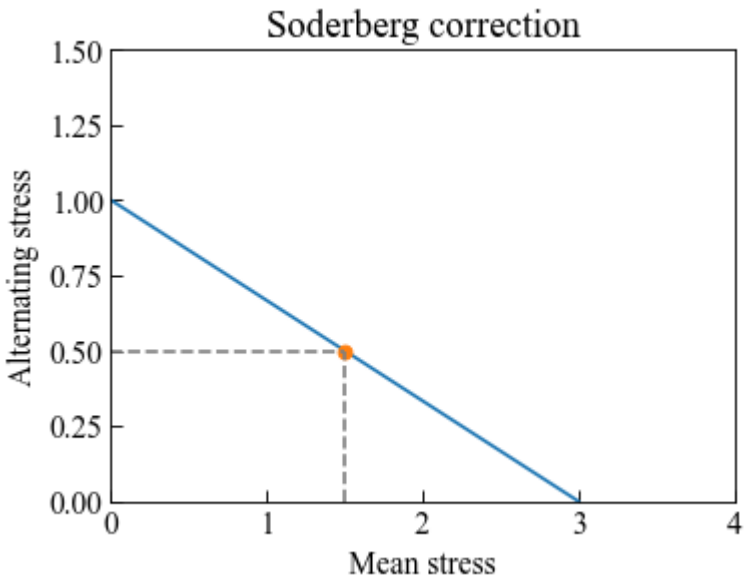
```
1.0
```

[9]:
```
fig, ax = plt.subplots()

x = [0, yieldStrength]
y = [soderbergResult, 0]
ax.plot( x, y, "-" )
ax.tick_params( axis='x', direction="in", length=5 )
ax.tick_params( axis='y', direction="in", length=5 )
plt.xlim( left=0, right=4 )
plt.ylim( bottom=0, top=1.5 )

sigmaMean = np.mean(stressRangeData)
```

(continues on next page)

(continued from previous page)

```
sigmaAlt = (stressRangeData[1] - stressRangeData[0]) / 2
point = ( sigmaMean, sigmaAlt )
ax.plot( point[0], point[1], "o" )
ax.axvline( point[0], ymin=0, ymax=point[1]/1.5, linestyle='--', color='gray') # plot␣
↪vertical line
ax.axhline( point[1], xmin=0, xmax=point[0]/4, linestyle='--', color='gray') # plot␣
↪horizontal line

ax.set_xlabel( "Mean stress" )
ax.set_ylabel( "Alternating stress" )
ax.set_title( "Soderberg correction" )

plt.tight_layout()
plt.show()
```



Gerber correction method

Function `gerberCorrection` implements the Goodman correction method.

The Goodman diagram, originally proposed in 1890, is a graphical representation of this effect. The Goodman correction could be expressed as

$$\frac{\sigma_a}{\sigma_{FL}} + \left(\frac{\sigma_m}{\sigma_u}\right)^2 = 1$$

where σ_u is the ultimate strength; σ_{mean} is the mean stress of the stress range; σ_a is the alternating stress; σ_{FL} is the fatigue limit.

If a safty factor n is considered, the equation becomes

$$\frac{n\sigma_a}{\sigma_{FL}} + \left(\frac{n\sigma_m}{\sigma_u}\right)^2 = 1$$

Function help

[10]:
```
from ffpack.lcc import gerberCorrection

help( gerberCorrection )
```

```
Help on function gerberCorrection in module ffpack.lcc.meanStressCorrection:

gerberCorrection(stressRange, ultimateStrength, n=1.0)
    The Gerber correction in this implementation is applicable to cases with stress
    ratio no less than -1.

    Parameters
    ----------
    stressRange: 1d array
        Stress range, e.g., [ lowerStress, upperStress ].
    ultimateStrength: scalar
        Ultimate strength.
    n: scalar, optional
        Safety factor, default to 1.0.

    Returns
    -------
    rst: scalar
        Fatigue limit.

    Raises
    ------
    ValueError
        If the stressRange dimension is not 1, or stressRange length is not 2.
        If stressRange[ 1 ] <= 0 or stressRange[ 0 ] >= stressRange[ 1 ].
        If ultimateStrength is not a scalar or ultimateStrength <= 0.
        If ultimateStrength is smaller than the mean stress.
        If safety factor n < 1.0.

    Examples
    --------
    >>> from ffpack.lcc import gerberCorrection
    >>> stressRange = [ 1.0, 2.0 ]
    >>> ultimateStrength = 3.0
    >>> rst = gerberCorrection( stressRange, ultimateStrength )
```

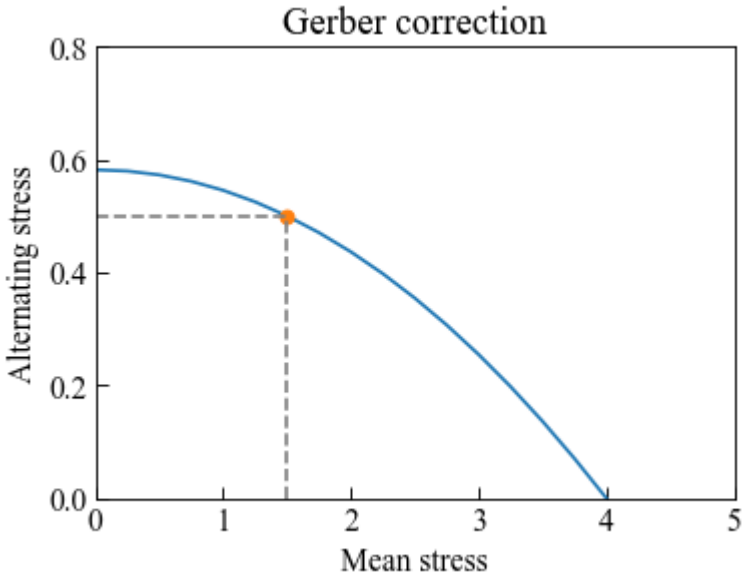
Example with default values

[11]:
```
stressRangeData = [ 1.0, 2.0 ]
ultimateStrength = 4.0

gerberResult = gerberCorrection( stressRangeData, ultimateStrength )
```

[12]:
```
print( gerberResult )
```

```
0.5818181818181818
```

[13]:
```
fig, ax = plt.subplots()

x = np.arange(0, ultimateStrength + 0.25, 0.25)

sigmaMean = np.mean(stressRangeData)
sigmaAlt = (stressRangeData[1] - stressRangeData[0]) / 2
def calculateSigmaAlt(mean):
    rst = 1 - (mean / ultimateStrength) ** 2
    return rst * gerberResult

y = [calculateSigmaAlt(mean) for mean in x]
ax.plot( x, y, "-" )

ax.tick_params( axis='x', direction="in", length=5 )
ax.tick_params( axis='y', direction="in", length=5 )
plt.xlim( left=0, right=5 )
plt.ylim( bottom=0, top=0.8 )

point = ( sigmaMean, sigmaAlt )
ax.plot( point[0], point[1], "o" )
ax.axvline( point[0], ymin=0, ymax=point[1]/0.8, linestyle='--', color='gray') # plot␣
↪vertical line
ax.axhline( point[1], xmin=0, xmax=point[0]/5, linestyle='--', color='gray') # plot␣
↪horizontal line

ax.set_xlabel( "Mean stress" )
ax.set_ylabel( "Alternating stress" )
ax.set_title( "Gerber correction" )

plt.tight_layout()
plt.show()
```



Comparison of the correction methods

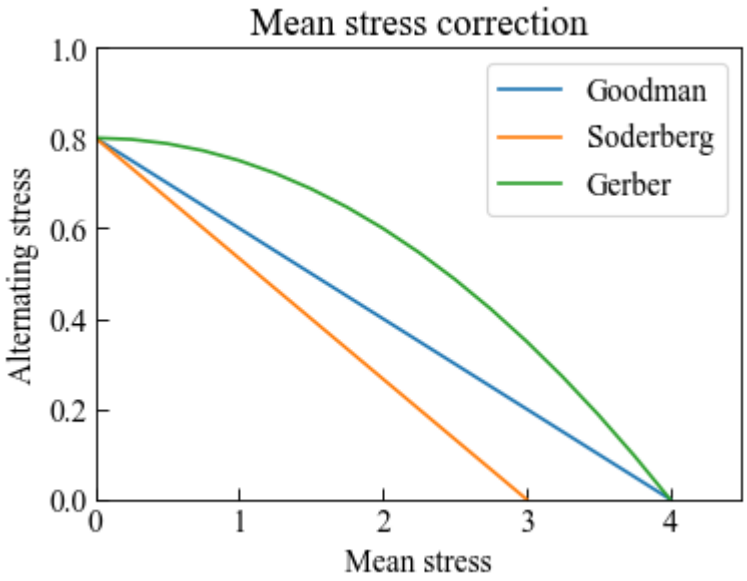
[14]:
```
sigmaFL = 0.8            # assumed fatigue limit
yieldStrength = 3.0      # assumed yield strength
ultimateStrength = 4.0   # assumed ultimate strength
```

[15]:
```
fig, ax = plt.subplots()

xYield = np.arange(0, yieldStrength + 0.25, 0.25)
xUltimate = np.arange(0, ultimateStrength + 0.25, 0.25)

def calcGoodman(mean):
    rst = 1 - mean / ultimateStrength
    return rst * sigmaFL

def calcSoderberg(mean):
    rst = 1 - mean / yieldStrength
    return rst * sigmaFL

def calcGerber(mean):
    rst = 1 - (mean / ultimateStrength) ** 2
    return rst * sigmaFL

yGoodman = [calcGoodman(mean) for mean in xUltimate]
ySoderberg = [calcSoderberg(mean) for mean in xYield]
yGerber = [calcGerber(mean) for mean in xUltimate]
ax.plot( xUltimate, yGoodman, "-", label='Goodman' )
ax.plot( xYield, ySoderberg, "-", label='Soderberg' )
ax.plot( xUltimate, yGerber, "-", label='Gerber' )
plt.legend()
```

(continues on next page)

(continued from previous page)

```
ax.tick_params( axis='x', direction="in", length=5 )
ax.tick_params( axis='y', direction="in", length=5 )
plt.xlim( left=0, right=4.5 )
plt.ylim( bottom=0, top=1 )

ax.set_xlabel( "Mean stress" )
ax.set_ylabel( "Alternating stress" )
ax.set_title( "Mean stress correction" )

plt.tight_layout()
plt.show()
```



1.5 Load sequence generator (lsg)

1.5.1 Random walk

Random walk is a random process describing a succession of random steps in the mathematical space.

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Uniform random walk

Function `randomWalkUniform` implements the uniform random walk function.

The uniform random walk starts from the mathematical origin. The function then updates the coordinates by randomly picking a dimension and a direction.

Function help

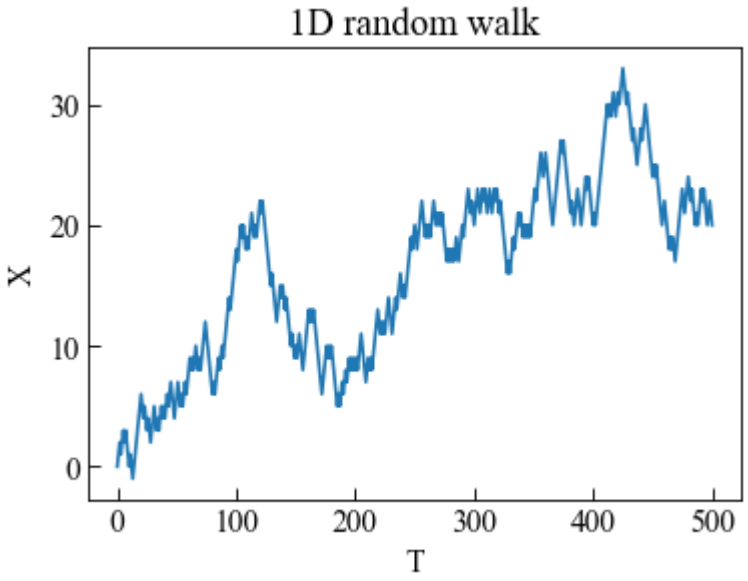
[2]:
```
from ffpack.lsg import randomWalkUniform
help( randomWalkUniform )
```

```
Help on function randomWalkUniform in module ffpack.lsg.randomWalk:

randomWalkUniform(numSteps, dim=1, randomSeed=None)
    Generate load sequence by a random walk.

    Parameters
    ----------
    numSteps: integer
        Number of steps for generating.
    dim: scalar, optional
        Data dimension.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Returns
    -------
    rst: 2d array
        A 2d (numSteps by dim) matrix holding the coordinates
        of the position at each step.

    Raises
    ------
    ValueError
        If the numSteps is less than 1 or the dim is less than 1.

    Examples
    --------
    >>> from ffpack.lsg import randomWalkUniform
    >>> rst = randomWalkUniform( 5 )
```

Example in 1D space

[3]:
```
urw1dResults = randomWalkUniform( 500, 1, randomSeed=2023 )
```

[4]:
```
fig, ax = plt.subplots()

ax.plot( np.array( urw1dResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "1D random walk" )

plt.tight_layout()
plt.show()
```



Example in 2D space

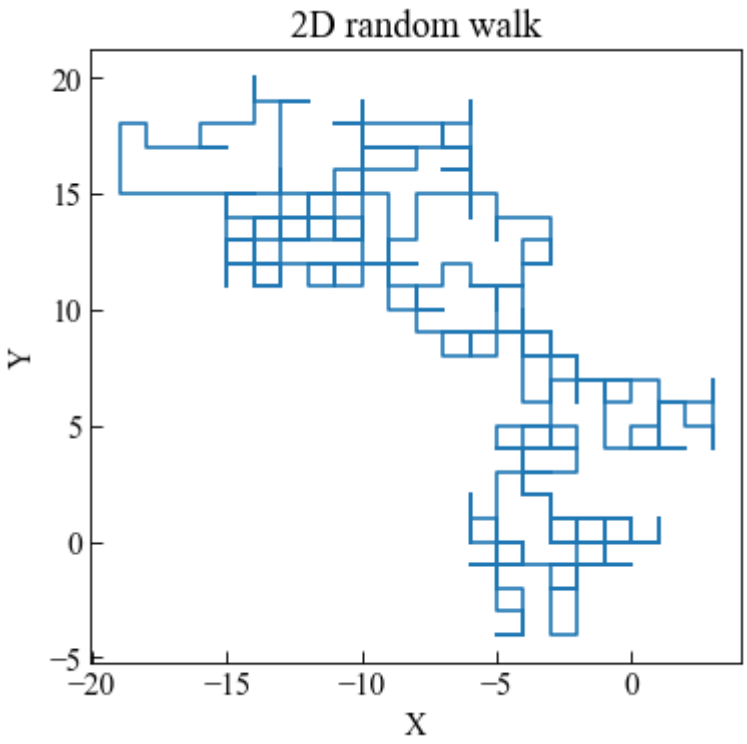
[5]:
```
urw2dResults = randomWalkUniform( 500, 2, randomSeed=2023 )
```

[6]:
```
fig, ax = plt.subplots( figsize=( 5, 5 ) )

ax.plot( np.array( urw2dResults )[ :, 0 ],
         np.array( urw2dResults )[ :, 1 ] )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "Y" )
ax.set_xlabel( "X" )
ax.set_title( "2D random walk" )
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



1.5.2 Autoregressive moving average

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Normal autoregressive (AR) model

Autoregressive model is a random process describing the time-varying procedure in which the output depends on the previous values. The p-th order autoregressive model can be expressed by the following equation,

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t$$

where X_t is the observed values; ϕ_i is the coefficient; ϵ_t is the white noise. For normal autoregressive mode, ϵ_t follows the normal distribution.

Therefore, in order to generate the data of the autoregressive model, the initial observed values and corresponding coefficients should be provided. The distribution for ϵ_t can be different based on the model choice.

Function `arNormal` implements the autoregressive model with normal distributed white noise for arbitrary order. The order depends on the length of the initial observed values.

Function help

[2]:
```
from ffpack.lsg import arNormal
help( arNormal )
```

```
Help on function arNormal in module ffpack.lsg.autoregressiveMovingAverage:

arNormal(numSteps, obs, phis, mu, sigma, randomSeed=None)
    Generate load sequence by an autoregressive model.

    The white noise is generated by the normal distribution.

    Parameters
    ----------
    numSteps: integer
        Number of steps for generating.
    obs: 1d array
        Initial observed values.
    phis: 1d array
        Coefficients for the autoregressive model.
    mu: scalar
        Mean of the white noise.
    sigma: scalar
        Standard deviation of the white noise.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Returns
    -------
    rst: 1d array
        Generated sequence includes the observed values.

    Raises
    ------
    ValueError
```

(continues on next page)

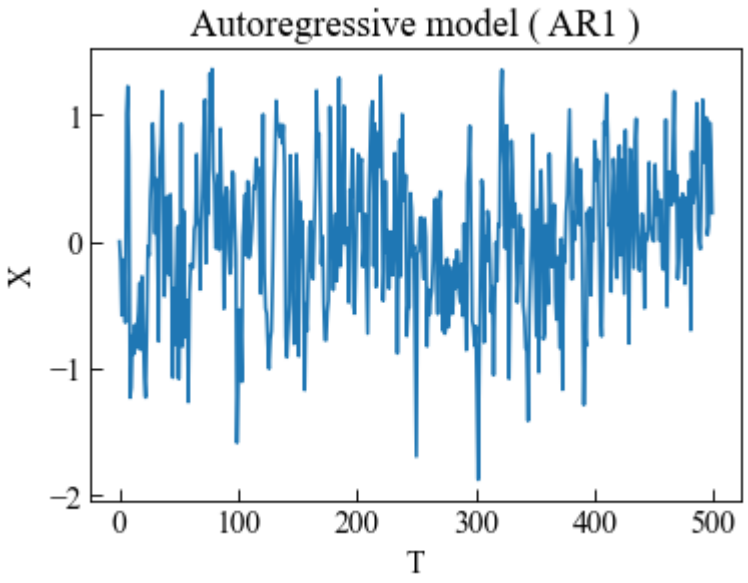
(continued from previous page)

```
        If the numSteps is less than 1.
        If lengths of obs and phis are not equal.

    Examples
    --------
    >>> from ffpack.lsg import arNormal
    >>> obs = [ 0, 1 ]
    >>> phis = [ 0.5, 0.3 ]
    >>> rst = arNormal( 500, obs, phis, 0, 0.5 )
```

Example with first order AR model

[3]:
```
obs = [ 0 ]
phis = [ 0.5 ]
arn1stResults = arNormal( 500, obs, phis, 0, 0.5, randomSeed=2023 )
```

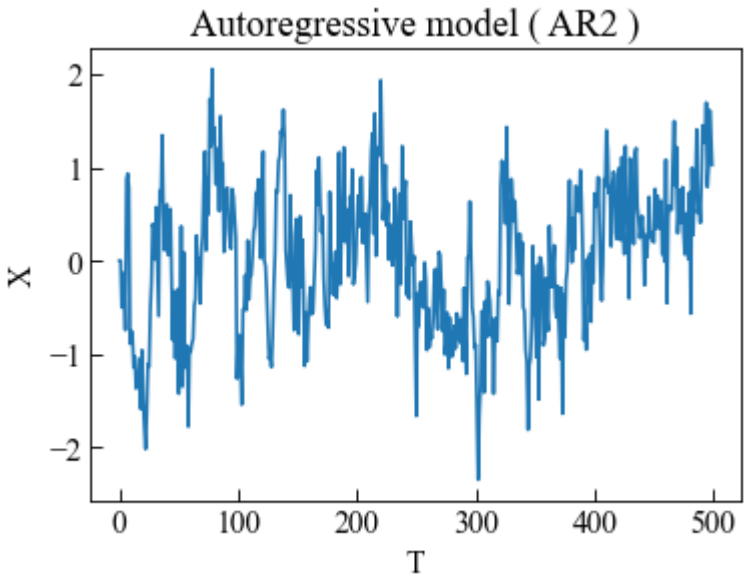
[4]:
```
fig, ax = plt.subplots()

ax.plot( np.array( arn1stResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "Autoregressive model ( AR1 )" )

plt.tight_layout()
plt.show()
```



Example with second order AR model

[5]:
```
obs = [ 0, 0 ]
phis = [ 0.5, 0.3 ]
arn2ndResults = arNormal( 500, obs, phis, 0, 0.5, randomSeed=2023 )
```

[6]:
```
fig, ax = plt.subplots()

ax.plot( np.array( arn2ndResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "Autoregressive model ( AR2 )" )

plt.tight_layout()
plt.show()
```



Normal moving average (MA) model

Moving average model is a common method to model the univariate time series. It can be represented by the following equation,

$$X_t = c + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

where X_t is the observed values; θ_i is the coefficient; ϵ_t are the white noise error term. For normal autoregressive mode, ϵ_t follows the normal distribution.

Function `maNormal` implements the moving average model with normal distributed white noise for arbitrary order. The order depends on the length of the coefficients for the white noise.

Function help

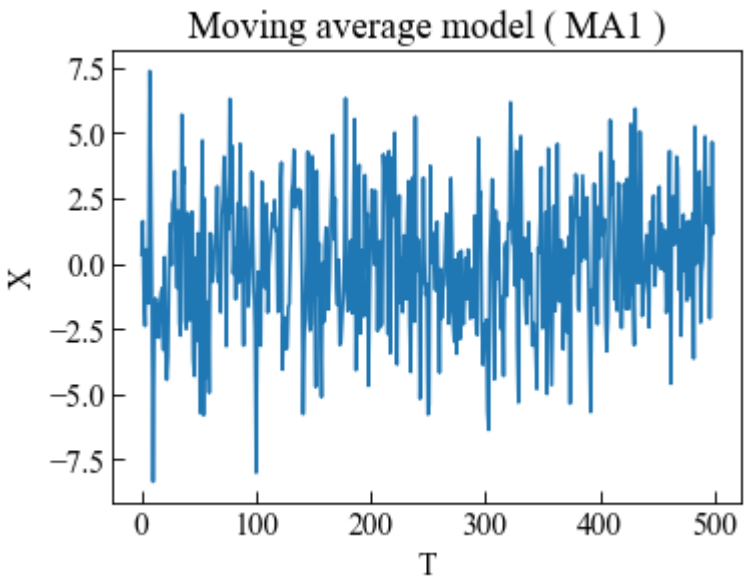
[7]:
```
from ffpack.lsg import maNormal
help( maNormal )
```

```
Help on function maNormal in module ffpack.lsg.autoregressiveMovingAverage:

maNormal(numSteps, c, thetas, mu, sigma, randomSeed=None)
    Generate load sequence by a moving-average model.

    The white noise is generated by the normal distribution.

    Parameters
    ----------
    numSteps: integer
        Number of steps for generating.
    c: scalar
        Mean of the series.
    thetas: 1d array
        Coefficients for the white noise in the moving-average model.
    mu: scalar
        Mean of the white noise.
    sigma: scalar
        Standard deviation of the white noise.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Returns
    -------
    rst: 1d array
        Generated sequence with moving-average model.

    Raises
    ------
    ValueError
        If the numSteps is less than 1.
        If mean of the series is not a scalar.
        If the thetas is empty.

    Examples
    --------
    >>> from ffpack.lsg import maNormal
    >>> thetas = [ 0.8, 0.5 ]
    >>> rst = maNormal( 500, 0, thetas, 0, 0.5 )
```

Example with first order MA model

[8]:
```
c = 0
thetas = [ 5 ]
man1stResults = maNormal( 500, c, thetas, 0, 0.5, randomSeed=2023 )
```

[9]:
```
fig, ax = plt.subplots()

ax.plot( np.array( man1stResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "Moving average model ( MA1 )" )

plt.tight_layout()
plt.show()
```



Example with second order MA model

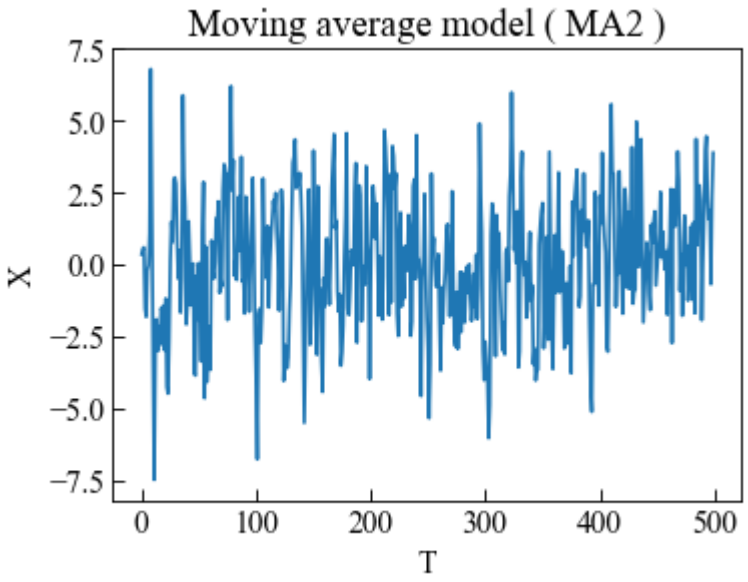
[10]:
```
c = 0
thetas = [ 2, 4 ]
man2ndResults = maNormal( 500, c, thetas, 0, 0.5, randomSeed=2023 )
```

[11]:
```
fig, ax = plt.subplots()

ax.plot( np.array( man2ndResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "Moving average model ( MA2 )" )

plt.tight_layout()
plt.show()
```



Normal ARMA model

Autoregressive moving average model is a combination of AR model and MA model. It can be represented by the following equation,

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

where X_t is the observed values; ϕ_i is the coefficient; θ_i is the coefficient; ϵ_t are the white noise error term. For normal autoregressive moving average mode, ϵ_t follows the normal distribution.

Function `armaNormal` implements the autoregressive moving average model with normal distributed white noise for arbitrary order. The AR order p and MA order q depend on the length of the initial observed values and coefficients for the white noise. It should be noted that the order for AR model and MA model can be different.

Function help

[12]:
```
from ffpack.lsg import armaNormal
help( armaNormal )
```

```
Help on function armaNormal in module ffpack.lsg.autoregressiveMovingAverage:

armaNormal(numSteps, obs, phis, thetas, mu, sigma, randomSeed=None)
    Generate load sequence by an autoregressive-moving-average model.
```

(continues on next page)

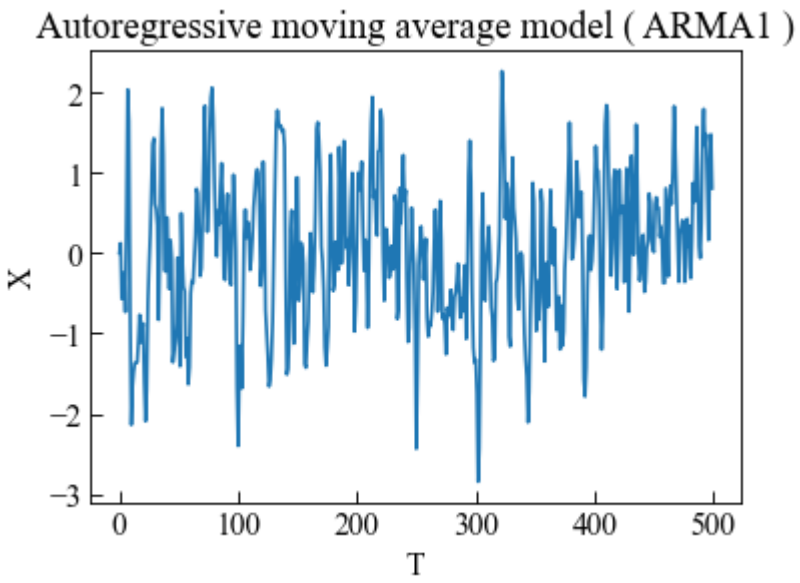
(continued from previous page)

```
The white noise is generated by the normal distribution.

Parameters
----------
numSteps: integer
    Number of steps for generating.
obs: 1d array
    Initial observed values, could be empty.
phis: 1d array
    Coefficients for the autoregressive part.
thetas: 1d array
    Coefficients for the white noise for the moving-average part.
mu: scalar
    Mean of the white noise.
sigma: scalar
    Standard deviation of the white noise.
randomSeed: integer, optional
    Random seed. If randomSeed is none or is not an integer, the random seed in
    global config will be used.

Returns
-------
rst: 1d array
    Generated sequence includes the observed values.

Raises
------
ValueError
    If the numSteps is less than 1.
    If the phis is empty.
    If the thetas is empty.

Examples
--------
>>> from ffpack.lsg import armaNormal
>>> obs = [ 0, 1 ]
>>> phis = [ 0.5, 0.3 ]
>>> thetas = [ 0.8, 0.5 ]
>>> rst = armaNormal( 500, obs, phis, thetas, 0, 0.5 )
```

Example with first order ARMA model

[13]:
```
obs = [ 0 ]
phis = [ 0.5 ]
thetas = [ 0.8 ]
arman1stResults = armaNormal( 500, obs, phis, thetas, 0, 0.5, randomSeed=2023 )
```

[14]:
```
fig, ax = plt.subplots()

ax.plot( np.array( arman1stResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "Autoregressive moving average model ( ARMA1 )" )

plt.tight_layout()
plt.show()
```



Example with second order ARMA model

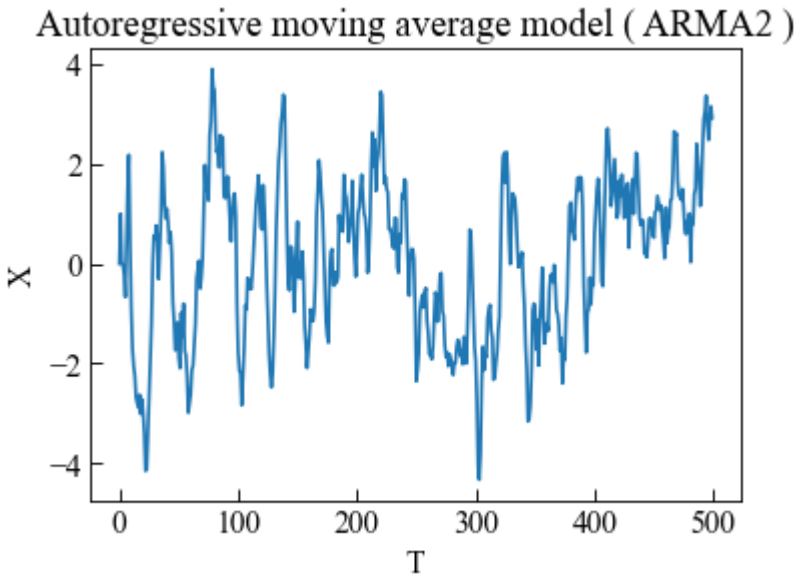
[15]:
```
obs = [ 0, 1 ]
phis = [ 0.5, 0.3 ]
thetas = [ 0.8, 0.5 ]
arman2ndResults = armaNormal( 500, obs, phis, thetas, 0, 0.5, randomSeed=2023 )
```

[16]:
```
fig, ax = plt.subplots()

ax.plot( np.array( arman2ndResults ) )
```

(continues on next page)

(continued from previous page)

```
ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax.set_title( "Autoregressive moving average model ( ARMA2 )" )

plt.tight_layout()
plt.show()
```



Normal ARIMA model

AutoRegressive integrated moving average is the combination of **differenced** AR model and MA model. It can be represented by the following equation,

$$X'_t = c + \sum_{i=1}^{p} \phi_i X'_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

where X'_t is the **differenced** observed values; ϕ_i is the coefficient; θ_i is the coefficient; ϵ_t are the white noise error term. For normal autoregressive integrated moving average mode, ϵ_t follows the normal distribution.

Function `arimaNormal` implements the **first** order differenced ARIMA model with normal distributed white noise. The AR order p and MA order q depend on the length of coefficients. It should be noted that the order for AR model and MA model can be different.

Function help

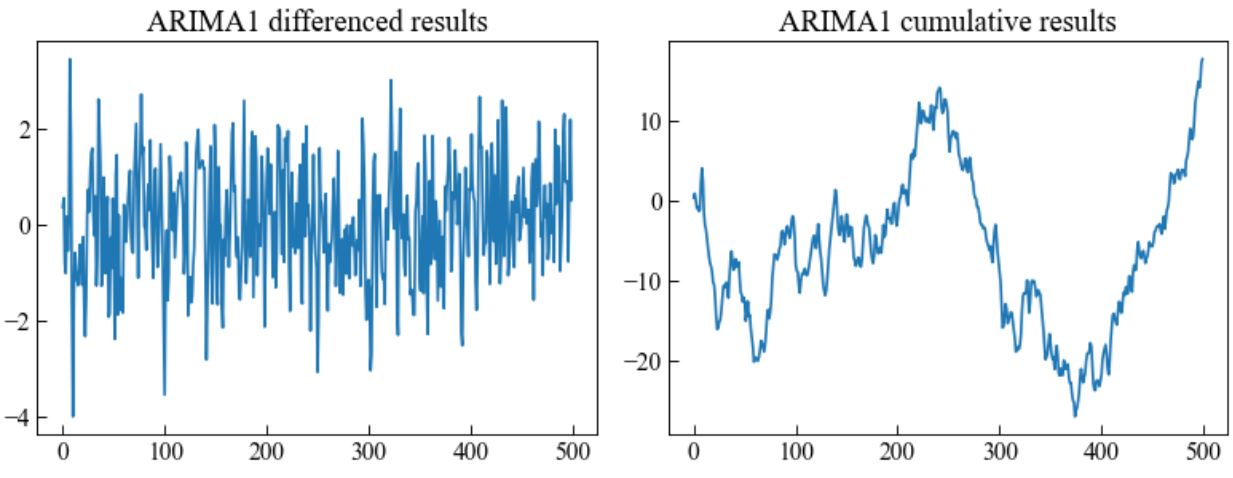
[17]:
```
from ffpack.lsg import arimaNormal
help( arimaNormal )
```

```
Help on function arimaNormal in module ffpack.lsg.autoregressiveMovingAverage:

arimaNormal(numSteps, c, phis, thetas, mu, sigma, randomSeed=None)
    Generate load sequence by an autoregressive integrated moving average model.

    The white noise is generated by the normal distribution.

    First-order diference is used in this function.

    Parameters
    ----------
    numSteps: integer
        Number of steps for generating.
    c: scalar
        Mean of the series.
    phis: 1d array
        Coefficients for the autoregressive part.
    thetas: 1d array
        Coefficients for the white noise for the moving-average part.
    mu: scalar
        Mean of the white noise.
    sigma: scalar
        Standard deviation of the white noise.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Returns
    -------
    rst: 1d array
        Generated sequence with the autoregressive integrated moving average model.

    Raises
    ------
    ValueError
        If the numSteps is less than 1.
        If mean of the series is not a scalar.
        If the phis is empty.
        If the thetas is empty.

    Examples
    --------
    >>> from ffpack.lsg import arimaNormal
    >>> phis = [ 0.5, 0.3 ]
    >>> thetas = [ 0.8, 0.5 ]
    >>> rst = arimaNormal( 500, 0.0, phis, thetas, 0, 0.5 )
```

Example with first order ARIMA model

[18]:
```
c = 0.0
phis = [ 0.1 ]
thetas = [ 2 ]
ariman1stDiffResults = arimaNormal( 500, c, phis, thetas, 0, 0.5, randomSeed=2023 )
ariman1stCumResults = np.cumsum( ariman1stDiffResults )
```

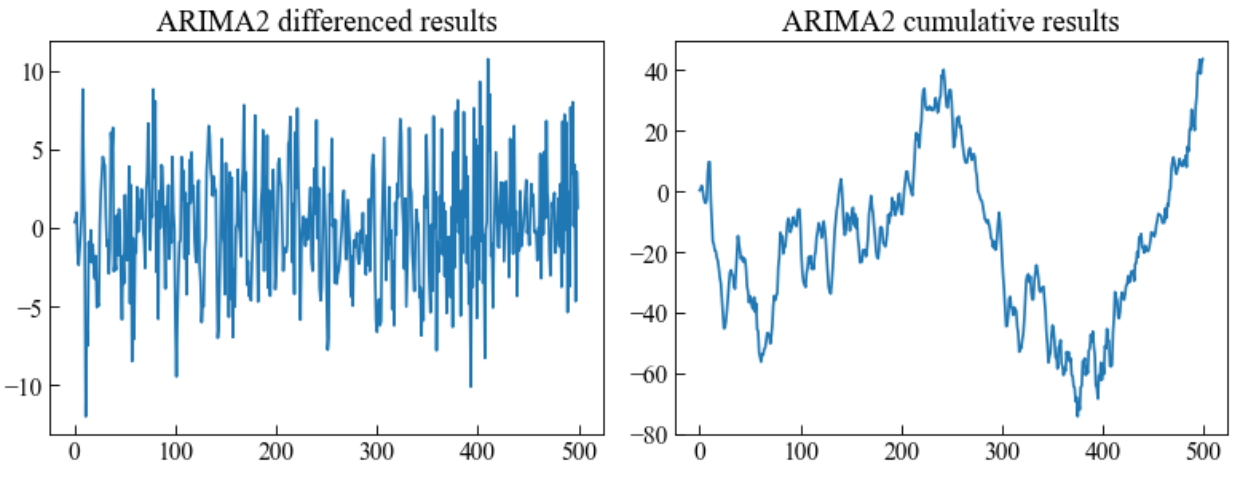
[19]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( np.array( ariman1stDiffResults ) )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax.set_ylabel( "X'" )
ax.set_xlabel( "T" )
ax1.set_title( "ARIMA1 differenced results" )

ax2.plot( np.array( ariman1stCumResults ) )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax2.set_title( "ARIMA1 cumulative results" )

plt.tight_layout()
plt.show()
```



Example with second order ARIMA model

[20]:
```
c = 0.0
phis = [ 0.1, 0.5 ]
thetas = [ 2, 5 ]
ariman2ndDiffResults = arimaNormal( 500, c, phis, thetas, 0, 0.5, randomSeed=2023 )
ariman2ndCumResults = np.cumsum( ariman2ndDiffResults )
```

[21]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( np.array( ariman2ndDiffResults ) )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax.set_ylabel( "X'" )
ax.set_xlabel( "T" )
ax1.set_title( "ARIMA2 differenced results" )

ax2.plot( np.array( ariman2ndCumResults ) )

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax.set_ylabel( "X" )
ax.set_xlabel( "T" )
ax2.set_title( "ARIMA2 cumulative results" )

plt.tight_layout()
plt.show()
```



1.5.3 Sequence from spectrum

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Spectral representation

Function `spectralRepresentation` implements the harmonic superposition function.

Function help

[2]:
```
from ffpack.lsg import spectralRepresentation
help( spectralRepresentation )
```

```
Help on function spectralRepresentation in module ffpack.lsg.sequenceFromSpectrum:

spectralRepresentation(fs, time, freq, psd, freqBandwidth=None, randomSeed=None)
    Generate a sequence from a given power spectrum density with spectral
    representation method.

    Parameters
    ----------
    fs: scalar
        Sampling frequency.
    time: scalar
        Total sampling time.
    freq: 1darray
        Frequency array for psd.
        The freq array should be in equally spaced increasing.
    psd: 1darray
        Power spectrum density array.
    freqBandwidth: scalar, optional
        Frequency bandwidth used to generate the time series from psd.
        Default to None, every frequency in freq will be used.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Returns
    -------
    ts: 1darray
        Array containing all the time data for the time series.
```

(continues on next page)

(continued from previous page)

```
    amps: 1darray
        Amplitude array containing the amplitudes of the time series
        corresponding to ts.

    Raises
    ------
    ValueError
        If the fs or time is not a scalar.
        If freq or psd is not a 1darray or has less than 3 elements.
        If freq and psd are in different lengths.
        If freq contains negative elements.
        If freq is not equally spaced increasing.

    Examples
    --------
    >>> from ffpack.lsg import spectralRepresentation
    >>> fs = 100
    >>> time = 10
    >>> freq = [ 0, 0.1, 0.2, 0.3, 0.4, 0.5 ]
    >>> psd = [ 0.01, 2, 0.05, 0.04, 0.01, 0.03 ]
    >>> ts, amps = spectralRepresentation( fs, time, freq, psd, freqBandwidth=None )
```

Example with generated sequence

Generate sequence with two peak frequencies for psd

[3]:
```
gfs = 500   # sampling frequency
fs1 = 20    # first signal component at 20 Hz
fs2 = 80    # second signal component at 80 Hz
T = 10      # 10s signal length
n0 = -10    # noise level (dB)
```

[4]:
```
t = np.r_[ 0: T: ( 1 / gfs ) ]  # sample time
gdata = np.sin( 2 * fs1 * np.pi * t ) + np.sin( 2 * fs2 * np.pi * t )
gdata += np.random.randn( len( gdata ) ) * 10**( n0 / 20.0 )
```

[5]:
```
from ffpack.lsm import periodogramSpectrum
gfreq, gpsd = periodogramSpectrum( gdata, gfs )
```

Use the psd to generate time series

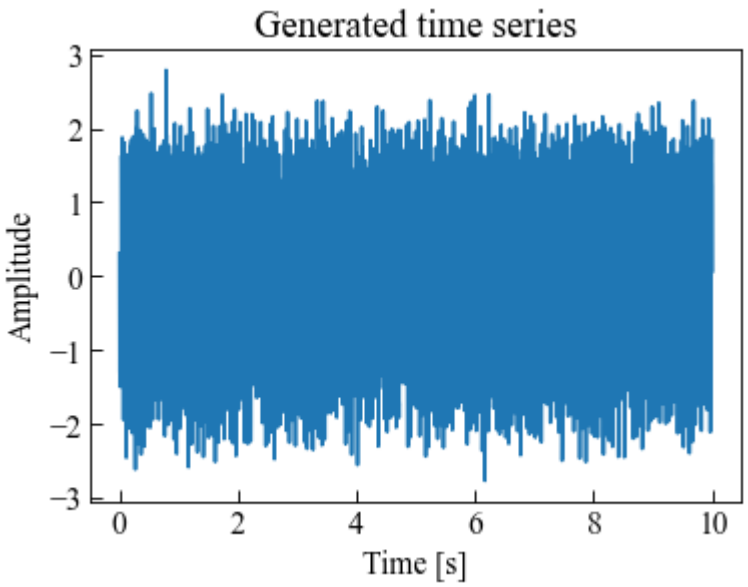
[6]:
```
fs = 500
time = 10
ts, amps = spectralRepresentation( fs, time, gfreq, gpsd, randomSeed=2023 )
```

[7]:
```
fig, ax = plt.subplots()

ax.plot( np.array( ts ),
         np.array( amps ) )
```

(continues on next page)

(continued from previous page)

```
ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')

ax.set_xlabel( "Time [s]" )
ax.set_ylabel( "Amplitude" )
ax.set_title( "Generated time series" )

plt.tight_layout()
plt.show()
```



Generated sequence to spectrum

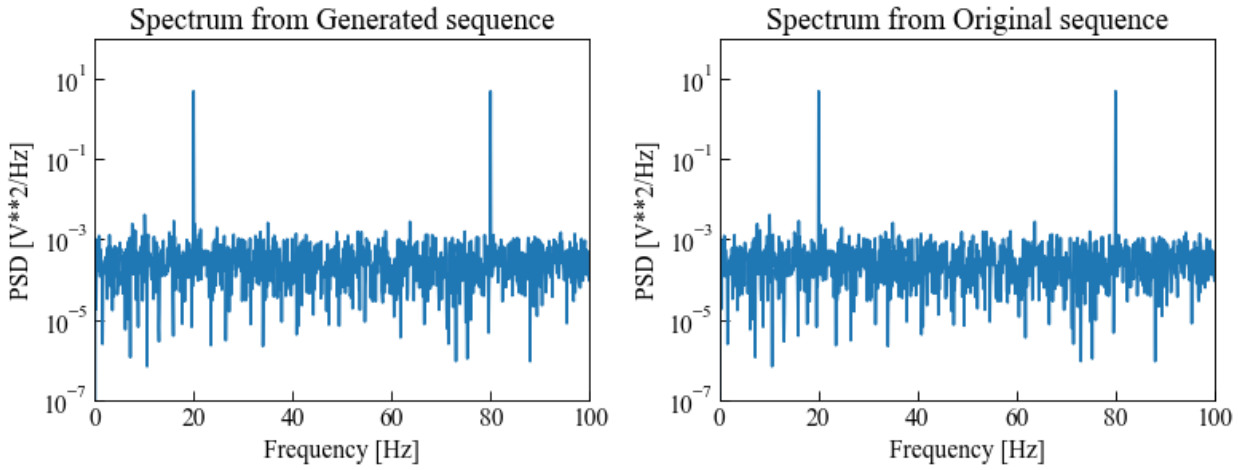
[8]:
```
sfreq, spsd = periodogramSpectrum( amps, fs )
```

[9]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )
plt.yscale("log")

ax1.semilogy( np.array( sfreq ),
              np.array( spsd ) )
ax1.tick_params(axis='x', direction="in", length=5)
ax1.tick_params(axis='y', direction="in", length=5)
ax1.tick_params(axis='x', direction="in", length=3, which='minor')
ax1.tick_params(axis='y', direction="in", length=3, which='minor')
ax1.set_ylim( [ 1e-7, 1e2 ] )
ax1.set_xlim( [ 0, 100 ] )
ax1.set_xlabel( "Frequency [Hz]" )
ax1.set_ylabel( "PSD [V**2/Hz]" )
ax1.set_title( "Spectrum from Generated sequence" )
```

(continues on next page)

(continued from previous page)

```
ax2.semilogy( np.array( gfreq ),
              np.array( gpsd ) )
ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.tick_params(axis='x', direction="in", length=3, which='minor')
ax2.tick_params(axis='y', direction="in", length=3, which='minor')
ax2.set_ylim( [ 1e-7, 1e2 ] )
ax2.set_xlim( [ 0, 100 ] )
ax2.set_xlabel( "Frequency [Hz]" )
ax2.set_ylabel( "PSD [V**2/Hz]" )
ax2.set_title( "Spectrum from Original sequence" )

plt.tight_layout()
plt.show()
```



[10]:
```
ind = np.argpartition( spsd, -2 )[ -2: ]
peak1 = min( sfreq[ ind ] )
peak2 = max( sfreq[ ind ] )
print( "Peak frequences - Generated sequence" )
print( [ peak1, peak2 ] )

ind = np.argpartition( gpsd, -2 )[ -2: ]
peak1 = min( gfreq[ ind ] )
peak2 = max( gfreq[ ind ] )
print( "\nPeak frequences - Original sequence" )
print( [ peak1, peak2 ] )
```

```
Peak frequences - Generated sequence
[20.0, 80.0]

Peak frequences - Original sequence
[20.0, 80.0]
```

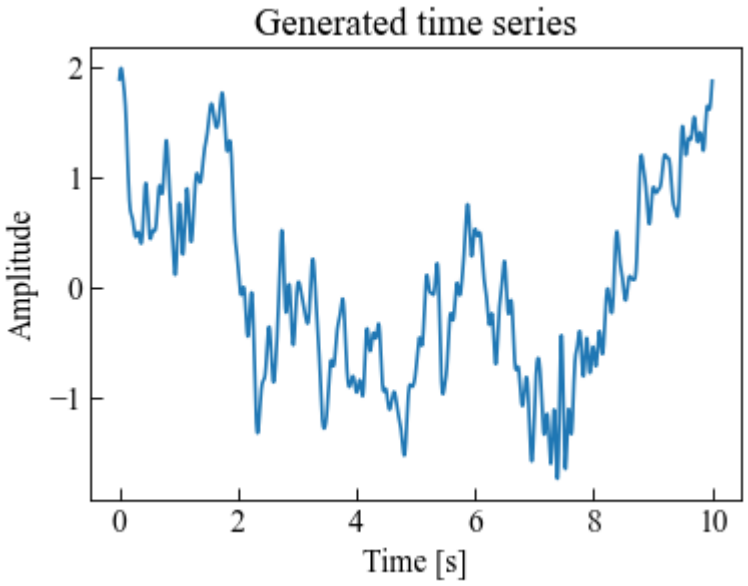
Example with Davenport spectrum

[11]:
```
from ffpack.lsm import davenportSpectrumWithRoughnessLength

dsrnfRange = np.linspace( 0.1, 10, num=100 )
```

[12]:
```
uz = 10
dsrnfResults = [ davenportSpectrumWithRoughnessLength( n, uz, normalized=False )
                 for n in dsrnfRange ]
```

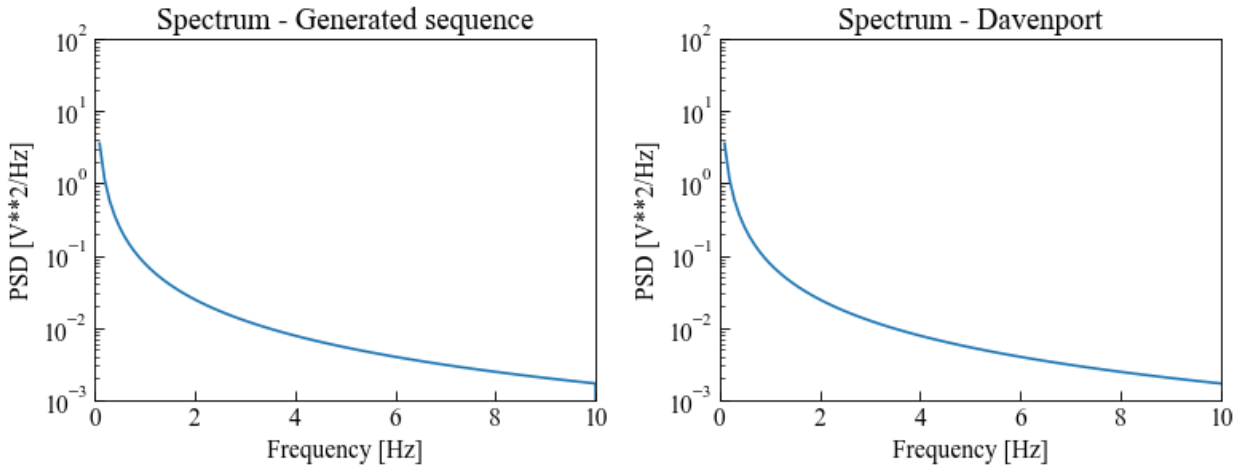
[13]:
```
dfs = 500
time = 10
dts, damps = spectralRepresentation( dfs, time, dsrnfRange,
                                     dsrnfResults, randomSeed=2023 )
```

[14]:
```
fig, ax = plt.subplots()

ax.plot( np.array( dts ),
         np.array( damps ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')

ax.set_xlabel( "Time [s]" )
ax.set_ylabel( "Amplitude" )
ax.set_title( "Generated time series" )

plt.tight_layout()
plt.show()
```



[15]:
```
dsfreq, dspsd = periodogramSpectrum( damps, dfs )
```

[16]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.semilogy( np.array( dsfreq[ 1: -10 ] ),
              np.array( dspsd[ 1: -10 ] ) )
ax1.tick_params(axis='x', direction="in", length=5)
ax1.tick_params(axis='y', direction="in", length=5)
ax1.tick_params(axis='x', direction="in", length=3, which='minor')
ax1.tick_params(axis='y', direction="in", length=3, which='minor')
ax1.set_ylim( [ 1e-3, 1e2 ] )
ax1.set_xlim( [ 0, 10 ] )
ax1.set_xlabel( "Frequency [Hz]" )
ax1.set_ylabel( "PSD [V**2/Hz]" )
ax1.set_title( "Spectrum - Generated sequence" )

ax2.semilogy( np.array( dsrnfRange ),
              np.array( dsrnfResults ) )
ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.tick_params(axis='x', direction="in", length=3, which='minor')
ax2.tick_params(axis='y', direction="in", length=3, which='minor')
ax2.set_ylim( [ 1e-3, 1e2 ] )
ax2.set_xlim( [ 0, 10 ] )
ax2.set_xlabel( "Frequency [Hz]" )
ax2.set_ylabel( "PSD [V**2/Hz]" )
ax2.set_title( "Spectrum - Davenport" )

plt.tight_layout()
plt.show()
```



1.6 Load spectra and matrices (lsm)

1.6.1 Wave spectra

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import warnings

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'

# Filter the plot warning
warnings.filterwarnings( "ignore" )
```

Jonswap Spectrum

The Jonswap spectrum can be expressed,

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \exp\left[-\beta\left(\frac{\omega_p}{\omega}\right)\right]\gamma^r$$

$$r = \exp\left[-\frac{(\omega-\omega_p)^2}{2\sigma^2\omega_p^2}\right]$$

where ω is the wave frequency; ω_p is the peak frequency; α is the intensity of the spectrum, default value = 0.0081; β is the shape factor, default value = 1.25; γ is the peak enhancement factor, default value = 3.3; g is the acceleration due to gravity, default value = 9.81.

Usually, the input paramters can be determined by the following equation from the JONSWAP experiment,

$$\omega_p = 22\left(\frac{g^2}{U_w F}\right)^{1/3}$$

where U_w is the wind speed at 10m above the sea surface; F is the distance from a lee shore.

$$\alpha = 0.076\left(\frac{U_w^2}{gF}\right)^{0.22}$$

where ω_p is the peak frequency; U_w is the wind speed at 10m above the sea surface.

$$\beta = \frac{5}{4}$$

$$\gamma = 3.3$$

Function `jonswapSpectrum` implements the Jonswap sepctrum.

Reference: * Hasselmann, K., Barnett, T.P., Bouws, E., Carlson, H., Cartwright, D.E., Enke, K., Ewing, J.A., Gienapp, A., Hasselmann, D.E., Kruseman, P. and Meerburg, A., 1973. Measurements of wind-wave growth and swell decay during the Joint North Sea Wave Project (JONSWAP). Ergaenzungsheft zur Deutschen Hydrographischen Zeitschrift, Reihe A.

Function help

[2]:
```
from ffpack.lsm import jonswapSpectrum
help( jonswapSpectrum )
```

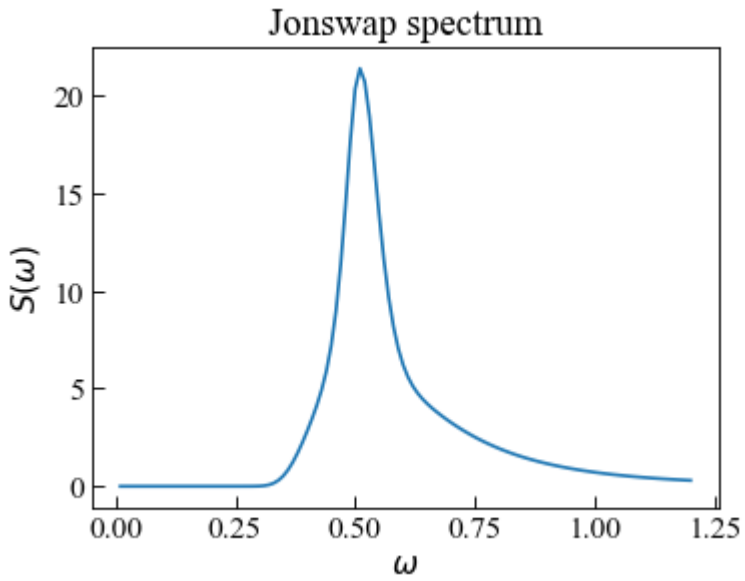
```
Help on function jonswapSpectrum in module ffpack.lsm.waveSpectra:

jonswapSpectrum(w, wp, alpha=0.0081, beta=1.25, gamma=3.3, g=9.81)
    JONSWAP (Joint North Sea Wave Project) spectrum is an empirical relationship
    that defines the distribution of energy with frequency within the ocean.

    Parameters
    ----------
    w: scalar
        Wave frequency.
    wp: scalar
        Peak wave frequency.
    alpha: scalar, optional
        Intensity of the Spectra.
    beta: scalar, optional
        Shape factor, fixed value 1.25.
    gamma: scalar, optional
        Peak enhancement factor.
    g: scalar, optional
        Acceleration due to gravity, a constant.
        9.81 m/s2 in SI units.

    Returns
    -------
    rst: scalar
        The wave spectrum density value at wave frequency w.

    Raises
    ------
    ValueError
        If w is not a scalar.
        If wp is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import jonswapSpectrum
    >>> w = 0.02
    >>> wp = 0.51
    >>> rst = jonswapSpectrum( w, wp, alpha=0.0081, beta=1.25, gamma=3.3, g=9.81 )
```

Example with default values

[3]:
```
jsfRange = np.linspace( 0.0, 1.2, num=121 )
```

[4]:
```
wp = 0.51
jsfResults = [ jonswapSpectrum( w, wp ) for w in jsfRange ]
```

[5]:
```
fig, ax = plt.subplots()

ax.plot( np.array( jsfRange ),
         np.array( jsfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "$S(\omega)$" )
ax.set_xlabel( "$\omega$" )
ax.set_title( "Jonswap spectrum" )

plt.tight_layout()
plt.show()
```



Pierson Moskowitz Spectrum

The Pierson Moskowitz spectrum can be expressed,

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \exp\left[-\beta\left(\frac{\omega_0}{\omega}\right)\right]\gamma^r$$

where α is the intensity of the spectrum, default value = 0.0081; β is the shape factor, default value = 0.74.

$$\omega_0 = \frac{g}{U_w}$$

where g is the acceleration due to gravity, default value = 9.81; U_w is the wind speed at 19.5m above the sea surface.

Reference:

- Pierson Jr, W.J. and Moskowitz, L., 1964. A proposed spectral form for fully developed wind seas based on the similarity theory of SA Kitaigorodskii. Journal of geophysical research, 69(24), pp.5181-5190.

Function help

[6]:
```
from ffpack.lsm import piersonMoskowitzSpectrum
help( piersonMoskowitzSpectrum )
```

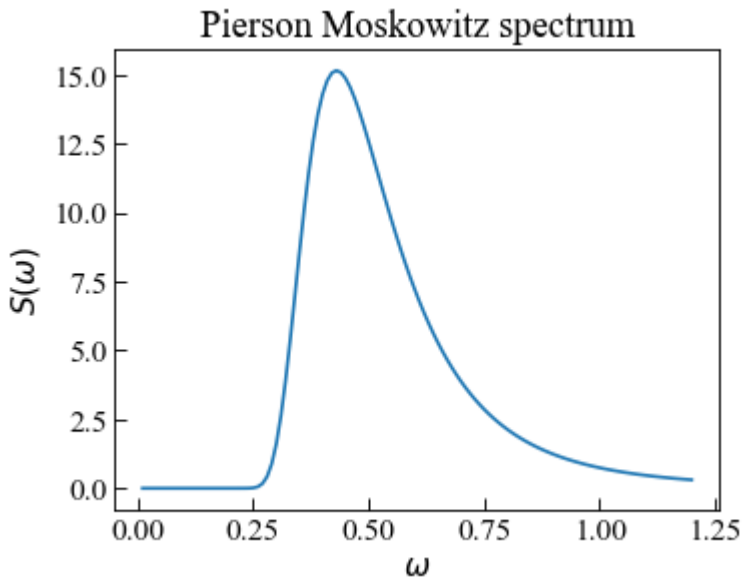
```
Help on function piersonMoskowitzSpectrum in module ffpack.lsm.waveSpectra:

piersonMoskowitzSpectrum(w, Uw, alpha=0.0081, beta=0.74, g=9.81)
    Pierson Moskowitz spectrum is an empirical relationship
    that defines the distribution of energy with frequency within the ocean.

    Parameters
    ----------
    w: scalar
        Wave frequency.
    Uw: scalar
        Wind speed at a height of 19.5m above the sea surface.
    alpha: scalar, optional
        Intensity of the Spectra.
    beta: scalar, optional
        Shape factor.
    g: scalar, optional
        Acceleration due to gravity, a constant.
        9.81 m/s2 in SI units.

    Returns
    -------
    rst: scalar
        The wave spectrum density value at wave frequency w.

    Raises
    ------
    ValueError
        If w is not a scalar.
        If wp is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import piersonMoskowitzSpectrum
    >>> w = 0.51
    >>> Uw = 20
    >>> rst = piersonMoskowitzSpectrum( w, Uw, alpha=0.0081,
    ...                                 beta=1.25, g=9.81 )
```

Example with default values

[7]:
```
pmsfRange = np.linspace( 0.0, 1.2, num=121 )
```

[8]:
```
Uw = 20
pmsfResults = [ piersonMoskowitzSpectrum( w, Uw ) for w in pmsfRange ]
```

[9]:
```
fig, ax = plt.subplots()

ax.plot( np.array( pmsfRange ),
         np.array( pmsfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "$S(\omega)$" )
ax.set_xlabel( "$\omega$" )
ax.set_title( "Pierson Moskowitz spectrum" )

plt.tight_layout()
plt.show()
```



ISSC spectrum

The ISSC spectrum, also known as Bretschneider or modified Pierson-Moskowitz, can be expressed,

$$S(\omega) = \frac{5}{16} \frac{Hs^2 \omega_p^4}{\omega^5} \exp\left[-\frac{5}{4}\left(\frac{\omega_p}{\omega}\right)^4\right]$$

where ω is the wave frequency; ω_p is the peak frequency; Hs is the significant wave height.

Function `isscSpectrum` implements the ISSC sepctrum.

Reference:

- Guidance Notes on Selecting Design Wave by Long Term Stochastic Method

Function help

[10]:
```
from ffpack.lsm import isscSpectrum
help( isscSpectrum )
```

```
Help on function isscSpectrum in module ffpack.lsm.waveSpectra:

isscSpectrum(w, wp, Hs)
    ISSC spectrum, also known as Bretschneider or modified Pierson-Moskowitz.

    Parameters
    ----------
    w: scalar
        Wave frequency.
    wp: scalar
        Peak wave frequency.
    Hs: scalar
        Significant wave height.

    Returns
    -------
    rst: scalar
        The wave spectrum density value at wave frequency w.

    Raises
    ------
    ValueError
        If w is not a scalar.
        If wp is not a scalar.
        If Hs is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import isscSpectrum
    >>> w = 0.02
    >>> wp = 0.51
    >>> Hs = 20
    >>> rst = isscSpectrum( w, wp, Hs )
```

Example with default values

[11]:
```
isfRange = np.linspace( 0.0, 1.2, num=121 )
```

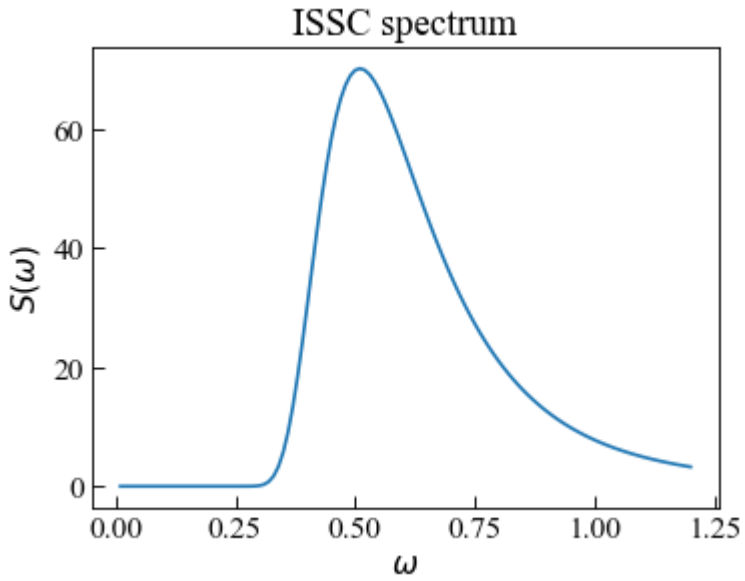
[12]:
```
wp = 0.51
Hs = 20
isfResults = [ isscSpectrum( w, wp, Hs ) for w in isfRange ]
```

[13]:
```
fig, ax = plt.subplots()

ax.plot( np.array( isfRange ),
```

(continues on next page)

(continued from previous page)

```
        np.array( isfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "$S(\omega)$" )
ax.set_xlabel( "$\omega$" )
ax.set_title( "ISSC spectrum" )

plt.tight_layout()
plt.show()
```



Gaussian Swell spectrum

The Gaussian Swell spectrum is typically used to model long period swell sea, and can be expressed,

$$S(\omega) = \frac{(Hs/4)^2}{2\pi\delta\sqrt{2\pi}} \exp\left[-\frac{(\omega-\omega_p)^2}{2(2\pi\delta)^2}\right]$$

where ω is the wave frequency; ω_p is the peak frequency; Hs is the significant wave height; δ is the peakedness parameter for Gaussian spectral width.

Function `gaussianSwellSpectrum` implements the ISSC sepctrum.

Reference:

- Guidance Notes on Selecting Design Wave by Long Term Stochastic Method

Function help

[14]:
```
from ffpack.lsm import gaussianSwellSpectrum
help( gaussianSwellSpectrum )
```

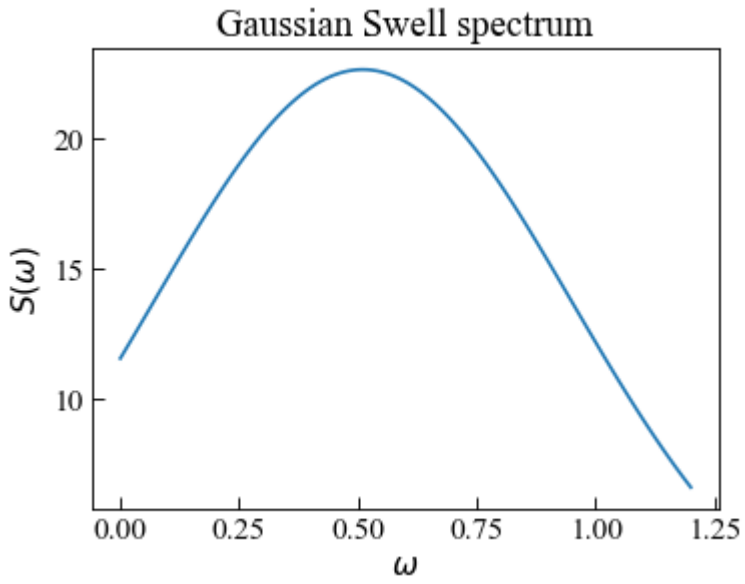
```
Help on function gaussianSwellSpectrum in module ffpack.lsm.waveSpectra:

gaussianSwellSpectrum(w, wp, Hs, sigma)
    Gaussian Swell spectrum, typically used to model long period
    swell seas [Guidance2016A]_.

    Parameters
    ----------
    w: scalar
        Wave frequency.
    wp: scalar
        Peak wave frequency.
    Hs: scalar
        Significant wave height.
    sigma: scalar
        peakedness parameter for Gaussian spectral width.

    Returns
    -------
    rst: scalar
        The wave spectrum density value at wave frequency w.

    Raises
    ------
    ValueError
        If w is not a scalar.
        If wp is not a scalar.
        If Hs is not a scalar.
        If sigma is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import gaussianSwellSpectrum
    >>> w = 0.02
    >>> wp = 0.51
    >>> Hs = 20
    >>> sigma = 0.07
    >>> rst = gaussianSwellSpectrum( w, wp, Hs, sigma )

    References
    ----------
    .. [Guidance2016A] Guidance Notes on Selecting Design Wave by Long
       Term Stochastic Method
```

Example with default values

[15]:
```
gsfRange = np.linspace( 0.0, 1.2, num=121 )
```

[16]:
```
wp = 0.51
Hs = 20
sigma = 0.07
gsfResults = [ gaussianSwellSpectrum( w, wp, Hs, sigma ) for w in gsfRange ]
```

[17]:
```
fig, ax = plt.subplots()

ax.plot( np.array( gsfRange ),
         np.array( gsfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "$S(\omega)$" )
ax.set_xlabel( "$\omega$" )
ax.set_title( "Gaussian Swell spectrum" )

plt.tight_layout()
plt.show()
```



Ochi-Hubble spectrum

The Ochi-Hubble 6-Parameter spectrum covers shapes of wave spectra associated with the growth and decay of a storm, including swells, and can be expressed,

$$S(\omega) = \frac{1}{4}\sum_{j=1}^{2} \frac{\left(-\frac{4\lambda_j+1}{4}\omega_{pj}^4\right)^{\lambda_j}}{\Gamma(\lambda_j)} \frac{H_{sj}^2}{\omega^{4\lambda_j+1}} \exp\left[-\frac{4\lambda_j+1}{4}\left(\frac{\omega_p}{\omega}\right)^4\right]$$

where $j = 1, 2$ stands for lower (swell part) and higher (wind seas part) frequency components; ω is the wave frequency; ω_p is the peak frequency; the six parameters $H_{s1}, H_{s2}, \omega_{p1}, \omega_{p2}, \lambda_1, \lambda_2$ are determined numerically to minimize the difference between theoretical and observed spectra.

The modal frequency of the first component, ω_{p1}, must be less than that of the second, ω_{p2}. The significant wave height of the first component, H_{s1}, should normally be greater than that of the second, H_{s2}, since most of the wave energy tends to be associated with the lower frequency component.

Function `ochiHubbleSpectrum` implements the Jonswap sepctrum.

Reference:

- Guidance Notes on Selecting Design Wave by Long Term Stochastic Method

Function help

[18]:
```
from ffpack.lsm import ochiHubbleSpectrum
help( ochiHubbleSpectrum )
```

```
Help on function ochiHubbleSpectrum in module ffpack.lsm.waveSpectra:

ochiHubbleSpectrum(w, wp1, wp2, Hs1, Hs2, lambda1, lambda2)
    Ochi-Hubble spectrum covers shapes of wave spectra associated with the growth
    and decay of a storm, including swells. [Guidance2016B]_.

    Parameters
    ----------
    w: scalar
        Wave frequency.
    wp1, wp2: scalar
        Peak wave frequency.
    Hs1, Hs2: scalar
        Significant wave height.
    lambda1, lambda2: scalar

    Returns
    -------
    rst: scalar
        The wave spectrum density value at wave frequency w.

    Raises
    ------
    ValueError
        If w is not a scalar.
        If wp1 or wp2 is not a scalar.
        If Hs1 or Hs2 is not a scalar.
```

(continues on next page)

(continued from previous page)

```
        If lambda1 or lambda2 is not a scalar.
        If wp1 is not smaller than wp2.

    Notes
    -----
    Hs1 should normally be greater than Hs2 since most of the wave energy tends to
    be associated with the lower frequency component.

    Examples
    --------
    >>> from ffpack.lsm import ochiHubbleSpectrum
    >>> w = 0.02
    >>> wp1 = 0.4
    >>> wp2 = 0.51
    >>> Hs1 = 20
    >>> Hs2 = 15
    >>> lambda1 = 7
    >>> lambda2 = 10
    >>> rst = ochiHubbleSpectrum( w, wp1, wp2, Hs1, Hs2, lambda1, lambda2 )

    References
    ----------
    .. [Guidance2016B] Guidance Notes on Selecting Design Wave by Long
       Term Stochastic Method
```

Example with default values

[19]:
```
ohfRange = np.linspace( 0.0, 1.2, num=121 )
```

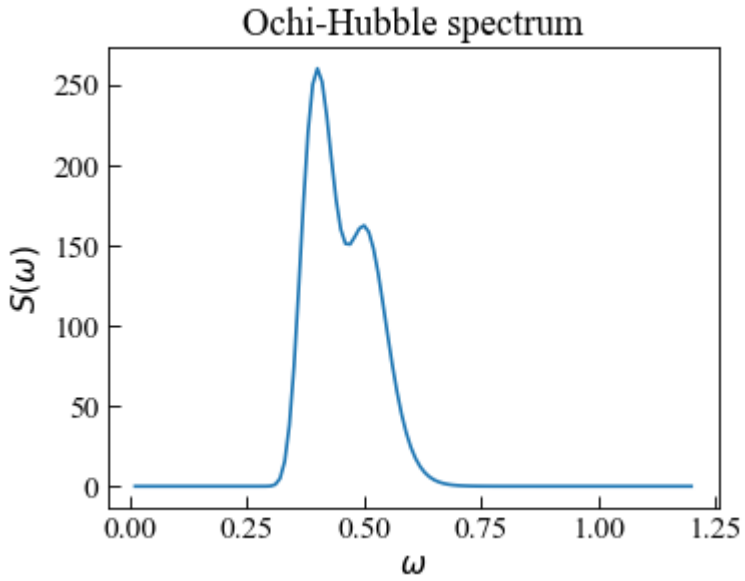
[20]:
```
wp1 = 0.4
wp2 = 0.51
Hs1 = 20
Hs2 = 15
lambda1 = 7
lambda2 = 10
ohfResults = [ ochiHubbleSpectrum( w, wp1, wp2, Hs1, Hs2, lambda1, lambda2 )
               for w in ohfRange ]
```

[21]:
```
fig, ax = plt.subplots()

ax.plot( np.array( ohfRange ),
         np.array( ohfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "$S(\omega)$" )
ax.set_xlabel( "$\omega$" )
ax.set_title( "Ochi-Hubble spectrum" )
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Wave spectra comparison

Note:

- the shapes of the spectra are sensitive to the parameters
- some parameters are fixed in some spectra, and therefore the parameters should be close to the same case
- the curves below are adjusted to be similar by trying the parameters

[22]:
```
waRange = np.linspace( 0.0, 1.2, num=121 )
```

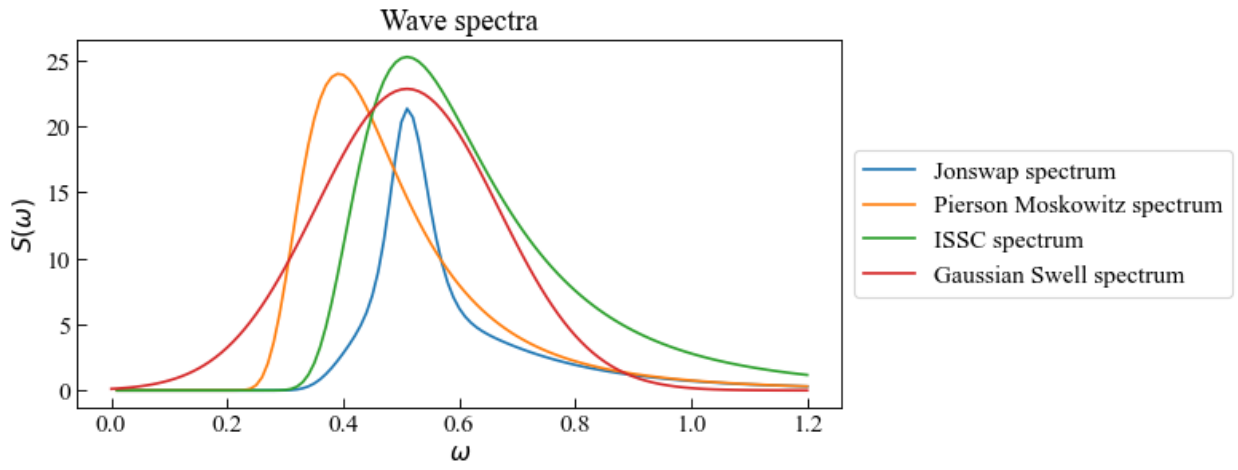
[23]:
```
wp = 0.51
jsfResults = [ jonswapSpectrum( w, wp, alpha=0.0081, beta=1.25, gamma=3.3, g=9.81 )
               for w in waRange ]
```

[24]:
```
Uw = 25
pmsfResults = [ piersonMoskowitzSpectrum( w, Uw, alpha=0.0081, beta=1.25, g=9.81 )
                for w in waRange ]
```

[25]:
```
wp = 0.51
Hs = 12
isfResults = [ isscSpectrum( w, wp, Hs ) for w in waRange ]
```

[26]:
```
wp = 0.51
Hs = 12
sigma = 0.025
gsfResults = [ gaussianSwellSpectrum( w, wp, Hs, sigma ) for w in waRange ]
```

[27]:
```
fig, ax = plt.subplots( figsize=(10, 4) )

ax.plot( np.array( jsfRange ),
         np.array( jsfResults ),
         label="Jonswap spectrum" )
ax.plot( np.array( pmsfRange ),
         np.array( pmsfResults ),
         label="Pierson Moskowitz spectrum" )
ax.plot( np.array( isfRange ),
         np.array( isfResults ),
         label="ISSC spectrum" )
ax.plot( np.array( gsfRange ),
         np.array( gsfResults ),
         label="Gaussian Swell spectrum" )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "$S(\omega)$" )
ax.set_xlabel( "$\omega$" )
ax.set_title( "Wave spectra" )

ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.tight_layout()
plt.show()
```



1.6.2 Wind spectra

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import warnings
```

(continues on next page)

(continued from previous page)

```
plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'

# Filter the plot warning
warnings.filterwarnings( "ignore" )
```

Davenport Spectrum with Drag Coefficient

The Davenport spectrum in the original paper by Davenport can be expressed,

$$\frac{nS(n)}{\kappa\Delta_1^2} = 4.0\frac{x^2}{(1+x^2)^{4/3}}$$

$$x = \frac{1200n}{\Delta_1}$$

where $S(n)$ is the power spectrum density ($m^2s^{-2}Hz^{-1}$); n is the frequency; Δ_1 is the velocity (m/s) at standard reference height of 10 m; κ is the drag coefficient referred to mean velocity at 10 m, default value = 0.005.

The normalized power spectrum density is defined as

$$\frac{nS(n)}{\kappa\Delta_1^2}$$

The normalized frequency is expressed as

$$\frac{10n}{\Delta_1}$$

The drag coefficient κ is related to the surface type and some recommended values are given as

Type of surface	κ
Open unobstructed country (e.g., prairie-type grassland, arctic tundra, desert)	0.005
Country broken by low clustered obstructions such as trees and houses*	0.015 - 0.020
Heavilly built-up urban centers with tall buildings	0.050

*below 10 m high

Function `davenportSpectrumWithDragCoef` implements the Davenport spectrum in the original paper by Davenport.

Reference:

- Davenport, A. G. (1961). The spectrum of horizontal gustiness near the ground in high winds. Quarterly Journal of the Royal Meteorological Society, 87(372), 194-211.

Function help

[2]:
```
from ffpack.lsm import davenportSpectrumWithDragCoef
help( davenportSpectrumWithDragCoef )
```

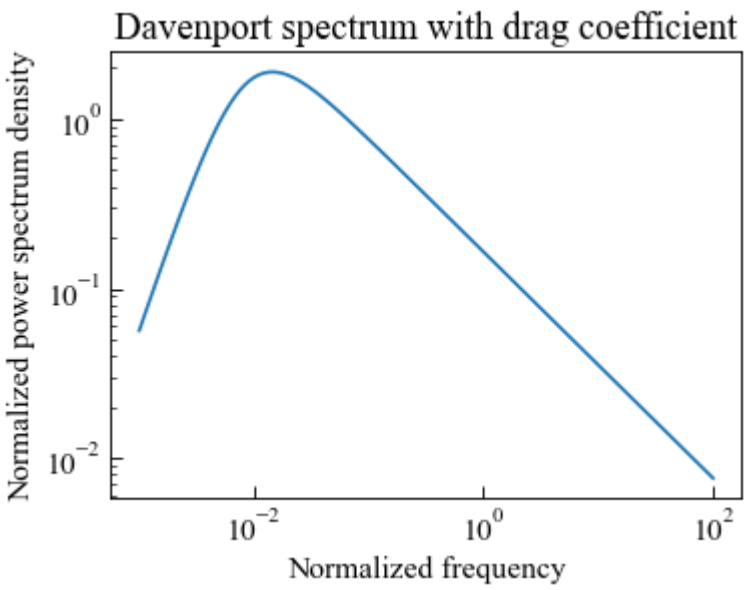
```
Help on function davenportSpectrumWithDragCoef in module ffpack.lsm.windSpectra:

davenportSpectrumWithDragCoef(n, delta1, kappa=0.005, normalized=True)
    Davenport spectrum in the original paper by Davenport [Davenport1961]_.

    Parameters
    ----------
    n: scalar
        Frequency ( Hz ) when normalized=False.
        Normalized frequency when normalized=True.
    delta1: scalar
        Velocity ( m/s ) at standard reference height of 10 m.
    kappa: scalar, optional
        Drag coefficient referred to mean velocity at 10 m.  Default value 0.005
        corresponding to open unobstructed country [Davenport1961]_.
        The recommended value for heavilly built-up urban centers with
        tall buildings is 0.05. The recommended value for country broken by
        low clustered obstructions is between 0.015 and 0.02.
    normalized: bool, optional
        If normalized is set to False, the power spectrum density will be returned.

    Returns
    -------
    rst: scalar
        Power spectrum density ( m^2 s^-2 Hz^-1 ) when normalized=False.
        Normalized power spectrum density when normalized=True.

    Raises
    ------
    ValueError
        If n is not a scalar.
        If delta1 is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import davenportSpectrumWithDragCoef
    >>> n = 2
    >>> delta1 = 10
    >>> rst = davenportSpectrumWithDragCoef( n, delta1, kappa=0.005,
    ...                                      normalized=True )

    References
    ----------
    .. [Davenport1961] Davenport, A.G., 1961. The spectrum of horizontal gustiness
       near the ground in high winds. Quarterly Journal of the Royal Meteorological
       Society, 87(372), pp.194-211.
```

Example with default values

[3]:
```
dsnRange = [ 10**i for i in np.linspace( -3, 2, num=121 ) ]
```

[4]:
```
delta1 = 10
dsnResults = [ davenportSpectrumWithDragCoef( n, delta1, normalized=True )
               for n in dsnRange ]
```

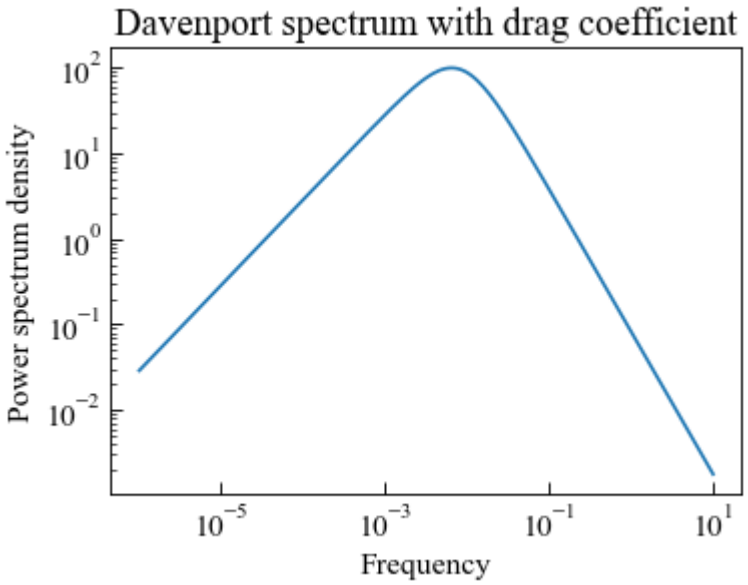
[5]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( dsnRange ),
         np.array( dsnResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Normalized frequency" )
ax.set_ylabel( "Normalized power spectrum density" )
ax.set_title( "Davenport spectrum with drag coefficient" )

plt.tight_layout()
plt.show()
```



[6]:
```
dsnfRange = [ 10**i for i in np.linspace( -6, 1, num=181 ) ]
```

[7]:
```
delta1 = 10
dsnfResults = [ davenportSpectrumWithDragCoef( n, delta1, normalized=False )
                for n in dsnfRange ]
```

[8]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( dsnfRange ),
         np.array( dsnfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Frequency" )
ax.set_ylabel( "Power spectrum density" )
ax.set_title( "Davenport spectrum with drag coefficient" )

plt.tight_layout()
plt.show()
```



Davenport Spectrum with Roughness Length

The Davenport spectrum in the paper by Hiriart et al. can be expressed,

$$\frac{nS(n)}{u_f^2} = 4.0\frac{x^2}{(1+x^2)^{4/3}}$$

$$x = \frac{1200n}{u_f}$$

where $S(n)$ is the power spectrum density ($m^2s^{-2}Hz^{-1}$); n is the frequency; u_f is the friction velocity (m/s); u_z is the mean wind speed (m/s) measured at height z; z is the height above the ground, default value = 10 m; z_0 is the roughness length, default value = 0.03 m corresponding to open exposure case in NIST database.

The friction velocity u_f is calculated as

$$u_f = \frac{ku_z}{ln(z/z_0)}$$

where k is the von Karman's constant and $k = 0.4$.

The normalized power spectrum density is defined as

$$\frac{nS(n)}{u_f^2}$$

The normalized frequency is expressed as

$$\frac{nz}{u_f}$$

Function `davenportSpectrumWithRoughnessLength` implements the Davenport spectrum in the paper by Hiriart et al.

Reference:

- Hiriart, D., Ochoa, J. L., & Garcia, B. (2001). Wind power spectrum measured at the San Pedro Mártir Sierra. Revista Mexicana de Astronomia y Astrofisica, 37(2), 213-220.
- Ho, T. C. E., Surry, D., & Morrish, D. P. (2003). NIST/TTU cooperative agreement-windstorm mitigation initiative: Wind tunnel experiments on generic low buildings. London, Canada: BLWTSS20-2003, Boundary-Layer Wind Tunnel Laboratory, Univ. of Western Ontario.

Function help

[9]:
```
from ffpack.lsm import davenportSpectrumWithRoughnessLength
help( davenportSpectrumWithRoughnessLength )
```

```
Help on function davenportSpectrumWithRoughnessLength in module ffpack.lsm.windSpectra:

davenportSpectrumWithRoughnessLength(n, uz, z=10, z0=0.03, normalized=True)
    Davenport spectrum in the paper by Hiriart et al. [Hiriart2001]_.

    Parameters
    ----------
    n: scalar
        Frequency ( Hz ) when normalized=False.
        Normalized frequency when normalized=True.
    uz: scalar
        Mean wind speed ( m/s ) measured at height z.
    z: scalar, optional
        Height above the ground ( m ), default to 10 m.
    z0: scalar, optional
        Roughness length ( m ), default to 0.03 m corresponding to open
        exposure case in [Ho2003]_.
    normalized: bool, optional
        If normalized is set to False, the power spectrum density will be returned.

    Returns
    -------
    rst: scalar
        Power spectrum density ( m^2 s^-2 Hz^-1 ) when normalized=False.
        Normalized power spectrum density when normalized=True.

    Raises
```

(continues on next page)

(continued from previous page)

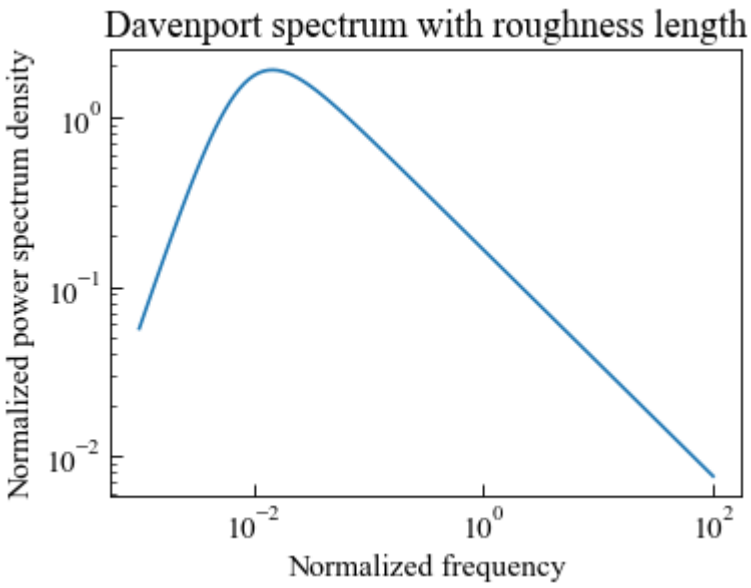
```
    ------
    ValueError
        If n is not a scalar.
        If uz is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import davenportSpectrumWithRoughnessLength
    >>> n = 2
    >>> uz = 10
    >>> rst = davenportSpectrumWithRoughnessLength( n, uz, z=10, z0=0.03,
    ...                                             normalized=True )

    References
    ----------
    .. [Hiriart2001] Hiriart, D., Ochoa, J.L. and Garcia, B., 2001. Wind power
       spectrum measured at the San Pedro Mártir Sierra. Revista Mexicana de
       Astronomia y Astrofisica, 37(2), pp.213-220.
    .. [Ho2003] Ho, T.C.E., Surry, D. and Morrish, D.P., 2003. NIST/TTU cooperative
       agreement-windstorm mitigation initiative: Wind tunnel experiments on generic
       low buildings. London, Canada: BLWTSS20-2003, Boundary-Layer Wind Tunnel
       Laboratory, Univ. of Western Ontario.
```

Example with default values

[10]:
```
dsrnRange = [ 10**i for i in np.linspace( -3, 2, num=121 ) ]
```

[11]:
```
uz = 10
dsrnResults = [ davenportSpectrumWithRoughnessLength( n, uz, normalized=True )
                for n in dsrnRange ]
```

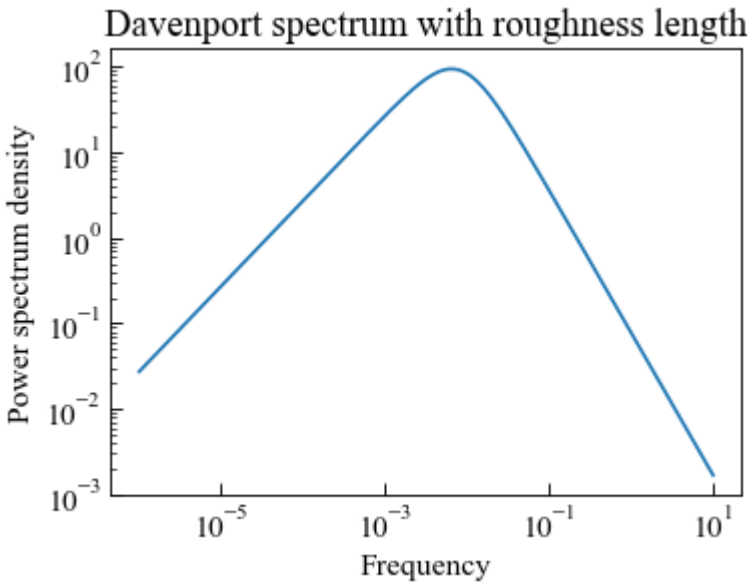
[12]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( dsrnRange ),
         np.array( dsrnResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Normalized frequency" )
ax.set_ylabel( "Normalized power spectrum density" )
ax.set_title( "Davenport spectrum with roughness length" )

plt.tight_layout()
plt.show()
```



[13]:
```
dsrnfRange = [ 10**i for i in np.linspace( -6, 1, num=181 ) ]
```

[14]:
```
uz = 10
dsrnfResults = [ davenportSpectrumWithRoughnessLength( n, uz, normalized=False )
                 for n in dsrnfRange ]
```

[15]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( dsrnfRange ),
         np.array( dsrnfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Frequency" )
ax.set_ylabel( "Power spectrum density" )
ax.set_title( "Davenport spectrum with roughness length" )

plt.tight_layout()
plt.show()
```



EC1 spectrum

The EC1 spectrum is implemented according to Annex B in BS EN 1991-1-4:2005 Eurocode 1: Actions on structures.

The wind distribution over frequencies is expressed by the non-dimensional power spectral density function $S_L(z,n)$, which should be determined as

$$S_L(z,n) = \frac{nS(z,n)}{\sigma_v^2} = \frac{6.8 f_L(z,n)^2}{(1+f_L(z,n)^2)^{5/3}}$$

$$f_L(z,n) = \frac{nL(z)}{v_m(z)}$$

where $S(z,n)$ is the one-sided variance spectrum ($m^2 s^{-2} Hz^{-1}$); $f_L(z,n)$ is the a non-dimensional frequency determined by the frequency n, the natural frequency in Hz; V_m is the mean velocity (m/s); $L(z)$ is the turbulence length scale and is determined as

$$L(z) = L_t \left(\frac{z}{z_t}\right)^{\alpha} \quad \text{for} \quad z \geq z_{min}$$

$$L(z) = L(z_{min}) \quad \text{for} \quad z < z_{min}$$

with a reference height of $z_t = 200\ m$, a reference length scale of $L_t = 300\ m$, and with $\alpha = 0.67 + 0.05 ln(z_0)$, where the roughness length z_0 is in m. The minimum height z_{min} is given in the following table,

Terrain category	z_0 (m)	z_{min} (m)
0 Sea or coastal area exposed to the open sea	0.003	1
1 Lakes or flat and horizontal area	0.01	1
2 Area with low vegetation such as grass and isolated obstacles	0.05	2
3 Area with regular cover of vegetation or buildings or with isolated obstacles	0.3	5
4 Area in which at least 15 % of the surface is covered with buildings	1.0	10

Terrain category	Description
0	Sea or coastal area exposed to the open sea
1	Lakes or flat and horizontal area with negligible vegetation and without obstacles
2	Area with low vegetation such as grass and isolated obstacles(trees, buildings) with separations of at least 20 obstacle heights
3	Area with regular cover of vegetation or buildings or with isolated obstacles with separations of maximum 20 obstacle heights (such as villages, suburban terrain, permanent forest)
4	Area in which at least 15 % of the surface is covered with buildings and their average height exceeds 15 m

Function `ec1Spectrum` implements the spectrum in Eurocode 1.

Reference:

- EN1991-1-4, 2005. Eurocode 1: Actions on structures.

Function help

[16]:
```
from ffpack.lsm import ec1Spectrum
help( ec1Spectrum )
```

```
Help on function ec1Spectrum in module ffpack.lsm.windSpectra:

ec1Spectrum(n, uz, sigma=0.03, z=10, tcat=0, normalized=True)
    EC1 spectrum is implemented according to Annex B [EN1991-1-42005]_.

    Parameters
    ----------
    n: scalar
        Frequency ( Hz ) when normalized=False.
        Normalized frequency when normalized=True.
    uz: scalar
        Mean wind speed ( m/s ) measured at height z.
    sigma: scalar, optional
        Standard derivation of wind.
    z: scalar, optional
        Height above the ground ( m ), default to 10 m.
    tcat: scalar, optional
        Terrain category, could be 0, 1, 2, 3, 4
        Default to 0 (sea or coastal area exposed to the open sea) in EC1 Table 4.1.
    normalized: bool, optional
        If normalized is set to False, the power spectrum density will be returned.

    Returns
    -------
    rst: scalar
        Power spectrum density ( m^2 s^-2 Hz^-1 ) when normalized=False.
        Normalized power spectrum density when normalized=True.

    Raises
    ------
```

(continues on next page)

(continued from previous page)

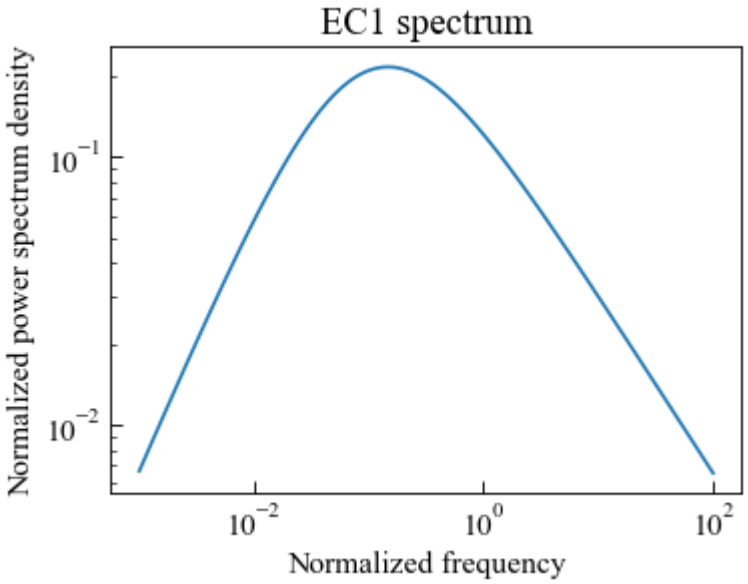
```
    ValueError
        If n is not a scalar.
        If uz is not a scalar.
        If tcat is not int or not within range of 0 to 4

    Examples
    --------
    >>> from ffpack.lsm import ec1Spectrum
    >>> n = 2
    >>> uz = 10
    >>> rst = ec1Spectrum( n, uz, sigma=0.03, z=10, tcat=0, normalized=True )

    References
    ----------
    .. [EN1991-1-42005] EN1991-1-4, 2005. Eurocode 1: Actions on structures.
```

Example with default values

[17]:
```
ec1nRange = [ 10**i for i in np.linspace( -3, 2, num=121 ) ]
```

[18]:
```
uz = 10
ec1nResults = [ ec1Spectrum( n, uz, normalized=True ) for n in ec1nRange ]
```

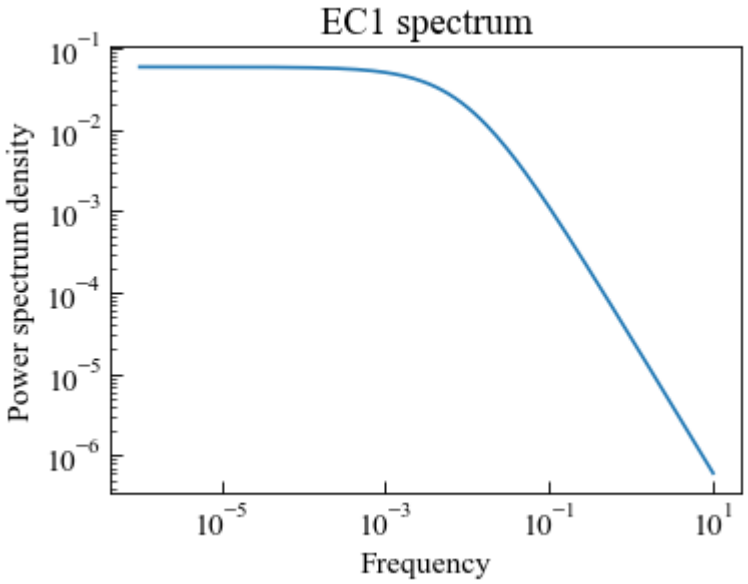
[19]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( ec1nRange ),
         np.array( ec1nResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Normalized frequency" )
ax.set_ylabel( "Normalized power spectrum density" )
ax.set_title( "EC1 spectrum" )

plt.tight_layout()
plt.show()
```



[20]:
```
ec1nfRange = [ 10**i for i in np.linspace( -6, 1, num=181 ) ]
```

[21]:
```
uz = 10
ec1nfResults = [ ec1Spectrum( n, uz, normalized=False ) for n in ec1nfRange ]
```

[22]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( ec1nfRange ),
         np.array( ec1nfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Frequency" )
ax.set_ylabel( "Power spectrum density" )
ax.set_title( "EC1 spectrum" )

plt.tight_layout()
plt.show()
```



IEC spectrum

The IEC spectrum is implemented according to IEC 61400-1 (2005), which is a modified version of the Kaimal wind spectrum.

The component power spectral densities are given in non-dimensional form by the equation:

$$\frac{fS_k(f)}{\sigma_k^2} = \frac{4fL_k/V_{hub}}{(1+6fL_k/V_{hub})^{5/3}}$$

where f is the frequency in Hz; k is the index referring to the velocity component direction (i.e. 1 = longitudinal, 2 = lateral, and 3 = upward); S_k is the single-sided velocity component spectrum; σ_k is the velocity component standard deviation; L_k is the velocity component integral scale parameter.

The turbulence spectral parameters are given in following table,

	$k=1$	$k=2$	$k=3$
Standard deviation σ_k	σ_1	$0.8\sigma_1$	$0.5\sigma_1$
Integral scale L_k	$8.1\Lambda_1$	$2.7\Lambda_1$	$0.66\Lambda_1$

where σ_1 and Λ_1 are the standard deviation and scale parameters, respectively, of the turbulence. The longitudinal turbulence scale parameter, Λ_1, at hub height z shall be given by

$$\Lambda_1 = 0.7z \text{ for } z \leq 60m$$
$$\Lambda_1 = 42m \text{ for } z > 60m$$

The normalized power spectrum density is defined as

$$\frac{fS_k(f)}{\sigma_k^2}$$

The normalized frequency is expressed as

$$\frac{fL_k}{V_{hub}}$$

Function `iecSpectrum` implements the spectrum in IEC 61400-1.

Reference:

- IEC, 2005. IEC 61400-1, Wind turbines - Part 1: Design requirements.

Function help

[23]:
```
from ffpack.lsm import iecSpectrum
help( iecSpectrum )
```

```
Help on function iecSpectrum in module ffpack.lsm.windSpectra:

iecSpectrum(f, vhub, sigma=0.03, z=10, k=1, normalized=True)
    IEC spectrum is implemented according to [IEC2005]_.

    Parameters
    ----------
    f: scalar
        Frequency ( Hz ) when normalized=False.
        Normalized frequency when normalized=True.
    vhub: scalar
        Mean wind speed ( m/s ).
    sigma: scalar, optional
        Standard derivation of the turblent wind speed component.
    z: scalar, optional
        Height above the ground ( m ), default to 10 m.
    k: scalar, optional
        Wind speed direction, could be 1, 2, 3
        ( 1 = longitudinal, 2 = lateral, and 3 = upward )
        Default to 1 (longitudinal).
    normalized: bool, optional
        If normalized is set to False, the power spectrum density will be returned.

    Returns
    -------
    rst: scalar
        Single-sided velocity component power spectrum density ( m^2 s^-2 Hz^-1 )
        when normalized=False.
        Normalized single-sided velocity component power spectrum density
        when normalized=True.

    Raises
    ------
    ValueError
        If n is not a scalar.
        If uz is not a scalar.
        If k is not int or not within range of 1 to 3

    Examples
    --------
    >>> from ffpack.lsm import iecSpectrum
    >>> n = 2
```

(continues on next page)

(continued from previous page)

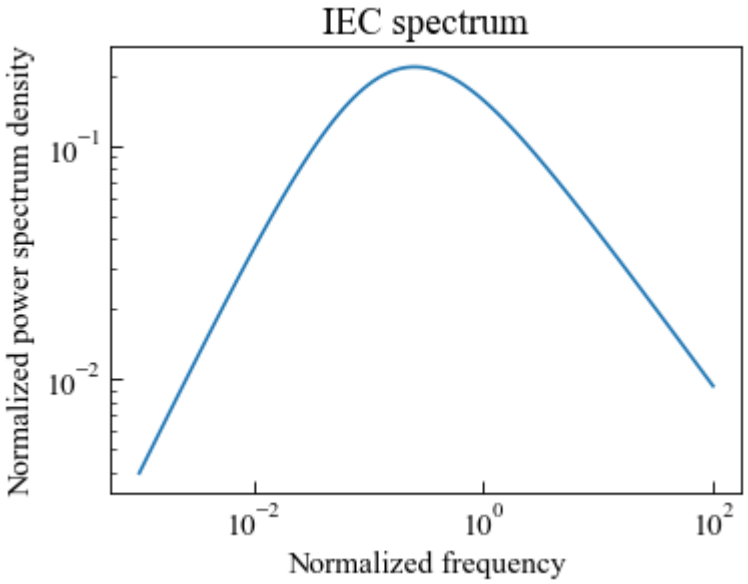
```
    >>> vhub = 10
    >>> rst = iecSpectrum( n, vhub, sigma=0.03, z=10, k=1, normalized=True )

    References
    ----------
    .. [IEC2005] IEC, 2005. IEC 61400-1, Wind turbines - Part 1: Design requirements.
```

Example with default values

[24]:
```
icenRange = [ 10**i for i in np.linspace( -3, 2, num=121 ) ]
```

[25]:
```
vhub = 10
icenResults = [ iecSpectrum( f, vhub, normalized=True ) for f in icenRange ]
```

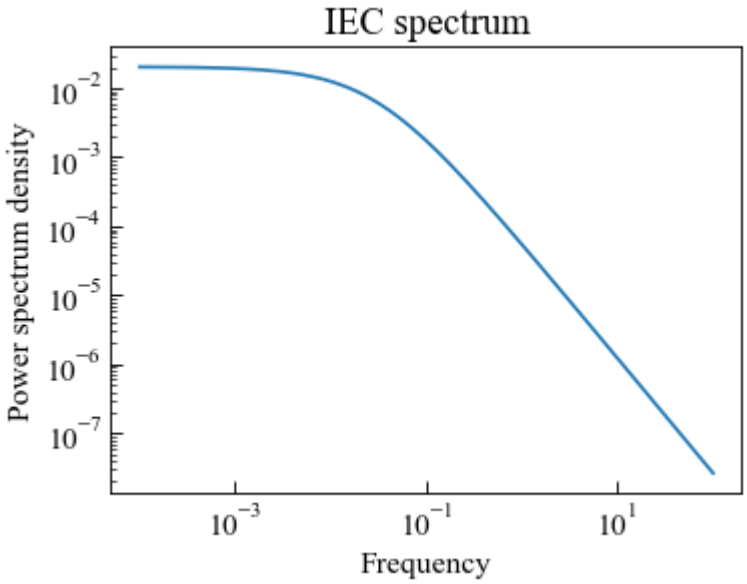
[26]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( icenRange ),
         np.array( icenResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Normalized frequency" )
ax.set_ylabel( "Normalized power spectrum density" )
ax.set_title( "IEC spectrum" )

plt.tight_layout()
plt.show()
```



[27]:
```
iecnfRange = [ 10**i for i in np.linspace( -4, 2, num=141 ) ]
```

[28]:
```
vhub = 10
iecnfResults = [ iecSpectrum( f, vhub, normalized=False ) for f in iecnfRange ]
```

[29]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( iecnfRange ),
         np.array( iecnfResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Frequency" )
ax.set_ylabel( "Power spectrum density" )
ax.set_title( "IEC spectrum" )

plt.tight_layout()
plt.show()
```



API spectrum

The API spectrum is implemented according to API Recommended practice 2A-WSD (RP 2A-WSD).

The 1 point wind spectrum for the energy density of the longitudinal wind speed fluctuations can be expressed by

$$S(f) = \frac{320 \left(\frac{U_0}{10}\right)^2 \left(\frac{z}{10}\right)^{0.45}}{\left(1 + \tilde{f}^n\right)^{\left(\frac{5}{3n}\right)}}$$

$$\tilde{f} = 172 f \left(\frac{z}{10}\right)^{2/3} \left(\frac{U_0}{10}\right)^{-0.75}$$

where $n = 0.468$; f is the frequency (Hz); $S(f)$ is the spectral energy density at frequency $(m^2 s^{-2} Hz^{-2})$; z is the height above sea level (m); U_0 is the 1 hour mean wind speed at 10 m above sea level (m/s).

Function `apiSpectrum` implements the spectrum in API 2007.

Reference:

- API, 2007. Recommended practice 2A-WSD (RP 2A-WSD): Recommnded practice for planning, designing and constructing fixed offshore platforms - working stress design.

Function help

[30]:
```
from ffpack.lsm import apiSpectrum
help( apiSpectrum )
```

```
Help on function apiSpectrum in module ffpack.lsm.windSpectra:

apiSpectrum(f, u0, z=10)
    API spectrum is implemented according to [API2007]_.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    f: scalar
        Frequency ( Hz ).
    u0: scalar
        1 hour mean wind speed ( m/s ) at 10 m above sea level.

    Returns
    -------
    rst: scalar
        Power spectrum density ( m^2 s^-2 Hz^-1 ).

    Raises
    ------
    ValueError
        If n is not a scalar.
        If uz is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import apiSpectrum
    >>> f = 2
    >>> u0 = 10
    >>> rst = apiSpectrum( f, u0 )

    References
    ----------
    .. [API2007] API, 2007. Recommended practice 2A-WSD (RP 2A-WSD):
       Recommnded practice for planning, designing and constructing fixed offshore
       platforms - working stress design.
```

Example with default values

[31]:
```
apifRange = [ 10**i for i in np.linspace( -6, 2, num=191 ) ]
```

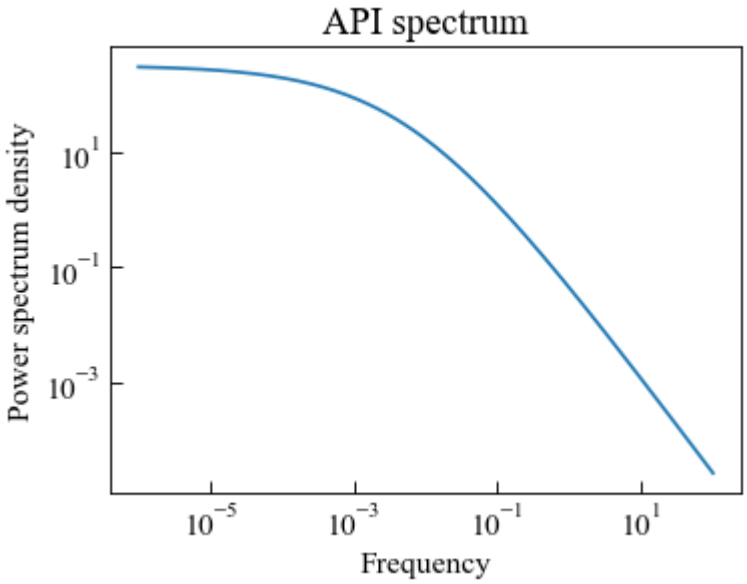
[32]:
```
u0 = 10
apifResults = [ apiSpectrum( f, u0 ) for f in apifRange ]
```

[33]:
```
fig, ax = plt.subplots()
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( apifRange ),
         np.array( apifResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel( "Frequency" )
ax.set_ylabel( "Power spectrum density" )
ax.set_title( "API spectrum" )

plt.tight_layout()
plt.show()
```



Wind spectra comparison

[34]:
```
wdRange = [ 10**i for i in np.linspace( -3, 2, num=121 ) ]
```

[35]:
```
delta1 = 10
dsnResults = [ davenportSpectrumWithDragCoef( n, delta1, normalized=True )
               for n in wdRange ]
```

[36]:
```
uz = 10
dsrnResults = [ davenportSpectrumWithRoughnessLength( n, uz, normalized=True )
                for n in wdRange ]
```

[37]:
```
uz = 10
ec1nResults = [ ec1Spectrum( n, uz, normalized=True ) for n in wdRange ]
```

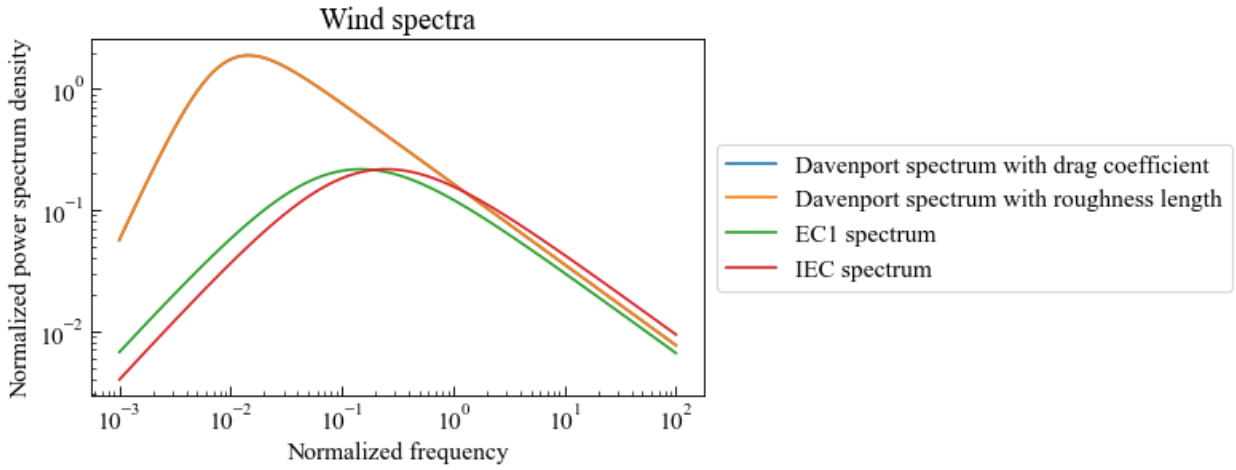
[38]:
```
vhub = 10
icenResults = [ iecSpectrum( f, vhub, normalized=True ) for f in wdRange ]
```

[39]:
```
fig, ax = plt.subplots( figsize=(10, 4) )
plt.xscale("log")
plt.yscale("log")

ax.plot( np.array( dsnRange ),
```

(continues on next page)

(continued from previous page)

```
         np.array( dsnResults ),
         label="Davenport spectrum with drag coefficient" )
ax.plot( np.array( dsrnRange ),
         np.array( dsrnResults ),
         label="Davenport spectrum with roughness length" )
ax.plot( np.array( ec1nRange ),
         np.array( ec1nResults ),
         label="EC1 spectrum" )
ax.plot( np.array( icenRange ),
         np.array( icenResults ),
         label="IEC spectrum" )


ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')
ax.set_xlabel( "Normalized frequency" )
ax.set_ylabel( "Normalized power spectrum density" )
ax.set_title( "Wind spectra" )

ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.tight_layout()
plt.show()
```



1.6.3 Sequence spectra

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import warnings
```

(continues on next page)

(continued from previous page)

```
plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'

# Filter the plot warning
warnings.filterwarnings( "ignore" )

# Numpy random seed
np.random.seed(0)
```

Periodogram spectrum

Function `periodogramSpectrum` transforms the sequence data to its power spectral density with the `scipy.signal.periodogram`.

Function help

[2]:
```
from ffpack.lsm import periodogramSpectrum
help( periodogramSpectrum )
```

```
Help on function periodogramSpectrum in module ffpack.lsm.sequenceSpectra:

periodogramSpectrum(data, fs)
    Power spectral density with `scipy.signal.periodogram`.

    Parameters
    ----------
    data: 1darray
        Sequence to calculate power spectral density.
    fs: scalar
        Sampling frequency.

    Returns
    -------
    freq: 1darray
        frequency components.
    psd: 1darray
        Power spectral density.

    Raises
    ------
    ValueError
        If data is not a 1darray.
        If fs is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import periodogramSpectrum
```

(continues on next page)

(continued from previous page)

```
    >>> data = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 2 ]
    >>> fs = 2
    >>> freq, psd = periodogramSpectrum( data, fs )
```

Example with generated data

[3]:
```
gfs = 1000  # 1kHz sampling frequency
fs1 = 10    # first signal component at 10 Hz
fs2 = 60    # second signal component at 60 Hz
T = 10      # 10s signal length
n0 = -10    # noise level (dB)
```

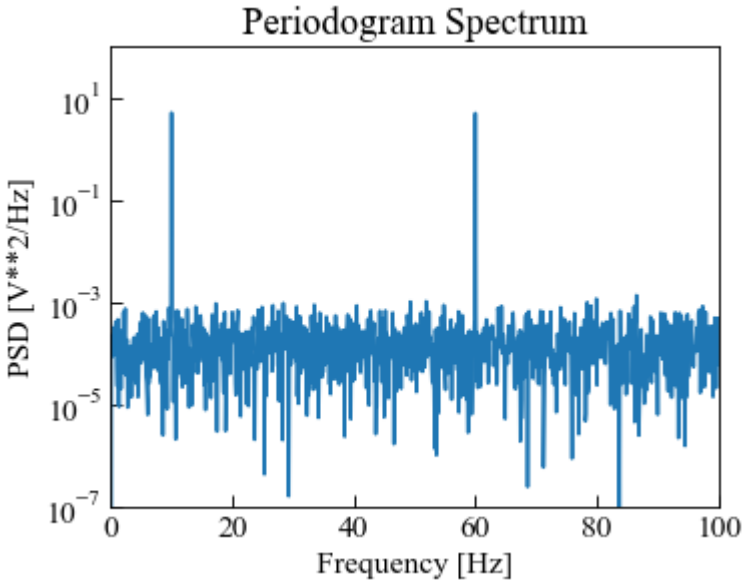
[4]:
```
t = np.r_[ 0: T: ( 1 / gfs ) ]  # sample time
gdata = np.sin( 2 * fs1 * np.pi * t ) + np.sin( 2 * fs2 * np.pi * t )

# white noise with power n0
gdata += np.random.randn( len( gdata ) ) * 10**( n0 / 20.0 )
```

[5]:
```
gfreq, gpsd = periodogramSpectrum( gdata, gfs )
```

[6]:
```
ind = np.argpartition( gpsd, -2 )[ -2: ]
peak1 = min( gfreq[ ind ] )
peak2 = max( gfreq[ ind ] )
print( [ peak1, peak2 ] )
```

```
[10.0, 60.0]
```

[7]:
```
fig, ax = plt.subplots()
plt.yscale("log")

ax.semilogy( np.array( gfreq ),
             np.array( gpsd ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')

ax.set_ylim( [ 1e-7, 1e2 ] )
ax.set_xlim( [ 0, 100 ] )

ax.set_xlabel( "Frequency [Hz]" )
ax.set_ylabel( "PSD [V**2/Hz]" )
ax.set_title( "Periodogram Spectrum" )

plt.tight_layout()
plt.show()
```



Welch spectrum

Function `welchSpectrum` transforms the sequence data to its power spectral density with the `scipy.signal.welch`.

Function help

[8]:
```
from ffpack.lsm import welchSpectrum
help( welchSpectrum )
```

```
Help on function welchSpectrum in module ffpack.lsm.sequenceSpectra:

welchSpectrum(data, fs, nperseg=1024)
    Power spectral density with `scipy.signal.welch`.

    Parameters
    ----------
    data: 1darray
        Sequence to calculate power spectral density.
    fs: scalar
        Sampling frequency.
    nperseg: scalar
        Length of each segment. Defaults to 1024.

    Returns
    -------
    freq: 1darray
        frequency components.
    psd: 1darray
        Power spectral density.

    Raises
```

(continues on next page)

(continued from previous page)

```
    ------
    ValueError
        If data is not a 1darray.
        If fs is not a scalar.

    Examples
    --------
    >>> from ffpack.lsm import welchSpectrum
    >>> data = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 2 ]
    >>> fs = 2
    >>> freq, psd = welchSpectrum( data, fs, nperseg=1024 )
```

Example with default values

[9]:
```
gfs = 1000  # 1kHz sampling frequency
fs1 = 10    # first signal component at 10 Hz
fs2 = 60    # second signal component at 60 Hz
T = 10      # 10s signal length
n0 = -10    # noise level (dB)
```

[10]:
```
t = np.r_[ 0: T: ( 1 / gfs ) ]  # sample time
gdata = np.sin( 2 * fs1 * np.pi * t ) + np.sin( 2 * fs2 * np.pi * t )

# white noise with power n0
gdata += np.random.randn( len( gdata ) ) * 10**( n0 / 20.0 )
```

[11]:
```
gfreq, gpsd = welchSpectrum( gdata, gfs )
```

[12]:
```
ind = np.argpartition( gpsd, -2 )[ -2: ]
peak1 = min( gfreq[ ind ] )
peak2 = max( gfreq[ ind ] )
print( [ peak1, peak2 ] )
```

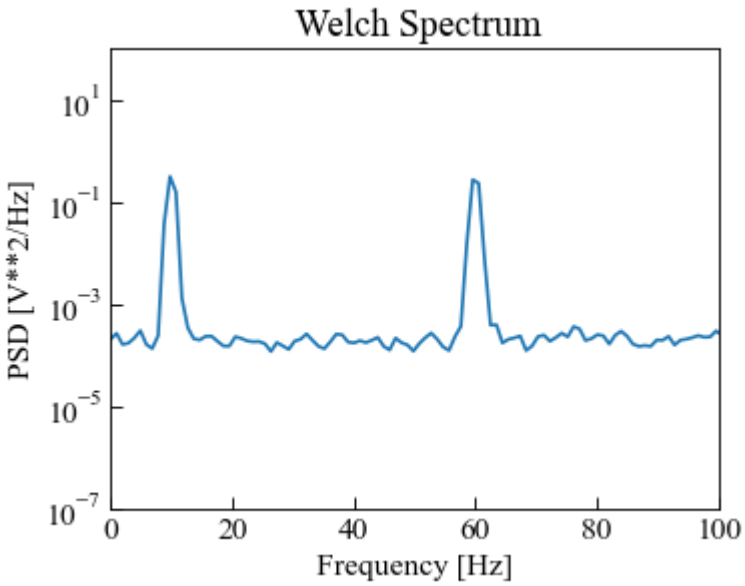
```
[9.765625, 59.5703125]
```

[13]:
```
fig, ax = plt.subplots()
plt.yscale("log")

ax.semilogy( np.array( gfreq ),
             np.array( gpsd ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.tick_params(axis='x', direction="in", length=3, which='minor')
ax.tick_params(axis='y', direction="in", length=3, which='minor')

ax.set_ylim( [ 1e-7, 1e2 ] )
ax.set_xlim( [ 0, 100 ] )
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel( "Frequency [Hz]" )
ax.set_ylabel( "PSD [V**2/Hz]" )
ax.set_title( " Welch Spectrum" )

plt.tight_layout()
plt.show()
```



1.6.4 Cycle counting matrix

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import warnings

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'

# Filter the plot warning
warnings.filterwarnings( "ignore" )
```

ASTM simple range counting matrix

Function help

[2]:
```
from ffpack.lsm import astmSimpleRangeCountingMatrix
help( astmSimpleRangeCountingMatrix )
```

```
Help on function astmSimpleRangeCountingMatrix in module ffpack.lsm.cycleCountingMatrix:

astmSimpleRangeCountingMatrix(data, resolution=0.5)
    Calculate ASTM simple range counting matrix.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate range counting matrix.
    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.

    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0.

    Examples
    --------
    >>> from ffpack.lsm import astmSimpleRangeCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = astmSimpleRangeCountingMatrix( data )
```

Example with default values

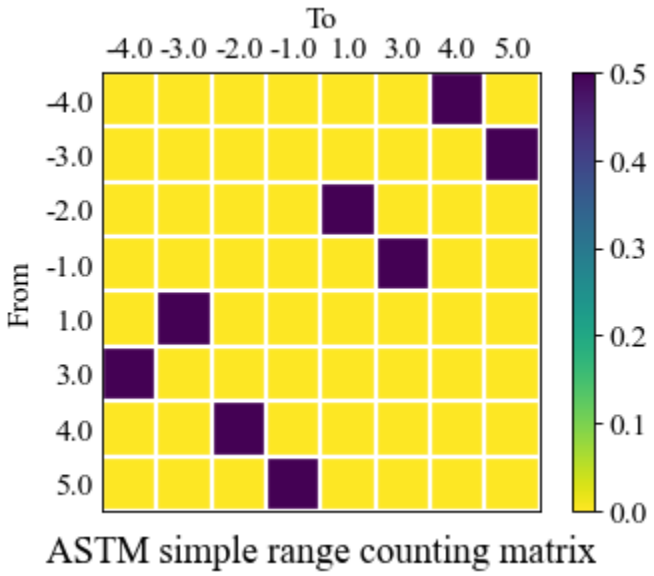
[3]:
```
asrcmData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
asrcmMat, asrcmIndex = astmSimpleRangeCountingMatrix( asrcmData )

asrcmMat = np.array( asrcmMat )
asrcmIndex = np.array( asrcmIndex ).astype( float )
```

[4]:
```
print( "ASTM simple range counting matrix" )
print( asrcmMat )
print()
print( "Matrix index" )
print( asrcmIndex )
```

```
ASTM simple range counting matrix
[[0.  0.  0.  0.  0.  0.  0.5 0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.5]
 [0.  0.  0.  0.  0.5 0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.5 0.  0. ]
 [0.  0.5 0.  0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.5 0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.5 0.  0.  0.  0. ]]

Matrix index
[-4. -3. -2. -1.  1.  3.  4.  5.]
```

[5]:
```
plt.set_cmap( "viridis_r" )
fig, ax = plt.subplots()

cax = ax.matshow( asrcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + asrcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + asrcmIndex.tolist() )
ax.set_xticks( np.arange( -.5, len( asrcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( asrcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "ASTM simple range counting matrix", y=-0.15 )

fig.colorbar(cax)
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



ASTM range pair counting matrix

Function help

[6]:
```
from ffpack.lsm import astmRangePairCountingMatrix
help( astmRangePairCountingMatrix )
```

```
Help on function astmRangePairCountingMatrix in module ffpack.lsm.cycleCountingMatrix:

astmRangePairCountingMatrix(data, resolution=0.5)
    Calculate ASTM range pair counting matrix.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate range pair counting matrix.
    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.
```

(continues on next page)

(continued from previous page)

```
    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.lsm import astmRangePairCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = astmRangePairCountingMatrix( data )
```

Example with default values

[7]:
```
arpcmData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
arpcmMat, arpcmIndex = astmRangePairCountingMatrix( arpcmData )

arpcmMat = np.array( arpcmMat )
arpcmIndex = np.array( arpcmIndex ).astype( float )
```

[8]:
```
print( "ASTM range pair counting matrix" )
print( arpcmMat )
print()
print( "Matrix index" )
print( arpcmIndex )
```

```
ASTM range pair counting matrix
[[0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]

Matrix index
[-3. -2. -1.  1.  3.  4.  5.]
```

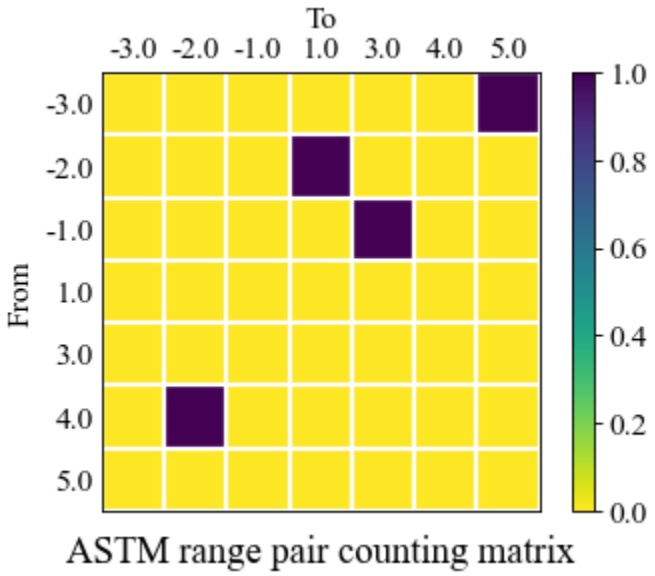
[9]:
```
plt.set_cmap( "viridis_r" )
fig, ax = plt.subplots()

cax = ax.matshow( arpcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + arpcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + arpcmIndex.tolist() )
ax.set_xticks( np.arange( -.5, len( arpcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( arpcmIndex ), 1 ), minor=True )
```

(continues on next page)

(continued from previous page)

```
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "ASTM range pair counting matrix", y=-0.15 )

fig.colorbar(cax)
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



ASTM rainflow counting matrix

Function help

[10]:
```
from ffpack.lsm import astmRainflowCountingMatrix
help( astmRainflowCountingMatrix )
```

```
Help on function astmRainflowCountingMatrix in module ffpack.lsm.cycleCountingMatrix:

astmRainflowCountingMatrix(data, resolution=0.5)
    Calculate ASTM rainflow counting matrix.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate rainflow counting matrix.
    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
```

(continues on next page)

(continued from previous page)

```
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.

    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.lsm import astmRainflowCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = astmRainflowCountingMatrix( data )
```

Example with default values

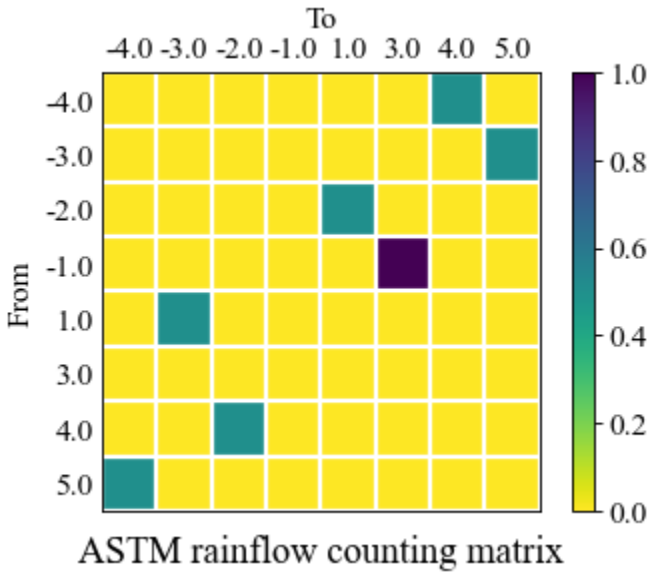
[11]:
```
arcmData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
arcmMat, arcmIndex = astmRainflowCountingMatrix( arcmData )

arcmMat = np.array( arcmMat )
arcmIndex = np.array( arcmIndex ).astype( float )
```

[12]:
```
print( "ASTM rainflow counting matrix" )
print( arcmMat )
print()
print( "Matrix index" )
print( arcmIndex )
```

```
ASTM rainflow counting matrix
[[0.  0.  0.  0.  0.  0.  0.5 0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.5]
 [0.  0.  0.  0.  0.5 0.  0.  0. ]
 [0.  0.  0.  0.  0.  1.  0.  0. ]
 [0.  0.5 0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.5 0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0.  0.  0.  0. ]]

Matrix index
[-4. -3. -2. -1.  1.  3.  4.  5.]
```

[13]:
```
plt.set_cmap( "viridis_r")
fig, ax = plt.subplots()

cax = ax.matshow( arcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + arcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + arcmIndex.tolist() )
ax.set_xticks( np.arange( -.5, len( arcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( arcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "ASTM rainflow counting matrix", y=-0.15 )

fig.colorbar( cax )
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



ASTM rainflow counting matrix for repeating histories

Function help

[14]:
```
from ffpack.lsm import astmRainflowRepeatHistoryCountingMatrix
help( astmRainflowRepeatHistoryCountingMatrix )
```

```
Help on function astmRainflowRepeatHistoryCountingMatrix in module ffpack.lsm.
↪cycleCountingMatrix:

astmRainflowRepeatHistoryCountingMatrix(data, resolution=0.5)
    Calculate ASTM simplified rainflow counting matrix for repeating histories.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate simplified rainflow counting matrix
        for repeating histories.
    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.

    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.lsm import astmRainflowRepeatHistoryCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = astmRainflowRepeatHistoryCountingMatrix( data )
```

Example with default values

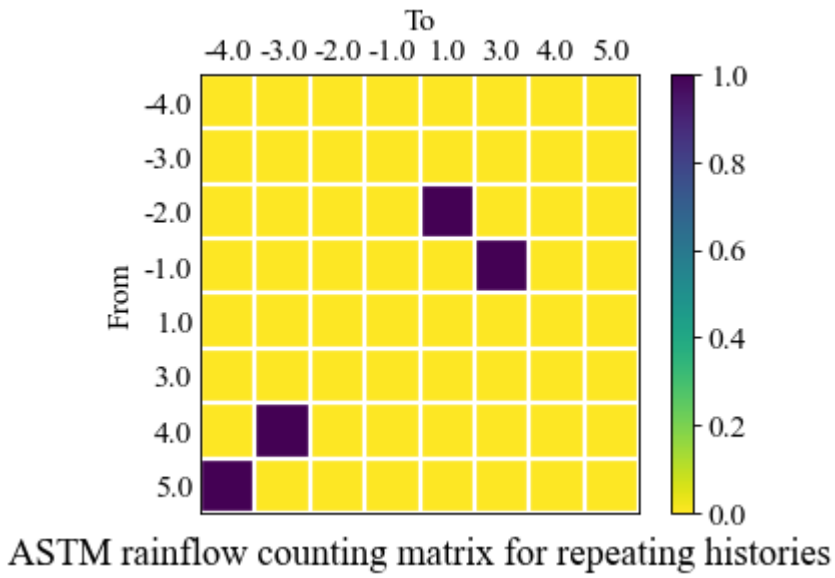
[15]:
```
arrhcmData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
arrhcmMat, arrhcmIndex = astmRainflowRepeatHistoryCountingMatrix( arrhcmData )

arrhcmMat = np.array( arrhcmMat )
arrhcmIndex = np.array( arrhcmIndex ).astype( float )
```

[16]:
```
print( "ASTM rainflow counting matrix for repeating histories" )
print( arrhcmMat )
print()
print( "Matrix index" )
print( arrhcmIndex )
```

```
ASTM rainflow counting matrix for repeating histories
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]]

Matrix index
[-4. -3. -2. -1.  1.  3.  4.  5.]
```

[17]:
```
plt.set_cmap( "viridis_r")
fig, ax = plt.subplots()

cax = ax.matshow( arrhcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + arrhcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + arrhcmIndex.tolist() )
ax.set_xticks( np.arange( -.5, len( arrhcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( arrhcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "ASTM rainflow counting matrix for repeating histories", y=-0.15 )

fig.colorbar( cax )
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



Johannesson min max counting matrix

Function help

[18]:
```
from ffpack.lsm import johannessonMinMaxCountingMatrix
help( johannessonMinMaxCountingMatrix )
```

```
Help on function johannessonMinMaxCountingMatrix in module ffpack.lsm.
↪cycleCountingMatrix:

johannessonMinMaxCountingMatrix(data, resolution=0.5)
    Calculate Johannesson minMax cycle counting matrix.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate rainflow counting matrix.
    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.
```

(continues on next page)

(continued from previous page)

```
    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.lsm import johannessonMinMaxCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = johannessonMinMaxCountingMatrix( data )
```

Example with default values

[19]:
```
jmmcmData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
jmmcmMat, jmmcmIndex = johannessonMinMaxCountingMatrix( jmmcmData )

jmmcmMat = np.array( jmmcmMat )
jmmcmIndex = np.array( jmmcmIndex ).astype( float )
```

[20]:
```
print( "Rychlik rainflow counting matrix" )
print( jmmcmMat )
print()
print( "Matrix index" )
print( jmmcmIndex )
```

```
Rychlik rainflow counting matrix
[[0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]

Matrix index
[-4. -3. -2. -1.  1.  3.  4.  5.]
```

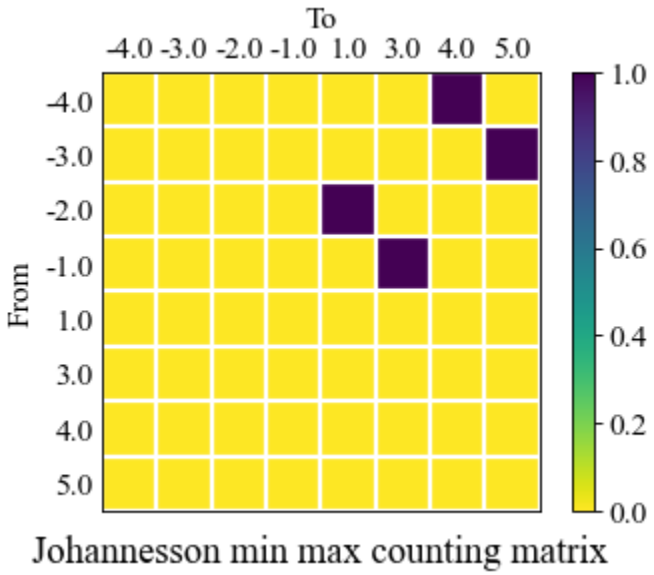
[21]:
```
plt.set_cmap( "viridis_r" )
fig, ax = plt.subplots()

cax = ax.matshow( jmmcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + jmmcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + jmmcmIndex.tolist() )
```

(continues on next page)

(continued from previous page)

```
ax.set_xticks( np.arange( -.5, len( jmmcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( jmmcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "Johannesson min max counting matrix", y=-0.15 )

fig.colorbar( cax )
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



Rychlik rainflow counting matrix

Function help

[22]:
```
from ffpack.lsm import rychlikRainflowCountingMatrix
help( rychlikRainflowCountingMatrix )
```

```
Help on function rychlikRainflowCountingMatrix in module ffpack.lsm.cycleCountingMatrix:

rychlikRainflowCountingMatrix(data, resolution=0.5)
    Calculate Rychlik rainflow counting matrix.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate rainflow counting matrix.
    resolution: bool, optional
        The desired resolution to round the data points.
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.

    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.lsm import rychlikRainflowCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = rychlikRainflowCountingMatrix( data )
```

Example with default values

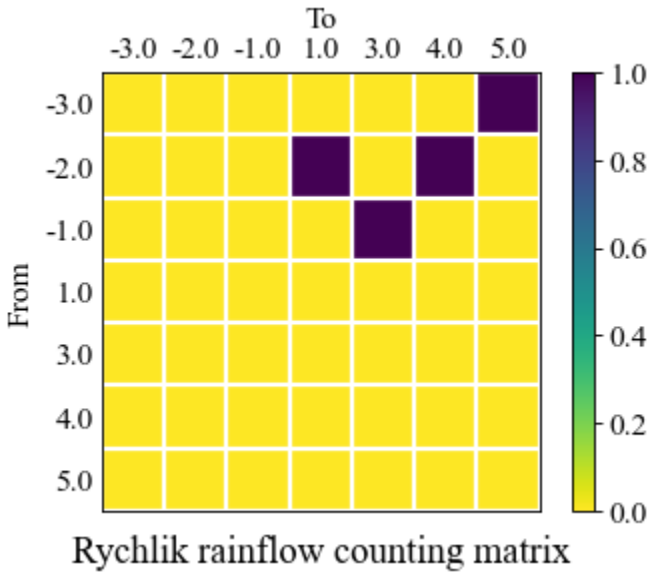
[23]:
```
rrcmData = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
rrcmMat, rrcmIndex = rychlikRainflowCountingMatrix( rrcmData )

rrcmMat = np.array( rrcmMat )
rrcmIndex = np.array( rrcmIndex ).astype( float )
```

[24]:
```
print( "Rychlik rainflow counting matrix" )
print( rrcmMat )
print()
print( "Matrix index" )
print( rrcmIndex )
```

```
Rychlik rainflow counting matrix
[[0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 1. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]

Matrix index
[-3. -2. -1.  1.  3.  4.  5.]
```

[25]:
```
plt.set_cmap( "viridis_r" )
fig, ax = plt.subplots()

cax = ax.matshow( rrcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + rrcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + rrcmIndex.tolist() )
ax.set_xticks( np.arange( -.5, len( rrcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( rrcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "Rychlik rainflow counting matrix", y=-0.15 )

fig.colorbar( cax )
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



Four point counting matrix

Function help

[26]:
```
from ffpack.lsm import fourPointCountingMatrix
help( fourPointCountingMatrix )
```

```
Help on function fourPointCountingMatrix in module ffpack.lsm.cycleCountingMatrix:

fourPointCountingMatrix(data, resolution=0.5)
    Calculate Four point cycle counting matrix.

    Parameters
    ----------
    data: 1d array
        Sequence data to calculate rainflow counting matrix.
    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.

    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.lsm import fourPointCountingMatrix
    >>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
    >>> rst, matrixIndexKey = fourPointCountingMatrix( data )
```

Example with default values

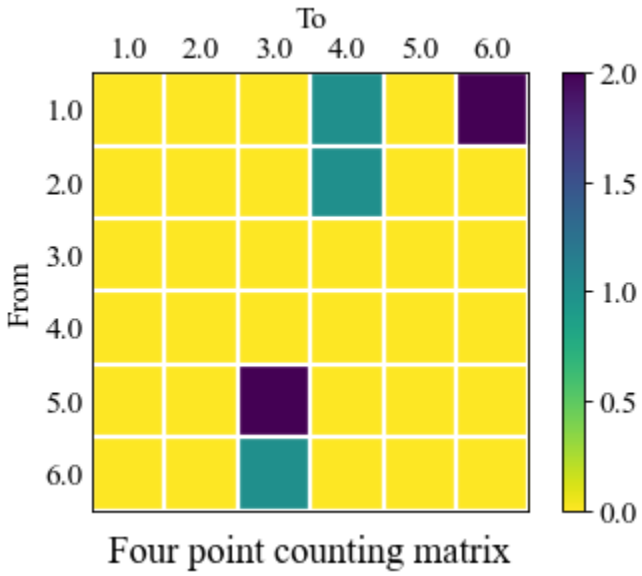
[27]:
```
fpcmData = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 4, 1, 5, 3, 6, 3, 6, 1, 5, 2 ]
fpcmMat, fpcmIndex = fourPointCountingMatrix( fpcmData )

fpcmMat = np.array( fpcmMat )
fpcmIndex = np.array( fpcmIndex ).astype( float )
```

[28]:
```
print( "Four point counting matrix" )
print( fpcmMat )
print()
print( "Matrix index" )
print( fpcmIndex )
```

```
Four point counting matrix
[[0. 0. 0. 1. 0. 2.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]]

Matrix index
[1. 2. 3. 4. 5. 6.]
```

[29]:
```
plt.set_cmap( "viridis_r" )
fig, ax = plt.subplots()

cax = ax.matshow( fpcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )
ax.set_xticklabels( [ '' ] + fpcmIndex.tolist() )
ax.set_yticklabels( [ '' ] + fpcmIndex.tolist() )
ax.set_xticks( np.arange( -.5, len( fpcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( fpcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )
ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "Four point counting matrix", y=-0.15 )

fig.colorbar( cax )
plt.tight_layout()
plt.show()
```

```
<Figure size 400x320 with 0 Axes>
```



1.7 Random and probabilistic model (rpm)

1.7.1 Metropolis–Hastings algorithm

The metropolis-Hastings algorithm is a typical algorithm that generates the samples for an arbitrary density function (i.e., target density function). It uses the proposal density (i.e., transition kernel) to generate the next sample candidate. The candidate is accepted with the acceptance ratio calculated by the target density function.

Notes

The proposal density function Q should satisfies the symmetry property, i.e., $Q(x|y) == Q(y|x)$. A usual choice of the proposal density function Q is Gaussian/normal distribution or uniform distribution.

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Metropolis–Hastings sampler

Class `MetropolisHastingsSampler` implements the Metropolis-Hastings sampler.

Based on the algorithm, three input parameters, namely `initialVal`, `targetPdf`, `proposalCSampler` are required to create a Metropolis-Hastings sampler.

`initialVal` is the first observed data point that is used to start the sampling procedure.

`targetPdf` is a target density function that returns the probability for a given value. It should be noted that the `targetPdf` function can be proportional to the target density function up to a multiplicative constant.

`proposalCSampler` is a sampler that can return a sample for a given observation. For example, for a given observation x_t, the sampler can be $x_{t+1} \sim \mathcal{N}(x_t, 1)$ and it can return a sample based on the current observation x_t.

The following examples show the application of the Metropolis-Hastings sampler for exponential-like and lognormal-like target distribution.

Class initialization help

[2]:
```
from ffpack.rpm import MetropolisHastingsSampler
help( MetropolisHastingsSampler.__init__ )
```

```
Help on function __init__ in module ffpack.rpm.metropolisHastings:

__init__(self, initialVal=None, targetPdf=None, proposalCSampler=None, sampleDomain=None,
↪ randomSeed=None, **sdKwargs)
    Initialize the Metropolis-Hastings sampler

    Parameters
    ----------
    initialVal: scalar or array_like
        Initial observed data point.
    targetPdf: function
        Target probability density function or target distribution function.
        targetPdf takes one input parameter and return the corresponding
        probability. It will be called as targetPdf( X ) where X is the same
        type as initialVal, and a scalar value of probability should be returned.
    proposalCSampler: function
        Proposal conditional sampler (i.e., transition kernel). proposalCSampler
        is a sampler that will return a sample for the given observed data point.
        A usual choice is to let proposalCSampler be a Gaussian/normal
        distribution centered at the observed data point. It will be called as
        proposalCSampler( X ) where X is the same type as initialVal, and a
        sample with the same type of initialVal should be returned.
    sampleDomain: function, optional
        Sample domain function. sampleDomain is a function to determine if a
        sample is in the sample domain. For example, if the sample doamin is
        [ 0, inf ] and the sample is -2, the sample will be rejected. For the
        sampling on field of real numbers, it should return True regardless of
        the sample value. It called as sampleDomain( cur, nxt, \**sdKwargs )
        where cur, nxt are the same type as initivalVal, and a boolean value
        should be returned.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
```

(continues on next page)

(continued from previous page)

```
        global config will be used.

    Raises
    ------
    ValueError
        If initialVal, targetPdf, or proposalCSampler is None.
        If targetPdf returns negative value.

    Examples
    --------
    >>> from ffpack.rpm import MetropolisHastingsSampler
    >>> initialVal = 1.0
    >>> targetPdf = lambda x : 0 if x < 0 else np.exp( -x )
    >>> proposalCSampler = lambda x : np.random.normal( x, 1 )
    >>> mhSampler = MetropolisHastingsSampler( initialVal, targetPdf,
    ...                                        proposalCsampler )
```

Example with exponential-like distribution

[3]:
```
mhsInitialVal = 1.0

# def mhsTargetPdf( x ):
#     return 0 if x < 0 else np.exp( -x )

# We can also use lambda function
mhsTargetPdf = lambda x : 0 if x < 0 else np.exp( -x )

# def mhsProposalCSampler( x ):
#     return np.random.normal( x, 1 )

# We can also use lambda function
mhsProposalCSampler = lambda x : np.random.normal( x, 1 )
```

[4]:
```
mhSampler = MetropolisHastingsSampler( initialVal=mhsInitialVal,
                                       targetPdf=mhsTargetPdf,
                                       proposalCSampler=mhsProposalCSampler,
                                       randomSeed=2023 )
```

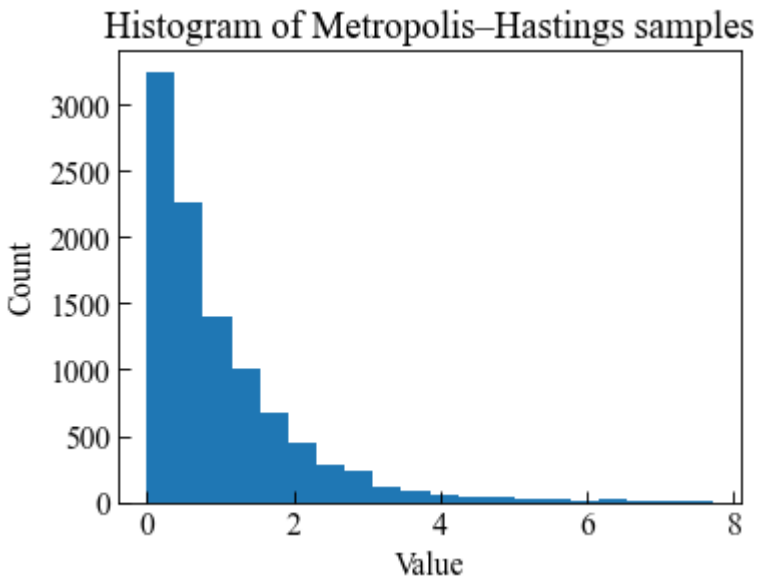
[5]:
```
mhsResults = np.zeros( 10000 )
for i in range( 10000 ):
    mhsResults[ i ] = mhSampler.getSample()
```

[6]:
```
fig, ax = plt.subplots()

ax.hist( np.array( mhsResults ), bins=20 )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel( "Count" )
ax.set_xlabel( "Value" )
ax.set_title( "Histogram of Metropolis-Hastings samples" )

plt.tight_layout()
plt.show()
```



Example with lognormal-like distribution

[7]:
```
mhsInitialVal = 1.0

def mhsTargetPdf( x ):
    return 0 if x < 0 else 1 / x * np.exp( -1 * np.power( np.log( x ), 2 ) )

def mhsProposalCSampler( x ):
    return np.random.normal( x, 1 )
```

[8]:
```
mhSampler = MetropolisHastingsSampler( initialVal=mhsInitialVal,
                                       targetPdf=mhsTargetPdf,
                                       proposalCSampler=mhsProposalCSampler,
                                       randomSeed=2023 )
```

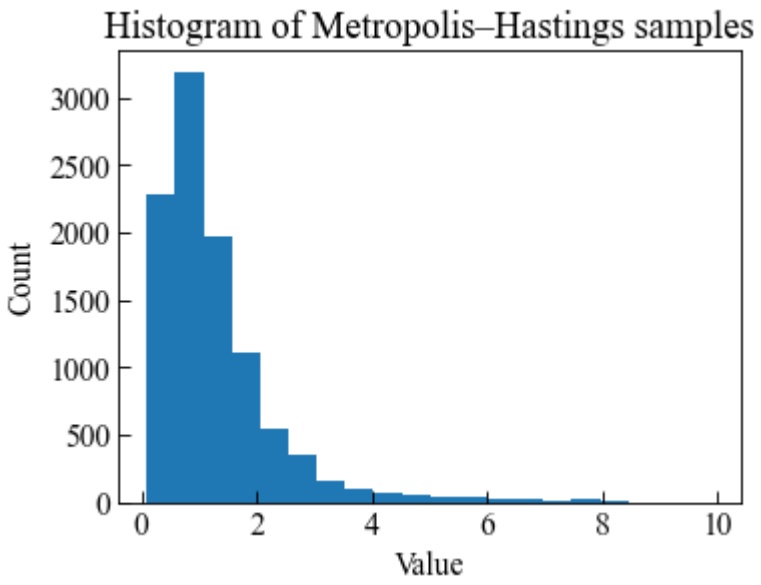
[9]:
```
mhsResults = np.zeros( 10000 )
for i in range( 10000 ):
    mhsResults[ i ] = mhSampler.getSample()
```

[10]:
```
fig, ax = plt.subplots()

ax.hist( np.array( mhsResults ), bins=20 )
```

(continues on next page)

(continued from previous page)

```
ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "Count" )
ax.set_xlabel( "Value" )
ax.set_title( "Histogram of Metropolis-Hastings samples" )

plt.tight_layout()
plt.show()
```



Au modified Metropolis–Hastings sampler

Class `AuModifiedMHSampler` implements the Metropolis-Hastings sampler.

The modified Metropolis-Hastings sampling algorithm proposed by Au and Beck samples data point at each dimension.

Reference:

- Au, S.K. and Beck, J.L., 2001. Estimation of small failure probabilities in high dimensions by subset simulation. Probabilistic engineering mechanics, 16(4), pp.263-277.

Class initialization help

[11]:
```
from ffpack.rpm import AuModifiedMHSampler
help( AuModifiedMHSampler.__init__ )
```

```
Help on function __init__ in module ffpack.rpm.metropolisHastings:

__init__(self, initialVal=None, targetPdf=None, proposalCSampler=None, sampleDomain=None,
↪ randomSeed=None, **sdKwargs)
    Initialize the Au modified Metropolis-Hastings sampler

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    initialVal: array_like
        Initial observed data point.
    targetPdf: function list
        Target probability density function list. Each element targetPdf[ i ] in
        the list is a callable function referring the independent marginal.
        targetPdf[ i ] takes one input parameter and return the corresponding
        probability. It will be called as targetPdf[ i ]( X[ i ] ) where X is a
        list in which the element is same type as initialVal[ i ], and a scalar
        value of probability should be returned by targetPdf[ i ]( X[ i ] ).
    proposalCSampler: function list
        Proposal conditional sampler list (i.e., transition kernel list). Each
        element proposalCSampler[ i ] in the list is a callable function
        referring a sampler that will return a sample for the given observed
        data point.  A usual choice is to let proposalCSampler[ i ] be a
        Gaussian/normal distribution centered at the observed data point.
        It will be called as proposalCSampler[ i ]( X[ i ] ) where X is a list
        in which each element is the same type as initialVal[ i ], and a
        sample with the same type of initialVal[ i ] should be returned.
    sampleDomain: function, optional
        Sample domain function. sampleDomain is a function to determine if a
        sample is in the sample domain. For example, if the sample doamin is
        [ 0, inf ] and the sample is -2, the sample will be rejected. For the
        sampling on field of real numbers, it should return True regardless of
        the sample value. It called as sampleDomain( cur, nxt, \**sdKwargs )
        where cur, nxt are lists in which each element is the same type as
        initivalVal[ i ], and a boolean value should be returned.
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Raises
    ------
    ValueError
        If initialVal, targetPdf, or proposalCSampler is None.
        If dims of initialVal, targetPdf, and proposalCSampler are not equal.
        If targetPdf returns negative value.

    Examples
    --------
    >>> from ffpack.rpm import AuModifiedMHSampler
    >>> initialValList = [ 1.0, 1.0 ]
    >>> targetPdf = [ lambda x : 0 if x < 0 else np.exp( -x ),
    ...               lambda x : 0 if x < 0 else np.exp( -x ) ]
    >>> proposalCSampler = [ lambda x : np.random.normal( x, 1 ),
    ...                      lambda x : np.random.normal( x, 1 ) ]
    >>> auMMHSampler = AuModifiedMHSampler( initialVal, targetPdf,
    ...                                     proposalCsampler )
```

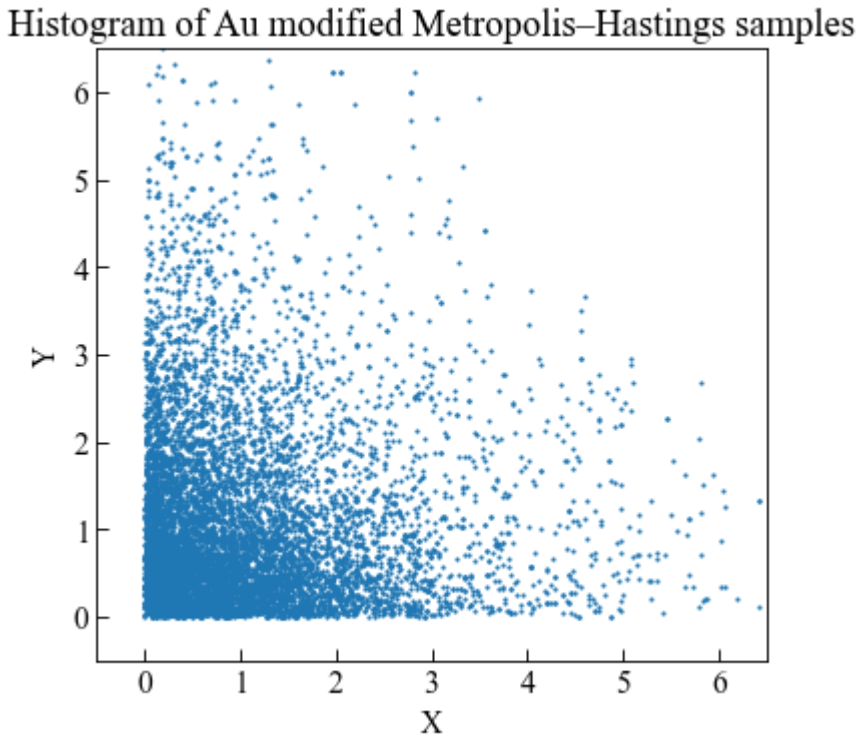
Example with exponential-like distribution

[12]:
```
aumhsInitialVal = [ 1.0, 1.0 ]


def autpdf( x ):
    return 0 if x < 0 else np.exp( -x )

aumhsTargetPdf = [ autpdf, autpdf ]

# We use normal distribution as proposal distribution
def aupcs( x ):
    return np.random.normal( x, 1 )

aumhsProposalCSampler = [ aupcs, aupcs ]
```

[13]:
```
aumhSampler = AuModifiedMHSampler( initialVal=aumhsInitialVal,
                                   targetPdf=aumhsTargetPdf,
                                   proposalCSampler=aumhsProposalCSampler,
                                   randomSeed=2023 )
```

[14]:
```
aumhsResults = np.zeros( [ 10000, 2 ] )
for i in range( 10000 ):
    aumhsResults[ i ] = aumhSampler.getSample()
```

[15]:
```
fig, ax = plt.subplots( figsize=( 5, 5 ) )

ax.plot( aumhsResults[ :, 0], aumhsResults[ :, 1 ], ".", markersize=2 )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "Y" )
ax.set_xlabel( "X" )
ax.set_title( "Histogram of Au modified Metropolis-Hastings samples" )
ax.set_xlim( [ -0.5, 6.5 ] )
ax.set_ylim( [ -0.5, 6.5 ] )

plt.tight_layout()
plt.show()
```



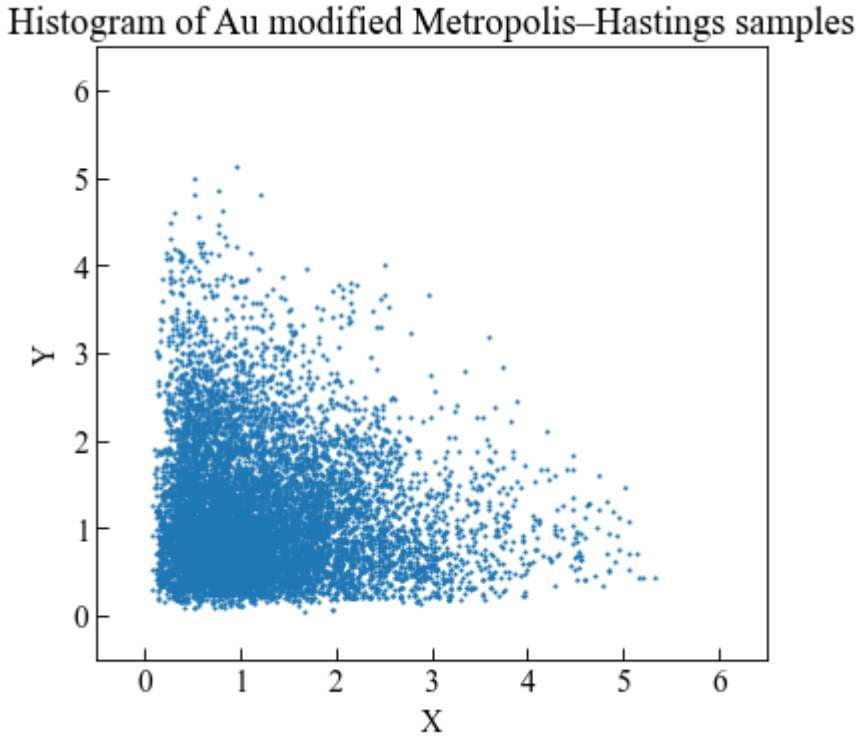
Example with lognormal-like distribution

[16]:
```
aumhsInitialVal = [ 1.0, 1.0 ]

def autpdf( x ):
    return 0 if x < 0 else 1 / x * np.exp( -1 * np.power( np.log( x ), 2 ) )

aumhsTargetPdf = [ autpdf, autpdf ]

# We use uniform distribution as proposal distribution
def aupcs( x ):
    return np.random.uniform( x-0.5, x+0.5 )

aumhsProposalCSampler = [ aupcs, aupcs ]
```

[17]:
```
aumhSampler = AuModifiedMHSampler( initialVal=aumhsInitialVal,
                                   targetPdf=aumhsTargetPdf,
                                   proposalCSampler=aumhsProposalCSampler,
                                   randomSeed=2023 )
```

[18]:
```
aumhsResults = np.zeros( [ 10000, 2 ] )
for i in range( 10000 ):
    aumhsResults[ i ] = aumhSampler.getSample()
```

[19]:
```
fig, ax = plt.subplots( figsize=( 5, 5 ) )

ax.plot( aumhsResults[ :, 0], aumhsResults[ :, 1 ], ".", markersize=2 )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "Y" )
ax.set_xlabel( "X" )
ax.set_title( "Histogram of Au modified Metropolis-Hastings samples" )
ax.set_xlim( [ -0.5, 6.5 ] )
ax.set_ylim( [ -0.5, 6.5 ] )

plt.tight_layout()
plt.show()
```



1.7.2 Nataf algorithm

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
```

(continues on next page)

(continued from previous page)

```
plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Nataf transformation

Nataf transformation is an isoprobabilistic transformation to create the joint distribution based on the marginal distributions and linear correlation coefficients. The Gaussian copula is assumed in the Nataf transformation for the joint distribution.

Here, we give a brief introduction to the Nataf transformation. The details can be found in the references. For the random variables $\mathbf{X} = (X_1, X_2, \ldots, X_n)$, we know the marginal distributions f_{X_i} for each random variable and the correlation coefficients ρ_{ij} for any two random variables X_i and X_j. Nataf transformation represents the joint distribution of the random variables $\mathbf{X}$ with the Gaussian copula. There are two steps in Nataf transformation:

1. Transfrom correlated random variables $\mathbf{X}$ into correlated standard normal variables $\mathbf{Z}$.
2. Transfrom correlated standard normal variables $\mathbf{Z}$ into independent standard normal variables $\mathbf{U}$.

To transform a random variable X_i to standard normal variables Z_i, the following transformation can be performed for each random variable,

$$z_i = \mathbf{\Phi}^{-1}(F_{X_i}(x_i))$$

Since the random variables $\mathbf{X}$ are correlated with the correlation coefficients ρ_{ij} for X_i and X_j, the key for step 1 in Nataf transformation is to solve the $\rho_{0,ij}$ for $\mathbf{Z}$. The $\rho_{0,ij}$ for Z_i and Z_j can be expressed with the implicit function,

$$\rho_{ij} = \int_R \int_R \left(\frac{F_{X_i}^{-1}(\Phi(z_i)) - \mu_i}{\sigma_i} \right) \left(\frac{F_{X_j}^{-1}(\Phi(z_j)) - \mu_j}{\sigma_j} \right) \phi(z_i, z_j; \rho_{0,ij}) \mathrm{d}z_i \mathrm{d}z_j$$

where $:nbsphinx-math:*phi* `( x, y; :nbsphinx-math:`rho`) $ is the standard bivariate normal distribution with correlation coefficient ρ.

$$\phi(x, y; \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)} \right)$$

After solving the aforementioned equation, we know the correlation coefficient $\rho_{0,ij}$ for Z_i and Z_j. The correlation matrix for $\mathbf{Z}$ is denoted by $\mathbf{R}_0$. Then, Cholesky decomposition of correlation matrix $\mathbf{R}_0$ is performed,

$$\mathbf{R}_0 = \mathbf{L}_0 \mathbf{L}_0^T$$

Step 2 in Nataf transformation can be performed,

$$\mathbf{U} = \mathbf{L}_0^{-1} \mathbf{Z}$$

Then, we map the correlated random variable $\mathbf{X}$ (data point in X space) to independent standard normal variables $\mathbf{U}$ (data point in U space) now.

Reference:

- Lemaire, M., 2013. Structural reliability. John Wiley & Sons.
- Bourinet, J.M., 2018. Reliability analysis and optimal design under uncertainty-Focus on adaptive surrogate-based approaches (Doctoral dissertation, Université Clermont Auvergne).
- Wang, C., 2021. Structural reliability and time-dependent reliability. Cham, Switzerland: Springer.

Class initialization help

[2]:
```
from ffpack.rpm import NatafTransformation
help( NatafTransformation.__init__ )
```

```
Help on function __init__ in module ffpack.rpm.nataf:

__init__(self, distObjs, corrMat, quadDeg=99, quadRange=8, randomSeed=None)
    Initialize the Nataf distribution.

    Parameters
    ----------
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
        objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
    corrMat: 2d matrix
        Correlation matrix of the marginal distributions.
    quadDeg: integer
        Quadrature degree.
    quadRange: scalar
        Quadrature range. The integral will be performed in the range
        [ -quadRange, quadRange ].
    randomSeed: integer, optional
        Random seed. If randomSeed is none or is not an integer, the random seed in
        global config will be used.

    Raises
    ------
    ValueError
        If distObjs is empty.
        If dimensions are not match for distObjs and corrMat.
        If corrMat is not 2d matrix.
        If corrMat is not positive definite.
        If corrMat is not symmetric.
        If corrMat diagonal is not 1.

    Examples
    --------
    >>> from ffpack.rpm import NatafTransformation
    >>> distObjs = [ stats.norm(), stats.norm() ]
    >>> corrMat = [ [ 1.0, 0.5 ], [ 0.5, 1.0 ] ]
    >>> natafDist = NatafTransformation( distObjs=distObjs, corrMat=corrMat )
```

Example with normal distribution

[3]:
```
# Define a standard bivariate normal distribution with correlation coefficient for␣
↪comparison
def standardBivariateNormalDistributionWithCorrelationCoefficient( x, y, rho ):
    return 1 / ( 2 * np.pi * np.sqrt( 1 - rho**2) ) * \
           np.exp( -1 / ( 2 * ( 1 - rho**2 ) ) *
           ( x**2 - 2 * rho * x * y + y**2 ) )
```

[4]:
```
# We create the Nataf transformation with 2 normal distributions
natafDistObjs = [ stats.norm(), stats.norm() ]
natafCorrMat = [ [ 1.0, 0.5 ], [ 0.5, 1.0 ] ]

natafNormDist = NatafTransformation( distObjs=natafDistObjs, corrMat=natafCorrMat )
```

[5]:
```
# Transformation from U space to X space
natafU = [ 1.0, 1.0 ]
natafX, natafJ = natafNormDist.getX( natafU )
print( "Data point coordinate in U space: " )
print( natafU )
print()
print( "Data point coordinate in X space: " )
print( natafX )
print()
print( "Jacobian matrix for transformation: " )
print( natafJ )
```

```
Data point coordinate in U space:
[1.0, 1.0]

Data point coordinate in X space:
[1.        1.3660254]

Jacobian matrix for transformation:
[[ 1.          0.        ]
 [-0.57735027  1.15470054]]
```

[6]:
```
# Transformation from X space to U space
natafX = [ 1.0, 1.0 ]
natafU, natafJ = natafNormDist.getU( natafX )
print( "Data point coordinate in X space: " )
print( natafX )
print()
print( "Data point coordinate in U space: " )
print( natafU )
print()
print( "Jacobian matrix for transformation: " )
print( natafJ )
```

```
Data point coordinate in X space:
[1.0, 1.0]

Data point coordinate in U space:
```

(continues on next page)

(continued from previous page)

```
[1.         0.57735027]

Jacobian matrix for transformation:
[[1.        0.       ]
 [0.5       0.8660254]]
```

[7]:
```
# Joint pdf for X
natafX = [ 1.0, 0.5 ]
natafPdfX = natafNormDist.pdf( natafX )
print( "pdf from Nataf transformation: " )
print( natafPdfX )
print()
print( "pdf from standard bivariate normal distribution with correlation coefficient:" )
print( standardBivariateNormalDistributionWithCorrelationCoefficient( 1.0, 0.5, 0.5 ) )
```

```
pdf from Nataf transformation:
0.11146595955294503

pdf from standard bivariate normal distribution with correlation coefficient:
0.11146595955293902
```

[8]:
```
# Joint cdf for X
natafX = [ 1.0, 0.5 ]
natafCdfX = natafNormDist.cdf( natafX )
print( "cdf from Nataf transformation: " )
print( natafCdfX )
```

```
cdf from Nataf transformation:
0.630283927552582
```

Example with exponential distribution

[9]:
```
# We create the Nataf transformation with 2 normal distributions
natafDistObjs = [ stats.expon(), stats.expon() ]
natafCorrMat = [ [ 1.0, 0.2 ], [ 0.2, 1.0 ] ]

natafExponDist = NatafTransformation( distObjs=natafDistObjs, corrMat=natafCorrMat )
```

[10]:
```
# Transformation from U space to X space
natafU = [ 1.0, 1.0 ]
natafX, natafJ = natafExponDist.getX( natafU )
print( "Data point coordinate in U space: " )
print( natafU )
print()
print( "Data point coordinate in X space: " )
print( natafX )
print()
print( "Jacobian matrix for transformation: " )
print( natafJ )
```

```
Data point coordinate in U space:
[1.0, 1.0]

Data point coordinate in X space:
[1.84102165 2.17177436]

Jacobian matrix for transformation:
[[ 0.65567954  0.        ]
 [-0.15726467  0.6077046 ]]
```

[11]:
```
# Transformation from X space to U space
natafX = [ 1.0, 1.0 ]
natafU, natafJ = natafExponDist.getU( natafX )
print( "Data point coordinate in X space: " )
print( natafX )
print()
print( "Data point coordinate in U space: " )
print( natafU )
print()
print( "Jacobian matrix for transformation: " )
print( natafJ )
```

```
Data point coordinate in X space:
[1.0, 1.0]

Data point coordinate in U space:
[0.33747496 0.26610302]

Jacobian matrix for transformation:
[[1.02440995 0.        ]
 [0.23892819 0.99615715]]
```

[12]:
```
# Joint pdf for X
natafX = [ 1.0, 0.5 ]
natafPdfX = natafExponDist.pdf( natafX )
print( "pdf from Nataf transformation: " )
print( natafPdfX )
```

```
pdf from Nataf transformation:
0.22315052067193208
```

[13]:
```
# Joint cdf for X
natafX = [ 1.0, 0.5 ]
natafCdfX = natafExponDist.cdf( natafX )
print( "cdf from Nataf transformation: " )
print( natafCdfX )
```

```
cdf from Nataf transformation:
0.2824109061173752
```

1.8 Risk and reliability model (rrm)

1.8.1 First order second moment

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Mean value FOSM

FOSM with mean value algorithm can be expressed by the following equation,

$$\beta = \frac{g(\mu_1, \mu_2, \ldots, \mu_n)}{\sqrt{\sum_{i=1}^{n} \alpha_i \sigma_i}}$$

where g is the limit state function (LSF); α_i is given by,

$$\alpha_i = \left.\frac{\partial g}{\partial X_i}\right|_{\mu_i}$$

Function `mvalFOSM` implements the FOSM with mean value algorithm.

Reference: Nowak, A.S. and Collins, K.R., 2012. Reliability of structures. CRC press.

Function help

[2]:
```
from ffpack.rrm import mvalFOSM
help( mvalFOSM )
```

```
Help on function mvalFOSM in module ffpack.rrm.firstOrderSecondMoment:

mvalFOSM(dim, g, dg, mus, sigmas, dx=1e-06)
    First order second moment method based on mean value algorithm.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    dg: array_like of function
        Gradient of the limit state function. It should be an array_like of function
        like dg = [ dg_dx1, dg_dx2, ... ]. To get the derivative of i-th random
```

(continues on next page)

(continued from previous page)

```
        variable at ( x1*, x2*, ... ), dg[ i ]( x1*, x2*, ... ) will be called.
        dg can be None, see the following Notes.
    mus: 1d array
        Mean of the random variables.
    sigmas: 1d array
        Variance of the random variables.
    dx : scalar, optional
        Spacing for auto differentiation. Not required if dg is provided.

    Returns
    -------
    beta: scalar
        Reliability index.
    pf: scalar
        probability of failure.

    Raises
    ------
    ValueError
        If the dim is less than 1.
        If the dim does not match the length of mus and sigmas.

    Notes
    -----
    If dg is None, the numerical differentiation will be used. The tolerance of the
    numerical differentiation can be changed in globalConfig.

    Examples
    --------
    >>> from ffpack.rrm import mvalFOSM
    >>> dim = 2
    >>> g = lambda X: 3 * X[ 0 ] - 2 * X[ 1 ]
    >>> dg = [ lambda X: 3, lambda X: -2 ]
    >>> mus = [ 1, 1 ]
    >>> sigmas = [ 3, 4 ]
    >>> beta, pf = mvalFOSM( dim, g, dg, mus, sigmas)
```

Example with explicit derivative of LSF

[3]:
```
# Define the dimension for the FOSM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: 3 * X[ 0 ] - 2 * X[ 1 ]

# Explicit derivative of LSF
# dg is a list in which each element is a partial derivative function of g w.r.t. X
# dg[0] = partial g / partial X[0]
# dg[1] = partial g / partial X[1]
```

(continues on next page)

(continued from previous page)

```
dg = [ lambda X: 3, lambda X: -2 ]

# Mean and standard deviation of the random variables
mus = [ 1, 1 ]
sigmas = [ 3, 4 ]

# Use mean value algorithm to get results
beta, pf = mvalFOSM( dim, g, dg, mus, sigmas)
```

[4]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
```

```
Reliability index:
0.08304547985373997

Failure probability:
0.46690768839408386
```

Example with automatic differentiation of LSF

[5]:
```
# Define the dimension for the FOSM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: 3 * X[ 0 ] - 2 * X[ 1 ]

# If dg is None, the internal automatic differentiation algorithm will be used
dg = None

# Mean and standard deviation of the random variables
mus = [ 1, 1 ]
sigmas = [ 3, 4 ]

# Use mean value algorithm to get results
beta, pf = mvalFOSM( dim, g, dg, mus, sigmas)
```

[6]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
```

```
Reliability index:
0.08304547985853711

Failure probability:
0.46690768839217667
```

1.8.2 First order reliability method

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Hasofer-Lind-Rackwitz-Fiessler FORM

Hasofer-Lind-Rackwitz-Fiessler (HLRF) algorithm is an iterative method to find the reliability index. It is shown that the HLRF algorithm is very effective in many situations even if the convergence is not assured in all cases.

Function `hlrfFORM` implements the FORM with Hasofer-Lind-Rackwitz-Fiessler algorithm. The Nataf transformation is used in the method to map the random variables from X space to U space.

Reference:

1. Wang, C., 2021. Structural reliability and time-dependent reliability. Cham, Switzerland: Springer.
2. Lemaire, M., 2013. Structural reliability. John Wiley & Sons.

Function help

[2]:
```
from ffpack.rrm import hlrfFORM
help( hlrfFORM )
```

```
Help on function hlrfFORM in module ffpack.rrm.firstOrderReliabilityMethod:

hlrfFORM(dim, g, dg, distObjs, corrMat, iter=1000, tol=1e-06, quadDeg=99, quadRange=8,␣
↪dx=1e-06)
    First order reliability method based on Hasofer-Lind-Rackwitz-Fiessler algorithm.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    dg: array_like of function
        Gradient of the limit state function. It should be an array_like of function
        like dg = [ dg_dx1, dg_dx2, ... ]. To get the derivative of i-th random
        variable at ( x1*, x2*, ... ), dg[ i ]( x1*, x2*, ... ) will be called.
        dg can be None, see the following Notes.
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
```

(continues on next page)

(continued from previous page)

```
    objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
corrMat: 2d matrix
    Correlation matrix of the marginal distributions.
iter: integer
    Maximum iteration steps.
tol: scalar
    Tolerance to demtermine if the iteration converges.
quadDeg: integer
    Quadrature degree for Nataf transformation
quadRange: scalar
    Quadrature range for Nataf transformation. The integral will be performed
    in the range [ -quadRange, quadRange ].
dx : scalar, optional
    Spacing for auto differentiation. Not required if dg is provided.

Returns
-------
beta: scalar
    Reliability index.
pf: scalar
    Probability of failure.
uCoord: 1d array
    Design point coordinate in U space.
xCoord: 1d array
    Design point coordinate in X space.

Raises
------
ValueError
    If the dim is less than 1.
    If the dim does not match the disObjs and corrMat.
    If corrMat is not 2d matrix.
    If corrMat is not positive definite.
    If corrMat is not symmetric.
    If corrMat diagonal is not 1.

Notes
-----
If dg is None, the numerical differentiation will be used. The tolerance of the
numerical differentiation can be changed in globalConfig.

Examples
--------
>>> from ffpack.rrm import hlrfFORM
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> dg = [ lambda X: -1, lambda X: -1 ]
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> beta, pf, uCoord, xCoord = hlrfFORM( dim, g, dg, distObjs, corrMat )
```

Example with explicit derivative of LSF

[3]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: 1.0 - X[ 0 ] - X[ 1 ]

# Explicit derivative of LSF
# dg is a list in which each element is a partial derivative function of g w.r.t. X
# dg[0] = partial g / partial X[0]
# dg[1] = partial g / partial X[1]
dg = [ lambda X: -1, lambda X: -1 ]

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm(), stats.norm() ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = hlrfFORM( dim, g, dg, distObjs, corrMat )
```

[4]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.7071067811865477

Failure probability:
0.23975006109347669

Design point coordinate in U space:
[0.5000000000000001, 0.5000000000000001]

Design point coordinate in X space:
[0.5000000000000001, 0.5000000000000001]
```

Example with automatic differentiation of LSF

[5]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: 1.0 - X[ 0 ] - X[ 1 ]

# If dg is None, the internal automatic differentiation algorithm will be used
dg = None

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm(), stats.norm() ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = hlrfFORM( dim, g, dg, distObjs, corrMat )
```

[6]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.7071053669729852

Failure probability:
0.23975050048498586

Design point coordinate in U space:
[0.4999990000000001, 0.4999990000000001]

Design point coordinate in X space:
[0.4999990000000025, 0.4999990000000025]
```

Constrained optimization FORM

Finding the design point or most probable point (MPP) for a given limit state function is basically a constrained optimization problem. It can be represented by the following equation,

$$\beta = \underset{\beta}{\operatorname{argmin}} ||\mathbf{U}||$$

$$s.t. g(\mathbf{X}) = g(\mathbf{T}^{-1}(\mathbf{U})) = g(\mathbf{U}) = 0$$

where transformation $\mathbf{T}$ is introduced to map the original random variables $\mathbf{X}$ (in X-space) to the standard, uncorrelated normal variables $\mathbf{U}$ with $\mathbf{U} = \mathbf{T}(\mathbf{X})$.

Function `coptFORM` implements the FORM with constrained optimization algorithm. The Nataf transformation is used in the method to map the random variables from X space to U space.

Function help

[7]:
```
from ffpack.rrm import coptFORM
help( coptFORM )
```

```
Help on function coptFORM in module ffpack.rrm.firstOrderReliabilityMethod:

coptFORM(dim, g, distObjs, corrMat, quadDeg=99, quadRange=8)
    First order reliability method based on constrained optimization.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
        objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
    corrMat: 2d matrix
        Correlation matrix of the marginal distributions.
    quadDeg: integer
        Quadrature degree for Nataf transformation
    quadRange: scalar
        Quadrature range for Nataf transformation. The integral will be performed
        in the range [ -quadRange, quadRange ].

    Returns
    -------
    beta: scalar
        Reliability index.
    pf: scalar
        Probability of failure.
    uCoord: 1d array
        Design point coordinate in U space.
    xCoord: 1d array
        Design point coordinate in X space.

    Raises
    ------
    ValueError
        If the dim is less than 1.
        If the dim does not match the disObjs and corrMat.
        If corrMat is not 2d matrix.
        If corrMat is not positive definite.
        If corrMat is not symmetric.
        If corrMat diagonal is not 1.

    Examples
```

(continues on next page)

(continued from previous page)

```
    --------
    >>> from ffpack.rrm import coptFORM
    >>> dim = 2
    >>> g = lambda X: -np.sum( X ) + 1
    >>> distObjs = [ stats.norm(), stats.norm() ]
    >>> corrMat = np.eye( dim )
    >>> beta, pf, uCoord, xCoord = coptFORM( dim, g, distObjs, corrMat )
```

Example with default values

[8]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: 1.0 - X[ 0 ] - X[ 1 ]

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm(), stats.norm() ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = coptFORM( dim, g, distObjs, corrMat )
```

[9]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.7071067706498352

Failure probability:
0.23975006436719704

Design point coordinate in U space:
[0.4999999925494192, 0.4999999925494193]

Design point coordinate in X space:
[0.4999999925494192, 0.4999999925494192]
```

1.8.3 Second order reliability method

For the second order reliability method, the random variables X first should be transformed to standard normal uncorrelated variables U. Then, the limit state function $g(U)$ can be approximated with a second order Taylor expansion,

$$g(U) \approx g(U^*) + \nabla g(U^*)^T (U - U^*) + \frac{1}{2}(U - U^*)^T \nabla^2 g(U^*)(U - U^*)$$

where U^* is the design point or most probable failure point (MPP); $\nabla^2 g(U^*)$ represent the Hessian matrix evaluated at the design point, it can be represented by,

$$\nabla^2 g(U^*)_{ij} = \frac{\partial g(U^*)}{\partial u_i \partial u_j}$$

Since $g(U^*) = 0$, the second order Taylor expansion can be represented by,

$$g(U) \approx \nabla g(U^*)^T (U - U^*) + \frac{1}{2}(U - U^*)^T \nabla^2 g(U^*)(U - U^*)$$

To solve the problem, a transformation $Y = HU$ is performed so that the last coordinate coincides with the vector U^* from the origin to the design point (β vector). H can be obtained by a Gram-Schmidt orthogonalization. Thus, the Taylor expansion is,

$$g(Y) \approx -y_n + \beta + \frac{1}{2}(Y - Y^*)^T H \frac{\nabla^2 g(U^*)}{||\nabla g(U^*)||} H^T (Y - Y^*)$$

where $Y^* = \{0, 0, \ldots, \beta\}^T$ is the design point in Y space corresponding to the design point U^* in U space; $(Y - Y^*) = \{y_1, y_2, \ldots, y_n - \beta\}^T$.

The main curvatures k_i can be obtained by transforming the $(n-1) \times (n-1)$ order matrix of $H \frac{\nabla^2 g(U^*)}{||\nabla g(U^*)||} H^T$ to a diagonal matrix (i.e., eigenvalues).

References:

- Choi, S.K., Canfield, R.A. and Grandhi, R.V., 2007. Reliability-Based Structural Design. Springer London.

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Breitung SORM

With the $n-1$ main curvatures k_i and first order reliability index β, Breitung proposed the parabolic approximation of probability of failure,

$$p_f = \Phi(-\beta) \prod_{i=1}^{n-1} (1 + k_i \beta)^{-1/2}$$

where $\Phi(\cdot)$ is the standard normal Cumulative Distribution Function (CDF); β is the first order reliability index; k_i is the main curvatures of the limit-state function at design point; It should be noted that Breitung formula is applicable for large β.

Function `breitungSORM` implements the SORM with Breitung algorithm. The Nataf transformation is used in the method to map the random variables from X space to U space.

References:

- Breitung, K., 1984. Asymptotic approximations for multinormal integrals. Journal of Engineering Mechanics, 110(3), pp.357-366.
- Hu, Z., Mansour, R., Olsson, M. and Du, X., 2021. Second-order reliability methods: a review and comparative study. Structural and Multidisciplinary Optimization, 64(6), pp.3233-3263.
- Bourinet, J.M., 2018. Reliability analysis and optimal design under uncertainty-Focus on adaptive surrogate-based approaches (Doctoral dissertation, Université Clermont Auvergne).

Function help

[2]:
```
from ffpack.rrm import breitungSORM
help( breitungSORM )
```

```
Help on function breitungSORM in module ffpack.rrm.secondOrderReliabilityMethod:

breitungSORM(dim, g, dg, distObjs, corrMat, quadDeg=99, quadRange=8, dx=1e-06)
    Second order reliability method based on Breitung algorithm.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    dg: array_like of function
        Gradient of the limit state function. It should be an array_like of function
        like dg = [ dg_dx1, dg_dx2, ... ]. To get the derivative of i-th random
        variable at ( x1*, x2*, ... ), dg[ i ]( x1*, x2*, ... ) will be called.
        dg can be None, see the following Notes.
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
        objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
    corrMat: 2d matrix
        Correlation matrix of the marginal distributions.
    quadDeg: integer
        Quadrature degree for Nataf transformation
    quadRange: scalar
        Quadrature range for Nataf transformation. The integral will be performed
        in the range [ -quadRange, quadRange ].
    dx : scalar, optional
        Spacing for auto differentiation. Not required if dg is provided.

    Returns
    -------
    beta: scalar
```

(continues on next page)

(continued from previous page)

```
        Reliability index.
    pf: scalar
        Probability of failure.
    uCoord: 1d array
        Design point coordinate in U space.
    xCoord: 1d array
        Design point coordinate in X space.

    Raises
    ------
    ValueError
        If the dim is less than 1.
        If the dim does not match the disObjs and corrMat.
        If corrMat is not 2d matrix.
        If corrMat is not positive definite.
        If corrMat is not symmetric.
        If corrMat diagonal is not 1.

    Notes
    -----
    If dg is None, the numerical differentiation will be used. The tolerance of the
    numerical differentiation can be changed in globalConfig.

    Examples
    --------
    >>> from ffpack.rrm import breitungSORM
    >>> dim = 2
    >>> g = lambda X: -np.sum( X ) + 1
    >>> dg = [ lambda X: -1, lambda X: -1 ]
    >>> distObjs = [ stats.norm(), stats.norm() ]
    >>> corrMat = np.eye( dim )
    >>> beta, pf, uCoord, xCoord = breitungSORM( dim, g, dg, distObjs, corrMat )
```

Example with explicit derivative of LSF

[3]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: X[ 0 ] ** 4 + 2 * X[ 1 ] ** 4 - 20

# Explicit derivative of LSF
# dg is a list in which each element is a partial derivative function of g w.r.t. X
# dg[0] = partial g / partial X[0]
# dg[1] = partial g / partial X[1]
dg = [ lambda X: 4 * X[ 0 ] ** 3, lambda X: 8 * X[ 1 ] ** 3 ]

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm( 5.0, 5.0 ), stats.norm( 5.0, 5.0 ) ]
```

(continues on next page)

(continued from previous page)

```
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = breitungSORM( dim, g, dg, distObjs, corrMat )
```

[4]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.9519628114174661

Failure probability:
0.06374492261292942

Design point coordinate in U space:
[-0.6398897170780924, -0.7048222075811494]

Design point coordinate in X space:
[1.8005514146095387, 1.4758889620942535]
```

Example with automatic differentiation of LSF

[5]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: X[ 0 ] ** 4 + 2 * X[ 1 ] ** 4 - 20

# If dg is None, the internal automatic differentiation algorithm will be used
dg = None

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm( 5.0, 5.0 ), stats.norm( 5.0, 5.0 ) ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = breitungSORM( dim, g, dg, distObjs, corrMat )
```

[6]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
```

(continues on next page)

(continued from previous page)

```
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.9519628114174661

Failure probability:
0.0637449226098400 4

Design point coordinate in U space:
[-0.6398897170780924, -0.7048222075811494]

Design point coordinate in X space:
[1.8005514146095387, 1.4758889620942535]
```

Tvedt SORM

Tvedt further derived a three-term approximation by ignoring terms of orders higher than two,

$$T_1 = \Phi(-\beta) \prod_{i=1}^{n-1} (1 + k_i \beta)^{-1/2}$$

$$T_2 = [\beta\Phi(-\beta) - \phi(\beta)] \left[\prod_{i=1}^{n-1} (1 + k_i \beta)^{-1/2} - \prod_{i=1}^{n-1} (1 + k_i (\beta + 1))^{-1/2} \right]$$

$$T_3 = (\beta + 1) [\beta\Phi(-\beta) - \phi(\beta)] \left[\prod_{i=1}^{n-1} (1 + k_i \beta)^{-1/2} - \text{Re} \left[\prod_{i=1}^{n-1} (1 + k_i (\beta + 1))^{-1/2} \right] \right]$$

$$p_f = T_1 + T_2 + T_3$$

where $\Phi(\cdot)$ is the standard normal Cumulative Distribution Function (CDF); $\phi(\cdot)$ is the standard normal Probabilistic Distribution Function (PDF); β is the first order reliability index; k_i is the main curvatures of the limit-state function at design point; Re is the real part of a complex number. It can be found that the T_1 is the Breitung's equation. The T_2 and T_3 can be interpreted as the correctors to the Breitung's formula to increase the accuracy for moderate values of β.

References:

- Tvedt, L., 1990. Distribution of quadratic forms in normal space—application to structural reliability. Journal of engineering mechanics, 116(6), pp.1183-1197.
- Hu, Z., Mansour, R., Olsson, M. and Du, X., 2021. Second-order reliability methods: a review and comparative study. Structural and Multidisciplinary Optimization, 64(6), pp.3233-3263.
- Bourinet, J.M., 2018. Reliability analysis and optimal design under uncertainty-Focus on adaptive surrogate-based approaches (Doctoral dissertation, Université Clermont Auvergne).

Function help

[7]:
```
from ffpack.rrm import tvedtSORM
help( tvedtSORM )
```

```
Help on function tvedtSORM in module ffpack.rrm.secondOrderReliabilityMethod:

tvedtSORM(dim, g, dg, distObjs, corrMat, quadDeg=99, quadRange=8, dx=1e-06)
    Second order reliability method based on Tvedt algorithm.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    dg: array_like of function
        Gradient of the limit state function. It should be an array_like of function
        like dg = [ dg_dx1, dg_dx2, ... ]. To get the derivative of i-th random
        variable at ( x1*, x2*, ... ), dg[ i ]( x1*, x2*, ... ) will be called.
        dg can be None, see the following Notes.
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
        objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
    corrMat: 2d matrix
        Correlation matrix of the marginal distributions.
    quadDeg: integer
        Quadrature degree for Nataf transformation
    quadRange: scalar
        Quadrature range for Nataf transformation. The integral will be performed
        in the range [ -quadRange, quadRange ].
    dx : scalar, optional
        Spacing for auto differentiation. Not required if dg is provided.

    Returns
    -------
    beta: scalar
        Reliability index.
    pf: scalar
        Probability of failure.
    uCoord: 1d array
        Design point coordinate in U space.
    xCoord: 1d array
        Design point coordinate in X space.

    Raises
    ------
    ValueError
        If the dim is less than 1.
        If the dim does not match the disObjs and corrMat.
        If corrMat is not 2d matrix.
        If corrMat is not positive definite.
        If corrMat is not symmetric.
```

(continues on next page)

(continued from previous page)

```
        If corrMat diagonal is not 1.

    Notes
    -----
    If dg is None, the numerical differentiation will be used. The tolerance of the
    numerical differentiation can be changed in globalConfig.

    Examples
    --------
    >>> from ffpack.rrm import tvedtSORM
    >>> dim = 2
    >>> g = lambda X: -np.sum( X ) + 1
    >>> dg = [ lambda X: -1, lambda X: -1 ]
    >>> distObjs = [ stats.norm(), stats.norm() ]
    >>> corrMat = np.eye( dim )
    >>> beta, pf, uCoord, xCoord = tvedtSORM( dim, g, dg, distObjs, corrMat )
```

Example with explicit derivative of LSF

[8]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: X[ 0 ] ** 4 + 2 * X[ 1 ] ** 4 - 20

# Explicit derivative of LSF
# dg is a list in which each element is a partial derivative function of g w.r.t. X
# dg[0] = partial g / partial X[0]
# dg[1] = partial g / partial X[1]
dg = [ lambda X: 4 * X[ 0 ] ** 3, lambda X: 8 * X[ 1 ] ** 3 ]

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm( 5.0, 5.0 ), stats.norm( 5.0, 5.0 ) ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = tvedtSORM( dim, g, dg, distObjs, corrMat )
```

[9]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.9519628114174661

Failure probability:
0.03604492191838041

Design point coordinate in U space:
[-0.6398897170780924, -0.7048222075811494]

Design point coordinate in X space:
[1.8005514146095387, 1.4758889620942535]
```

Example with automatic differentiation of LSF

[10]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: X[ 0 ] ** 4 + 2 * X[ 1 ] ** 4 - 20

# If dg is None, the internal automatic differentiation algorithm will be used
dg = None

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm( 5.0, 5.0 ), stats.norm( 5.0, 5.0 ) ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = tvedtSORM( dim, g, dg, distObjs, corrMat )
```

[11]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.9519628114174661

Failure probability:
0.03604492191636101

Design point coordinate in U space:
[-0.6398897170780924, -0.7048222075811494]

Design point coordinate in X space:
```

(continues on next page)

(continued from previous page)

```
[1.8005514146095387, 1.4758889620942535]
```

Hohenbichler and Rackwitz SORM

Hohenbichler and Rackwitz derived a closed form expression based on a Taylor expansion,

$$p_f = \Phi(-\beta) \prod_{i=1}^{n-1} \left(1 + k_i \frac{\phi(\beta)}{\Phi(\beta)}\right)^{-1/2}$$

where $\Phi(\cdot)$ is the standard normal Cumulative Distribution Function (CDF); $\phi(\cdot)$ is the standard normal Probabilistic Distribution Function (PDF); β is the first order reliability index; k_i is the main curvatures of the limit-state function at design point; The Hohenbichler and Rackwitz's formula also aims at improving the reliability estimate for moderate β. It should be noted that the Hohenbichler and Rackwitz's formula is asymptotically equivalent to Breitung's formula for a large β.

References:

- Hohenbichler, M. and Rackwitz, R., 1988. Improvement of second-order reliability estimates by importance sampling. Journal of Engineering Mechanics, 114(12), pp.2195-2199.
- Hu, Z., Mansour, R., Olsson, M. and Du, X., 2021. Second-order reliability methods: a review and comparative study. Structural and Multidisciplinary Optimization, 64(6), pp.3233-3263.
- Bourinet, J.M., 2018. Reliability analysis and optimal design under uncertainty-Focus on adaptive surrogate-based approaches (Doctoral dissertation, Université Clermont Auvergne).

Function help

[12]:
```
from ffpack.rrm import hrackSORM
help( hrackSORM )
```

```
Help on function hrackSORM in module ffpack.rrm.secondOrderReliabilityMethod:

hrackSORM(dim, g, dg, distObjs, corrMat, quadDeg=99, quadRange=8, dx=1e-06)
    Second order reliability method based on Hohenbichler and Rackwitz algorithm.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    dg: array_like of function
        Gradient of the limit state function. It should be an array_like of function
        like dg = [ dg_dx1, dg_dx2, ... ]. To get the derivative of i-th random
        variable at ( x1*, x2*, ... ), dg[ i ]( x1*, x2*, ... ) will be called.
        dg can be None, see the following Notes.
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
        objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
    corrMat: 2d matrix
        Correlation matrix of the marginal distributions.
```

(continues on next page)

(continued from previous page)

```
    quadDeg: integer
        Quadrature degree for Nataf transformation
    quadRange: scalar
        Quadrature range for Nataf transformation. The integral will be performed
        in the range [ -quadRange, quadRange ].
    dx : scalar, optional
        Spacing for auto differentiation. Not required if dg is provided.

    Returns
    -------
    beta: scalar
        Reliability index.
    pf: scalar
        Probability of failure.
    uCoord: 1d array
        Design point coordinate in U space.
    xCoord: 1d array
        Design point coordinate in X space.

    Raises
    ------
    ValueError
        If the dim is less than 1.
        If the dim does not match the disObjs and corrMat.
        If corrMat is not 2d matrix.
        If corrMat is not positive definite.
        If corrMat is not symmetric.
        If corrMat diagonal is not 1.

    Notes
    -----
    If dg is None, the numerical differentiation will be used. The tolerance of the
    numerical differentiation can be changed in globalConfig.

    Examples
    --------
    >>> from ffpack.rrm import tvedtSORM
    >>> dim = 2
    >>> g = lambda X: -np.sum( X ) + 1
    >>> dg = [ lambda X: -1, lambda X: -1 ]
    >>> distObjs = [ stats.norm(), stats.norm() ]
    >>> corrMat = np.eye( dim )
    >>> beta, pf, uCoord, xCoord = hrackSORM( dim, g, dg, distObjs, corrMat )
```

Example with explicit derivative of LSF

[13]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: X[ 0 ] ** 4 + 2 * X[ 1 ] ** 4 - 20

# Explicit derivative of LSF
# dg is a list in which each element is a partial derivative function of g w.r.t. X
# dg[0] = partial g / partial X[0]
# dg[1] = partial g / partial X[1]
dg = [ lambda X: 4 * X[ 0 ] ** 3, lambda X: 8 * X[ 1 ] ** 3 ]

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm( 5.0, 5.0 ), stats.norm( 5.0, 5.0 ) ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = hrackSORM( dim, g, dg, distObjs, corrMat )
```

[14]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.9519628114174661

Failure probability:
0.09883455895117937

Design point coordinate in U space:
[-0.6398897170780924, -0.7048222075811494]

Design point coordinate in X space:
[1.8005514146095387, 1.4758889620942535]
```

Example with automatic differentiation of LSF

[15]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: X[ 0 ] ** 4 + 2 * X[ 1 ] ** 4 - 20

# If dg is None, the internal automatic differentiation algorithm will be used
dg = None

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm( 5.0, 5.0 ), stats.norm( 5.0, 5.0 ) ]
corrMat = np.eye( dim )

beta, pf, uCoord, xCoord = hrackSORM( dim, g, dg, distObjs, corrMat )
```

[16]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
0.9519628114174661

Failure probability:
0.09883455894748124

Design point coordinate in U space:
[-0.6398897170780924, -0.7048222075811494]

Design point coordinate in X space:
[1.8005514146095387, 1.4758889620942535]
```

1.8.4 Simulation based reliability method

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
```

(continues on next page)

(continued from previous page)

```
plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Subset simulation

The subset simulation proposed by Au and Beck is used to determine the small failure probability in high dimensional space. The basic idea for subset simulation is to express the failure probability as a series of intermediate probability,

$$p_f = P(E) = P(E_m) = P(E_m|E_{m-1})P(E_{m-1}|E_{m-2})\dots P(E_1)$$

where $P(E_i)$ denotes the probability of event E_i; $P(E_{m-1}|E_{m-2})$ denotes the conditional probability of event E_{m-1} given E_{m-2}.

The intermediate event E_{m-1} can be considered the p_0-quantile of the previous event E_{m-2}, where p_0 is a predefined probability level. Usually, the optimal value for the probability level is $p_0 = 0.1 - 0.3$. After the intermediate event E_{m-1} is obtained by the p_0-quantile of the previous event E_{m-2}, it can be consider as the initial sampling points for MCMC chains to determine the next intermediate event E_m.

Function `subsetSimulation` implements the subset simulation algorithm.

Reference:

- Au, S.K. and Beck, J.L., 2001. Estimation of small failure probabilities in high dimensions by subset simulation. Probabilistic engineering mechanics, 16(4), pp.263-277.

Function help

[2]:
```
from ffpack.rrm import subsetSimulation
help( subsetSimulation )
```

```
Help on function subsetSimulation in module ffpack.rrm.simulationBasedReliabilityMethod:

subsetSimulation(dim, g, distObjs, corrMat, numSamples, maxSubsets, probLevel=0.1,␣
↪quadDeg=99, quadRange=8, randomSeed=None)
    Second order reliability method based on Breitung algorithm.

    Parameters
    ----------
    dim: integer
        Space dimension ( number of random variables ).
    g: function
        Limit state function. It will be called like g( [ x1, x2, ... ] ).
    distObjs: array_like of distributions
        Marginal distribution objects. It should be the freezed distribution
        objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
    corrMat: 2d matrix
        Correlation matrix of the marginal distributions.
    numSamples: integer
        Number of samples in each subset.
    maxSubsets: scalar
```

(continues on next page)

(continued from previous page)

```
    Maximum number of subsets used to compute the failure probability.
probLevel: scalar, optional
    Probability level for intermediate subsets.
quadDeg: integer, optional
    Quadrature degree for Nataf transformation
quadRange: scalar, optional
    Quadrature range for Nataf transformation. The integral will be performed
    in the range [ -quadRange, quadRange ].
randomSeed: integer, optional
    Random seed. If randomSeed is none or is not an integer, the random seed in
    global config will be used.

Returns
-------
pf: scalar
    Probability of failure.
allLsfValue: array_like
    Values of limit state function in each subset.
allUSamples: array_like
    Samples of U space in each subset.
allXSamples: array_like
    Samples of X space in each subset.

Raises
------
ValueError
    If the dim is less than 1.
    If the dim does not match the disObjs and corrMat.
    If corrMat is not 2d matrix.
    If corrMat is not positive definite.
    If corrMat is not symmetric.
    If corrMat diagonal is not 1.

Notes
-----
Nataf transformation is used for the marginal distributions.

Examples
--------
>>> from ffpack.rrm import subsetSimulation
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> numSamples, maxSubsets = 500, 10
>>> pf = subsetSimulation( dim, g, distObjs, corrMat, numSamples, maxSubsets )
```

Example with linear LSF

[3]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: -np.sum( X ) / np.sqrt( dim ) + 3.0

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm(), stats.norm() ]
corrMat = np.eye( dim )
numSamples, maxSubsets = 3000, 10

pf, allLsfValues, allUSamples, allXSamples = \
        subsetSimulation( dim, g, distObjs, corrMat, numSamples,
                          maxSubsets, randomSeed=2023 )
```

[4]:
```
print( "Failure probability: " )
print( pf )
```

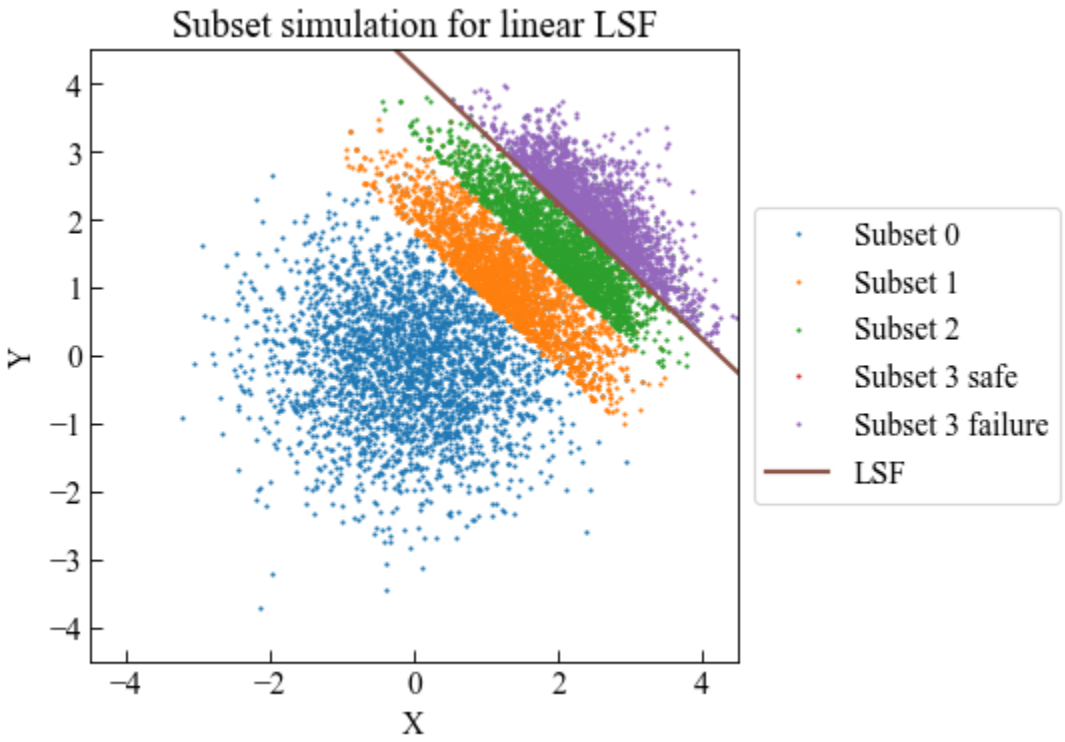
```
Failure probability:
0.00097066666666666669
```

[5]:
```
fig, ax = plt.subplots( figsize=( 7, 5 ) )

# Plot samples in each subset
for idx, uSamples in enumerate( allUSamples ):
    if idx != len( allUSamples ) - 1:
        ax.plot( uSamples[ :, 0], uSamples[ :, 1 ],
                 ".", markersize=2, label='Subset %d' % idx )
    else:
        # For the last subset, we plot the non-failure points and failure points
        fidx = np.searchsorted( allLsfValues[ idx ], 0.0, side='right' )
        ax.plot( uSamples[ fidx: , 0], uSamples[ fidx: , 1 ],
                 ".", markersize=2, label='Subset %d safe' % idx )
        ax.plot( uSamples[ : fidx, 0], uSamples[ : fidx, 1 ],
                 ".", markersize=2, label='Subset %d failure' % idx )

# Plot limit state function
gx = np.linspace( -4.5, 4.5, 200)
gy = [ 3 * np.sqrt( dim ) - x for x in gx ]
ax.plot( gx, gy, linewidth=2, label='LSF' )

ax.tick_params( axis='x', direction="in", length=5 )
ax.tick_params( axis='y', direction="in", length=5 )
ax.set_ylabel( "Y" )
ax.set_xlabel( "X" )
ax.set_title( "Subset simulation for linear LSF" )
ax.set_xlim( [ -4.5, 4.5 ] )
ax.set_ylim( [ -4.5, 4.5 ] )
ax.legend( loc='center left', bbox_to_anchor= (1, 0.5 ) )

plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



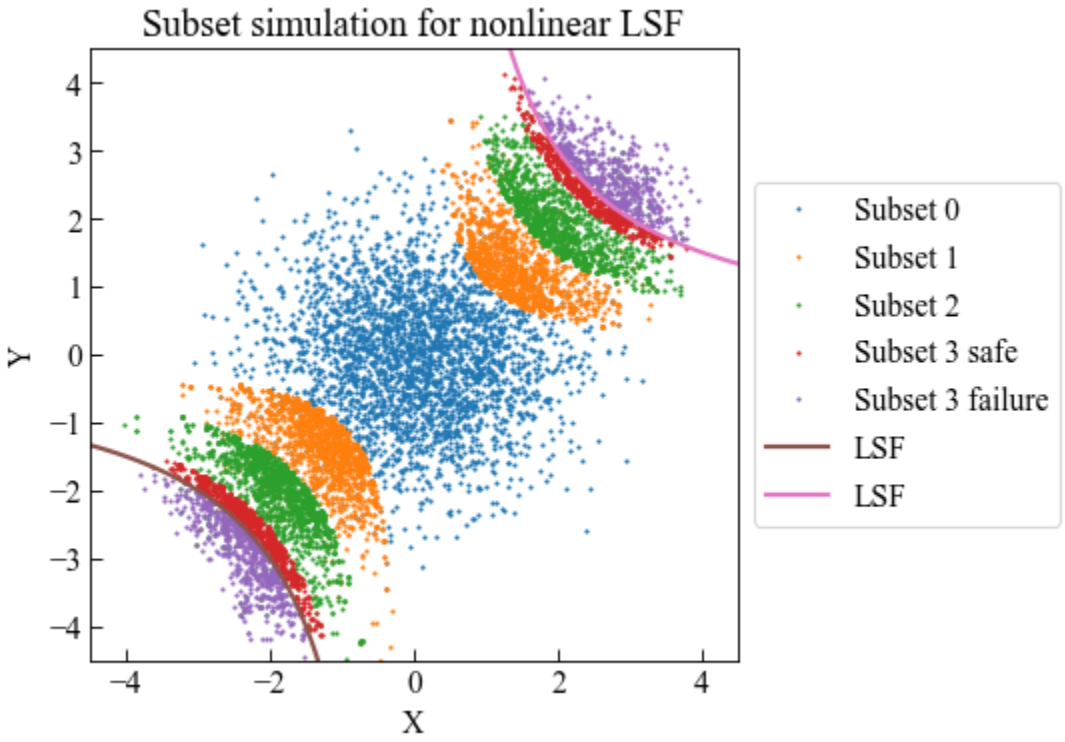
Example with nonlinear LSF

[6]:
```
# Define the dimension for the FORM problem
dim = 2

# Define the limit state function (LSF) g
g = lambda X: -( X[0] * X[1] ) + 6

# Marginal distributions and correlation Matrix of the random variables
distObjs = [ stats.norm(), stats.norm() ]
corrMat = np.eye( dim )
numSamples, maxSubsets = 3000, 10

pf, allLsfValues, allUSamples, allXSamples = \
        subsetSimulation( dim, g, distObjs, corrMat, numSamples,
                          maxSubsets, randomSeed=2023 )
```

[7]:
```
print( "Failure probability: " )
print( pf )
```

```
Failure probability:
0.0004250000000000001
```

[8]:
```
fig, ax = plt.subplots( figsize=( 7, 5 ) )

# Plot samples in each subset
for idx, uSamples in enumerate( allUSamples ):
    if idx != len( allUSamples ) - 1:
        ax.plot( uSamples[ :, 0], uSamples[ :, 1 ],
                 ".", markersize=2, label='Subset %d' % idx )
    else:
        # For the last subset, we plot the non-failure points and failure points
        fidx = np.searchsorted( allLsfValues[ idx ], 0.0, side='right' )
        ax.plot( uSamples[ fidx: , 0], uSamples[ fidx: , 1 ],
                 ".", markersize=2, label='Subset %d safe' % idx )
        ax.plot( uSamples[ : fidx, 0], uSamples[ : fidx, 1 ],
                 ".", markersize=2, label='Subset %d failure' % idx )

# Plot limit state function
gx = np.linspace( -4.5, 4.5, 200)
gy = [ 6.0 / x for x in gx ]
ax.plot( gx[ : 100], gy[ : 100], linewidth=2, label='LSF' )
ax.plot( gx[ 100: ], gy[ 100: ], linewidth=2, label='LSF' )

ax.tick_params( axis='x', direction="in", length=5 )
ax.tick_params( axis='y', direction="in", length=5 )
ax.set_ylabel( "Y" )
ax.set_xlabel( "X" )
ax.set_title( "Subset simulation for nonlinear LSF" )
ax.set_xlim( [ -4.5, 4.5 ] )
ax.set_ylim( [ -4.5, 4.5 ] )
ax.legend( loc='center left', bbox_to_anchor= (1, 0.5 ) )

plt.tight_layout()
plt.show()
```



1.9 Utility methods (utils)

1.9.1 Aggregation

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
plt.style.use( "default" )

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Cycle counting aggregation

After the cycle counting, we usually obtain a 2D matrix with the format: `[ [ value, count ],[ value, count ], ... ]`. However, the counting results are very noisy if no preprocessing procedure is applied to the load sequence. For example, we might count two cycles with a range of 0.96, a half cycle with a range of 0.98, etc. Then, the resutls will be `[ [ 0.96, 2 ],[ 0.98, 0.5 ], ... ]`. We expected the counting results to be aggregated as `[ [ 1, 2.5 ], ... ]`. This function can aggregate based on the bin size and generate the cleaned counting results.

Function help

[2]:
```
from ffpack.utils import cycleCountingAggregation
help( cycleCountingAggregation )
```

```
Help on function cycleCountingAggregation in module ffpack.utils.lccUtils:

cycleCountingAggregation(data, binSize=1.0)
    Count the number of occurrences of each cycle digitized to the nearest bin.

    Parameters
    ----------
    data: 2d array
        Input cycle counting data [ [ value, count ], ... ] for bin collection

    binSize: scalar, optional
        bin size is the difference between each level,
        for example, binSize=1.0, the levels will be 0.0, 1.0, 2.0, 3.0 ...

    Returns
    -------
    rst: 2d array
        Aggregated [ [ aggregatedValue, count ] ] by the binSize


    Raises
    ------
    ValueError
        If the data dimension is not 2.
        If the data is empty

    Notes
    -----
    When a value is in the middle, it will be counted downward
    for example, 0.5 when binSize=1.0, the count will be counted to 0.0

    Examples
    --------
    >>> from ffpack.utils import cycleCountingAggregation
    >>> data = [ [ 1.7, 2.0 ], [ 2.2, 2.0 ] ]
    >>> rst = cycleCountingAggregation( data )
```

Example with default values

[3]:
```
ccaLccData = [ [ 1.7, 2.0 ], [ 2.2, 2.0 ] ]
ccaResults = cycleCountingAggregation( ccaLccData )
```

[4]:
```
print( ccaResults )
```

```
[[2.0, 4.0]]
```

1.9.2 Counting matrix

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
plt.style.use( "default" )

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Counting results to counting matrix

Function `countingRstToCountingMatrix` returns the cycle counting matrix and corresponding index from the cycle counting results.

Function help

[2]:
```
from ffpack.utils import countingRstToCountingMatrix
help( countingRstToCountingMatrix )
```

```
Help on function countingRstToCountingMatrix in module ffpack.utils.lsmUtils:

countingRstToCountingMatrix(countingRst)
    Calculate counting matrix from rainflow counting result.

    Parameters
    ----------
    countingRst: 2d array
        Cycle counting result in form of [ [ rangeStart1, rangeEnd1, count1 ],
        [ rangeStart2, rangeEnd2, count2 ], ... ].

    Returns
    -------
    rst: 2d array
        A matrix contains the counting results.
```

(continues on next page)

(continued from previous page)

```
    matrixIndexKey: 1d array
        A sorted array contains the index keys for the counting matrix.

    Raises
    ------
    ValueError
        If the data dimension is not 2.
        If the data is not empty and not in dimension of n by 3.

    Examples
    --------
    >>> from ffpack.lsm import countingRstToCountingMatrix
    >>> countingRst = [ [ -2.0, 1.0, 1.0 ], [ 5.0, -1.0, 3.0 ], [ -4.0, 4.0, 0.5 ] ]
    >>> rst, matrixIndexKey = countingRstToCountingMatrix( countingRst )
```

Example with default values

[3]:
```
cctcmRst = [ [ -2.0, 1.0, 1.0 ], [ 5.0, -1.0, 3.0 ], [ -4.0, 4.0, 0.5 ] ]
cctcmMat, cctcmMatIndexKey = countingRstToCountingMatrix( cctcmRst )

cctcmMat = np.array( cctcmMat )
cctcmMatIndex = np.array( cctcmMatIndexKey ).astype( float )
```

[4]:
```
print( "Cycle counting matrix" )
print( cctcmMat )
print()
print( "Matrix index" )
print( cctcmMatIndex )
```

```
Cycle counting matrix
[[0.  0.  0.  0.  0.5 0. ]
 [0.  0.  0.  1.  0.  0. ]
 [0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0. ]
 [0.  0.  3.  0.  0.  0. ]]

Matrix index
[-4. -2. -1.  1.  4.  5.]
```

1.9.3 Degitization

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
plt.style.use( "default" )

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Sequence digitization

Function `sequenceDigitization` can digitize a sequence with specific resolution.

Function help

[2]:
```
from ffpack.utils import sequenceDigitization
help( sequenceDigitization )
```

```
Help on function sequenceDigitization in module ffpack.utils.generalUtils:

sequenceDigitization(data, resolution=1.0)
    Digitize the sequence data to a specific resolution

    The sequence data are digitized by the round method.

    Parameters
    ----------
    data: 1d array
        Sequence data to digitize.

    resolution: bool, optional
        The desired resolution to round the data points.

    Returns
    -------
    rst: 1d array
        A list contains the digitized data.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2 with keedEnds == False
        If the data length is less than 3 with keedEnds == True
```

(continues on next page)

(continued from previous page)

```
    Notes
    -----
    The default round function will round half to even: 1.5, 2.5 => 2.0:

    Examples
    --------
    >>> from ffpack.utils import sequenceDigitization
    >>> data = [ -1.0, 2.3, 1.8, 0.6, -0.4, 0.8, -1.6, -2.5, 3.4, 0.3, 0.1 ]
    >>> rst = sequenceDigitization( data )
```

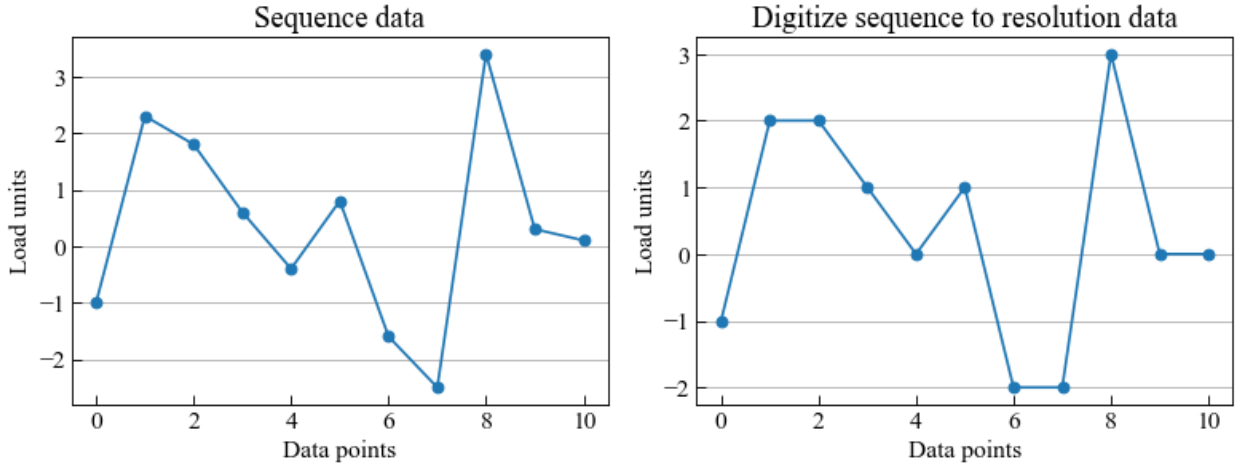
Example with default values

[3]:
```
dstrSequenceData = [ -1.0, 2.3, 1.8, 0.6, -0.4, 0.8, -1.6, -2.5, 3.4, 0.3, 0.1 ]

dstrResults = sequenceDigitization( dstrSequenceData, resolution=1.0 )
```

[4]:
```
print( dstrResults )
```

```
[-1.0, 2.0, 2.0, 1.0, -0.0, 1.0, -2.0, -2.0, 3.0, 0.0, 0.0]
```

[5]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( dstrSequenceData, 'o-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )
ax1.grid( axis='y', color="0.7" )

ax2.plot( dstrResults, 'o-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Load units" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Digitize sequence to resolution data" )
ax2.grid( axis='y', color="0.7" )

plt.tight_layout()
plt.show()
```



1.9.4 Fitter

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
plt.style.use( "default" )

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

SN curve fitter

Class `SnCurveFitter` creates a fitter for the SN curve.

This fitter (not fitting) has a query function `getN` and represents the SN curve itself. For the given experimental data, the fitter assumes the linear relation between the `S` and `log(N)` if `S` is larger than the fatigue limit. Therefore, if a given `S` is less than or equal fatigue limit, the `getN` will return `-1` representing no fatigue damage is caused by the current `S`. Otherwise, `getN` will return the estimated `N` value representing the fatigue failure cycles under the current `S`.

Class initialization help

[2]:
```
from ffpack.utils import SnCurveFitter
help( SnCurveFitter.__init__ )
```

```
Help on function __init__ in module ffpack.utils.fdrUtils:

__init__(self, data, fatigueLimit)
    Initialize a fitter for a SN curve based on the experimental data.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    data: 2d array
        Experimental data for fitting in a 2D matrix,
        e.g., [ [ N1, S1 ], [ N2, S2 ], ..., [ Ni, Si ] ]

    fatigueLimit: scalar
        Fatigue limit indicating the minimum S that can cause fatigue.

    Raises
    ------
    ValueError
        If the data dimension is not 2.
        If the data length is less than 2.
        If the fatigueLimit is less than or equal 0.
        If N_i or S_i is less than or equal 0.

    Examples
    --------
    >>> from ffpack.utils import SnCurveFitter
    >>> data = [ [ 10, 3 ], [ 1000, 1 ] ]
    >>> fatigueLimit = 0.5
    >>> snCurveFitter = SnCurveFitter( data, fatigueLimit )
```

Function `getN` help

[3]:
```
help( SnCurveFitter.getN )
```

```
Help on function getN in module ffpack.utils.fdrUtils:

getN(self, S)
    Query fatigue life N for a given S

    Parameters
    ----------
    S: scalar
        Input S for fatigue life query.

    Returns
    -------
    rst: scalar
        Fatigue life under the query S.
        If S is less than or equal fatigueLimit, -1 will be returned.

    Raises
    ------
    ValueError
        If the S is less than or equal 0.
```

(continues on next page)

(continued from previous page)

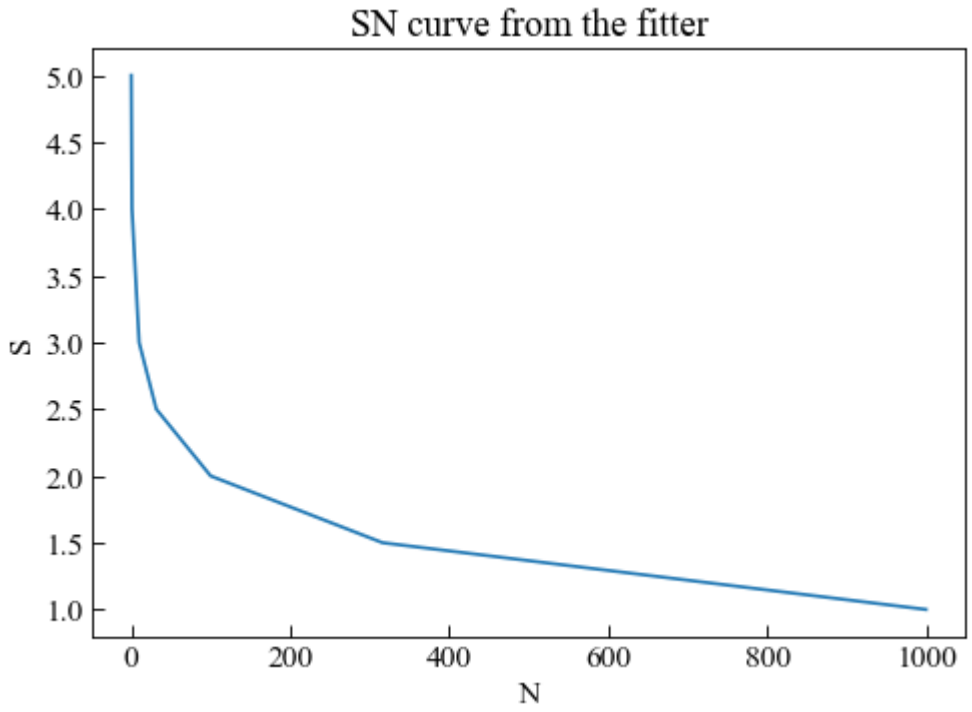
```
    Examples
    --------
    >>> rst = snCurveFitter.getN( 2 )
```

Example with default values

[4]:
```
data = [ [ 10, 3 ], [ 1000, 1 ] ]
fatigueLimit = 0.5
snCurveFitter = SnCurveFitter( data, fatigueLimit )

queryS = [ 0.2, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0 ]
calN = [ snCurveFitter.getN( s ) for s in queryS ]

for index, s in enumerate( queryS ):
    print( "Fatigue failure cycles at S == %s: " % s, "{:.2f}".format( calN[ index ] ) )

plotStartIndex = next( x[ 0 ] for x in enumerate( queryS ) if x[ 1 ] > fatigueLimit )
```

```
Fatigue failure cycles at S == 0.2:  -1.00
Fatigue failure cycles at S == 0.5:  -1.00
Fatigue failure cycles at S == 1.0:  1000.00
Fatigue failure cycles at S == 1.5:  316.23
Fatigue failure cycles at S == 2.0:  100.00
Fatigue failure cycles at S == 2.5:  31.62
Fatigue failure cycles at S == 3.0:  10.00
Fatigue failure cycles at S == 4.0:  1.00
Fatigue failure cycles at S == 5.0:  0.10
```

[5]:
```
fig, ax = plt.subplots()

ax.plot( np.array( calN )[ plotStartIndex: ],
         np.array( queryS )[ plotStartIndex: ] )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "S" )
ax.set_xlabel( "N" )
ax.set_title( "SN curve from the fitter" )

plt.tight_layout()
plt.show()
```



1.9.5 Sequence filters

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]
plt.style.use( "default" )

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

Sequence peakValley filter

Function `sequencePeakValleyFilter` returns the peaks and valleys of a sequence.

Function help

[2]:
```
from ffpack.utils import sequencePeakValleyFilter
help( sequencePeakValleyFilter )
```

```
Help on function sequencePeakValleyFilter in module ffpack.utils.generalUtils:

sequencePeakValleyFilter(data, keepEnds=False)
    Remove the intermediate value and only get the peaks and valleys of the data

    The peak and valley refer the data points that are EXACTLY above and below
    the neighbors, not equal.

    Parameters
    ----------
    data: 1darray
        Sequence data to get peaks and valleys.
    keepEnds: bool, optional
        If two ends of the original data should be preserved.

    Returns
    -------
    rst: 1darray
        A list contains the peaks and valleys of the data.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2 with keedEnds == False.
        If the data length is less than 3 with keedEnds == True.

    Examples
    --------
    >>> from ffpack.utils import sequencePeakValleyFilter
    >>> data = [ -0.5, 1.0, -2.0, 3.0, -1.0, 4.5, -2.5, 3.5, -1.5, 1.0 ]
    >>> rst = sequencePeakValleyFilter( data )
```

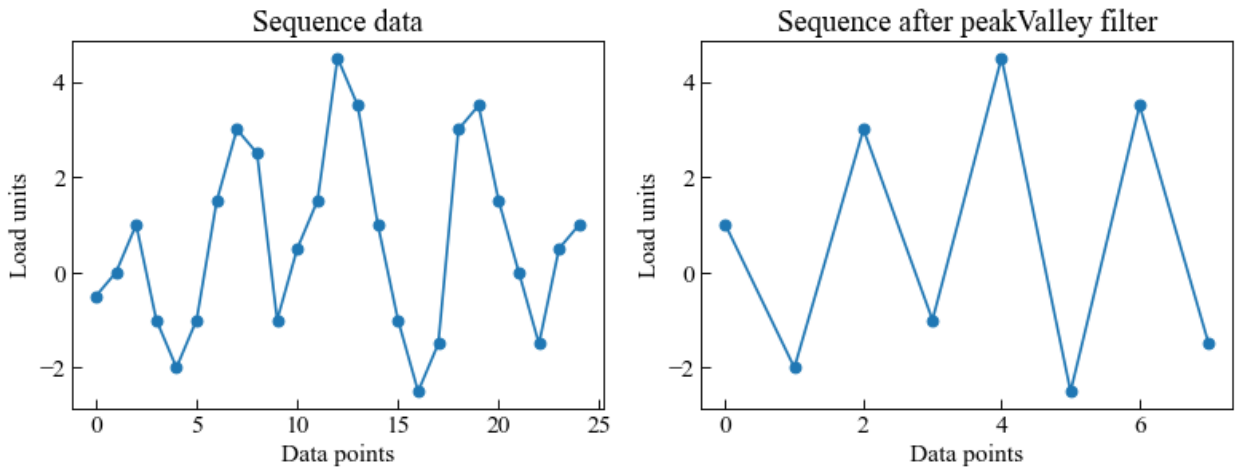
Example with default values

[3]:
```
gspvSequenceData = [ -0.5, 0.0, 1.0, -1.0, -2.0, -1.0, 1.5, 3.0, 2.5, -1.0, 0.5, 1.5, 4.
↪5,
                     3.5, 1.0, -1.0, -2.5, -1.5, 3.0, 3.5, 1.5, 0.0, -1.5, 0.5, 1.0 ]

gspvResults = sequencePeakValleyFilter( gspvSequenceData, keepEnds=False )
```

[4]:
```
print( gspvResults )
```

```
[1.0, -2.0, 3.0, -1.0, 4.5, -2.5, 3.5, -1.5]
```

[5]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( gspvSequenceData, 'o-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.plot( gspvResults, 'o-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Load units" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Sequence after peakValley filter" )

plt.tight_layout()
plt.show()
```



Example with keep end points

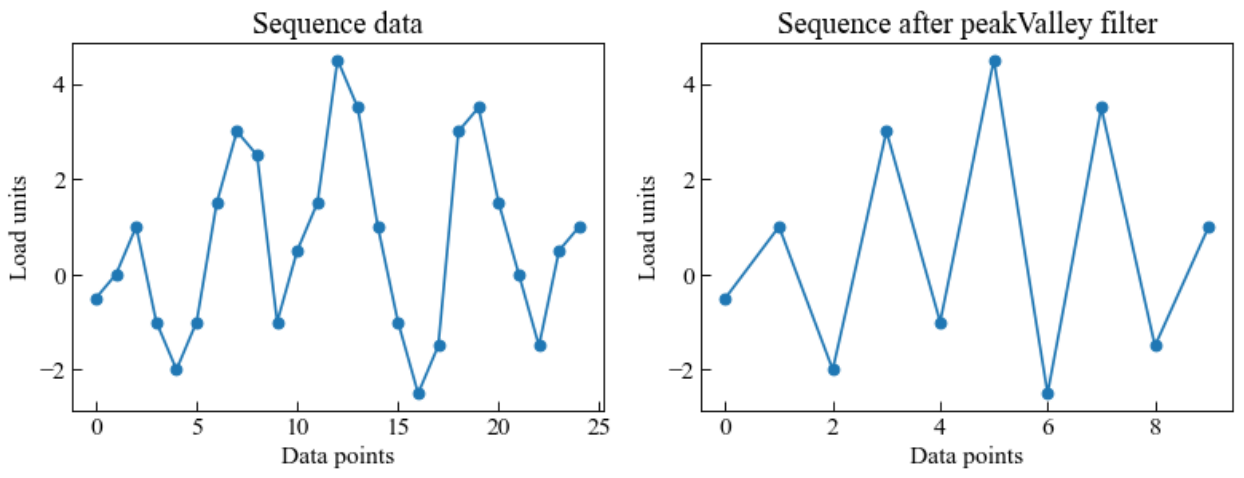
[6]:
```
from ffpack.utils import sequencePeakValleyFilter
```

[7]:
```
gspvSequenceData = [ -0.5, 0.0, 1.0, -1.0, -2.0, -1.0, 1.5, 3.0, 2.5, -1.0, 0.5, 1.5, 4.
↪5,
                     3.5, 1.0, -1.0, -2.5, -1.5, 3.0, 3.5, 1.5, 0.0, -1.5, 0.5, 1.0 ]

gspvResults = sequencePeakValleyFilter( gspvSequenceData, keepEnds=True )
```

[8]:
```
print( gspvResults )
```

```
[-0.5, 1.0, -2.0, 3.0, -1.0, 4.5, -2.5, 3.5, -1.5, 1.0]
```

[9]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( gspvSequenceData, 'o-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.plot( gspvResults, 'o-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Load units" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Sequence after peakValley filter" )

plt.tight_layout()
plt.show()
```



Sequence hysteresis filter

Function `sequenceHysteresisFilter` returns the sequence after hysteresis filtering.

Function help

[10]:
```
from ffpack.utils import sequenceHysteresisFilter
help( sequenceHysteresisFilter )
```

```
Help on function sequenceHysteresisFilter in module ffpack.utils.generalUtils:

sequenceHysteresisFilter(data, gateSize)
    Filter data within the gateSize.

    Any cycle that has an amplitude smaller than the gate is removed from the data.
    This is done by scan the data, i.e., point i, to check if the next points,
    i.e., i + 1, i + 2, ... are within the gate from point i.

    Parameters
    ----------
    data: 1darray
        Sequence data to get peaks and valleys.
    gateSize: scalar
        Gate size to filter the data.

    Returns
    -------
    rst: 1darray
        A list contains the filtered data.

    Raises
    ------
    ValueError
        If the data dimension is not 1.
        If the data length is less than 2.
        If gateSize is not a scalar or not positive.

    Examples
    --------
    >>> from ffpack.utils import sequenceHysteresisFilter
    >>> data = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 2 ]
    >>> gateSize = 3.0
    >>> rst = sequenceHysteresisFilter( data, gateSize )
```

Example with default values

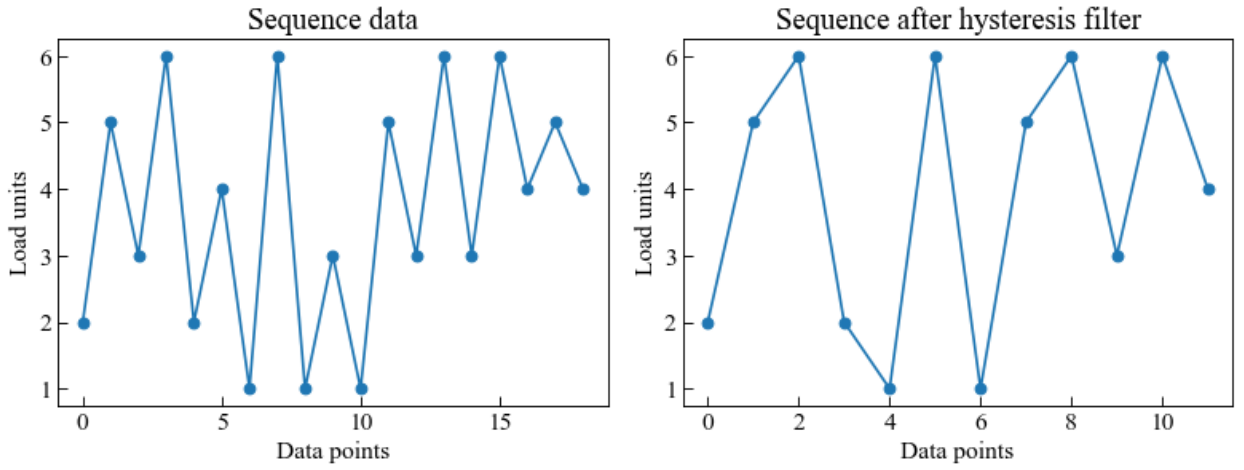
[11]:
```
hfSequenceData = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 4 ]
```

[12]:
```
gateSize = 3
hfResults = sequenceHysteresisFilter( hfSequenceData, gateSize )
```

[13]:
```
print( hfResults )
```

```
[2.0, 5.0, 6.0, 2.0, 1.0, 6.0, 1.0, 5.0, 6.0, 3.0, 6.0, 4.0]
```

[14]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( hfSequenceData, 'o-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.plot( hfResults, 'o-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Load units" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Sequence after hysteresis filter" )

plt.tight_layout()
plt.show()
```



[15]:
```
hfSequenceData = [ -0.5, 0.5, 0.2, 0.8, 1.0, -1.0, -2.0, -1.0, 0.8, 0.0, 1.5, 3.0, 2.5,
                   2.0, 2.5, 1.5, -1.0, 0.5, 1.2, 0.8, 1.5, 2.5, 2.0, 3.5, 3.0, 4.5, 3.5,
                   4.0, 4.2, 3.3, 2.8, 3.0, 2.5, 3.0, 2.0, 1.0, 0.0, 0.5, -1.0, -2.0,
                   -1.5, -2.0, -2.5, -1.5, -2.0, 0.0, 1.0, 1.2, 0.5, 2.0, 3.0, 4.0, 3.5,
```

(continues on next page)

(continued from previous page)

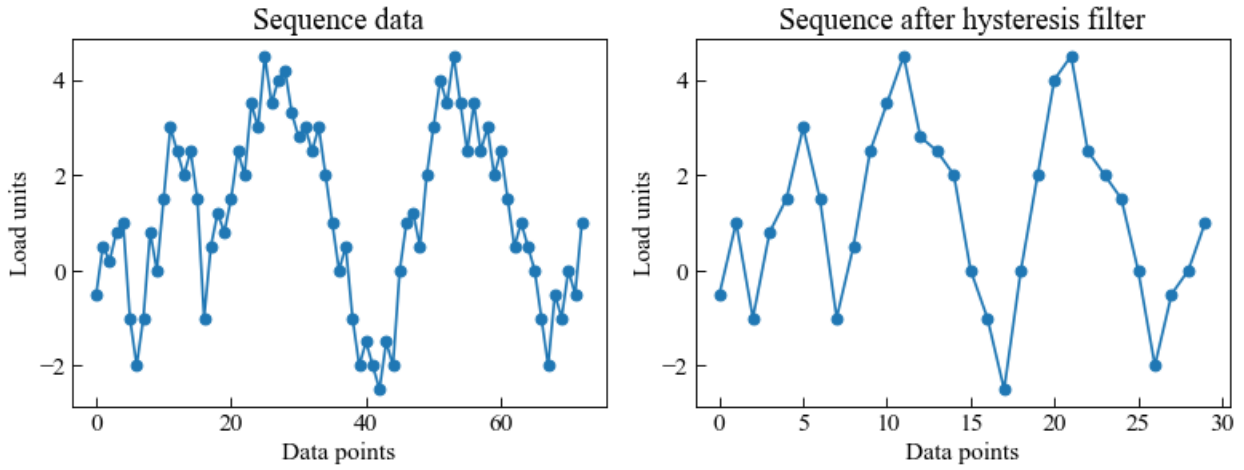
```
                    4.5, 3.5, 2.5, 3.5, 2.5, 3.0, 2.0, 2.5, 1.5, 0.5, 1.0, 0.5, 0.0, -1.0,
                    -0.5 -1.5, -0.5, -1.0, 0.0, -0.5, 1.0 ]
```

[16]:
```
gateSize = 1.5
hfResults = sequenceHysteresisFilter( hfSequenceData, gateSize )
```

[17]:
```
print( hfResults )
```

```
[-0.5, 1.0, -1.0, 0.8, 1.5, 3.0, 1.5, -1.0, 0.5, 2.5, 3.5, 4.5, 2.8, 2.5, 2.0, 0.0, -1.0,
↪ -2.5, 0.0, 2.0, 4.0, 4.5, 2.5, 2.0, 1.5, 0.0, -2.0, -0.5, 0.0, 1.0]
```

[18]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( hfSequenceData, 'o-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Load units" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data" )

ax2.plot( hfResults, 'o-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Load units" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Sequence after hysteresis filter" )

plt.tight_layout()
plt.show()
```



1.10 Fatigue damage application

1.10.1 Fagitue damage application

This is an application of fatigue damage calculation with generated load sequence for a 300CVM steel.

Material properties: yield stress $\sigma_y = 2098$ MPa and ultimate stress $\sigma_u = 2590$ MPa.

Number of cycles to failure at different stress amplitudes is given below.

Stress amplitude (MPa)	Fatigue life
2086	891
2000	1,160
1655	3,809
1103	48,645
965	112,573
900	174,400
827	296,400

Reference:

- Manson, S.S., Freche, J.C. and Ensign, C.R., 1967. Application of a double linear damage rule to cumulative fatigue (Vol. 3839). National Aeronautics and Space Administration.

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

[2]:
```
# Set random seed for repeatable results
from ffpack.config import globalConfig

globalConfig.setSeed( 2023 )
```

Material properties

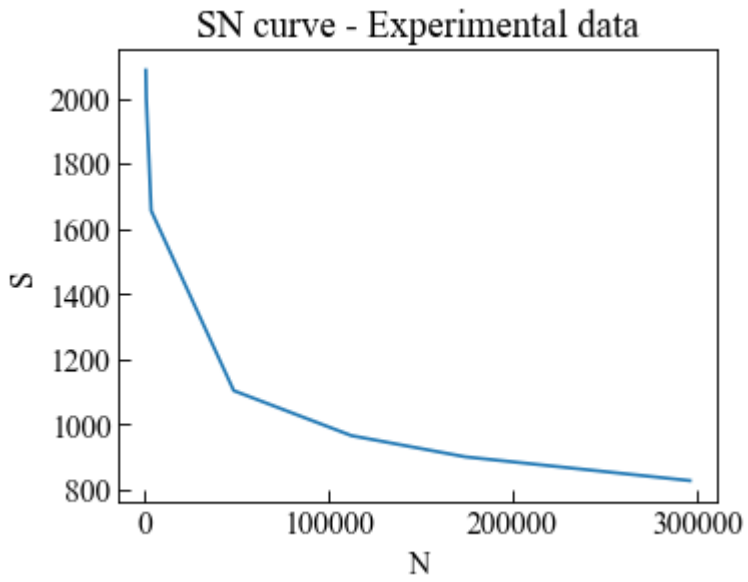
[3]:
```
ultimateStrength = 2590

snData = [ [ 891, 2086 ], [ 1160, 2000 ], [ 3809, 1655 ], [ 48645, 1103 ],
           [ 112573, 965 ], [ 174400, 900 ], [ 296400, 827 ] ]
```

[4]:
```
fig, ax = plt.subplots()

ax.plot( np.array( snData )[:,0], np.array( snData )[:,1] )
```

(continues on next page)

(continued from previous page)

```
ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "S" )
ax.set_xlabel( "N" )
ax.set_title( "SN curve - Experimental data" )

plt.tight_layout()
plt.show()
```



Generate load sequence data

The load sequence data is generated with a second order ARMA model. The initial observation values, the coefficients for autoregressive and moving average, the normal distribution coefficients for white noise are assumed.

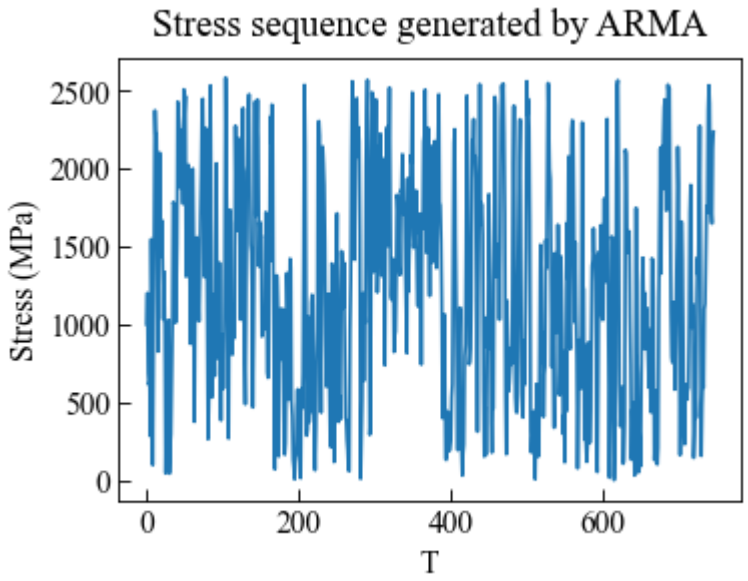
[5]:
```
# second order ARMA model
from ffpack.lsg import armaNormal

numStep = 1000
obs = [ 1000, 1200 ]
phis = [ 0.4, 0.2 ]
thetas = [ 0.8, 0.5 ]
mu = 200
sigma = 500
arman2ndResults = armaNormal( numStep, obs, phis, thetas, mu, sigma )
```

[6]:
```
# filer stresses greater than ultimate stress and those below 0 for processing
arman2ndResults = [x for x in arman2ndResults if x < 2590 and x > 0 ]
```

[7]:
```
len(arman2ndResults)
```

[7]:
```
748
```

[8]:
```
fig, ax = plt.subplots()

ax.plot( np.array( arman2ndResults ) )

ax.tick_params(axis='x', direction="in", length=5)
ax.tick_params(axis='y', direction="in", length=5)
ax.set_ylabel( "Stress (MPa)" )
ax.set_xlabel( "T" )
ax.set_title( "Stress sequence generated by ARMA", pad=10 )

plt.tight_layout()
plt.show()
```



Digitize sequence

The generated sequence is digitized with a resolution of 100 MPa.

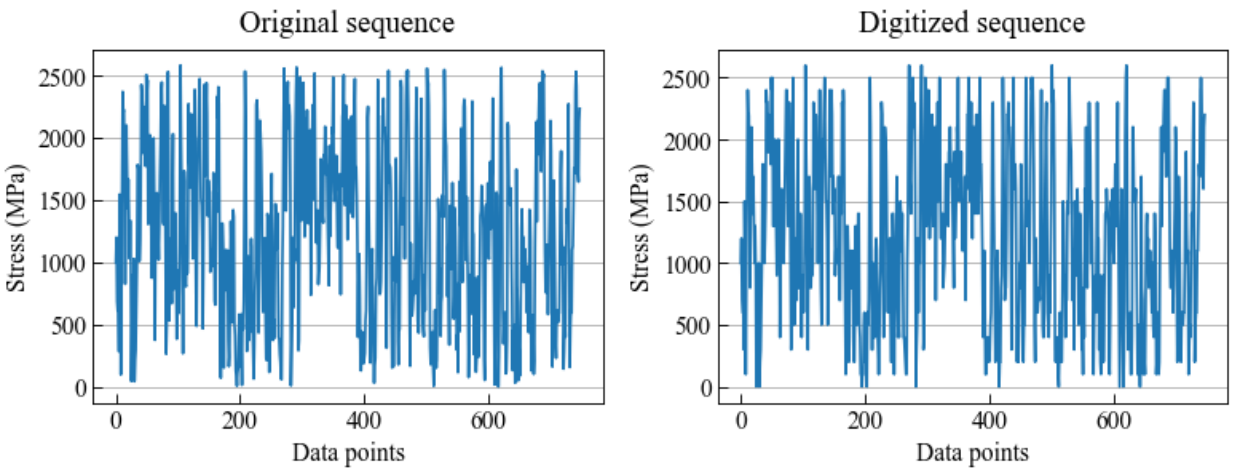
[9]:
```
from ffpack.utils import sequenceDigitization

dstrResults = sequenceDigitization( arman2ndResults, resolution=100 )
```

[10]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( arman2ndResults, '-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Stress (MPa)" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Original sequence", pad=10 )
```

(continues on next page)

(continued from previous page)

```
ax1.grid( axis='y', color="0.7" )

ax2.plot( dstrResults, '-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Stress (MPa)" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Digitized sequence", pad=10 )
ax2.grid( axis='y', color="0.7" )

plt.tight_layout()
plt.show()
```



Filtering

The digitized sequence is firstly filtered using the `sequencePeakValleyFilter` to retain only the peek and valley values. The resulting sequence is further filtered using the `sequenceHysteresisFilter` to eliminate small vibrations.

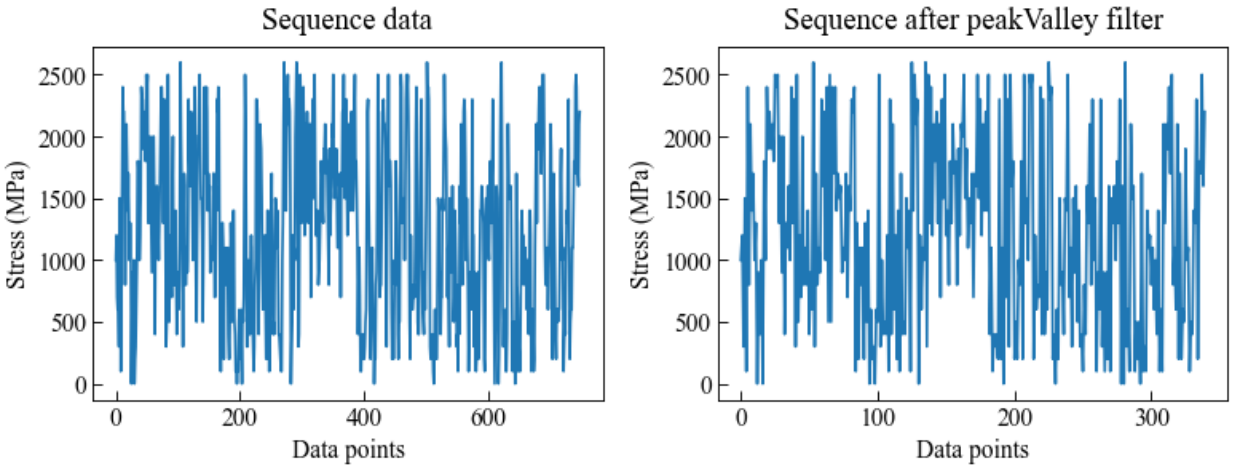
[11]:
```
from ffpack.utils import sequencePeakValleyFilter

gspvResults = sequencePeakValleyFilter( dstrResults, keepEnds=True )
```

[12]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( dstrResults, '-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Stress (MPa)" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data", pad=10 )

ax2.plot( gspvResults, '-' )
```

(continues on next page)

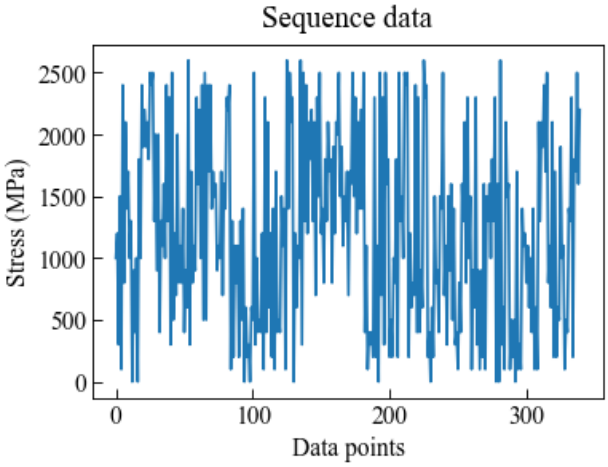
(continued from previous page)

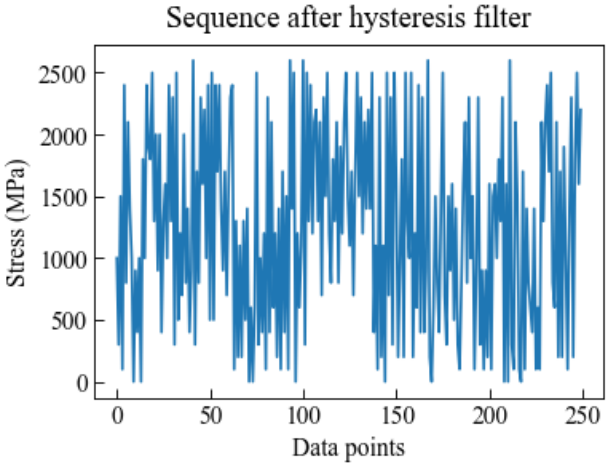
```
ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Stress (MPa)" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Sequence after peakValley filter", pad=10 )

plt.tight_layout()
plt.show()
```



[13]:
```
from ffpack.utils import sequenceHysteresisFilter

gateSize = 500   # assumed
hfResults = sequenceHysteresisFilter( gspvResults, gateSize )
```

[14]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )

ax1.plot( gspvResults, '-' )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Stress (MPa)" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data", pad=10 )

ax2.plot( hfResults, '-' )

ax2.tick_params(axis='x', direction="in", length=5)
ax2.tick_params(axis='y', direction="in", length=5)
ax2.set_ylabel( "Stress (MPa)" )
ax2.set_xlabel( "Data points" )
ax2.set_title( "Sequence after hysteresis filter", pad=10 )

plt.tight_layout()
plt.show()
```





Rainflow counting and mean stress correction

After filtering, the stress sequence is processed using the `astmRainflowCounting` method to obtain the cycle counts. To account for the impact of the superimposed tensile stress, the stresses are corrected using the `goodmanCorrection` method. The results are saved in both non-corrected and corrected forms for later fatigue damage evaluation.

[15]:
```
from ffpack.lcc import astmRainflowCounting

astmRfcCountingRaw = astmRainflowCounting( hfResults, aggregate=False )

astmRfcCountingRaw[:6][:]
```

[15]:
```
[[1000.0, 300.0, 0.5],
 [300.0, 1500.0, 0.5],
 [1500.0, 100.0, 0.5],
 [800.0, 2100.0, 1],
 [100.0, 2400.0, 0.5],
 [900.0, 400.0, 1]]
```

[16]:
```
# show stress amplitude for rainflow counting result

astmRfcCountingAmp = []
for row in astmRfcCountingRaw:
    astmRfcCountingAmp.append([abs(row[0] - row[1])/2, row[2]])

astmRfcCountingAmp[:6][:]
```

[16]:
```
[[350.0, 0.5],
 [600.0, 0.5],
 [700.0, 0.5],
 [650.0, 1],
 [1150.0, 0.5],
 [250.0, 1]]
```

[17]:
```
# mean stress correction
from ffpack.lcc import goodmanCorrection
```

(continues on next page)

(continued from previous page)

```
goodmanCorrected = []

for i in range(0, len(astmRfcCountingRaw)):
    stressRangeData = astmRfcCountingRaw[i][:2]
    if stressRangeData[0] > stressRangeData[1]:
        temp = stressRangeData[0]
        stressRangeData[0] = stressRangeData[1]
        stressRangeData[1] = temp
    goodmanResult = goodmanCorrection( stressRangeData, ultimateStrength )
    goodmanCorrected.append([goodmanResult, astmRfcCountingRaw[i][2] ])
```

[18]:
```
goodmanCorrected[:6][:]
```

[18]:
```
[[467.2680412371134, 0.5],
 [919.5266272189349, 0.5],
 [1012.8491620111732, 0.5],
 [1476.7543859649122, 1],
 [2222.7611940298502, 0.5],
 [333.7628865979381, 1]]
```

The stress amplitudes in each row of the corrected result, named `goodmanCorrected`, are larger than the original amplitudes in `astmRfcCountingAmp`.

[19]:
```
# rainflow counting results after aggregation
astmRfcCountingResults = astmRainflowCounting( hfResults )
astmRfcCountingResults
```

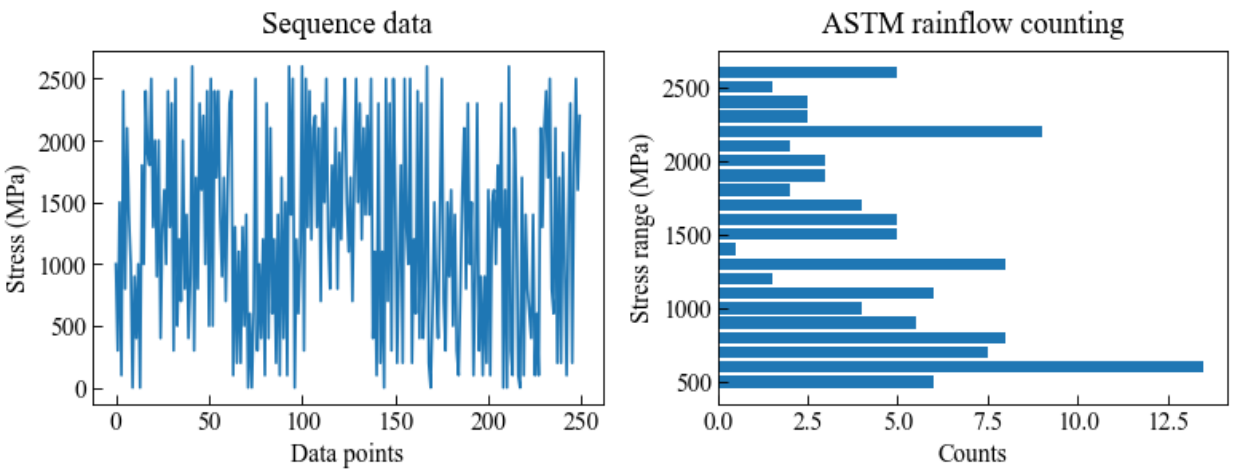
[19]:
```
[[500.0, 6.0],
 [600.0, 13.5],
 [700.0, 7.5],
 [800.0, 8.0],
 [900.0, 5.5],
 [1000.0, 4.0],
 [1100.0, 6.0],
 [1200.0, 1.5],
 [1300.0, 8.0],
 [1400.0, 0.5],
 [1500.0, 5.0],
 [1600.0, 5.0],
 [1700.0, 4.0],
 [1800.0, 2.0],
 [1900.0, 3.0],
 [2000.0, 3.0],
 [2100.0, 2.0],
 [2200.0, 9.0],
 [2300.0, 2.5],
 [2400.0, 2.5],
 [2500.0, 1.5],
 [2600.0, 5.0]]
```

[20]:
```
fig, ( ax1, ax2 ) = plt.subplots( 1, 2, figsize=( 10, 4 ) )
```

(continues on next page)

(continued from previous page)

```
ax1.plot( hfResults, "-" )

ax1.tick_params( axis='x', direction="in", length=5 )
ax1.tick_params( axis='y', direction="in", length=5 )
ax1.set_ylabel( "Stress (MPa)" )
ax1.set_xlabel( "Data points" )
ax1.set_title( "Sequence data", pad=10 )

arr_str = np.array( astmRfcCountingResults )[ :, 0 ].astype(int).astype(str)
ax2.barh( arr_str,
          np.array( astmRfcCountingResults )[ :, 1 ] )

# show tick labels every 5
from matplotlib.ticker import MultipleLocator
ax2.yaxis.set_major_locator(MultipleLocator(5))

ax2.tick_params( axis='x', direction="in", length=5 )
ax2.tick_params( axis='y', direction="in", length=5 )
ax2.set_ylabel( "Stress range (MPa)" )
ax2.set_xlabel( "Counts" )
ax2.set_title( "ASTM rainflow counting", pad=10 )

plt.tight_layout()
plt.show()
```



Rainflow counting matrix

[21]:
```
from ffpack.lsm import astmRainflowCountingMatrix

arcmMat, arcmIndex = astmRainflowCountingMatrix( hfResults )

arcmMat = np.array( arcmMat )
arcmIndex = np.array([s.replace(',', '') for s in arcmIndex]).astype(float)
```

[22]:
```
print( "ASTM rainflow counting matrix" )
print( arcmMat )
print()
print( "Matrix index" )
print( arcmIndex )
```

```
ASTM rainflow counting matrix
[[0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  1.  0.  0.  0.  0.  0.  1.  0.
  0.  0.  0.  0.  0.  0.  0.  1.5 2.5]
 [0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  2.  0.  1.  1.
  0.  0.  0.  1.  0.  5.  0.5 0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  1.  0.  0.  1.
  1.  1.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.  0.5 1.  0.
  0.  0.  0.  0.  0.  0.  0.  2.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  1.  0.  0.
  0.  0.  0.  1.  0.  1.  1.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.
  0.  0.  1.  0.  0.  0.  1.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.
  0.  0.  0.  1.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  1.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.
  0.  0.  0.  2.  0.  0.  0.  0.  0. ]
 [0.  0.  1.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.
  0.  0.  1.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.5 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  2.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  1.  0.  0.  0.  0. ]
 [0.  0.  1.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  1.  1.  0.  1.  1.  0.  0. ]
 [0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  1.  0. ]
 [0.  0.5 0.  0.  1.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  1.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  1.5 0.  0.  0.  0. ]
 [0.  1.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  2.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  1.  1.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
```

(continues on next page)

(continued from previous page)

```
 [0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  1.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  1.  1.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.
  1.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  2.  2.  2.  1.  0.  0.  2.  1.  0.  1.  0.  1.  0.  0.  0.  0.5 0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [2.5 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0. ]]

Matrix index
[   0.  100.  200.  300.  400.  500.  600.  700.  800.  900. 1000. 1100.
 1200. 1300. 1400. 1500. 1600. 1700. 1800. 1900. 2000. 2100. 2200. 2300.
 2400. 2500. 2600.]
```

[23]:
```
plt.set_cmap( "viridis_r")
fig, ax = plt.subplots(figsize=( 5, 5 ))

cax = ax.matshow( arcmMat )

ax.tick_params( axis='x', direction="in", length=5, top=False, bottom=False )
ax.tick_params( axis='y', direction="in", length=5, left=False, right=False )
ax.tick_params( axis='x', which="minor", top=False, bottom=False )
ax.tick_params( axis='y', which="minor", left=False, right=False )

# show tick labels every 5
from matplotlib.ticker import MultipleLocator
ax.yaxis.set_major_locator(MultipleLocator(5))
ax.xaxis.set_major_locator(MultipleLocator(5))
ax.set_xticklabels( [ '' ] + arcmIndex.astype(int)[::5].tolist() )
ax.set_yticklabels( [ '' ] + arcmIndex.astype(int)[::5].tolist() )

# show every tick
ax.set_xticks( np.arange( -.5, len( arcmIndex ), 1 ), minor=True )
ax.set_yticks( np.arange( -.5, len( arcmIndex ), 1 ), minor=True )
ax.grid( which="minor", color='w', linestyle='-', linewidth=2 )

ax.set_ylabel( "From" )
ax.set_xlabel( "To" )
ax.xaxis.set_label_position( "top" )
ax.set_title( "ASTM rainflow counting matrix", y=-0.15 )

ax.set_aspect('auto')
fig.colorbar( cax )
plt.tight_layout()
plt.show()
```

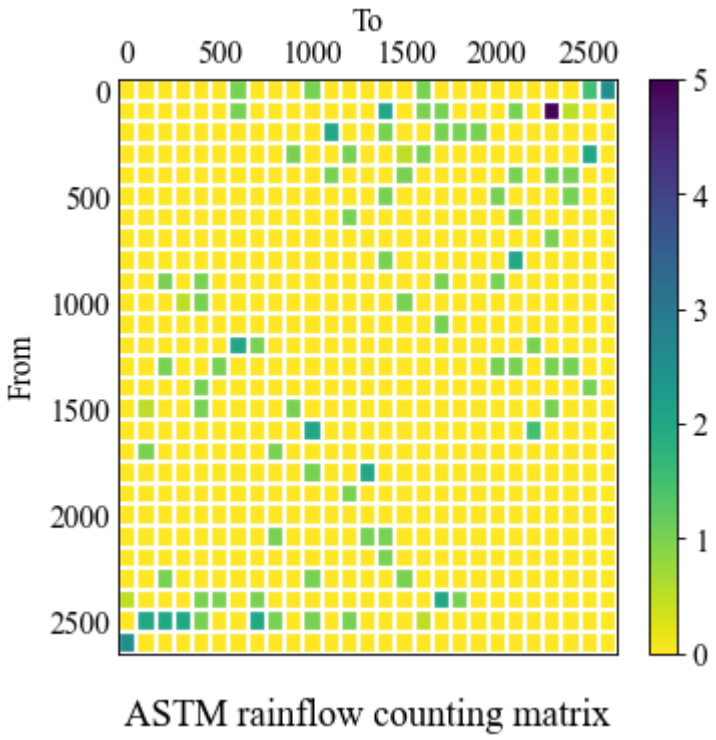
```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/ipykernel_
↪launcher.py:15: UserWarning: FixedFormatter should only be used together with␣
↪FixedLocator
  from ipykernel import kernelapp as app
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/ipykernel_
↪launcher.py:16: UserWarning: FixedFormatter should only be used together with␣
```

(continues on next page)

(continued from previous page)

```
↪FixedLocator
```

```
<Figure size 400x320 with 0 Axes>
```



Fatigue damage model

Classic Palmgren-miner damage model is used to evaluate the fatigue limit under the loading sequence.

[24]:
```
from ffpack.fdm import minerDamageModelClassic

# use non-corrected stresses
import copy
cmdrLccData = copy.deepcopy(astmRfcCountingResults)
# change the stress range in rainflow counting result to stress amplitute
for row in cmdrLccData:
    row[0] /= 2

cmdrSnData = snData
cmdrFatigueLimit = 50

cmdrResults = minerDamageModelClassic( cmdrLccData, cmdrSnData, cmdrFatigueLimit )
```

[25]:
```
print( "{:.4f}".format(cmdrResults) )
```

```
0.0007
```

[26]:
```
# use goodman corrected stresses

cmdrLccData = goodmanCorrected
cmdrSnData = snData
cmdrFatigueLimit = 50

cmdrResults = minerDamageModelClassic( cmdrLccData, cmdrSnData, cmdrFatigueLimit )
```

[27]:
```
print( "{:.4f}".format(cmdrResults) )
```

```
0.1822
```

The application of the Goodman correction on the stresses resulted in a higher calculated fatigue limit compared to the non-corrected stresses. This implies that the presence of superimposed static stress increases the fatigue damage.

1.11 Reliability application

1.11.1 Reliability application

This is an application of fatigue crack growth probabilistic analysis of a finite width rectangular plate with an edge crack subject to constant amplitude in [1].

The number of cycles to failure N_f is given as

$$N_f = \frac{a_f^{1-m/2} - a_i^{1-m/2}}{c(1.1215\Delta\sigma)^m \pi^{m/2}(1-m/2)}$$

where $m = 3.32$, c is the Paris constant, $\Delta\sigma$ is the load, a_i is the initial crack size, a_f is the final crack size and is expressed as

$$a_f = \frac{1}{\pi}\left(\frac{K_{IC}}{1.1215\Delta\sigma}\right)^2$$

where K_{IC} is the fracture toughness.

The random variables follow certain distributions given in the following table.

Random Variable	Distribution	Mean	Standard Deviation
Load (ksi), $\Delta\sigma$	Lognormal	100	10
Initial Crack Size (in), a_i	Lognormal	0.01	0.005
Paris constant, c	Lognormal	1.2e-10	1.2e-11
Fracture Toughness $ksi/\sqrt{in}$, K_{IC}	Normal	60	6

Reference:

[1] Choi, S.K., Canfield, R.A. and Grandhi, R.V., 2007. Reliability-Based Structural Optimization (pp. 153-202). Springer London.

[1]:
```
# Import auxiliary libraries for demonstration

import matplotlib as mpl
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import numpy as np

plt.rcParams[ "figure.figsize" ] = [ 5, 4 ]

plt.rcParams[ "figure.dpi" ] = 80
plt.rcParams[ "font.family" ] = "Times New Roman"
plt.rcParams[ "font.size" ] = '14'
```

[2]:
```
# Set random seed for repeatable results
from ffpack.config import globalConfig

globalConfig.setSeed( 2023 )
```

Parameter distributions

[3]:
```
import math
from scipy import stats

m = 3.32

meanStd = [ [ 100, 10 ], [ 0.01, 0.005 ], [ 1.2e-10, 0.12e-10 ], [ 60, 6 ] ]
```

We need to convert the mean and standard derivation for the lognorm distribution.

A random variable X follows lognormal distribution, and random variable $Y = lnX$ follows a normal distribution.

The PDF of Y is give by

$$f_Y(y) = \frac{1}{\sqrt{2\pi}\sigma_Y} \exp\left[-\frac{1}{2}\left(\frac{y-\mu_Y}{\sigma_Y}\right)^2\right], -\infty < y < \infty$$

The equation can be rewritten in terms of X as

$$f_X(x) = \frac{1}{\sqrt{2\pi}x\sigma_Y} \exp\left[-\frac{1}{2}\left(\frac{lnx-\mu_Y}{\sigma_Y}\right)^2\right], 0 \le x < \infty$$

where

$$\sigma_Y^2 = \ln\left[\left(\frac{\sigma_X}{\mu_X}\right)^2 + 1\right]$$

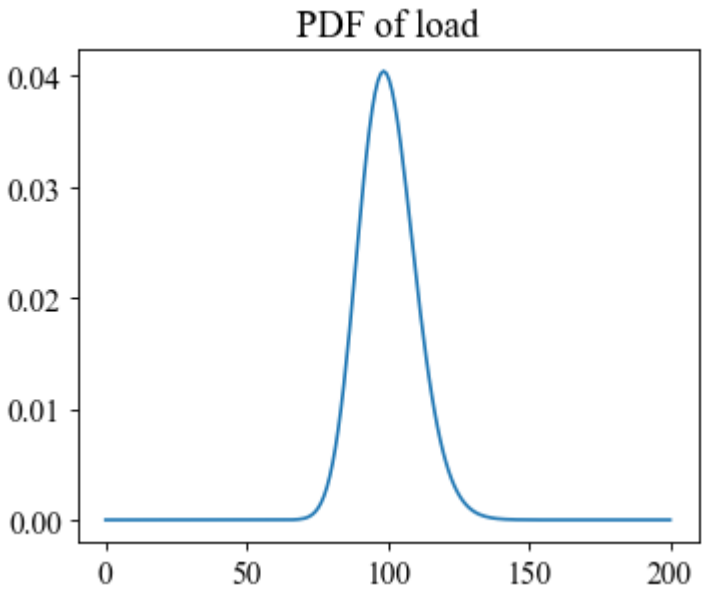
$$\mu_Y = \ln\mu_X - \frac{1}{2}\sigma_Y^2$$

[4]:
```
from numpy import log

def convertLognormal(mean, std):
    sigma2 = log( 1 + std**2 / mean**2 )
    mu = log( mean ) - sigma2 / 2
    return ( mu, sigma2 ** 0.5 )
```

[5]:
```
mu, std = convertLognormal(meanStd[0][0], meanStd[0][1])

X1 = stats.lognorm( s=std, scale=np.exp(mu) )
print(X1.mean())
print(X1.var() ** 0.5)

xxx = np.linspace(0, 200, 200)

fig, ax = plt.subplots()

ax.plot(xxx, X1.pdf(xxx))
ax.set_title( "PDF of load" )
```

```
100.00000000000001
10.000000000000005
```

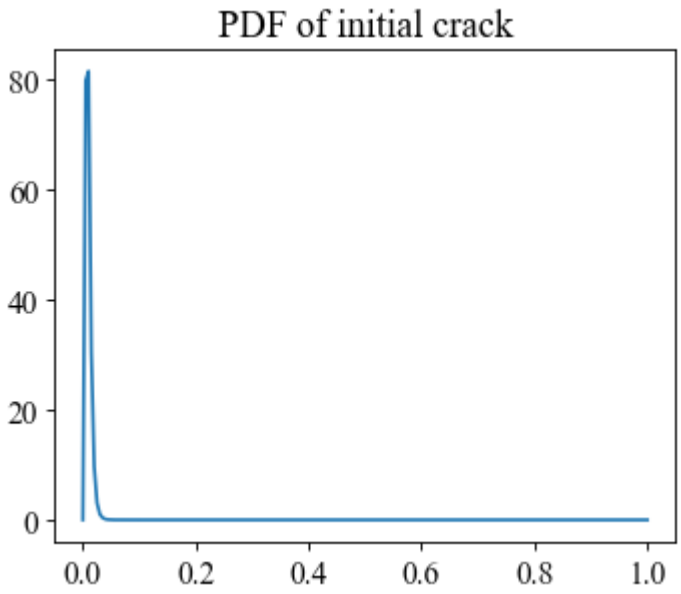
[5]:
```
Text(0.5, 1.0, 'PDF of load')
```



[6]:
```
mu, std = convertLognormal(meanStd[1][0], meanStd[1][1])
X2 = stats.lognorm( s=std, scale=np.exp(mu) )
print(X2.mean())
print(X2.var() ** 0.5)

xxx = np.linspace(0, 1, 200)

fig, ax = plt.subplots()

ax.plot(xxx, X2.pdf(xxx))
ax.set_title( "PDF of initial crack" )
```

```
0.010000000000000002
0.005
```

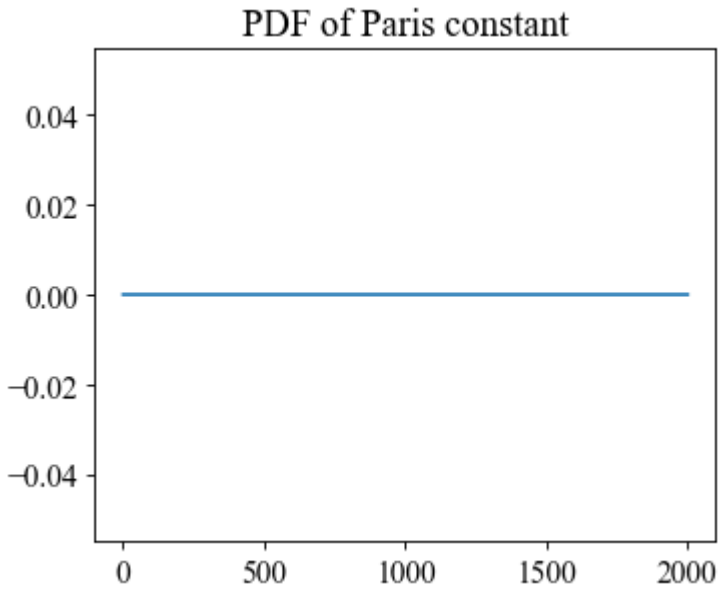
[6]:
```
Text(0.5, 1.0, 'PDF of initial crack')
```



[7]:
```
mu, std = convertLognormal(meanStd[2][0], meanStd[2][1])
X3 = stats.lognorm( s=std, scale=np.exp(mu) )
print(X3.mean())
print(X3.var() ** 0.5)

xxx = np.linspace(0, 2000, 2000)

fig, ax = plt.subplots()
ax.plot(xxx, X3.pdf(xxx))
ax.set_title( "PDF of Paris constant" )
```

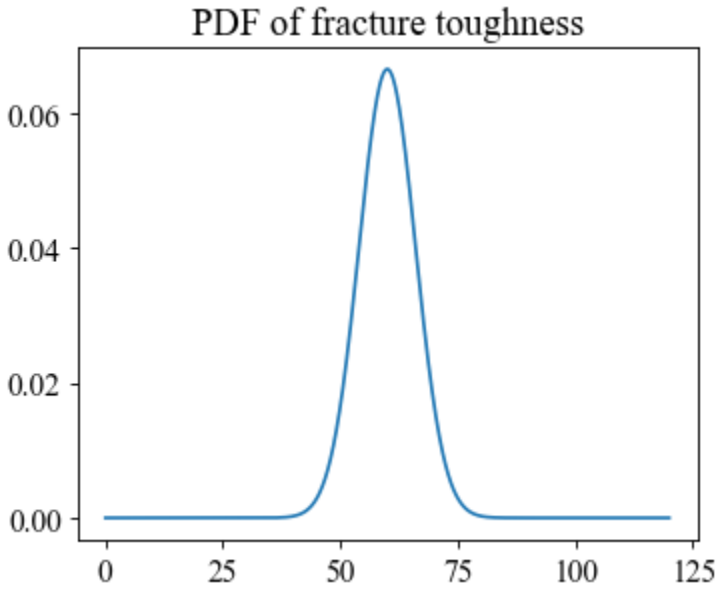
```
1.2000000000000013e-10
1.200000000000002e-11
```

[7]:
```
Text(0.5, 1.0, 'PDF of Paris constant')
```



[8]:
```
X4 = stats.norm( loc=meanStd[3][0], scale=meanStd[3][1] )

xxx = np.linspace(0, 120, 1200)

fig, ax = plt.subplots()
ax.plot(xxx, X4.pdf(xxx))
ax.set_title( "PDF of fracture toughness" )
```

[8]:
```
Text(0.5, 1.0, 'PDF of fracture toughness')
```



Limit state function

[9]:
```
# Define the dimension for the FORM problem
dim = 4

# Define the limit state function (LSF) g
def g( X ):
    numerator = ( ( X[3] / 1.1215 / X[0] ) ** 2 / math.pi ) ** ( 1 - m / 2 ) - X[1] ** (␣
↪1 - m / 2 )
    denominator = X[2] * ( 1.1215 * X[0] ) ** m * math.pi ** ( m / 2 ) * ( 1 - m / 2 )
    return numerator / denominator - 3000

# use internal automatic differentiation algorithm
dg = None
```

Marginal distribution

[10]:
```
# Marginal distributions and correlation Matrix of the random variables
distObjs = [ X1, X2, X3, X4 ]
corrMat = np.eye( dim )
```

[11]:
```
# define an array to record the reliability index and failure probability
results = []
```

FORM

[12]:
```
# FORM: Constrained optimization FORM

from ffpack.rrm import coptFORM

beta, pf, uCoord, xCoord = coptFORM( dim, g, distObjs, corrMat )
results.append( [ beta, pf ] )
```

[13]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
1.0037934895478329

Failure probability:
0.15773908156041105
```

(continues on next page)

(continued from previous page)

```
Design point coordinate in U space:
[0.6472382726814411, 0.7428977098133317, 0.16674351823826586, -0.09478174884750643]

Design point coordinate in X space:
[106.13988658677613, 0.012704342671365916, 1.2140711290863364e-10, 59.431309506914964]
```

SORM

[14]:
```
# SORM: Breitung SORM

from ffpack.rrm import breitungSORM

beta, pf, uCoord, xCoord = breitungSORM( dim, g, dg, distObjs, corrMat )
results.append( [ beta, pf ] )
```

[15]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
1.0037934895478329

Failure probability:
0.1460989862149804

Design point coordinate in U space:
[0.6472382726814411, 0.7428977098133317, 0.16674351823826586, -0.09478174884750643]

Design point coordinate in X space:
[106.13988658677613, 0.012704342671365916, 1.2140711290863364e-10, 59.431309506914964]
```

[16]:
```
# SORM: Tvedt SORM

from ffpack.rrm import tvedtSORM

beta, pf, uCoord, xCoord = tvedtSORM( dim, g, dg, distObjs, corrMat )
results.append( [ beta, pf ] )
```

[17]:
```
print( "Reliability index: " )
print( beta )
print()
```

(continues on next page)

(continued from previous page)

```
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
1.0037934895478329

Failure probability:
0.13184401228952822

Design point coordinate in U space:
[0.6472382726814411, 0.7428977098133317, 0.16674351823826586, -0.09478174884750643]

Design point coordinate in X space:
[106.13988658677613, 0.012704342671365916, 1.2140711290863364e-10, 59.431309506914964]
```

[18]:
```
# SORM: Hohenbichler and Rackwitz SORM

from ffpack.rrm import hrackSORM

beta, pf, uCoord, xCoord = hrackSORM( dim, g, dg, distObjs, corrMat )
results.append( [ beta, pf ] )
```

[19]:
```
print( "Reliability index: " )
print( beta )
print()
print( "Failure probability: " )
print( pf )
print()
print( "Design point coordinate in U space: " )
print( uCoord )
print()
print( "Design point coordinate in X space: " )
print( xCoord )
```

```
Reliability index:
1.0037934895478329

Failure probability:
0.1540947049684117

Design point coordinate in U space:
[0.6472382726814411, 0.7428977098133317, 0.16674351823826586, -0.09478174884750643]

Design point coordinate in X space:
[106.13988658677613, 0.012704342671365916, 1.2140711290863364e-10, 59.431309506914964]
```

Result comparison

[20]:
```
# results in reference [1]
ref = [ 0.15774, 0.14473, 0.13987, np.nan ]
```

[21]:
```
# combine the two arrays using np.concatenate
combined_array = np.concatenate((results, np.array(ref).reshape(-1, 1)), axis=1)

# format the array elements as strings
data_str = [['{:.4f}'.format(item) for item in row ] for row in combined_array]

# define the first column
extra = [ 'FORM', 'SORM - Breitung', 'SORM - Tvedt', 'SORM - Hrack']

# define the column headers
headers = [ 'Method', 'Reliability index', 'Pf - Ours', 'Pf - Ref']

# Print the array as a formatted table
print('{:<20} {:<20} {:<15} {:<15}'.format(*headers))  # Print the header row

for i in range( len( data_str ) ):
    print('{:<20} {:<20} {:<15} {:<15}'
          .format( np.array(extra)[i] , data_str[i][0], data_str[i][1], data_str[i][2] )␣
↪)
```

```
Method               Reliability index    Pf - Ours       Pf - Ref
FORM                 1.0038               0.1577          0.1577
SORM - Breitung      1.0038               0.1461          0.1447
SORM - Tvedt         1.0038               0.1318          0.1399
SORM - Hrack         1.0038               0.1541          nan
```

The failure probabilities with different methods are very close to the results in reference [1]. The reliability indices given by different methods are consistent.

1.12 Module API

1.12.1 `fdm` fatigue damage model

Palmgren-miner damage model

Palmgren-Miner damage model is one of the famous fatigue damage models for fatigue estimation. The model is defined in a simple and intuitive way and it is very popular now.

Reference: Miner, M.A., 1945. Cumulative damage in fatigue.

`ffpack.fdm.minerModel.`**`minerDamageModelClassic`**(*lccData*, *snData*, *fatigueLimit*)

Classical Palmgren-miner damage model calculates the damage results based on the SN curve.

Parameters

- **`lccData`** (*2d array*) – Load cycle counting results in a 2D matrix, e.g., [[value, count], …]

- **snData** (*2d array*) – Experimental SN data in 2D matrix, e.g., [[N1, S1], [N2, S2], ..., [Ni, Si]]
- **fatigueLimit** (*scalar*) – Fatigue limit indicating the minimum S that can cause fatigue.

Returns
rst – Fatigue damage calculated based on the Palmgren-miner model.

Return type
scalar

Raises
ValueError – If the lccData dimension is not 2. If the lccData length is less than 1.

Examples

```
>>> from ffpack.fdr import minerDamageModelClassic
>>> lccData = [ [ 1, 100 ], [ 2, 10 ] ]
>>> snData = [ [ 10, 3 ], [ 1000, 1 ] ]
>>> fatigueLimit = 0.5
>>> rst = minerDamageModelClassic( lccData, snData, fatigueLimit )
```

ffpack.fdm.minerModel.**minerDamageModelNaive**(*fatigueData*)

Naive Palmgren-miner damage model directly calcuates the damage results.

Parameters
fatigueData (*2d array*) – Paired counting and experimental data under each load condition, e.g., [[C1, F1], [C2, F2], ..., [Ci, Fi]] where Ci and Fi represent the counting cycles and failure cycles under the same load condition.

Returns
rst – Fatigue damage calculated based on the Palmgren-miner model

Return type
scalar

Raises
ValueError – If fatigueData length is less than 1. If counting cycles is less than 0. If number of failure cycles is less than or equal 0. If number of counting cycles is large than failure cycles.

Examples

```
>>> from ffpack.fdm import minerDamageModelNaive
>>> fatigueData = [ [ 10, 100 ], [ 200, 2000 ] ]
>>> rst = minerDamageModelNaive( fatigueData )
```

1.12.2 lcc load counting and correction

ASTM Counting

This module implements the standard cycle counting methods in ASTM E1049-85(2017) Standard Practices for Cycle Counting in Fatigue Analysis

ffpack.lcc.astmCounting.**astmLevelCrossingCounting**(*data*, *refLevel=0.0*, *levels=None*, *aggregate=True*)

ASTM level crossing counting in E1049-85: sec 5.1.1.

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **refLevel** (*scalar, optional*) – Reference level.
- **levels** (*1d array*) – Self-defined levels for counting.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [crossPoint1, corssPoint2, …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data length is less than 2 or the data dimension is not 1.

Examples

```
>>> from ffpack.lcc import astmLevelCrossingCounting
>>> data = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
>>>          -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]
>>> rst = astmLevelCrossingCounting( data )
```

ffpack.lcc.astmCounting.**astmPeakCounting**(*data*, *refLevel=None*, *aggregate=True*)

ASTM peak counting in E1049-85: sec 5.2.1.

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **refLevel** (*scalar, optional*) – Reference level.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [peak1, peak2, …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data length is less than 2 or the data dimension is not 1.

Examples

```
>>> from ffpack.lcc import astmPeakCounting
>>> data = [ 0.0, 1.5, 0.5, 3.5, 0.5, 2.5, -1.5, -0.5, -2.5,
>>>          -2.0, -2.7, -2.5, -3.5, 1.5, 0.5, 3.5, -0.5 ]
>>> rst = astmPeakCounting( data )
```

ffpack.lcc.astmCounting.**astmRainflowCounting**(*data*, *aggregate=True*)

ASTM rainflow counting in E1049-85: sec 5.4.4.

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data length is less than 2 or the data dimension is not 1.

Examples

```
>>> from ffpack.lcc import astmRainflowCounting
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst = astmRainflowCounting( data )
```

ffpack.lcc.astmCounting.**astmRainflowRepeatHistoryCounting**(*data*, *aggregate=True*)

ASTM simplified rainflow counting for repeating histories in E1049-85: sec 5.4.5.

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data length is less than 2 or the data dimension is not 1. If the data is not repeatable: first data point is different from the last data point.

Examples

```
>>> from ffpack.lcc import astmRainflowRepeatHistoryCounting
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst = astmRainflowRepeatHistoryCounting( data )
```

ffpack.lcc.astmCounting.**astmRangePairCounting**(*data*, *aggregate=True*)

ASTM range pair counting in E1049-85: sec 5.4.3.

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data length is less than 2 or the data dimension is not 1.

Examples

```
>>> from ffpack.lcc import astmRangePairCounting
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst = astmRangePairCounting( data )
```

ffpack.lcc.astmCounting.**astmSimpleRangeCounting**(*data*, *aggregate=True*)

ASTM simple range counting in E1049-85: sec 5.3.1.

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data length is less than 2 or the data dimension is not 1.

Examples

```
>>> from ffpack.lcc import astmSimpleRangeCounting
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst = astmSimpleRangeCounting( data )
```

Johannesson Counting

Johannesson proposed a minMax cycle counting method.

Reference: Johannesson, P., 1998. Rainflow cycles for switching processes with Markov structure. Probability in the Engineering and Informational Sciences, 12(2), pp.143-175.

`ffpack.lcc.johannessonCounting.`**`johannessonMinMaxCounting`**(*data*, *aggregate=True*)

Johannesson min-max counting

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **aggragate** (*bool, optional*) – if aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], …], will be returned.

Returns

rst – Sorted counting results.

Return type

2d array

Raises

ValueError – If the data dimension is not 1 If the data length is less than 2

Examples

```
>>> from ffpack.lcc import johannessonMinMaxCounting
>>> data = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
>>>          -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]
>>> rst = johannessonMinMaxCounting( data )
```

Rychlik Counting

Rychlik proposed a toplevel-up cycle counting method and proved that the proposed method is equivalent to the classical rainflow counting method. Compared to the classical rainflow counting method, the proposed method keeps the original sequence information which is quite useful if the sequence information is required for further analysis.

Reference: Rychlik, I., 1987. A new definition of the rainflow cycle counting method. International journal of fatigue, 9(2), pp.119-121.

`ffpack.lcc.rychlikCounting.`**`rychlikRainflowCounting`**(*data*, *aggregate=True*)

Rychilk rainflow counting (toplevel-up cycle TUC)

Parameters

- **data** (*1d array*) – Load sequence data for counting.

- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], ...], will be returned.

Returns
rst – Sorted counting results.

Return type
2d array

Raises
`ValueError` – If the data dimension is not 1 If the data length is less than 2

Examples

```
>>> from ffpack.lcc import rychlikRainflowCycleCounting
>>> data = [ -0.8, 1.3, 0.7, 3.4, 0.7, 2.5, -1.4, -0.5, -2.3,
>>>          -2.2, -2.6, -2.4, -3.3, 1.5, 0.6, 3.4, -0.5 ]
>>> rst = rychlikRainflowCycleCounting( data )
```

Rychlik Counting

ffpack.lcc.fourPointCounting.**fourPointRainflowCounting**(*data*, *aggregate=True*)

Four point rainflow counting in [Lee2011].

Parameters

- **data** (*1d array*) – Load sequence data for counting.
- **aggragate** (*bool, optional*) – If aggregate is set to False, the original sequence for internal counting, e.g., [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], ...], will be returned.

Returns
rst – Sorted counting results.

Return type
2d array

Raises
`ValueError` – If the data length is less than 4 or the data dimension is not 1.

Examples

```
>>> from ffpack.lcc import fourPointRainflowCounting
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst = fourPointRainflowCounting( data )
```

References

Mean Stress Correction

This module implements the mean stress methods to quantify the interaction of mean and alternating stresses on the fatigue life of a material.

ffpack.lcc.meanStressCorrection.**gerberCorrection**(*stressRange*, *ultimateStrength*, *n=1.0*)

The Gerber correction in this implementation is applicable to cases with stress ratio no less than -1.

Parameters

- **stressRange** (*1d array*) – Stress range, e.g., [lowerStress, upperStress].
- **ultimateStrength** (*scalar*) – Ultimate strength.
- **n** (*scalar, optional*) – Safety factor, default to 1.0.

Returns
rst – Fatigue limit.

Return type
scalar

Raises
ValueError – If the stressRange dimension is not 1, or stressRange length is not 2. If stressRange[1] <= 0 or stressRange[0] >= stressRange[1]. If ultimateStrength is not a scalar or ultimateStrength <= 0. If ultimateStrength is smaller than the mean stress. If safety factor n < 1.0.

Examples

```
>>> from ffpack.lcc import gerberCorrection
>>> stressRange = [ 1.0, 2.0 ]
>>> ultimateStrength = 3.0
>>> rst = gerberCorrection( stressRange, ultimateStrength )
```

ffpack.lcc.meanStressCorrection.**goodmanCorrection**(*stressRange*, *ultimateStrength*, *n=1.0*)

The Goodman correction in this implementation is applicable to cases with stress ratio no less than -1.

Parameters

- **stressRange** (*1d array*) – Stress range, e.g., [lowerStress, upperStress].
- **ultimateStrength** (*scalar*) – Ultimate tensile strength.
- **n** (*scalar, optional*) – Safety factor, default to 1.0.

Returns
rst – Fatigue limit.

Return type
scalar

Raises
ValueError – If the stressRange dimension is not 1, or stressRange length is not 2. If stressRange[1] <= 0 or stressRange[0] >= stressRange[1]. If ultimateStrength is not a scalar or ultimateStrength <= 0. If ultimateStrength is smaller than the mean stress. If n < 1.0.

Examples

```
>>> from ffpack.lcc import goodmanCorrection
>>> stressRange = [ 1.0, 2.0 ]
>>> ultimateStrength = 4.0
>>> rst = goodmanCorrection( stressRange, ultimateStrength )
```

ffpack.lcc.meanStressCorrection.**soderbergCorrection**(*stressRange*, *yieldStrength*, *n=1.0*)

The Soderberg correction in this implementation is applicable to cases with stress ratio no less than -1.

Parameters

- **stressRange** (*1d array*) – Stress range, e.g., [lowerStress, upperStress].
- **yieldStrength** (*scalar*) – Yield strength.
- **n** (*scalar, optional*) – Safety factor, default to 1.0.

Returns

rst – Fatigue limit.

Return type

scalar

Raises

ValueError – If the stressRange dimension is not 1, or stressRange length is not 2. If stressRange[1] <= 0 or stressRange[0] >= stressRange[1]. If yieldStrength is not a scalar or yieldStrength <= 0. If yieldStrength is smaller than the mean stress. If safety factor n < 1.0.

Examples

```
>>> from ffpack.lcc import soderbergCorrection
>>> stressRange = [ 1.0, 2.0 ]
>>> yieldStrength = 3.0
>>> rst = soderbergCorrection( stressRange, yieldStrength )
```

1.12.3 lsg load sequence generator

Random walk

ffpack.lsg.randomWalk.**randomWalkUniform**(*numSteps*, *dim=1*, *randomSeed=None*)

Generate load sequence by a random walk.

Parameters

- **numSteps** (*integer*) – Number of steps for generating.
- **dim** (*scalar, optional*) – Data dimension.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns

rst – A 2d (numSteps by dim) matrix holding the coordinates of the position at each step.

Return type

2d array

Raises
ValueError – If the numSteps is less than 1 or the dim is less than 1.

Examples

```
>>> from ffpack.lsg import randomWalkUniform
>>> rst = randomWalkUniform( 5 )
```

Autoregressive moving average

ffpack.lsg.autoregressiveMovingAverage.**arNormal**(*numSteps*, *obs*, *phis*, *mu*, *sigma*, *randomSeed=None*)

Generate load sequence by an autoregressive model.

The white noise is generated by the normal distribution.

Parameters

- **numSteps** (*integer*) – Number of steps for generating.
- **obs** (*1d array*) – Initial observed values.
- **phis** (*1d array*) – Coefficients for the autoregressive model.
- **mu** (*scalar*) – Mean of the white noise.
- **sigma** (*scalar*) – Standard deviation of the white noise.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns
rst – Generated sequence includes the observed values.

Return type
1d array

Raises
ValueError – If the numSteps is less than 1. If lengths of obs and phis are not equal.

Examples

```
>>> from ffpack.lsg import arNormal
>>> obs = [ 0, 1 ]
>>> phis = [ 0.5, 0.3 ]
>>> rst = arNormal( 500, obs, phis, 0, 0.5 )
```

ffpack.lsg.autoregressiveMovingAverage.**arimaNormal**(*numSteps*, *c*, *phis*, *thetas*, *mu*, *sigma*, *randomSeed=None*)

Generate load sequence by an autoregressive integrated moving average model.

The white noise is generated by the normal distribution.

First-order diference is used in this function.

Parameters

- **numSteps** (*integer*) – Number of steps for generating.

- **c** (*scalar*) – Mean of the series.
- **phis** (*1d array*) – Coefficients for the autoregressive part.
- **thetas** (*1d array*) – Coefficients for the white noise for the moving-average part.
- **mu** (*scalar*) – Mean of the white noise.
- **sigma** (*scalar*) – Standard deviation of the white noise.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns
rst – Generated sequence with the autoregressive integrated moving average model.

Return type
1d array

Raises
ValueError – If the numSteps is less than 1. If mean of the series is not a scalar. If the phis is empty. If the thetas is empty.

Examples

```
>>> from ffpack.lsg import arimaNormal
>>> phis = [ 0.5, 0.3 ]
>>> thetas = [ 0.8, 0.5 ]
>>> rst = arimaNormal( 500, 0.0, phis, thetas, 0, 0.5 )
```

ffpack.lsg.autoregressiveMovingAverage.**armaNormal**(*numSteps*, *obs*, *phis*, *thetas*, *mu*, *sigma*, *randomSeed=None*)

Generate load sequence by an autoregressive-moving-average model.

The white noise is generated by the normal distribution.

Parameters

- **numSteps** (*integer*) – Number of steps for generating.
- **obs** (*1d array*) – Initial observed values, could be empty.
- **phis** (*1d array*) – Coefficients for the autoregressive part.
- **thetas** (*1d array*) – Coefficients for the white noise for the moving-average part.
- **mu** (*scalar*) – Mean of the white noise.
- **sigma** (*scalar*) – Standard deviation of the white noise.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns
rst – Generated sequence includes the observed values.

Return type
1d array

Raises
ValueError – If the numSteps is less than 1. If the phis is empty. If the thetas is empty.

Examples

```
>>> from ffpack.lsg import armaNormal
>>> obs = [ 0, 1 ]
>>> phis = [ 0.5, 0.3 ]
>>> thetas = [ 0.8, 0.5 ]
>>> rst = armaNormal( 500, obs, phis, thetas, 0, 0.5 )
```

ffpack.lsg.autoregressiveMovingAverage.**maNormal**(*numSteps*, *c*, *thetas*, *mu*, *sigma*, *randomSeed=None*)

Generate load sequence by a moving-average model.

The white noise is generated by the normal distribution.

Parameters

- **numSteps** (*integer*) – Number of steps for generating.
- **c** (*scalar*) – Mean of the series.
- **thetas** (*1d array*) – Coefficients for the white noise in the moving-average model.
- **mu** (*scalar*) – Mean of the white noise.
- **sigma** (*scalar*) – Standard deviation of the white noise.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns

rst – Generated sequence with moving-average model.

Return type

1d array

Raises

ValueError – If the numSteps is less than 1. If mean of the series is not a scalar. If the thetas is empty.

Examples

```
>>> from ffpack.lsg import maNormal
>>> thetas = [ 0.8, 0.5 ]
>>> rst = maNormal( 500, 0, thetas, 0, 0.5 )
```

Sequence from spectrum

ffpack.lsg.sequenceFromSpectrum.**spectralRepresentation**(*fs*, *time*, *freq*, *psd*, *freqBandwidth=None*, *randomSeed=None*)

Generate a sequence from a given power spectrum density with spectral representation method.

Parameters

- **fs** (*scalar*) – Sampling frequency.
- **time** (*scalar*) – Total sampling time.
- **freq** (*1darray*) – Frequency array for psd. The freq array should be in equally spaced increasing.

- **psd** (*1darray*) – Power spectrum density array.
- **freqBandwidth** (*scalar, optional*) – Frequency bandwidth used to generate the time series from psd. Default to None, every frequency in freq will be used.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns

- **ts** (*1darray*) – Array containing all the time data for the time series.
- **amps** (*1darray*) – Amplitude array containing the amplitudes of the time series corresponding to ts.

Raises

ValueError – If the fs or time is not a scalar. If freq or psd is not a 1darray or has less than 3 elements. If freq and psd are in different lengths. If freq contains negative elements. If freq is not equally spaced increasing.

Examples

```
>>> from ffpack.lsg import spectralRepresentation
>>> fs = 100
>>> time = 10
>>> freq = [ 0, 0.1, 0.2, 0.3, 0.4, 0.5 ]
>>> psd = [ 0.01, 2, 0.05, 0.04, 0.01, 0.03 ]
>>> ts, amps = spectralRepresentation( fs, time, freq, psd, freqBandwidth=None )
```

1.12.4 lsm load spectra and matrices

Wave spectra

ffpack.lsm.waveSpectra.**gaussianSwellSpectrum**(*w*, *wp*, *Hs*, *sigma*)

Gaussian Swell spectrum, typically used to model long period swell seas [Guidance2016A].

Parameters

- **w** (*scalar*) – Wave frequency.
- **wp** (*scalar*) – Peak wave frequency.
- **Hs** (*scalar*) – Significant wave height.
- **sigma** (*scalar*) – peakedness parameter for Gaussian spectral width.

Returns

rst – The wave spectrum density value at wave frequency w.

Return type

scalar

Raises

ValueError – If w is not a scalar. If wp is not a scalar. If Hs is not a scalar. If sigma is not a scalar.

Examples

```
>>> from ffpack.lsm import gaussianSwellSpectrum
>>> w = 0.02
>>> wp = 0.51
>>> Hs = 20
>>> sigma = 0.07
>>> rst = gaussianSwellSpectrum( w, wp, Hs, sigma )
```

References

ffpack.lsm.waveSpectra.**isscSpectrum**(*w*, *wp*, *Hs*)

ISSC spectrum, also known as Bretschneider or modified Pierson-Moskowitz.

Parameters

- **w** (*scalar*) – Wave frequency.
- **wp** (*scalar*) – Peak wave frequency.
- **Hs** (*scalar*) – Significant wave height.

Returns

rst – The wave spectrum density value at wave frequency w.

Return type

scalar

Raises

ValueError – If w is not a scalar. If wp is not a scalar. If Hs is not a scalar.

Examples

```
>>> from ffpack.lsm import isscSpectrum
>>> w = 0.02
>>> wp = 0.51
>>> Hs = 20
>>> rst = isscSpectrum( w, wp, Hs )
```

ffpack.lsm.waveSpectra.**jonswapSpectrum**(*w*, *wp*, *alpha=0.0081*, *beta=1.25*, *gamma=3.3*, *g=9.81*)

JONSWAP (Joint North Sea Wave Project) spectrum is an empirical relationship that defines the distribution of energy with frequency within the ocean.

Parameters

- **w** (*scalar*) – Wave frequency.
- **wp** (*scalar*) – Peak wave frequency.
- **alpha** (*scalar, optional*) – Intensity of the Spectra.
- **beta** (*scalar, optional*) – Shape factor, fixed value 1.25.
- **gamma** (*scalar, optional*) – Peak enhancement factor.
- **g** (*scalar, optional*) – Acceleration due to gravity, a constant. 9.81 m/s2 in SI units.

Returns

rst – The wave spectrum density value at wave frequency w.

Return type
scalar

Raises
ValueError – If w is not a scalar. If wp is not a scalar.

Examples

```
>>> from ffpack.lsm import jonswapSpectrum
>>> w = 0.02
>>> wp = 0.51
>>> rst = jonswapSpectrum( w, wp, alpha=0.0081, beta=1.25, gamma=3.3, g=9.81 )
```

ffpack.lsm.waveSpectra.**ochiHubbleSpectrum**(*w*, *wp1*, *wp2*, *Hs1*, *Hs2*, *lambda1*, *lambda2*)

Ochi-Hubble spectrum covers shapes of wave spectra associated with the growth and decay of a storm, including swells. [Guidance2016B].

Parameters

- **w** (*scalar*) – Wave frequency.
- **wp1** (*scalar*) – Peak wave frequency.
- **wp2** (*scalar*) – Peak wave frequency.
- **Hs1** (*scalar*) – Significant wave height.
- **Hs2** (*scalar*) – Significant wave height.
- **lambda1** (*scalar*) –
- **lambda2** (*scalar*) –

Returns
rst – The wave spectrum density value at wave frequency w.

Return type
scalar

Raises
ValueError – If w is not a scalar. If wp1 or wp2 is not a scalar. If Hs1 or Hs2 is not a scalar. If lambda1 or lambda2 is not a scalar. If wp1 is not smaller than wp2.

Notes

Hs1 should normally be greater than Hs2 since most of the wave energy tends to be associated with the lower frequency component.

Examples

```
>>> from ffpack.lsm import ochiHubbleSpectrum
>>> w = 0.02
>>> wp1 = 0.4
>>> wp2 = 0.51
>>> Hs1 = 20
>>> Hs2 = 15
>>> lambda1 = 7
>>> lambda2 = 10
>>> rst = ochiHubbleSpectrum( w, wp1, wp2, Hs1, Hs2, lambda1, lambda2 )
```

References

ffpack.lsm.waveSpectra.**piersonMoskowitzSpectrum**(*w*, *Uw*, *alpha=0.0081*, *beta=0.74*, *g=9.81*)

Pierson Moskowitz spectrum is an empirical relationship that defines the distribution of energy with frequency within the ocean.

Parameters

- **w** (*scalar*) – Wave frequency.
- **Uw** (*scalar*) – Wind speed at a height of 19.5m above the sea surface.
- **alpha** (*scalar, optional*) – Intensity of the Spectra.
- **beta** (*scalar, optional*) – Shape factor.
- **g** (*scalar, optional*) – Acceleration due to gravity, a constant. 9.81 m/s2 in SI units.

Returns

rst – The wave spectrum density value at wave frequency w.

Return type

scalar

Raises

ValueError – If w is not a scalar. If wp is not a scalar.

Examples

```
>>> from ffpack.lsm import piersonMoskowitzSpectrum
>>> w = 0.51
>>> Uw = 20
>>> rst = piersonMoskowitzSpectrum( w, Uw, alpha=0.0081,
...                                 beta=1.25, g=9.81 )
```

Wind spectra

ffpack.lsm.windSpectra.**apiSpectrum**(*f*, *u0*, *z=10*)

API spectrum is implemented according to [API2007].

Parameters

- **f** (*scalar*) – Frequency (Hz).
- **u0** (*scalar*) – 1 hour mean wind speed (m/s) at 10 m above sea level.

Returns

rst – Power spectrum density (m^2 s^-2 Hz^-1).

Return type

scalar

Raises

ValueError – If n is not a scalar. If uz is not a scalar.

Examples

```
>>> from ffpack.lsm import apiSpectrum
>>> f = 2
>>> u0 = 10
>>> rst = apiSpectrum( f, u0 )
```

References

ffpack.lsm.windSpectra.**davenportSpectrumWithDragCoef**(*n*, *delta1*, *kappa=0.005*, *normalized=True*)

Davenport spectrum in the original paper by Davenport [Davenport1961].

Parameters

- **n** (*scalar*) – Frequency (Hz) when normalized=False. Normalized frequency when normalized=True.
- **delta1** (*scalar*) – Velocity (m/s) at standard reference height of 10 m.
- **kappa** (*scalar, optional*) – Drag coefficient referred to mean velocity at 10 m. Default value 0.005 corresponding to open unobstructed country [Davenport1961]. The recommended value for heavilly built-up urban centers with tall buildings is 0.05. The recommended value for country broken by low clustered obstructions is between 0.015 and 0.02.
- **normalized** (*bool, optional*) – If normalized is set to False, the power spectrum density will be returned.

Returns

rst – Power spectrum density (m^2 s^-2 Hz^-1) when normalized=False. Normalized power spectrum density when normalized=True.

Return type

scalar

Raises

ValueError – If n is not a scalar. If delta1 is not a scalar.

Examples

```
>>> from ffpack.lsm import davenportSpectrumWithDragCoef
>>> n = 2
>>> delta1 = 10
>>> rst = davenportSpectrumWithDragCoef( n, delta1, kappa=0.005,
...                                      normalized=True )
```

References

ffpack.lsm.windSpectra.**davenportSpectrumWithRoughnessLength**(*n*, *uz*, *z=10*, *z0=0.03*, *normalized=True*)

Davenport spectrum in the paper by Hiriart et al. [Hiriart2001].

Parameters

- **n** (*scalar*) – Frequency (Hz) when normalized=False. Normalized frequency when normalized=True.
- **uz** (*scalar*) – Mean wind speed (m/s) measured at height z.
- **z** (*scalar, optional*) – Height above the ground (m), default to 10 m.
- **z0** (*scalar, optional*) – Roughness length (m), default to 0.03 m corresponding to open exposure case in [Ho2003].
- **normalized** (*bool, optional*) – If normalized is set to False, the power spectrum density will be returned.

Returns

rst – Power spectrum density (m^2 s^-2 Hz^-1) when normalized=False. Normalized power spectrum density when normalized=True.

Return type

scalar

Raises

ValueError – If n is not a scalar. If uz is not a scalar.

Examples

```
>>> from ffpack.lsm import davenportSpectrumWithRoughnessLength
>>> n = 2
>>> uz = 10
>>> rst = davenportSpectrumWithRoughnessLength( n, uz, z=10, z0=0.03,
...                                             normalized=True )
```

References

ffpack.lsm.windSpectra.**ec1Spectrum**(*n*, *uz*, *sigma=0.03*, *z=10*, *tcat=0*, *normalized=True*)

EC1 spectrum is implemented according to Annex B [EN1991-1-42005].

Parameters

- **n** (*scalar*) – Frequency (Hz) when normalized=False. Normalized frequency when normalized=True.
- **uz** (*scalar*) – Mean wind speed (m/s) measured at height z.
- **sigma** (*scalar, optional*) – Standard derivation of wind.
- **z** (*scalar, optional*) – Height above the ground (m), default to 10 m.
- **tcat** (*scalar, optional*) – Terrain category, could be 0, 1, 2, 3, 4 Default to 0 (sea or coastal area exposed to the open sea) in EC1 Table 4.1.
- **normalized** (*bool, optional*) – If normalized is set to False, the power spectrum density will be returned.

Returns

rst – Power spectrum density (m^2 s^-2 Hz^-1) when normalized=False. Normalized power spectrum density when normalized=True.

Return type

scalar

Raises

ValueError – If n is not a scalar. If uz is not a scalar. If tcat is not int or not within range of 0 to 4

Examples

```
>>> from ffpack.lsm import ec1Spectrum
>>> n = 2
>>> uz = 10
>>> rst = ec1Spectrum( n, uz, sigma=0.03, z=10, tcat=0, normalized=True )
```

References

ffpack.lsm.windSpectra.**iecSpectrum**(*f*, *vhub*, *sigma=0.03*, *z=10*, *k=1*, *normalized=True*)

IEC spectrum is implemented according to [IEC2005].

Parameters

- **f** (*scalar*) – Frequency (Hz) when normalized=False. Normalized frequency when normalized=True.
- **vhub** (*scalar*) – Mean wind speed (m/s).
- **sigma** (*scalar, optional*) – Standard derivation of the turblent wind speed component.
- **z** (*scalar, optional*) – Height above the ground (m), default to 10 m.
- **k** (*scalar, optional*) – Wind speed direction, could be 1, 2, 3 (1 = longitudinal, 2 = lateral, and 3 = upward) Default to 1 (longitudinal).

- **normalized** (*bool, optional*) – If normalized is set to False, the power spectrum density will be returned.

Returns

rst – Single-sided velocity component power spectrum density (m^2 s^-2 Hz^-1) when normalized=False. Normalized single-sided velocity component power spectrum density when normalized=True.

Return type

scalar

Raises

ValueError – If n is not a scalar. If uz is not a scalar. If k is not int or not within range of 1 to 3

Examples

```
>>> from ffpack.lsm import iecSpectrum
>>> n = 2
>>> vhub = 10
>>> rst = iecSpectrum( n, vhub, sigma=0.03, z=10, k=1, normalized=True )
```

References

Sequence spectra

ffpack.lsm.sequenceSpectra.**periodogramSpectrum**(*data*, *fs*)

Power spectral density with *scipy.signal.periodogram*.

Parameters

- **data** (*1darray*) – Sequence to calculate power spectral density.
- **fs** (*scalar*) – Sampling frequency.

Returns

- **freq** (*1darray*) – frequency components.
- **psd** (*1darray*) – Power spectral density.

Raises

ValueError – If data is not a 1darray. If fs is not a scalar.

Examples

```
>>> from ffpack.lsm import periodogramSpectrum
>>> data = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 2 ]
>>> fs = 2
>>> freq, psd = periodogramSpectrum( data, fs )
```

ffpack.lsm.sequenceSpectra.**welchSpectrum**(*data*, *fs*, *nperseg=1024*)

Power spectral density with *scipy.signal.welch*.

Parameters

- **data** (*1darray*) – Sequence to calculate power spectral density.

- **fs** (*scalar*) – Sampling frequency.
- **nperseg** (*scalar*) – Length of each segment. Defaults to 1024.

Returns

- **freq** (*1darray*) – frequency components.
- **psd** (*1darray*) – Power spectral density.

Raises

ValueError – If data is not a 1darray. If fs is not a scalar.

Examples

```
>>> from ffpack.lsm import welchSpectrum
>>> data = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 2 ]
>>> fs = 2
>>> freq, psd = welchSpectrum( data, fs, nperseg=1024 )
```

Cycle counting matrix

ffpack.lsm.cycleCountingMatrix.**astmRainflowCountingMatrix**(*data*, *resolution=0.5*)

Calculate ASTM rainflow counting matrix.

Parameters

- **data** (*1d array*) – Sequence data to calculate rainflow counting matrix.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

ValueError – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.lsm import astmRainflowCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = astmRainflowCountingMatrix( data )
```

ffpack.lsm.cycleCountingMatrix.**astmRainflowRepeatHistoryCountingMatrix**(*data*, *resolution=0.5*)

Calculate ASTM simplified rainflow counting matrix for repeating histories.

Parameters

- **data** (*1d array*) – Sequence data to calculate simplified rainflow counting matrix for repeating histories.

- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

ValueError – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.lsm import astmRainflowRepeatHistoryCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = astmRainflowRepeatHistoryCountingMatrix( data )
```

ffpack.lsm.cycleCountingMatrix.**astmRangePairCountingMatrix**(*data*, *resolution=0.5*)

Calculate ASTM range pair counting matrix.

Parameters

- **data** (*1d array*) – Sequence data to calculate range pair counting matrix.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

ValueError – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.lsm import astmRangePairCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = astmRangePairCountingMatrix( data )
```

ffpack.lsm.cycleCountingMatrix.**astmSimpleRangeCountingMatrix**(*data*, *resolution=0.5*)

Calculate ASTM simple range counting matrix.

Parameters

- **data** (*1d array*) – Sequence data to calculate range counting matrix.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

`ValueError` – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0.

Examples

```
>>> from ffpack.lsm import astmSimpleRangeCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = astmSimpleRangeCountingMatrix( data )
```

ffpack.lsm.cycleCountingMatrix.**fourPointCountingMatrix**(*data*, *resolution=0.5*)

Calculate Four point cycle counting matrix.

Parameters

- **data** (*1d array*) – Sequence data to calculate rainflow counting matrix.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

`ValueError` – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.lsm import fourPointCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = fourPointCountingMatrix( data )
```

ffpack.lsm.cycleCountingMatrix.**johannessonMinMaxCountingMatrix**(*data*, *resolution=0.5*)

Calculate Johannesson minMax cycle counting matrix.

Parameters

- **data** (*1d array*) – Sequence data to calculate rainflow counting matrix.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

`ValueError` – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.lsm import johannessonMinMaxCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = johannessonMinMaxCountingMatrix( data )
```

`ffpack.lsm.cycleCountingMatrix.`**rychlikRainflowCountingMatrix**(*data*, *resolution=0.5*)

Calculate Rychlik rainflow counting matrix.

Parameters

- **data** (*1d array*) – Sequence data to calculate rainflow counting matrix.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

`ValueError` – If the data dimension is not 1. If the data length is less than 2.

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.lsm import rychlikRainflowCountingMatrix
>>> data = [ -2.0, 1.0, -3.0, 5.0, -1.0, 3.0, -4.0, 4.0, -2.0 ]
>>> rst, matrixIndexKey = rychlikRainflowCountingMatrix( data )
```

1.12.5 rpm random and probabilistic model

Metropolis Hastings algorithm

class ffpack.rpm.metropolisHastings.**AuModifiedMHSampler**(*initialVal=None*, *targetPdf=None*, *proposalCSampler=None*, *sampleDomain=None*, *randomSeed=None*, ***sdKwargs*)

Modified Metropolis-Hastings sampler based on Au and Beck algorithm [Au2001].

References

__init__(*initialVal=None*, *targetPdf=None*, *proposalCSampler=None*, *sampleDomain=None*, *randomSeed=None*, ***sdKwargs*)

Initialize the Au modified Metropolis-Hastings sampler

Parameters

- **initialVal** (*array_like*) – Initial observed data point.
- **targetPdf** (*function list*) – Target probability density function list. Each element targetPdf[i] in the list is a callable function referring the independent marginal. targetPdf[i] takes one input parameter and return the corresponding probability. It will be called as targetPdf[i](X[i]) where X is a list in which the element is same type as initialVal[i], and a scalar value of probability should be returned by targetPdf[i](X[i]).
- **proposalCSampler** (*function list*) – Proposal conditional sampler list (i.e., transition kernel list). Each element proposalCSampler[i] in the list is a callable function referring a sampler that will return a sample for the given observed data point. A usual choice is to let proposalCSampler[i] be a Gaussian/normal distribution centered at the observed data point. It will be called as proposalCSampler[i](X[i]) where X is a list in which each element is the same type as initialVal[i], and a sample with the same type of initialVal[i] should be returned.
- **sampleDomain** (*function, optional*) – Sample domain function. sampleDomain is a function to determine if a sample is in the sample domain. For example, if the sample doamin is [0, inf] and the sample is -2, the sample will be rejected. For the sampling on field of real numbers, it should return True regardless of the sample value. It called as sampleDomain(cur, nxt, **sdKwargs) where cur, nxt are lists in which each element is the same type as initivalVal[i], and a boolean value should be returned.
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Raises

ValueError – If initialVal, targetPdf, or proposalCSampler is None. If dims of initialVal, targetPdf, and proposalCSampler are not equal. If targetPdf returns negative value.

Examples

```
>>> from ffpack.rpm import AuModifiedMHSampler
>>> initialValList = [ 1.0, 1.0 ]
>>> targetPdf = [ lambda x : 0 if x < 0 else np.exp( -x ),
...               lambda x : 0 if x < 0 else np.exp( -x ) ]
>>> proposalCSampler = [ lambda x : np.random.normal( x, 1 ),
...                      lambda x : np.random.normal( x, 1 ) ]
>>> auMMHSampler = AuModifiedMHSampler( initialVal, targetPdf,
...                                     proposalCsampler )
```

getSample()

Get a sample.

Returns
rst – Data point sample.

Return type
array_like

Examples

```
>>> sample = auMMHSampler.getSample()
```

class ffpack.rpm.metropolisHastings.**MetropolisHastingsSampler**(*initialVal=None*, *targetPdf=None*, *proposalCSampler=None*, *sampleDomain=None*, *randomSeed=None*, ***sdKwargs*)

Metropolis-Hastings sampler to sample data [Bourinet2018].

References

__init__(*initialVal=None*, *targetPdf=None*, *proposalCSampler=None*, *sampleDomain=None*, *randomSeed=None*, ***sdKwargs*)

Initialize the Metropolis-Hastings sampler

Parameters

- **initialVal** (*scalar or array_like*) – Initial observed data point.
- **targetPdf** (*function*) – Target probability density function or target distribution function. targetPdf takes one input parameter and return the corresponding probability. It will be called as targetPdf(X) where X is the same type as initialVal, and a scalar value of probability should be returned.
- **proposalCSampler** (*function*) – Proposal conditional sampler (i.e., transition kernel). proposalCSampler is a sampler that will return a sample for the given observed data point. A usual choice is to let proposalCSampler be a Gaussian/normal distribution centered at the observed data point. It will be called as proposalCSampler(X) where X is the same type as initialVal, and a sample with the same type of initialVal should be returned.
- **sampleDomain** (*function, optional*) – Sample domain function. sampleDomain is a function to determine if a sample is in the sample domain. For example, if the sample doamin is [0, inf] and the sample is -2, the sample will be rejected. For the sampling on field of real numbers, it should return True regardless of the sample value. It called as

sampleDomain(cur, nxt, **sdKwargs) where cur, nxt are the same type as initivalVal, and a boolean value should be returned.

- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Raises

ValueError – If initialVal, targetPdf, or proposalCSampler is None. If targetPdf returns negative value.

Examples

```
>>> from ffpack.rpm import MetropolisHastingsSampler
>>> initialVal = 1.0
>>> targetPdf = lambda x : 0 if x < 0 else np.exp( -x )
>>> proposalCSampler = lambda x : np.random.normal( x, 1 )
>>> mhSampler = MetropolisHastingsSampler( initialVal, targetPdf,
...                                        proposalCsampler )
```

getSample()

Get a sample.

Returns

rst – Data point sample.

Return type

scalar or array_like of scalar

Examples

```
>>> sample = mhSampler.getSample()
```

Nataf transformation

This module implements the Nataf distribution. The Gaussian quadrature and root finding algorithm are used to determine the integral resutls and rho' of Eq.(12) in reference[1]. Reference[2] uses the Gauss–Legendre quadrature to calculate the integral results. With the transformation, Gauss–Hermite quadrature can also be used to calculate the integral results, as indicated in reference[3]. However, the iteration method proposed in reference[3] requires Cholesky decomposition in the iteration. This module follows the Nataf transformation method implemented in reference[2].

class ffpack.rpm.nataf.**NatafTransformation**(*distObjs*, *corrMat*, *quadDeg=99*, *quadRange=8*, *randomSeed=None*)

Nataf distribution for correlated marginal distributions.

__init__(*distObjs*, *corrMat*, *quadDeg=99*, *quadRange=8*, *randomSeed=None*)

Initialize the Nataf distribution.

Parameters

[1] Liu, P.L. and Der Kiureghian, A., 1986. Multivariate distribution models with prescribed marginals and covariances. Probabilistic engineering mechanics, 1(2), pp.105-112.

[2] Ehre, M., Geyer, S., Kamariotis, A., Papaioannou, I., Sardi, L., 2022. Documentation of the ERA Distribution Classes.

[3] Li, H., Lü, Z. and Yuan, X., 2008. Nataf transformation based point estimate method. Chinese Science Bulletin, 53(17), pp.2586-2592.

- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **quadDeg** (*integer*) – Quadrature degree.
- **quadRange** (*scalar*) – Quadrature range. The integral will be performed in the range [-quadRange, quadRange].
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Raises
ValueError – If distObjs is empty. If dimensions are not match for distObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Examples

```
>>> from ffpack.rpm import NatafTransformation
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = [ [ 1.0, 0.5 ], [ 0.5, 1.0 ] ]
>>> natafDist = NatafTransformation( distObjs=distObjs, corrMat=corrMat )
```

cdf(*X*)

Get cdf value for X.

Parameters
X (*1d array*) – Data point in X space.

Returns
rst – Value of cumulative distribution function at data point X.

Return type
scalar

Raises
ValueError – If length of X does not match dim. If X is not 1d array.

Examples

```
>>> X = [ 0.5, 1.5 ]
>>> rst = natafDist.cdf( X )
```

getSample()

Get a sample in X space from Nataf distribution

Returns
X – Data point in X space.

Return type
1d array

Examples

```
>>> X = natafDist.getSample()
```

getU(*X*)

Get data point in U space and Jacobian.

Parameters
X (*1d array*) – Data point in X space.

Returns

- **U** (*1d array*) – Data point in U space.
- **J** (*2d matrix*) – Jacobian from X space to U space.

Raises
ValueError – If length of X does not match dim. If X is not 1d array.

Notes

X -> Z -> U

Examples

```
>>> X = [ 0.5, 1.5 ]
>>> U, J = natafDist.getU( X )
```

getX(*U*)

Get data point in X space and Jacobian.

Parameters
U (*1d array*) – Data point in U space.

Returns

- **X** (*1d array*) – Data point in X space.
- **J** (*2d matrix*) – Jacobian from U space to X space.

Raises
ValueError – If length of U does not match dim. If U is not 1d array.

Notes

U -> Z -> X

Examples

```
>>> U = [ 0.5, 1.5 ]
>>> X, J = natafDist.getX( U )
```

pdf(*X*)

Get pdf value for X.

Parameters
X (*1d array*) – Data point in X space.

Returns
rst – Value of probability density function at data point X.

Return type
scalar

Raises
ValueError – If length of X does not match dim. If X is not 1d array.

Examples

```
>>> X = [ 0.5, 1.5 ]
>>> rst = natafDist.pdf( X )
```

1.12.6 rrm risk and reliability model

First order second moment

ffpack.rrm.firstOrderSecondMoment.**mvalFOSM**(*dim*, *g*, *dg*, *mus*, *sigmas*, *dx=1e-06*)

First order second moment method based on mean value algorithm.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, …]).
- **dg** (*array_like of function*) – Gradient of the limit state function. It should be an array_like of function like dg = [dg_dx1, dg_dx2, …]. To get the derivative of i-th random variable at (x1*, x2*, …), dg[i](x1*, x2*, …) will be called. dg can be None, see the following Notes.
- **mus** (*1d array*) – Mean of the random variables.
- **sigmas** (*1d array*) – Variance of the random variables.
- **dx** (*scalar, optional*) – Spacing for auto differentiation. Not required if dg is provided.

Returns

- **beta** (*scalar*) – Reliability index.
- **pf** (*scalar*) – probability of failure.

Raises
ValueError – If the dim is less than 1. If the dim does not match the length of mus and sigmas.

Notes

If dg is None, the numerical differentiation will be used. The tolerance of the numerical differentiation can be changed in globalConfig.

Examples

```
>>> from ffpack.rrm import mvalFOSM
>>> dim = 2
>>> g = lambda X: 3 * X[ 0 ] - 2 * X[ 1 ]
>>> dg = [ lambda X: 3, lambda X: -2 ]
>>> mus = [ 1, 1 ]
>>> sigmas = [ 3, 4 ]
>>> beta, pf = mvalFOSM( dim, g, dg, mus, sigmas)
```

First order reliability method

ffpack.rrm.firstOrderReliabilityMethod.**coptFORM**(*dim*, *g*, *distObjs*, *corrMat*, *quadDeg=99*, *quadRange=8*)

First order reliability method based on constrained optimization.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, ...]).
- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **quadDeg** (*integer*) – Quadrature degree for Nataf transformation
- **quadRange** (*scalar*) – Quadrature range for Nataf transformation. The integral will be performed in the range [-quadRange, quadRange].

Returns

- **beta** (*scalar*) – Reliability index.
- **pf** (*scalar*) – Probability of failure.
- **uCoord** (*1d array*) – Design point coordinate in U space.
- **xCoord** (*1d array*) – Design point coordinate in X space.

Raises

ValueError – If the dim is less than 1. If the dim does not match the disObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Examples

```
>>> from ffpack.rrm import coptFORM
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> beta, pf, uCoord, xCoord = coptFORM( dim, g, distObjs, corrMat )
```

ffpack.rrm.firstOrderReliabilityMethod.**hlrfFORM**(*dim*, *g*, *dg*, *distObjs*, *corrMat*, *iter=1000*, *tol=1e-06*, *quadDeg=99*, *quadRange=8*, *dx=1e-06*)

First order reliability method based on Hasofer-Lind-Rackwitz-Fiessler algorithm.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, …]).
- **dg** (*array_like of function*) – Gradient of the limit state function. It should be an array_like of function like dg = [dg_dx1, dg_dx2, …]. To get the derivative of i-th random variable at (x1*, x2*, …), dg[i](x1*, x2*, …) will be called. dg can be None, see the following Notes.
- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **iter** (*integer*) – Maximum iteration steps.
- **tol** (*scalar*) – Tolerance to demtermine if the iteration converges.
- **quadDeg** (*integer*) – Quadrature degree for Nataf transformation
- **quadRange** (*scalar*) – Quadrature range for Nataf transformation. The integral will be performed in the range [-quadRange, quadRange].
- **dx** (*scalar, optional*) – Spacing for auto differentiation. Not required if dg is provided.

Returns

- **beta** (*scalar*) – Reliability index.
- **pf** (*scalar*) – Probability of failure.
- **uCoord** (*1d array*) – Design point coordinate in U space.
- **xCoord** (*1d array*) – Design point coordinate in X space.

Raises

ValueError – If the dim is less than 1. If the dim does not match the disObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Notes

If dg is None, the numerical differentiation will be used. The tolerance of the numerical differentiation can be changed in globalConfig.

Examples

```
>>> from ffpack.rrm import hlrfFORM
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> dg = [ lambda X: -1, lambda X: -1 ]
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> beta, pf, uCoord, xCoord = hlrfFORM( dim, g, dg, distObjs, corrMat )
```

Second order reliability method

ffpack.rrm.secondOrderReliabilityMethod.**breitungSORM**(*dim*, *g*, *dg*, *distObjs*, *corrMat*, *quadDeg=99*, *quadRange=8*, *dx=1e-06*)

Second order reliability method based on Breitung algorithm.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, …]).
- **dg** (*array_like of function*) – Gradient of the limit state function. It should be an array_like of function like dg = [dg_dx1, dg_dx2, …]. To get the derivative of i-th random variable at (x1*, x2*, …), dg[i](x1*, x2*, …) will be called. dg can be None, see the following Notes.
- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **quadDeg** (*integer*) – Quadrature degree for Nataf transformation
- **quadRange** (*scalar*) – Quadrature range for Nataf transformation. The integral will be performed in the range [-quadRange, quadRange].
- **dx** (*scalar, optional*) – Spacing for auto differentiation. Not required if dg is provided.

Returns

- **beta** (*scalar*) – Reliability index.
- **pf** (*scalar*) – Probability of failure.
- **uCoord** (*1d array*) – Design point coordinate in U space.
- **xCoord** (*1d array*) – Design point coordinate in X space.

Raises

ValueError – If the dim is less than 1. If the dim does not match the disObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Notes

If dg is None, the numerical differentiation will be used. The tolerance of the numerical differentiation can be changed in globalConfig.

Examples

```
>>> from ffpack.rrm import breitungSORM
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> dg = [ lambda X: -1, lambda X: -1 ]
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> beta, pf, uCoord, xCoord = breitungSORM( dim, g, dg, distObjs, corrMat )
```

ffpack.rrm.secondOrderReliabilityMethod.**hrackSORM**(*dim*, *g*, *dg*, *distObjs*, *corrMat*, *quadDeg=99*, *quadRange=8*, *dx=1e-06*)

Second order reliability method based on Hohenbichler and Rackwitz algorithm.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, …]).
- **dg** (*array_like of function*) – Gradient of the limit state function. It should be an array_like of function like dg = [dg_dx1, dg_dx2, …]. To get the derivative of i-th random variable at (x1*, x2*, …), dg[i](x1*, x2*, …) will be called. dg can be None, see the following Notes.
- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **quadDeg** (*integer*) – Quadrature degree for Nataf transformation
- **quadRange** (*scalar*) – Quadrature range for Nataf transformation. The integral will be performed in the range [-quadRange, quadRange].
- **dx** (*scalar, optional*) – Spacing for auto differentiation. Not required if dg is provided.

Returns

- **beta** (*scalar*) – Reliability index.
- **pf** (*scalar*) – Probability of failure.
- **uCoord** (*1d array*) – Design point coordinate in U space.
- **xCoord** (*1d array*) – Design point coordinate in X space.

Raises

ValueError – If the dim is less than 1. If the dim does not match the disObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Notes

If dg is None, the numerical differentiation will be used. The tolerance of the numerical differentiation can be changed in globalConfig.

Examples

```
>>> from ffpack.rrm import tvedtSORM
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> dg = [ lambda X: -1, lambda X: -1 ]
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> beta, pf, uCoord, xCoord = hrackSORM( dim, g, dg, distObjs, corrMat )
```

ffpack.rrm.secondOrderReliabilityMethod.**tvedtSORM**(*dim*, *g*, *dg*, *distObjs*, *corrMat*, *quadDeg=99*, *quadRange=8*, *dx=1e-06*)

Second order reliability method based on Tvedt algorithm.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, …]).
- **dg** (*array_like of function*) – Gradient of the limit state function. It should be an array_like of function like dg = [dg_dx1, dg_dx2, …]. To get the derivative of i-th random variable at (x1*, x2*, …), dg[i](x1*, x2*, …) will be called. dg can be None, see the following Notes.
- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **quadDeg** (*integer*) – Quadrature degree for Nataf transformation
- **quadRange** (*scalar*) – Quadrature range for Nataf transformation. The integral will be performed in the range [-quadRange, quadRange].
- **dx** (*scalar, optional*) – Spacing for auto differentiation. Not required if dg is provided.

Returns

- **beta** (*scalar*) – Reliability index.
- **pf** (*scalar*) – Probability of failure.
- **uCoord** (*1d array*) – Design point coordinate in U space.
- **xCoord** (*1d array*) – Design point coordinate in X space.

Raises

ValueError – If the dim is less than 1. If the dim does not match the disObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Notes

If dg is None, the numerical differentiation will be used. The tolerance of the numerical differentiation can be changed in globalConfig.

Examples

```
>>> from ffpack.rrm import tvedtSORM
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> dg = [ lambda X: -1, lambda X: -1 ]
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> beta, pf, uCoord, xCoord = tvedtSORM( dim, g, dg, distObjs, corrMat )
```

Simulation based reliability method

ffpack.rrm.simulationBasedReliabilityMethod.**subsetSimulation**(*dim*, *g*, *distObjs*, *corrMat*, *numSamples*, *maxSubsets*, *probLevel=0.1*, *quadDeg=99*, *quadRange=8*, *randomSeed=None*)

Second order reliability method based on Breitung algorithm.

Parameters

- **dim** (*integer*) – Space dimension (number of random variables).
- **g** (*function*) – Limit state function. It will be called like g([x1, x2, ...]).
- **distObjs** (*array_like of distributions*) – Marginal distribution objects. It should be the freezed distribution objects with pdf, cdf, ppf. We recommend to use scipy.stats functions.
- **corrMat** (*2d matrix*) – Correlation matrix of the marginal distributions.
- **numSamples** (*integer*) – Number of samples in each subset.
- **maxSubsets** (*scalar*) – Maximum number of subsets used to compute the failure probability.
- **probLevel** (*scalar, optional*) – Probability level for intermediate subsets.
- **quadDeg** (*integer, optional*) – Quadrature degree for Nataf transformation
- **quadRange** (*scalar, optional*) – Quadrature range for Nataf transformation. The integral will be performed in the range [-quadRange, quadRange].
- **randomSeed** (*integer, optional*) – Random seed. If randomSeed is none or is not an integer, the random seed in global config will be used.

Returns

- **pf** (*scalar*) – Probability of failure.
- **allLsfValue** (*array_like*) – Values of limit state function in each subset.
- **allUSamples** (*array_like*) – Samples of U space in each subset.
- **allXSamples** (*array_like*) – Samples of X space in each subset.

Raises

ValueError – If the dim is less than 1. If the dim does not match the disObjs and corrMat. If corrMat is not 2d matrix. If corrMat is not positive definite. If corrMat is not symmetric. If corrMat diagonal is not 1.

Notes

Nataf transformation is used for the marginal distributions.

Examples

```
>>> from ffpack.rrm import subsetSimulation
>>> dim = 2
>>> g = lambda X: -np.sum( X ) + 1
>>> distObjs = [ stats.norm(), stats.norm() ]
>>> corrMat = np.eye( dim )
>>> numSamples, maxSubsets = 500, 10
>>> pf = subsetSimulation( dim, g, distObjs, corrMat, numSamples, maxSubsets )
```

1.12.7 `utils` utility methods

Aggregation

ffpack.utils.aggregation.**cycleCountingAggregation**(*data*, *binSize=1.0*)

Count the number of occurrences of each cycle digitized to the nearest bin.

Parameters

- **data** (*2d array*) – Input cycle counting data [[value, count], …] for bin collection
- **binSize** (*scalar, optional*) – bin size is the difference between each level, for example, binSize=1.0, the levels will be 0.0, 1.0, 2.0, 3.0 …

Returns

rst – Aggregated [[aggregatedValue, count]] by the binSize

Return type

2d array

Raises

ValueError – If the data dimension is not 2. If the data is empty

Notes

When a value is in the middle, it will be counted downward for example, 0.5 when binSize=1.0, the count will be counted to 0.0

Examples

```
>>> from ffpack.utils import cycleCountingAggregation
>>> data = [ [ 1.7, 2.0 ], [ 2.2, 2.0 ] ]
>>> rst = cycleCountingAggregation( data )
```

Counting matrix

ffpack.utils.countingMatrix.**countingRstToCountingMatrix**(*countingRst*)

Calculate counting matrix from rainflow counting result.

Parameters

countingRst (*2d array*) – Cycle counting result in form of [[rangeStart1, rangeEnd1, count1], [rangeStart2, rangeEnd2, count2], …].

Returns

- **rst** (*2d array*) – A matrix contains the counting results.
- **matrixIndexKey** (*1d array*) – A sorted array contains the index keys for the counting matrix.

Raises

ValueError – If the data dimension is not 2. If the data is not empty and not in dimension of n by 3.

Examples

```
>>> from ffpack.lsm import countingRstToCountingMatrix
>>> countingRst = [ [ -2.0, 1.0, 1.0 ], [ 5.0, -1.0, 3.0 ], [ -4.0, 4.0, 0.5 ] ]
>>> rst, matrixIndexKey = countingRstToCountingMatrix( countingRst )
```

Derivatives

ffpack.utils.derivatives.**centralDiffWeights**(*Np*, *ndiv=1*)

Return weights for an Np-point central derivative[1].

This function came from scipy.misc module, we put it here since scipy.misc module is completely removed in SciPy v1.12.0.

Assumes equally-spaced function points.

If weights are in the vector w, then derivative is w[0] * f(x - ho * dx) + … + w[-1] * f(x + h0 * dx)

Parameters

- **Np** (*integer*) – Number of points for the central derivative.
- **ndiv** (*integer, optional*) – Number of divisions. Default is 1.

Returns

w – Weights for an Np-point central derivative. Its size is *Np*.

Return type

ndarray

[1] https://en.wikipedia.org/wiki/Finite_difference

Notes

Can be inaccurate for a large number of points.

Examples

```
>>> def f( x ):
...     return 2 * x**2 + 3
>>> x = 3.0 # derivative point
>>> h = 0.1 # differential step
>>> Np = 3 # point number for central derivative
>>> weights = centralDiffWeights( Np ) # weights for first derivative
>>> vals = [ f( x + ( i - Np / 2 ) * h) for i in range( Np )]
>>> sum( w * v for (w, v) in zip( weights, vals ) ) / h
11.79999999999998
This value is close to the analytical solution:
f'(x) = 4x, so f'(3) = 12
```

References

ffpack.utils.derivatives.**derivative**(*func*, *x0*, *dx=1.0*, *n=1*, *args=()*, *order=3*)

Find the n-th derivative of a function at a point.

This function came from scipy.misc module, we put it here since scipy.misc module is completely removed in SciPy v1.12.0.

Given a function, use a central difference formula with spacing *dx* to compute the nth derivative at *x0*.

Parameters

- **func** (*function*) – Input function.
- **x0** (*float*) – The point at which the nth derivative is found.
- **dx** (*float, optional*) – Spacing.
- **n** (*int, optional*) – Order of the derivative. Default is 1.
- **args** (*tuple, optional*) – Arguments
- **order** (*int, optional*) – Number of points to use, must be odd.

Notes

Decreasing the step size too small can result in round-off error.

Examples

```
>>> def f(x):
...     return x**3 + x**2
>>> derivative( f, 1.0, dx=1e-6 )
4.9999999999217337
```

ffpack.utils.derivatives.**gradient**(*func*, *nvar*, *n=1*, *dx=0.001*, *order=3*)

Find n-th gradient of a scalar-valued differentiable function.

Parameters

- **func** (*function*) – Input scalar-valued differentiable function.
- **nvar** (*integer*) – The number of input variables for the input function. Input function will be called like func(X) = func([X[0], X[1], ..., X[nvar - 1]]).
- **n** (*integer, optional*) – Order of the derivative. Default is 1.
- **dx** (*scalar, optional*) – Spacing for derivative calculation.
- **order** (*integer, optional*) – Number of points used for central derivative weights, must be odd.

Returns

rst – n-th gradient of function, i.e., [$\partial^n f/\partial X_0^n, \ldots, \partial^n f/\partial X_{nvar}^n$]. In general, the i-th element in the list is the n-th derivative of the func w.r.t. i-th input variable. Therefore, rst[i] = $\partial^n f/\partial X_i^n$ and it can be called like rst[i](X0) to evaluate the n-th derivative w.r.t. i-th variable at point X0. It should be noted that X0 MUST be a list, NOT a numpy array.

Return type

1d array

Notes

Decreasing the step size too small can result in round-off error.

Examples

```
>>> def f( X ):
...     return X[ 0 ]**3 + X[ 1 ]**2
>>> gradient( f, nvar=2, n=1 )
Output will be a function list of the 1st derivative of func
>>> [ lambda X: 3 * X[ 0 ]**2, lambda X: 2 * X[ 1 ] ]
>>> gradient( f, nvar=2, n=2 )
Output will be a function list of the 2nd derivative of func
>>> [ lambda X: 6 * X[ 0 ], lambda X: 2 ]
```

ffpack.utils.derivatives.**gramSchmidOrth**(*A*, *alignVec=None*)

Perform Gram-Schmidt orthogonization to matrix.

Parameters

- **A** (*2d matrix*) – Input matrix. The orthogonization is performed w.r.t. each column vector. The input matrix must be a square matrix.

- **alignVec** (*1d array*) – If alignVec exists, the alignVec will be the first vector for orthogonization.

Returns

- **B** (*2d matrix*) – The matrix in which each column vector is orthogonized.
- **J** (*2d matrix*) – The transformation matrix to perform the orthogonization, e.g., JA = B

Notes

A must be square and of full-rank, i.e., all rows (or, equivalently, columns) must be linearly independent.

Examples

```
>>> A = [ [ 1, 0 ], [ 0, 1 ] ]
>>> alignVec = [ 0.5, 0.5 ]
>>> B, J = gramSchmidOrth( A, alignVec )
```

ffpack.utils.derivatives.**hessianMatrix**(*func*, *nvar*, *dx=0.001*, *order=3*)

Find Hessian matrix for a scalar-valued differentiable function.

Parameters

- **func** (*function*) – Input scalar-valued differentiable function.
- **nvar** (*integer*) – The number of input variables for the input function. Input function will be called like func(X) = func([X[0], X[1], ..., X[nvar - 1]]).
- **dx** (*scalar, optional*) – Spacing for derivative calculation.
- **order** (*integer, optional*) – Number of points used for central derivative weights, must be odd.

Returns

rst – Hessian matrix. rst[i][j] = $\partial f/(\partial X_i \partial X_j)$. It can be called like rst[i][j](X0) to evaluate the value at point X0. It should be noted that X0 MUST be a list, NOT a numpy array.

Return type

2d array

Notes

Decreasing the step size too small can result in round-off error.

Examples

```
>>> def f( X ):
...     return X[ 0 ]**3 + X[ 1 ]**2
>>> hessianMatrix( f, nvar=2 )
Output will be a function list of the 1st derivative of func
>>> [ [ lambda X: 6 * X[ 0 ], lambda X: 0 ],
...   [ lambda X: 0, lambda X: 2] ]
```

Digitization

ffpack.utils.digitization.**sequenceDigitization**(*data*, *resolution=1.0*)

Digitize the sequence data to a specific resolution

The sequence data are digitized by the round method.

Parameters

- **data** (*1d array*) – Sequence data to digitize.
- **resolution** (*bool, optional*) – The desired resolution to round the data points.

Returns

rst – A list contains the digitized data.

Return type

1d array

Raises

ValueError – If the data dimension is not 1. If the data length is less than 2 with keedEnds == False If the data length is less than 3 with keedEnds == True

Notes

The default round function will round half to even: 1.5, 2.5 => 2.0:

Examples

```
>>> from ffpack.utils import sequenceDigitization
>>> data = [ -1.0, 2.3, 1.8, 0.6, -0.4, 0.8, -1.6, -2.5, 3.4, 0.3, 0.1 ]
>>> rst = sequenceDigitization( data )
```

Fitter

class ffpack.utils.fitter.**SnCurveFitter**(*data*, *fatigueLimit*)

Fitter for a SN curve based on the experimental data.

__init__(*data*, *fatigueLimit*)

Initialize a fitter for a SN curve based on the experimental data.

Parameters

- **data** (*2d array*) – Experimental data for fitting in a 2D matrix, e.g., [[N1, S1], [N2, S2], . . . , [Ni, Si]]
- **fatigueLimit** (*scalar*) – Fatigue limit indicating the minimum S that can cause fatigue.

Raises

ValueError – If the data dimension is not 2. If the data length is less than 2. If the fatigue-Limit is less than or equal 0. If N_i or S_i is less than or equal 0.

Examples

```
>>> from ffpack.utils import SnCurveFitter
>>> data = [ [ 10, 3 ], [ 1000, 1 ] ]
>>> fatigueLimit = 0.5
>>> snCurveFitter = SnCurveFitter( data, fatigueLimit )
```

getN(*S*)

Query fatigue life N for a given S

Parameters
S (*scalar*) – Input S for fatigue life query.

Returns
rst – Fatigue life under the query S. If S is less than or equal fatigueLimit, -1 will be returned.

Return type
scalar

Raises
ValueError – If the S is less than or equal 0.

Examples

```
>>> rst = snCurveFitter.getN( 2 )
```

Sequence filter

ffpack.utils.sequenceFilter.**sequenceHysteresisFilter**(*data*, *gateSize*)

Filter data within the gateSize.

Any cycle that has an amplitude smaller than the gate is removed from the data. This is done by scan the data, i.e., point i, to check if the next points, i.e., i + 1, i + 2, ... are within the gate from point i.

Parameters

- **data** (*1darray*) – Sequence data to get peaks and valleys.
- **gateSize** (*scalar*) – Gate size to filter the data.

Returns
rst – A list contains the filtered data.

Return type
1darray

Raises
ValueError – If the data dimension is not 1. If the data length is less than 2. If gateSize is not a scalar or not positive.

Examples

```
>>> from ffpack.utils import sequenceHysteresisFilter
>>> data = [ 2, 5, 3, 6, 2, 4, 1, 6, 1, 3, 1, 5, 3, 6, 3, 6, 4, 5, 2 ]
>>> gateSize = 3.0
>>> rst = sequenceHysteresisFilter( data, gateSize )
```

ffpack.utils.sequenceFilter.**sequencePeakValleyFilter**(*data*, *keepEnds=False*)

Remove the intermediate value and only get the peaks and valleys of the data

The peak and valley refer the data points that are EXACTLY above and below the neighbors, not equal.

Parameters

- **data** (*1darray*) – Sequence data to get peaks and valleys.
- **keepEnds** (*bool, optional*) – If two ends of the original data should be preserved.

Returns

rst – A list contains the peaks and valleys of the data.

Return type

1darray

Raises

ValueError – If the data dimension is not 1. If the data length is less than 2 with keedEnds == False. If the data length is less than 3 with keedEnds == True.

Examples

```
>>> from ffpack.utils import sequencePeakValleyFilter
>>> data = [ -0.5, 1.0, -2.0, 3.0, -1.0, 4.5, -2.5, 3.5, -1.5, 1.0 ]
>>> rst = sequencePeakValleyFilter( data )
```

1.12.8 config package setting

Global config

class ffpack.config.**GlobalConfig**

Global config for FFPACK

__init__()

Initialize a global config instance with default values.

seed

Seed for random number generator. Default value is None.

Type

scalar

atol

Absolute tolerance in digits. Default value is 8.

Type

scalar

rtol

Relative tolerance in digits. Default value is 5.

Type
scalar

dtol

Derivative tolerance in digits. Default value is 6.

Type
scalar

Examples

```
>>> from ffpack.config import globalConfig
>>> globalConfig.atol = 7
```

setSeed(*seed*)

Set seed for random number generator

Parameters
seed (*scalar*) – Input seed for random number generator

Notes

Set seed to None can clean the seed

Examples

```
>>> from ffpack.config import globalConfig
>>> globalConfig.setSeed( 0 )
```

BIBLIOGRAPHY

[Lee2011] Lee, Y.L., Barkey, M.E. and Kang, H.T., 2011. Metal fatigue analysis handbook: practical problem-solving techniques for computer-aided engineering. Elsevier.

[Guidance2016A] Guidance Notes on Selecting Design Wave by Long Term Stochastic Method

[Guidance2016B] Guidance Notes on Selecting Design Wave by Long Term Stochastic Method

[API2007] API, 2007. Recommended practice 2A-WSD (RP 2A-WSD): Recommnded practice for planning, designing and constructing fixed offshore platforms - working stress design.

[Davenport1961] Davenport, A.G., 1961. The spectrum of horizontal gustiness near the ground in high winds. Quarterly Journal of the Royal Meteorological Society, 87(372), pp.194-211.

[Hiriart2001] Hiriart, D., Ochoa, J.L. and Garcia, B., 2001. Wind power spectrum measured at the San Pedro Mártir Sierra. Revista Mexicana de Astronomia y Astrofisica, 37(2), pp.213-220.

[Ho2003] Ho, T.C.E., Surry, D. and Morrish, D.P., 2003. NIST/TTU cooperative agreement-windstorm mitigation initiative: Wind tunnel experiments on generic low buildings. London, Canada: BLWTSS20-2003, Boundary-Layer Wind Tunnel Laboratory, Univ. of Western Ontario.

[EN1991-1-42005] EN1991-1-4, 2005. Eurocode 1: Actions on structures.

[IEC2005] IEC, 2005. IEC 61400-1, Wind turbines - Part 1: Design requirements.

[Au2001] Au, S.K. and Beck, J.L., 2001. Estimation of small failure probabilities in high dimensions by subset simulation. Probabilistic engineering mechanics, 16(4), pp.263-277.

[Bourinet2018] Bourinet, J.M., 2018. Reliability analysis and optimal design under uncertainty-Focus on adaptive surrogate-based approaches (Doctoral dissertation, Université Clermont Auvergne).

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

M

N

O

P

R

S

T

W